Radare2

Mensi Mohamed Amine

Abstract

Radare2 is an open-source reverse engineering tool for analyzing binaries, offering features like disassembly, debugging, and decompiling across multiple architectures and file formats.

1 Introduction

Radare2 is a versatile toolkit for reverse engineering binaries, supporting tasks like disassembly, debugging, and patching. It is popular among security professionals for its flexibility, though its command-line interface requires a learning curve.

Contents

2 Radare2 Workflow

This is a comprehensive guide to analyzing binaries using Radare2. It includes navigation, disassembly, debugging, memory modification, and automation.

2.1 Opening a Binary

```
r2 ./test_binary          # Non-debug mode
r2 -d ./test_binary       # Debug mode
r2 -A ./test_binary       # Auto-analysis mode
```

Flags:

- `-A`: Run full analysis on load.
- `-d`: Debug mode.

2.2 Performing Analysis

Basic and Full Analysis:

```
aa                  # Basic analysis
aaa                 # Full analysis (functions, xrefs, vars)
afl                 # List functions
afn newname sym.f   # Rename function
```

2.3 Navigating the Binary

View and move around functions:

```
afl # List functions
s sym.main # Seek to main
pdf # Disassemble current function
s 0xADDRESS # Seek to address
s+ 0x10             # Seek forward
s- 0x10 # Seek backward
```

2.4 Viewing Data and Strings

Strings and Sections:

```
izz                 # List strings
iS                  # List sections
s .rodata           # Seek to section
px 64               # Print 64 bytes in hex
```

2.5 Entry Points, Imports, and Exports

View binary metadata:

```
ie                  # Entry point
ii                  # Imports
is                  # Symbols
ii~puts             # Search for 'puts' in imports
is~main             # Search 'main' in symbols
```

2.6 Exploring Cross-References (Xrefs)

```
axt                 # Xrefs to current address
axf                 # Xrefs from function
axtj                # JSON-formatted xrefs
```

Example:

```
s sym.main
axt
```

2.7 Analyzing Function Calls

Disassembly and call chains:

```
pdf~call            # Show lines with 'call'
agf                 # Graph function
```

2.8 Debugging

Start with debugging:

```
r2 -d ./test_binary
```

Commands:

```
db sym.main         # Set breakpoint at main
dc                  # Continue
ds                  # Step into
dr                  # Show registers
```

2.9 Viewing and Modifying Memory

Read/Write Memory:

```
px 64               # Print 64 bytes at current address
wx 41424344         # Write ABCD as hex
wa nop              # Assemble and write NOP
wa jmp 0xADDRESS    # Assemble a JMP instruction
```

2.10 Visual Mode

Activate GUI-like views:

```
V                   # Visual mode
p                   # Disassembly
x                   # Hex view
VV                  # Visual graph mode
```

2.11 Saving the Analysis

Projects:

```
Ps myproject        # Save project
Po myproject        # Open project
```

2.12 Other Useful Commands

```
/bin/sh            # Search for strings
/ 414243           # Hex pattern search (ABCD)
afn myfunc sym.f   # Rename a function
```

2.13 Example Session Summary

```
r2 -A ./test_binary
afl                # List functions
s sym.main         # Seek to main
pdf                # Print disassembly
axt                # Cross-references
izz                # List strings
ii                 # Show imports
db sym.main        # Set breakpoint
dc                 # Continue execution
```

3 File Analysis in radare2

This section provides a **detailed breakdown of File Analysis in Radare2**, with **commands, explanations, and examples**, to help you thoroughly understand how to analyze binaries effectively.

3.1 Opening a Binary

```
r2 ./a.out
```

Opens the binary in **read-only** mode by default.
Use `-w` to enable **write mode** (for patching):

```
r2 -w ./a.out
```

3.2 Initial Analysis Commands

3.2.1 `aa` — Analyze All

```
[0x00000000]> aa
```

Performs:

- Function detection
- Cross-references
- Symbols
- Strings
- Stack frame analysis
- Switches and jump tables

Tip: Use this first after loading a binary.

3.2.2 `aaa` — Aggressive Analysis

```
[0x00000000]> aaa
```

More exhaustive than `aa`. Includes:

- Deeper function analysis
- Block discovery

3.2.3 `aap` — Analyze with Prologues

```
[0x00000000]> aap
```

Uses known function prologues; useful for obfuscated binaries.

3.3 Function Analysis

3.3.1 af — Analyze Function

```
[0x00400510]> af
```

Analyzes the function at the current seek location.

3.3.2 afl — List Functions

```
[0x00400510]> afl
```

Example output:

```
0x00400510    42  3   sym.main
0x00400450    17  1   sym._init
```

3.3.3 afn — Rename a Function

```
[0x00400510]> afn my_function
```

Renames the current function.

3.3.4 pdf — Print Disassembled Function

```
[0x00400510]> pdf
[0x00400510]> pdf @ sym.main
```

Shows:

- Instructions
- Basic block structure
- Function arguments

3.4 Cross-References (Xrefs)

3.4.1 axt — Xrefs To Location

```
[0x00400520]> axt @ sym.main
```

3.4.2 axf — Xrefs From Function

```
[0x00400520]> axf @ sym.main
```

3.5 Control Flow Graphs

1. Open binary: `r2 ./a.out`
2. Type: `aaa`
3. Seek: `s sym.main`
4. Enter graph mode: `VV`

Navigate using arrow keys. Press `?` for help.

3.6 Sections and Segments

3.6.1 `iS` — List Sections

```
[0x00400000]> iS
```

Example:

```
idx=00 vaddr=0x00400000 paddr=0x00000000 sz=0x000003f0 name=.text
```

3.6.2 `iM` — Memory Maps

```
[0x00400000]> iM
```

3.7 Entrypoint, Symbols, and Headers

- `ie` — Show entry point
- `is` — Show symbols (like `nm`)
- `iH` — File header info (ELF, PE, Mach-O)

3.8 Strings and Imports/Exports

- `iz` — Strings in binary
- `ii` — Imported functions
- `iE` — Exported functions

3.9 Analyze Binary Metadata

- `i` — General binary info
- `iL` — Linked libraries

3.10 Example: End-to-End Analysis

```
r2 -w ./a.out
```

Inside r2:

```
aaa             # Full analysis
afl             # List all functions
s sym.main      # Seek to main
pdf             # Disassemble main
axt             # Find calls to main
iz              # List strings
ii              # List imports
```

4 Navigation in radare2

This section provides a **detailed guide to Navigation in Radare2**, packed with explanations and real-world examples. Understanding how to move efficiently through the binary is essential for effective reverse engineering.

4.1 Seek: The Core Navigation Command

4.1.1 `s` — Seek to an Address or Symbol

```
[0x00000000]> s 0x00400510       # Seek to an absolute address
[0x00000000]> s main             # Seek to symbol 'main'
[0x00000000]> s entry0           # Seek to entry point
```

- `s` moves the "seek pointer," which is like the cursor.
- Most commands act on the current seek location.

4.2 Relative Seeking

4.2.1 `s+ <offset>` — Move Forward

```
[0x00400500]> s+ 16             # Move 16 bytes forward
```

4.2.2 `s- <offset>` — Move Backward

```
[0x00400510]> s- 0x10           # Move 16 bytes backward
```

4.3 Bookmarks and Marks

4.3.1 `f` — Flag (bookmark) an Address

```
[0x00400510]> f myfunc          # Flag current address as 'myfunc'
```

Now you can seek to it with:

```
s myfunc
```

You can list all flags with:

```
fs *                            # Show all flagspaces and flags
```

4.3.2 `f- <flag>` — Remove a Flag

```
[0x00400510]> f- myfunc
```

4.4 Show Current Position

4.4.1 `s.` — Show Current Seek

```
[0x00400510]> s.
```

Returns:

```
0x00400510
```

4.5 Named Symbols & Flag Spaces

4.5.1 `fs` — Manage Flagspaces

```
[0x00400510]> fs symbols
```

Switch to the `symbols` flagspace (e.g., for imported/exported names).

4.5.2 `fs *` — List All Flagspaces

```
[0x00400510]> fs *
```

4.6 Program Counter Related (During Debugging)

4.6.1 `dr pc` — Show PC (Instruction Pointer)

```
[0x00400510]> dr pc
```

Returns something like:

```
0x00400510
```

You can also seek to the PC:

```
s `dr pc`
```

4.7 Function & Block Traversal

4.7.1 `afl` — List Functions

```
[0x00400000]> afl
```

Use any function name from here with `s` to navigate.

4.7.2 `pdf` — Print Disassembly of Function

```
[0x00400510]> pdf
```

Use arrow keys in visual mode or `s <addr>` to jump to basic blocks.

4.8 Hex View & Memory Navigation

4.8.1 `px` — Hex Dump

```
[0x00400510]> px 32           # Dump 32 bytes
```

4.8.2 `x` — Show Instruction at Current Address

```
[0x00400510]> x
```

4.9 Visual Navigation (Keyboard-Based)

Enter visual mode:

```
[0x00400000]> V
```

Use the following keys in visual mode:

- `j` / `k`: Scroll down/up
- `l` / `h`: Step into/step back
- `p`: Change panels (code, hex, graph)
- `g <addr>`: Jump to address
- `q`: Exit visual mode

4.10 Jump to Strings, Imports, Functions

4.10.1 From Strings List:

```
[0x00400000]> iz
vaddr=0x00400600 len=12 str="Welcome!"

# Jump to string:
[0x00400000]> s 0x00400600
```

4.10.2 From Imports:

```
[0x00400000]> ii
0x004003e0    sym.imp.puts

# Jump:
[0x00400000]> s sym.imp.puts
```

4.11 Example Navigation Session

```
r2 ./a.out
[0x00000000]> aaa               # Full analysis
[0x00000000]> afl               # List functions
[0x00000000]> s main            # Go to main
[0x00400510]> pdf               # View main function
[0x00400510]> s+ 0x20           # Move forward 32 bytes
[0x00400530]> px 16             # Show 16 bytes in hex
[0x00400530]> V                 # Enter visual mode
```

5 Disassembly and Assembly in radare2

This section provides a **comprehensive guide to Disassembly and Assembly in Radare2**, covering **commands, deep explanations, and practical examples** to help you reverse engineer, understand, or modify machine code effectively.

5.1 Disassembly in Radare2

Disassembly is the process of converting raw machine code into human-readable assembly instructions.

5.1.1 `pd` — Print Disassembly

```
[0x00400510]> pd 10
```

- `pd <n>`: Print disassembly of `n` instructions from the current address.
- Use `pD` to include comments and better formatting.

Example:

```
[0x00400510]> pd 5
            0x00400510      push rbp
            0x00400511      mov rbp, rsp
            0x00400514      sub rsp, 0x10
            0x00400518      mov dword [rbp - 4], 0
            0x0040051f      nop
```

5.1.2 `pd @ <addr>` — Disassemble at Specific Address

```
[0x00400510]> pd 5 @ sym.main
```

5.1.3 `pD` — Print Disassembly with Extra Info

```
[0x00400510]> pD 5
```

Includes:

- Comments
- Function info
- Variable hints

5.1.4 `pdr` — Print Disassembly in Raw Mode

```
[0x00400510]> pdr 5
```

Just raw instructions, no metadata — useful for script output.

5.1.5 `pdf` — Print Disassembled Function

```
[0x00400510]> pdf
```

- Disassembles the **entire function** at the current location. - Shows basic blocks, branches, and structure.

5.2 Assembly in Radare2

Assembly in Radare2 means modifying or inserting instructions — often used for **patching** or **binary exploitation**.

5.2.1 `wa` — Write Assembly Instruction

```
[0x00400520]> wa nop
```

- Assembles and writes a `nop` at the current address.
Multiple instructions:

```
[0x00400520]> wa push rax; pop rax
```

5.2.2 `wa @ <addr>` — Assemble at Specific Address

```
[0x00400520]> wa mov eax, 0 @ 0x00400530
```

5.2.3 `waf` — Assemble a Function from Text

```
[0x00400520]> waf asm_function.asm
```

- Assembles from a provided file (containing assembly instructions).

5.2.4 `w` — Manual Write Commands (Byte-Level)

- `wx`: Write hex bytes
- `wo`: Write opcode

Example for writing hex bytes:

```
[0x00400520]> wx 90              # Write one `nop` byte (0x90)
[0x00400520]> wx 4889e5          # `mov rbp, rsp`
```

Example for writing opcodes:

```
[0x00400520]> wo mov eax, 1
```

5.2.5 `aw` — Assemble and Write Helper Commands

```
[0x00400520]> awc mov eax, 0     # Assemble instruction and convert to C
```

5.3 Enable Write Mode

To patch or assemble code, open the binary in write mode:

```
r2 -w ./a.out
```

Then:

```
[0x00400510]> wa nop
```

Without `-w`, you'll get permission errors when trying to write.

5.4 Patching: Replace Instruction

Step-by-step:

1. **Open in write mode**:

```
r2 -w ./a.out
```

2. **Analyze and seek**:

```
aaa
s main
pdf
```

3. **Replace an instruction**:

```
wa nop                        # Overwrite current instruction
```

4. **Write to disk** (optional, not needed in `-w` mode):

```
wq
```

5.5 Visual Assembly Mode

1. Enter visual mode:

```
V
```

2. Hit `A` to open the **assemble editor**.
3. Type an instruction (e.g., `jmp 0x00400600`).
4. Press **Enter** to write.

5.6 Verify Assembly/Disassembly

5.6.1 `ao` — Analyze Opcode at Current Address

```
[0x00400510]> ao
```

Returns detailed info about the instruction:

- Mnemonic
- Opcode

- Operands
- Length
- Type (`mov`, `jmp`, etc.)

5.6.2 `aoe` — Encode (Assemble) and Show Machine Bytes

```
[0x00400000]> aoe mov eax, 0
```

Returns:

```
opcode: b8 00 00 00 00
```

5.6.3 `aop` — Parse Instruction at Current Address

```
[0x00400000]> aop
```

More analysis-level parsing info (useful in scripting).

5.7 Example: Patching `main()` to Return Immediately

Original:

```
[0x00400510]> pdf
...
0x00400510    push rbp
0x00400511    mov rbp, rsp
...
```

Patch:

```
[0x00400510]> wa ret
```

Overwrites `push rbp` with a `ret`, effectively making `main()` do nothing.

5.8 Summary of Key Commands

Command	Purpose
`pd <n>`	Disassemble `n` instructions
`pdf`	Disassemble function
`wa`	Write (assemble) instruction
`wx <hex>`	Write hex bytes
`ao`	Show info about instruction
`aoe`	Assemble instruction to hex
`V / A`	Visual mode + assembly editor

6 Functions and Symbols in radare2

Here's a **deep dive into Functions and Symbols in Radare2**, with detailed explanations and real examples to help you master identifying, analyzing, renaming, and working with functions and symbols in binary analysis.

6.1 What Are Functions and Symbols?

- **Functions**: Blocks of code that perform specific tasks. In binaries, they may or may not have names.
- **Symbols**: Names attached to addresses — they could be functions, variables, imports, etc.

Radare2 uses **flags** to label addresses (like `sym.main`, `sym.imp.puts`, etc.).

6.2 Analyze and Discover Functions

6.2.1 `aaa` — Analyze Everything (including functions)

```
[0x00000000]> aaa
```

- Detects functions, variables, cross-references, etc.
- Use this first when opening a binary.

6.2.2 `af` — Analyze Function at Current Address

```
[0x00400510]> af
```

- If `aaa` missed a function, you can manually define one here.

6.3 Listing and Describing Functions

6.3.1 `afl` — List All Recognized Functions

```
[0x00000000]> afl
```

Example output:

```
0x00400510   42  3  fcn.00400510
0x00400440   24  1  sym._init
0x00400550   50  2  sym.main
```

Fields:

- Address
- Size (bytes)
- Number of basic blocks
- Function name

6.3.2 `afl main` — Filter Functions by Name

```
[0x00000000]> afl~main
```

6.3.3 `afi` — Info About Current Function

```
[0x00400510]> afi
```

Includes:

- Size
- Calls
- Arguments
- Stack size
- Calling convention

6.3.4 `afcf` — Calling Convention Finder

```
[0x00400510]> afcf
```

- Shows guessed calling convention: `cdecl`, `stdcall`, etc.

6.4 Renaming and Labeling Functions

6.4.1 `afn` — Rename a Function

```
[0x00400510]> afn decrypt_data
```

Now it will show up as `sym.decrypt`$_data$.

6.4.2 `f` — Flag a Custom Symbol (non-function)

```
[0x00400600]> f my_secret_data
```

6.5 Cross-References and Calls

6.5.1 `axt` — Xrefs To (who is calling this?)

```
[0x00400510]> axt @ sym.main
```

6.5.2 `axf` — Xrefs From (who is it calling?)

```
[0x00400510]> axf @ sym.main
```

6.6 Navigating Between Functions

6.6.1 `s sym.main` — Seek to Function by Name

```
[0x00000000]> s sym.main
```

6.6.2 `pdf` — Print Disassembly of Current Function

```
[0x00400510]> pdf
```

6.6.3 `afbj` — Show Function's Basic Blocks in JSON

```
[0x00400510]> afbj
```

`Use this for scripting or automation.`

6.7 Import & Export Symbols

6.7.1 `ii` — List Imported Functions (from shared libraries)

```
[0x00000000]> ii
```

`Example:`

```
0x004003e0    sym.imp.puts
0x004003f0    sym.imp.printf
```

6.7.2 `iE` — List Exported Symbols

```
[0x00000000]> iE
```

`Common in shared libraries (.so, .dll).`

6.8 Auto-Name Functions by Signature

6.8.1 `afvs` — Function Variable Signatures (after `aaa`)

```
[0x00400510]> afvs
```

`Radare2 may suggest variable types or function arguments.`

6.8.2 `afcf`, `afv`, `afvj` — More on Args and Vars

```
[0x00400510]> afv     # Function local vars
[0x00400510]> afvj    # Same in JSON
```

6.9 Define Custom Functions (Manual Analysis)

If you discover code not marked as a function:

```
[0x00400700]> af        # Create a function at this address
[0x00400700]> afn decrypt_buffer
```

6.10 Example Workflow

```
r2 ./a.out
[0x00000000]> aaa                 # Analyze all
[0x00000000]> afl                 # List functions
[0x00000000]> s sym.main         # Seek to main
[0x00400510]> pdf                # View disassembly
[0x00400510]> afn my_main        # Rename main
[0x00400510]> axf                # List function calls
[0x00400510]> afv                # Local vars and args
```

6.11 Summary of Commands

Command	Description
afl	List all functions
af	Define function at current address
afn	Rename a function
afi	Show info about function
pdf	Disassemble current function
axt / axf	Cross-references to/from a function
ii / iE	Imports and exports
afv, afcf	Vars and calling convention

7 Visual Mode in radare2

Here's a comprehensive guide to Visual Mode in Radare2, covering its features, commands, modes, and practical examples. Visual mode (V) is one of the most powerful and interactive ways to analyze binaries in Radare2.

7.1 What Is Visual Mode?

Visual Mode in Radare2 allows interactive navigation and exploration of disassembly, hex dumps, graphs, and more | directly in the terminal UI.

You enter Visual Mode with:

```
[0x00400000]> V
```

There are multiple sub-modes, which we'll cover in detail.

7.2 How to Enter Visual Mode

Start by opening a binary:

```
r2 ./a.out
[0x00000000]> aaa       # Analyze everything
[0x00000000]> s main    # Seek to main function
[0x00400510]> V         # Enter visual mode
```

7.3 Visual Mode Controls Overview

Key	Action
q	Quit current visual mode
V	Enter/exit main visual mode
p	Cycle between panels (disasm, hex, etc.)
P	Change panel layout (list, graph, etc.)
Tab	Change focus between panels
hjkl	Move cursor (vi-style)
g, G	Jump to top or bottom
A	Open assembler editor
:cmd	Run Radare2 command from visual
x	Toggle Xrefs
Space	Cycle through visual styles (ASM, bytes)
Enter	Follow jump/call address

7.4 Panel Views (p and P)

Press p to cycle panels:

1. Disassembly (default)
2. Hexdump
3. Stack
4. Registers
5. Graph
6. Functions list

Press P to change layout style:

- Flat view
- Split view
- Graph view (CFG)

7.5 Graph View (Control Flow Graph)

To enter:

```
[0x00400510]> VV
```

Or:

- Press P until you see the graph.
- Use arrow keys or hjkl to navigate basic blocks.
- Press Enter to follow a jump or call.

Example:

- jmp 0x00400530 | Press Enter to go to that basic block.
- u | Go back to previous location.

7.6 Assembler in Visual Mode

Press A to enter the Assembly editor:

- Type a new instruction (e.g., mov eax, 0)
- Press Enter to assemble
- Press q to quit editor

Only works if opened with -w flag (write mode):

```
r2 -w ./a.out
```

7.7 Visual Function Navigation

- afl (outside visual): list all functions
- In visual mode:
  - F: Show function list (use j/k to select)
  - Press Enter to jump to selected function

7.8 Follow Function Calls / Jumps

While in disassembly:

- Highlight a call or jmp instruction
- Press Enter to follow it
- Press u to go back

7.9 Contextual Info Windows

Use Tab to switch between:

- Hexdump view
- Stack view
- Registers
- Function calls
- Comments/Xrefs

7.10 Common Visual Mode Tasks

7.10.1 1. Patch Instruction

```
r2 -w ./a.out
aaa
s main
V
Navigate to instruction
Press A
Type nop
Press Enter
Press q to exit editor
```

7.10.2 2. Follow a Jump in CFG

```
aaa
s main
VV                    # Graph mode
Navigate to jmp or call
Press Enter
```

7.10.3 3. Rename a Function or Flag

- Press :afn newname or :f myflag @ current_address

7.11 Useful Visual Mode Commands (: Prefix)

You can enter any command using : inside visual mode:

```
:pdf
:s sym.main
:afn decrypt_func
:wx 90
```

7.12 Summary: Visual Mode Tips

Action	Keys / Commands
Enter visual mode	V
Enter graph mode	VV
Exit visual mode	q
Switch panels	p / Tab
Follow a jump/call	Enter
Go back	u
Open assembler	A
Run command	:cmd
View function list	F

7.13 Final Example Workflow

```
r2 -w ./a.out
[0x00000000]> aaa
[0x00000000]> s main
[0x00400510]> V
(p to switch panels, A to edit code, VV for graph, etc.)
```

7.14 Would you like a visual cheat sheet or an interactive training binary to practice visual mode features?

8 Searching in radare2

Here's a detailed guide to Searching in Radare2, covering how to find strings, instructions, byte patterns, functions, references, and more | with examples and explanations for each use case.

8.1 Why Search in Radare2?

Searching lets you:

- Find hardcoded strings in binaries.
- Locate specific byte sequences or opcodes.
- Search for function calls, syscalls, or magic values.
- Identify references to memory locations or symbols.

8.2 Searching for Strings

iz { List Strings in Binary

```
[0x00000000]> izz
```

- Extracts and lists all printable strings. - Shows offset, size, and encoding.
Example:

```
[Strings]
0x00400600  13  str.Hello_world  ascii   Hello world
```

iz <pattern> { Filter Strings by Pattern

```
[0x00000000]> iz~flag
```

Finds strings that contain "flag".

8.3 Searching in Memory

Radare2 can search for:

Search Type	Command
String	/
Hex bytes	/x
Assembly opcode	/c
Wide string (UTF-16)	/w
Flags/symbols	/f

Search for a String: /

```
[0x00000000]> / flag
```

- Finds the ASCII string flag in memory. - Use /! to invert the match.
Search for Hex Bytes: /x

```
[0x00000000]> /x 48 89 e5
```

- Finds the hex pattern for mov rbp, rsp.
Or shorthand:

```
[0x00000000]> /x 4889e5
```

You can also search for wildcards:

```
[0x00000000]> /x 48 ?? e5
```

?? = wildcard byte.
Search for Assembly Instruction: /c

```
[0x00000000]> /c mov eax, 0
```

- Finds occurrences of this instruction (assembles it and searches for the opcode).
Search for Wide Strings: /w

```
[0x00000000]> /w password
```

- Useful for Windows binaries (UTF-16).
Search for Function/Flag Names: /f

```
[0x00000000]> /f main
```

8.4 Navigating Search Results

After a search:

- n → Next result
- N → Previous result
- s <addr> → Seek to specific result address

Example:

```
[0x00000000]> / flag
hits: 1
0x00400600 hit0_0 flag
[0x00000000]> s 0x00400600
```

8.5 Search All Memory Ranges: e search.in=io.maps

```
[0x00000000]> e search.in=io.maps
```

Tells Radare2 to search in all mapped memory.

8.6 List All Previous Searches: sl

```
[0x00000000]> sl
```

- View all search hits stored in flags. - Example output:

```
0x00400600 hit0_0
0x00400710 hit1_0
```

8.7 Binary Grepping with Operator

You can apply to almost any command output.
Examples:

```
[0x00000000]> izz~flag         # Search strings
[0x00000000]> afl~main         # Search function names
[0x00000000]> /x 90~:          # Filter hex search results
```

8.8 Combine Search with Analysis

Find where a string is used:

```
[0x00000000]> izz              # Find the string (e.g., "Welcome")
[0x00000000]> axt @ 0x00400600 # Find Xrefs to the string
```

Or for function references:

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

8.9 Search Example: Patch strcmp("password")

1. List strings:

   ```
   izz~password
   ```

2. Seek to string address (e.g., 0x00400650):

   ```
   s 0x00400650
   ```

3. Find references:

   ```
   axt @ 0x00400650
   ```

4. Patch logic at that location.

8.10 Scripted Search (Loop)

Search multiple values:

```
[0x00000000]> for i in `seq 0 255`; do /x $(printf "%02x" $i); done
```

8.11 Summary of Key Search Commands

Command	Description
`izz`	`List all strings`
`/ pattern`	`Search ASCII string`
`/x hex`	`Search hex bytes`
`/c instr`	`Search for assembled opcode`
`/f name`	`Search for flag/function by name`
`/w str`	`Search for wide strings (UTF-16)`
`n / N`	`Next / previous result`
`sl`	`List saved search hits (flags)`
`axt @ addr`	`Find all Xrefs to an address`

8.12 Would you like a search cheatsheet PDF, or an example binary for practice with challenges (like "Find the key string" or "Patch a function")?

9 Debugging in radare2

Here's a complete guide to Debugging in Radare2, packed with detailed explanations and examples. This includes how to start a debugging session, set breakpoints, inspect registers, step through code, analyze memory, and patch live processes.

9.1 What Is Radare2's Debugging Mode?

Radare2 has a powerful built-in debugger that works with:

- Native ELF binaries (Linux)
- PE binaries (Windows via WSL or Wine)
- Raw binaries (if you specify architecture manually)
- It supports tracing, breakpoints, memory inspection, register control, and more.

9.2 Starting a Debugging Session

Start with Debug Mode Enabled

```
r2 -d ./a.out
```

This loads the binary under the debugger. The process does not run automatically. You'll see something like:

```
Process with PID 12345 started...
pid = 12345 tid = 12345
```

Launch and Analyze

```
[0x7ffff7dd0000]> aaa      # Analyze binary
[0x7ffff7dd0000]> s main   # Seek to main (if symbol exists)
```

9.3 Breakpoints

Set a Breakpoint

```
[0x00400510]> db 0x00400510        # Set breakpoint at address
```

Or:

```
[0x00400510]> db sym.main          # Breakpoint at symbol
```

List Breakpoints

```
[0x00400510]> db
```

Delete Breakpoint

```
[0x00400510]> db- 0x00400510
```

Clear All

```
[0x00400510]> db-*
```

9.4 Running & Controlling Execution

Command	Description
dc	Continue execution
ds	Step into instruction
dso	Step over instruction
dsc	Step until call
dret	Step until return
drr	Print registers
dr <reg>	Show a register
dr eax=0	Set register value

Example: Step-by-Step Debugging

```
r2 -d ./a.out
[0x7ffff7dd0000]> aaa
[0x7ffff7dd0000]> db sym.main
[0x7ffff7dd0000]> dc              # Run until breakpoint
[0x00400510]> pdf                 # Disassemble current function
[0x00400510]> ds                  # Step into
[0x00400511]> dso                 # Step over
[0x00400512]> dr rax              # View rax
```

9.5 4. Inspecting Memory and Stack

Read 16 bytes of memory:

```
[0x00400510]> px 16 @ rsp
```

View stack:

```
[0x00400510]> pxw @ rsp
```

View memory mappings (like /proc/PID/maps):

```
[0x00400510]> dm
```

9.6 Modifying Memory or Registers

Write a single instruction

```
[0x00400510]> wa nop
```

Modify a register

```
[0x00400510]> dr eax=0x1337
```

Write hex bytes to memory

```
[0x00400510]> wx 909090
```

9.7 Watching Variables and Addresses

Add a watchpoint:

```
[0x00400510]> dwp 0x601050 4 r  # Read watch on 4 bytes
```

- You can specify read (r), write (w), or execute (x) access.

9.8 Using Visual Debugger (TUI Mode)

```
r2 -d ./a.out
[0x7ffff7dd0000]> aaa
[0x7ffff7dd0000]> V
```

- Press p to switch panels. - Press c to continue. - Press s to step. - Use Tab to switch focus (e.g., disasm → registers → stack). - Press q to quit visual mode.

9.9 Trace Execution Automatically

Trace All Instructions:

```
[0x00400510]> dtr*      # Trace everything
```

Trace Until Function Return:

```
[0x00400510]> dtr@
```

Use dtt to dump the trace log.

9.10 Debugging a Forking Program

Radare2 defaults to tracing only the parent. To handle child processes:

```
e dbg.follow.child = true
```

9.11 Debugging a Remote Process

Start the remote target (e.g., gdbserver):

```
gdbserver :1234 ./vuln
```

Connect with Radare2:

```
r2 -d gdb://127.0.0.1:1234
```

Then you can use all the same d* commands.

9.12 Summary of Debugging Commands

Command	Description
`-d`	Start in debug mode
`db`	Set breakpoint
`dc, ds`	Continue / Step
`dso, dsc`	Step over / Step until call
`dr`	Show/set registers
`px, pxw`	View memory, stack
`dm`	Memory map
`wa`	Write assembly
`wx`	Write hex bytes
`dtr*`	Trace execution
`dwp`	Watchpoint (memory access tracking)
`V`	Enter visual (TUI) debugger

9.13 Would you like a real vulnerable binary for practice, or a step-by-step challenge ("find and patch a password check")?

10 Memory and Sections in radare2

A comprehensive guide to Memory and Sections in Radare2, covering how to inspect, analyze, and interact with memory regions, sections, and segments in binaries | with detailed commands and examples.

10.1 What Are Sections and Segments?

In binary files (like ELF or PE), memory is organized into:

- Sections: .text, .data, .bss, .rodata, etc. (Used during linking)
- Segments: Memory areas loaded by the OS (LOAD segments)
- Pages/Maps: Actual memory mappings used at runtime

Radare2 allows analysis at all three levels.

10.2 View All Sections: iS

```
[0x00000000]> iS
```

Outputs a table of sections:

```
[Sections]
idx=00 vaddr=0x00400000 paddr=0x00000000 sz=0x001000 vsz=0x001000 perm=m-r-x name=.text
idx=01 vaddr=0x00401000 paddr=0x00001000 sz=0x000200 vsz=0x000200 perm=--r-- name=.rodata
```

Useful fields:

Field	Meaning
vaddr	Virtual address when loaded
paddr	Physical (file offset)
sz	Size in file
vsz	Size in memory
perm	Permissions (rwx)
name	Section name

10.3 View All Segments: iP

```
[0x00000000]> iP
```

Lists program headers (segments) like LOAD, DYNAMIC, INTERP, etc.
Example:

```
[Segments]
00 0x0 LOAD 0x00400000 sz=0x6000 vsz=0x6000 perm=rx
01 0x1 LOAD 0x00600000 sz=0x2000 vsz=0x3000 perm=rw
```

These are what the OS uses when loading the binary.

10.4 View Runtime Memory Maps: `dm`

```
[0x00000000]> dm
```

This shows memory mappings at runtime. Useful in debug mode or forensics.
Example:

```
0x00400000 - 0x0040a000 m-r-x /bin/ls
0x00600000 - 0x00602000 m--rw /bin/ls
```

Filter by permission:

```
[0x00000000]> dm~rx
```

10.5 Explore Section Contents

Seek to a section:

```
[0x00000000]> s section..text
```

You can use tab completion after section.
Print contents as hex:

```
[0x00400000]> px 64       # Print 64 bytes
```

Or as disassembly:

```
[0x00400000]> pd 10
```

10.6 Analyze Specific Section

Disassemble only .text section:

```
[0x00000000]> s section..text
[0x00400000]> pd 50
```

Search for strings in .rodata:

```
[0x00000000]> s section..rodata
[0x00401000]> iz
```

10.7 Mark/Flag a Section or Memory Range

Flag a section start:

```
[0x00400000]> f mycode 0x100
```

Creates a flag mycode at 0x00400000 with length 0x100.

10.8 Identify Section for Any Address: `iSj`

Use this to find which section an address belongs to.

```
[0x00400123]> iSj~{0x00400123}
```

Or programmatically:

```
[0x00400123]> ?v $S~text
```

10.9 Modify Section Permissions (Debug Mode)

Radare2 allows patching memory if permissions allow. If not, change them:
Example:

```
[0x00400000]> dmmu 0x00400000 rwx
```

dmmu = set memory map permissions.
Only available in debug mode (r2 -d).

10.10 Patch Data in a Section

Overwrite .rodata string

- Seek to .rodata:

```
s section..rodata
```

- Find string:

```
px 128
```

- Overwrite:

```
wx 48656c6c6f205253     # "Hello RS" in hex
```

Or:

```
wa mov eax, 0x1337      # Assemble into memory
```

(Use -w mode)

10.11 Dump a Section

Dump .text to file:

```
[0x00000000]> s section..text
[0x00400000]> wt text.bin 0x1000
```

10.12 Summary of Section & Memory Commands

Command	Purpose
`iS`	Show all sections
`iP`	Show program headers (segments)
`dm`	Show runtime memory maps
`s section..x`	Seek to section
`px / pd`	Print hex / disassembly
`f`	Flag memory area
`dmmu`	Change memory permissions (debug only)
`wx / wa`	Write hex or assembly
`wt file size`	Write memory to file

10.13 Would you like a live memory visualization (CFG + sections) or an example binary to explore section data?

11 Strings and Imports/Exports in radare2

Here's a comprehensive guide to working with Strings, Imports, and Exports in Radare2, with detailed explanations and practical examples to help you reverse-engineer binaries effectively.

11.1 PART 1: STRINGS

11.1.1 Listing All Strings

```
[0x00000000]> iz
```

Shows all detected strings in the binary.
Output example:

```
[Strings]
0x00400600  13  str.Hello_world  ascii   Hello world
0x00400610  10  str.enter_pass   ascii   Enter password:
```

Columns:

- offset: memory address of the string
- len: length in bytes
- type: ASCII, UTF-16, etc.
- string: actual contents

11.1.2 Analyze & Index Strings

To ensure all strings are found:

```
[0x00000000]> aaa          # Analyze everything
[0x00000000]> izz          # Find + list all strings
```

This is more comprehensive than iz.

11.1.3 Filter Strings

```
[0x00000000]> iz~flag
```

Shows only strings containing the word "flag".

11.1.4 Seek to a String

```
[0x00000000]> s 0x00400600
[0x00400600]> ps
```

ps → print string at current location.

11.1.5 Find References to Strings

```
[0x00000000]> axt @ 0x00400600
```

Finds Xrefs to the string. Useful for locating code that uses it.

11.1.6 Rename a String (Optional)

```
f str.password = 0x00400600
```

You can also reflag strings manually for clarity.

11.2 PART 2: IMPORTS

Imports = functions or symbols the binary needs from external libraries (like libc).

11.2.1 List Imports

```
[0x00000000]> ii
```

Output example:

```
[Imports]
ordinal=001 plt=0x00400420 bind=GLOBAL type=FUNC name=printf
ordinal=002 plt=0x00400430 bind=GLOBAL type=FUNC name=scanf
```

Columns:

- plt: address of function stub
- name: function name
- type: usually FUNC
- bind: linkage type

11.2.2 Filter Imports

```
[0x00000000]> ii~puts
```

Finds any imported symbol named "puts".

11.2.3 Jump to Import Function Stub (PLT)

```
[0x00000000]> s sym.imp.printf
[0x00400420]> pd 5
```

Disassemble the PLT stub.

11.2.4 Find Calls to Imports

```
[0x00000000]> axt @ sym.imp.printf
```

Find all places where printf is called.

11.3 PART 3: EXPORTS

Exports = symbols or functions provided by the binary (used by others, e.g., in a shared library).

11.3.1 List Exports

```
[0x00000000]> iE
```

Output example:

```
[Exports]
vaddr=0x00400500 paddr=0x00000500 ord=0 type=FUNC name=exported_func
```

- vaddr: virtual address of function
- name: exported symbol name

11.3.2 Seek to Exported Function

```
[0x00000000]> s sym.exported_func
[0x00400500]> pdf
```

Disassemble the full function.

11.3.3 Filter Exported Names

```
[0x00000000]> iE~init
```

Find all exports containing "init".

11.4 BONUS: Identify Unknown Imports Automatically

If symbols are stripped, Radare2 can try to resolve symbols heuristically:

```
[0x00000000]> aaaa
```

This includes:

- Symbolic renaming
- Import detection
- Cross-references

11.5 Summary Cheat Sheet

Command	Description
izz	Index and list all strings
iz keyword	Filter strings
ps	Print string at current location
axt @ addr	Find Xrefs to string or function
ii	List imports
ii name	Filter imports by name
iE	List exports
s sym.xxx	Seek to import/export by name
pdf	Disassemble function

11.6 Would you like an exercise binary with hidden strings and stripped imports for a hands-on challenge?

12 Scripting and Automation in Radare2

Here's a detailed guide to scripting and automation in Radare2, covering r2pipe, scripting languages, macros, and .rc files | with practical examples and pro tips to automate reverse engineering tasks.

12.1 Why Script in Radare2?

Scripting allows you to:

- Automate analysis (loop over functions, extract data, patch)
- Generate reports or summaries
- Run batch reverse engineering on many binaries
- Integrate with other tools via Python, C, Go, Node.js

12.2 Types of Radare2 Scripting

Method	Description
.rc files	Run command sequences automatically
r2pipe	Use Radare2 from Python, Go, C, JS, etc.
!pipe	Inline scripts inside Radare2
Macros	Radare2 command-line aliases using $ syntax
Loops	Use !for, !while, etc. inside r2 shell

12.3 .rc Files (Startup Scripts)

You can write a sequence of Radare2 commands in a file and run them with:

```
r2 -i script.rc binary
```

Example: analyze.rc

```
aaa
s sym.main
pdf
iz~password
```

Run with:

```
r2 -i analyze.rc ./binary
```

12.4 Python Scripting with r2pipe

Radare2 has bindings for Python via r2pipe.

12.4.1 Install r2pipe

```
pip install r2pipe
```

12.4.2 Example: List All Functions

```python
import r2pipe

r2 = r2pipe.open('./a.out')
r2.cmd('aaa')  # Analyze all

funcs = r2.cmdj('aflj')  # JSON output of function list
for f in funcs:
    print(f"Function: {f['name']} @ 0x{f['offset']:x}")
```

12.4.3 Find Strings with "flag"

```python
strings = r2.cmdj('izj')
for s in strings:
    if 'flag' in s['string']:
        print(f"Found flag string at {hex(s['vaddr'])}: {s['string']}")
```

12.5 Command-Line Macros with $

Define reusable command blocks:

12.6 Loops and Shell Commands

12.6.1 For Loop Over Functions

This is using shell scripting + r2 -qc for one-liners.

12.7 Batch Automation with -qc

Run Radare2 commands non-interactively:

```
r2 -qc "aaa; afl; iz~flag" ./binary
```

This can be used in your scripts or CI pipelines.

12.8 Other Languages with r2pipe

You can use:

- Node.js: npm install r2pipe
- Go: go get github.com/radareorg/r2pipe/go
- Rust/C: via subprocess or lib bindings
- Bash: simple piping with r2 -qc

12.9 Patch via Python

Example: patch a function call:

```
r2.cmd('s sym.main')
r2.cmd('wa nop')  # Write NOP at current location
```

Or hex patch:

```
r2.cmd('wx 90 90 90 90')  # Overwrite with NOPs
```

12.10 Script to Dump All Strings Used in Functions

```
import r2pipe
r2 = r2pipe.open("a.out")
r2.cmd("aaa")

funcs = r2.cmdj("aflj")
for f in funcs:
    r2.cmd(f"s {f['offset']}")
    xrefs = r2.cmdj("axtj")
    if not xrefs:
        continue
    for x in xrefs:
        if x['type'] == 'DATA':
            s = r2.cmd(f"ps @ {x['to']}")
            if s.strip():
                print(f"{f['name']} uses string: {s.strip()}")
```

12.11 Save Your Own Commands

Save commands for reuse:

```
echo 's sym.main\npdf\niz~flag' > myscript.rc
```

Then:

```
r2 -i myscript.rc ./binary
```

Or use it interactively:

12.12 Summary of Key Scripting Tools

Tool	Use Case
.rc files	Sequence of commands
r2 -i file	Run commands from file
r2pipe	Use Python/Go/JS with Radare2
r2 -qc	Run quick commands and exit
$macro	Define reusable command blocks
!for/!while	Shell loops inside Radare2

12.13 Would you like a template Python script for automating analysis of multiple binaries or generating HTML reports?