Disassembling with radare2

Tomáš Antecký (tomas@antecky.cz)
antecky.cz/r2

radare2

- An open source reverse engineering framework
  - http://rada.re
  - https://github.com/radare/radare2
- Started at 2006 (today over 15 000+ commits, just for **`radare2`** repository)
- Used for static/dynamic analysis, binary patching, forensic analysis,...
- Set of libraries/binaries primarily written in C
- Runs on: Linux, *BSD, Windows, OSX, iOS, Android....
- Supports many:
  - Architectures: x86, mips, arm, sparc, powerpc, avr,...
  - Binary formats: ELF, mach0, PE, DEX, ART, Wasm, Swf, COFF,...
- Can handle tampered binaries
- Mainly used through CLI, but there are graphical frontends
- Scriptable (bindings to Python, Ruby, JavaScript, Perl, Java, C#,...)

Installation

- **radare2** packages provided by distributions are obsolete
- Recommend way of installation is by using Git:

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

System-wide installation (requires root)

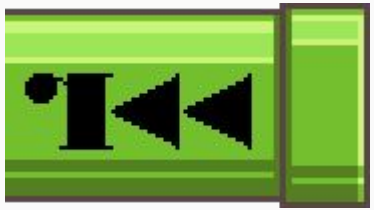
```
$ sys/install.sh
```

User based installation (into $HOME)

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

- Language bindings (**r2pipe**) are installed separately:

```
$ pip install r2pipe
$ npm install r2pipe
$ gem install r2pipe
```



Major binaries in radare2 suite

- **rabin2**
  - Binary identification
- **rasm2**
  - Inline assembler/disassembler
- **radiff2**
  - Binary diffing
- **r2pm**
  - Package/plugin manager
- **r2**
  - The “main” binary
  - Console interface
  - Many shell like features (file/command redirection, history, shortcuts, command substitution, tab completion...)
- and many more...

```
$ rabin2 -I /bin/ls
arch     x86
binsz    124726
bintype  elf
...

$ rasm2 -a arm -b 64 'movk x0, 0x1337'
e06682f2
$ rasm2 -a arm -b 64 -d e06682f2
movk x0, 0x1337

$ radiff2 genuine cracked
0x000081e0 85c00f94c0 => 9090909090 0x000081e0
0x0007c805 85c00f84c0 => 9090909090 0x0007c805
```

```
$ r2 /bin/ls # open binary (read-only)
$ r2 -w /bin/ls # enable writing
$ r2 -d /bin/ls # run with a debugger
$ r2 -n /bin/ls # open as a flat file
$ r2 - # open without a file
```

antecky.cz/r2

Demo

- Quick demonstration of **`radare2`** capabilities
- Static and dynamic analysis
- A simple crackme/CTF challenge
- Goal is to obtain a password/flag stored inside a binary
- Source code at antecky.cz/r2 (spoiler alert)
- Build with help of **`radare2`** (see **`prepare.py`**)
- Each step is in this presentation as well
- So no worries, if don't catch anything

Demo

- A binary called **`runme`**
- ELF64 for Linux, statically linked
- Requires password

```
~/LinuxDays2017 $ ./runme
password: NoIdea
Wrong!
```

- Let's try **`objdump`** and **`gdb`** first

```
~/LinuxDays2017 $ objdump -d ./runme
objdump: ./runme: File format not recognized
~/LinuxDays2017 $ gdb -q -ex run runme
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
(gdb) quit
```

```
~/LinuxDays2017 $ rabin2 -I ./runme
Warning: Cannot initialize section headers
Warning: Cannot initialize strings table
Warning: Cannot initialize dynamic strings
arch     x86
binsz    1021
bintype  elf
bits     64
canary   false
class    ELF64
crypto   false
endian   little
havecode true
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      false
relocs   true
rpath    NONE
static   true
stripped false
subsys   linux
va       true
```

Demo

- Next run **`$ r2 runme`**
- **`?`** is the most important command
- It works with subcommands as well (e.g. **`i?`**)
- **`i`** shows the same info as **`rabin2`**
- **`j`** suffix shows JSON for many commands
- It can be prettified by adding **`~{}`** (see **`?@?`** for more)
- A binary entry points are displayed by **`ie`**
- **`S=`** shows program's segments in a fancy way

```
[0x00600120]> ij~{}
{
  "core": {
    "type": "EXEC (Executable file)",
    "file": "runme",
    "fd": 3,
    "size": 1304,
    "humansz": "1.3K",
    "iorw": false,
    "mode": "-r-x",
    "obsz": 0,
    "block": 256,
    "format": "elf64"
```

```
[0x00600120]> S=
00  0x00400000 |#####-----------| 0x00400116   278 mr--  LOAD0
01* 0x00600120 |----############| 0x006003fd   733 mrwx  LOAD1
=>  0x00600120 |----------------| 0x0060011f
```

- The text/code section is writable
- Self modifying code?

antecky.cz/r2

Demo

- It’s always a good idea to search for interesting strings
- **`izz`** searches for string in the whole binary
- ASCII and Unicode strings are found at once
- We can combine the command with an internal **`less`** as well (i.e. **`izz~..`**)
- Only strings inside data segment (**`LOAD0`**) seems to be interesting
- The results can be filtered by using an internal **`grep`** (**`izz~LOAD0`**)

```
[0x00600120]> izz~LOAD0
vaddr=0x004000e8 paddr=0x000000e8 ordinal=000 sz=11 len=10 section=LOAD0 type=ascii string=password:
vaddr=0x004000f3 paddr=0x000000f3 ordinal=001 sz=8 len=7 section=LOAD0 type=ascii string=Wrong!\n
vaddr=0x004000fb paddr=0x000000fb ordinal=002 sz=11 len=10 section=LOAD0 type=ascii string=Good job!\n
```

- In this case there is nothing useful

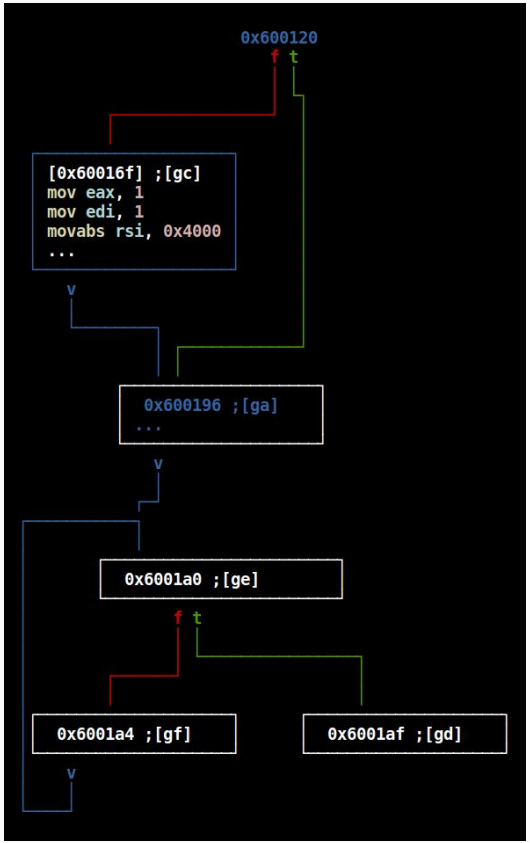
Demo

- **`Visual mode`** can be entered by running **`V`** command
- **`p/P`** rotates between views
- The second view/panel is the **`Disassembly view`**
- Once again **`?`** displays help
- **`hjkl`** keys are used for move around
- **`q`** is used to go back to the command line
- In order to run a command inside **`Visual mode`** press **`:`**
- **`c`** activates cursor for easier movement
- Command **`s entry0`** seeks back to the entry point

antecky.cz/r2

Demo

- Next pressing **V** brings **Function graph**, however a function has to be analyzed first
- Analysis can be done by running **af** command
  - **aa** can analyze the whole file (not recommend for large binaries)
- **p/P** rotates between views
- **hjkl** keys are used for move around
- **+/-** changes zoom level
- **tab/TAB** cycles between nodes
- **y/Y** folds current node
- **t/f** follows conditional jump
- **g?** jumps to particular node (e.g. **gc**)
- **.** centers current node



antecky.cz/r2

Demo

- Let's focus on the three syscalls at the beginning
- Linux x86-64 kernel syscall calling convention:
  - syscall number and return value is inside **`rax`**
  - **`rdi/rsi/rdx/r10/r8/r9`** for syscall arguments
- **`asl`** command can be used to translate a syscall number to its name
- The first syscall writes "**`password`**:" string into **`stdout`** (**`mov edi, 1`**)
  - Use **`x @ 0x4000e8`** to examine memory at given address

```
[0x00600120]> x @ 0x4000e8
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004000e8  7061 7373 776f 7264 3a20 0057 726f 6e67  password: .Wrong
0x004000f8  210a 0047 6f6f 6420 6a6f 6221 0a00 0300  !..Good job!....
0x00400108  0000 0000 0000 0000 0000 0000 0000 ffff  ................
```

  - Zero terminated strings can be printed by running **`psz`**

```
[0x00600120]> psz @ 0x4000e8
password:
```

```
0x00600120      mov eax, 1
0x00600125      mov edi, 1
0x0060012a      movabs rsi, 0x4000e8
0x00600134      mov edx, 0xb
0x00600139      syscall
0x0060013b      sub rsp, 0x10
0x0060013f      mov eax, 0
0x00600144      mov edi, 0
0x00600149      mov rsi, rsp
0x0060014c      mov edx, 0x10
0x00600151      syscall
0x00600153      mov eax, 0x23
0x00600158      mov edi, 0x400106
0x0060015d      xor esi, esi
0x0060015f      syscall
```

```
[0x00600120]> asl 1
write
[0x00600120]> asl 0
read
[0x00600120]> asl 0x23
nanosleep
```

antecky.cz/r2

Demo

- The second syscall reads from **`stdin`** (**`mov edi, 0`**) to stack (**`mov rsi, rsp`**)
- The third syscall is **`nanosleep`** and due to it the binary sleeps for given amount of time
- Length of sleep is specified by a struct at **`0x400106`**
  - In this case it is hardcoded to 3 seconds

```
0x00600120      mov eax, 1
0x00600125      mov edi, 1
0x0060012a      movabs rsi, 0x4000e8
0x00600134      mov edx, 0xb
0x00600139      syscall
0x0060013b      sub rsp, 0x10
0x0060013f      mov eax, 0
0x00600144      mov edi, 0
0x00600149      mov rsi, rsp
0x0060014c      mov edx, 0x10
0x00600151      syscall
0x00600153      mov eax, 0x23
0x00600158      mov edi, 0x400106
0x0060015d      xor esi, esi
0x0060015f      syscall
```

```
[0x00600120]> x 16 @ 0x400106
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00400106  0300 0000 0000 0000 0000 0000 0000 0000  ................
```

```
[0x00600120]> asl 1
write
[0x00600120]> asl 0
read
[0x00600120]> asl 0x23
nanosleep
```

- We can insert comments by pressing `;`

```
Enter a comment: ('-' to remove, '!' to use $EDITOR)
comment:
            0x00600158    *  mov edi, 0x400106      ; length of sleep
            0x0060015d       xor esi, esi
            0x0060015f       syscall
```

Demo

- This brute force “protection” can be disabled by patching the binary:
  - Creating a backup of the binary (**`!cp runme runme.bak`**)
  - Enabling writing (**`oo+`**)
  - Display bytes/opcodes for each instruction (**`e asm.bytes=1`**)
  - Placing cursor at **`0x0060015f`** where is the instruction **`syscall`** (2 bytes long)

```
[0x00600153 25% 230 (0xc:-1=1)]> pd $r
            0x00600153      b823000000      mov eax, 0x23 ; '#' ; 35
            0x00600158      bf06014000      mov edi, 0x400106 ; length of sleep
            0x0060015d      31f6            xor esi, esi
            0x0060015f    * 0f05            syscall
            0x00600161      4889e1          mov rcx, rsp
```

  - Pressing **`A`** for interactive assembler and writing 2 **`nop`** instruction
  - Confirming changes by running **`radiff2`**

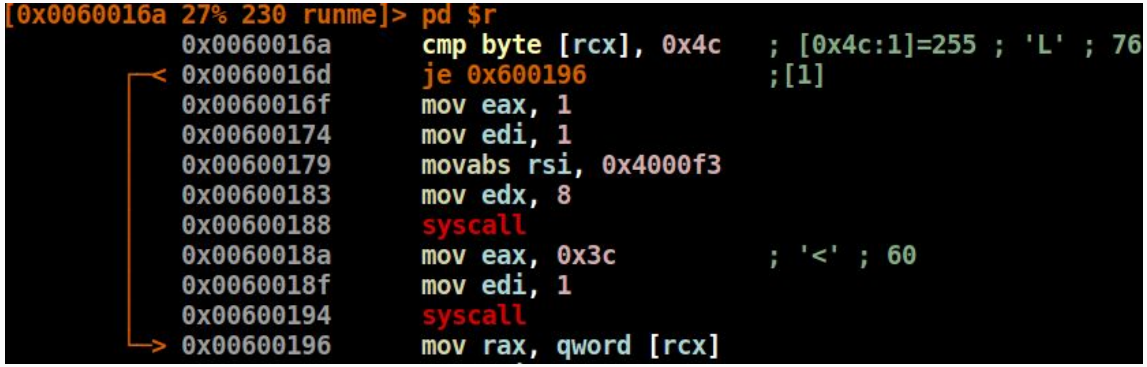
```
[0x00600120]> !radiff2 runme.bak runme
0x0000015f 0f05 => 9090 0x0000015f
```

```
Write some x86-64 assembly...

2> nop;nop
* 9090

            0x0060015f      90              nop
            0x00600160      90              nop
            0x00600161      4889e1          mov rcx, rsp
```

antecky.cz/r2

Demo

- Next the first character on the stack is compared with character “**L**”



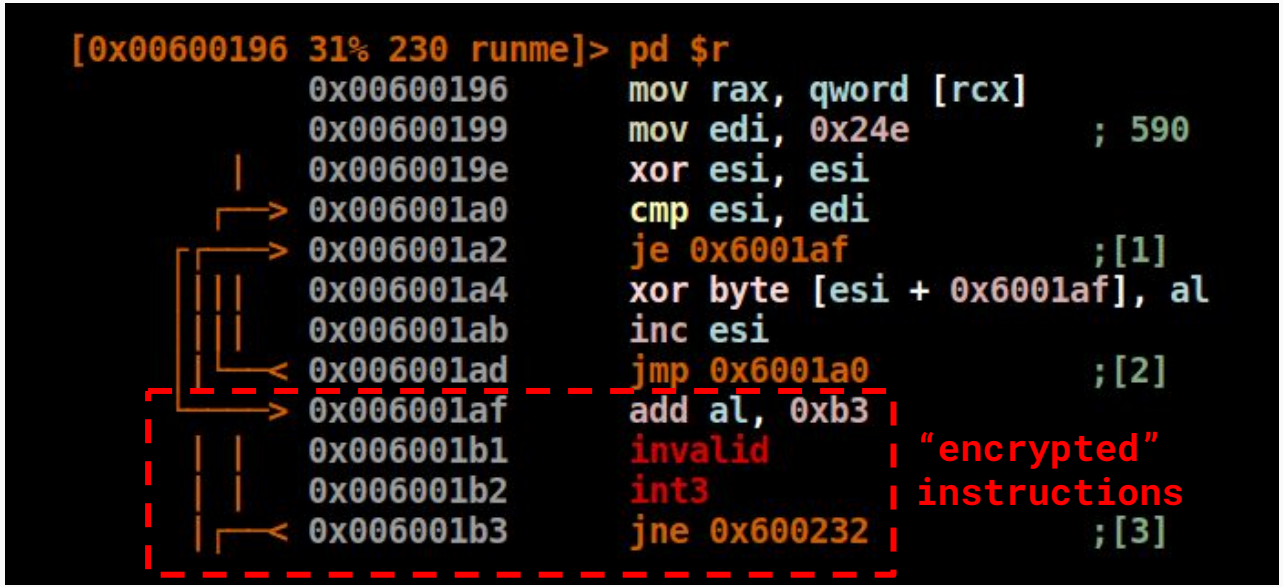
- If it is not equal the provided character a string “**Wrong**!” is printed (**`psz @ 0x4000f3`**)
- Finally the program exits (**`asl 0x3c`**) with a status code **1** (**`mov edi, 1`**)

```
~/LinuxDays2017 $ ./runme
password: NoIdea
Wrong!
~/LinuxDays2017 $ echo $?
1
```

antecky.cz/r2

Demo

- In the second branch the character (“**L**”) is used in a loop to decrypt/xor an instruction starting at **0x006001af** and further down



Demo

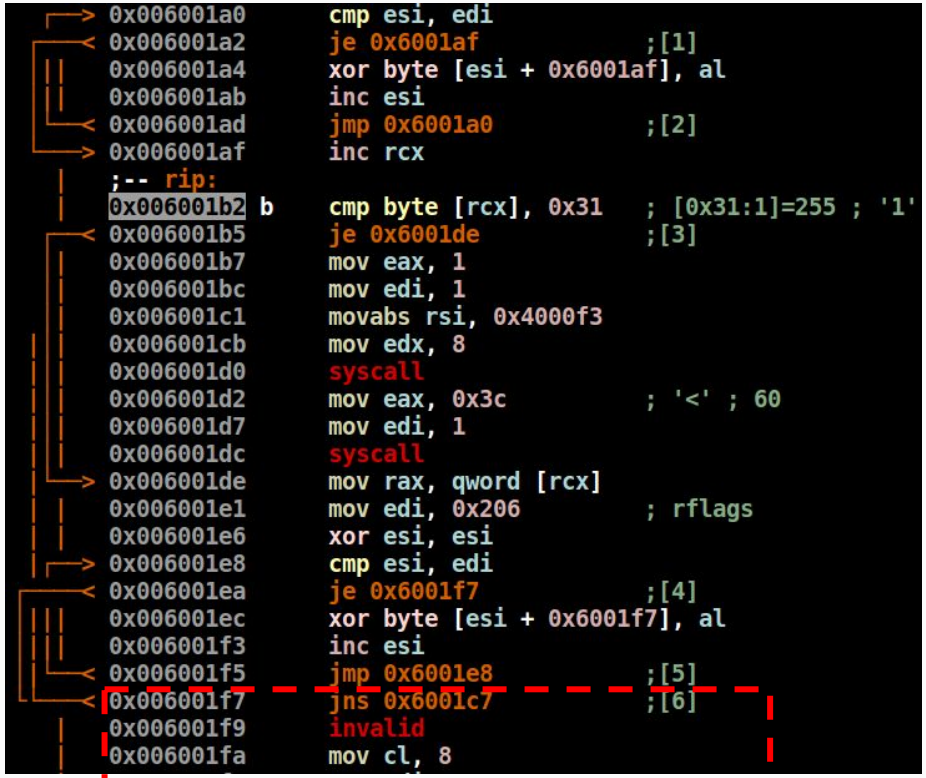
- Now it is time to switch to dynamic analysis:
  - Start the binary with an attached debugger (**ood**)
  - Place a breakpoint at **0x0060016a** (**db 0x0060016a**)
  - Continue (**dc/F9**) until the breakpoint is hit
  - Provide some garbage input as a password
  - After that the breakpoint is hit
  - Write "**L**" character (**0x4c** byte) to the stack (**wx 0x4c @ rcx**)
  - Confirm it by running **px 1 @ rcx**
  - Perform several single step (**ds/F7**) to see valid instructions emerging
  - A command can be repeated several times by providing number prefix (e.g. **300ds**)

```
0x0060015f      nop
0x00600160      nop
0x00600161      mov rcx, rsp
0x00600164      dec rcx
0x00600167      inc rcx
0x0060016a b    cmp byte [rcx], 0x4c    ;'L'
0x0060016d      je 0x600196             ;[1]
0x0060016f      mov eax, 1
0x00600174      mov edi, 1
```

```
[0x0060016a]> px 1 @ rcx
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffd1e50fcf0  4c                                       L
```

Demo

- It can be seen that a new decrypted block is the same as the previous one
- Except a compared character is different (now it is “**1**”)
- Manual password extraction can be tedious
- There are several ways how to automate this process



“encrypted” instructions

antecky.cz/r2

Demo

- To automate the process of password extraction Python 3 was chosen
- Requires installed **`r2pipe`** package
  - **`$ pip3 install r2pipe`**
- See a script **`solve.py`**
- Works the same way as the manual method described earlier
- All used commands should be clear by now

```
~/LinuxDays2017 $ ./solve.py 2> /dev/null
L1nuxDayZ
```

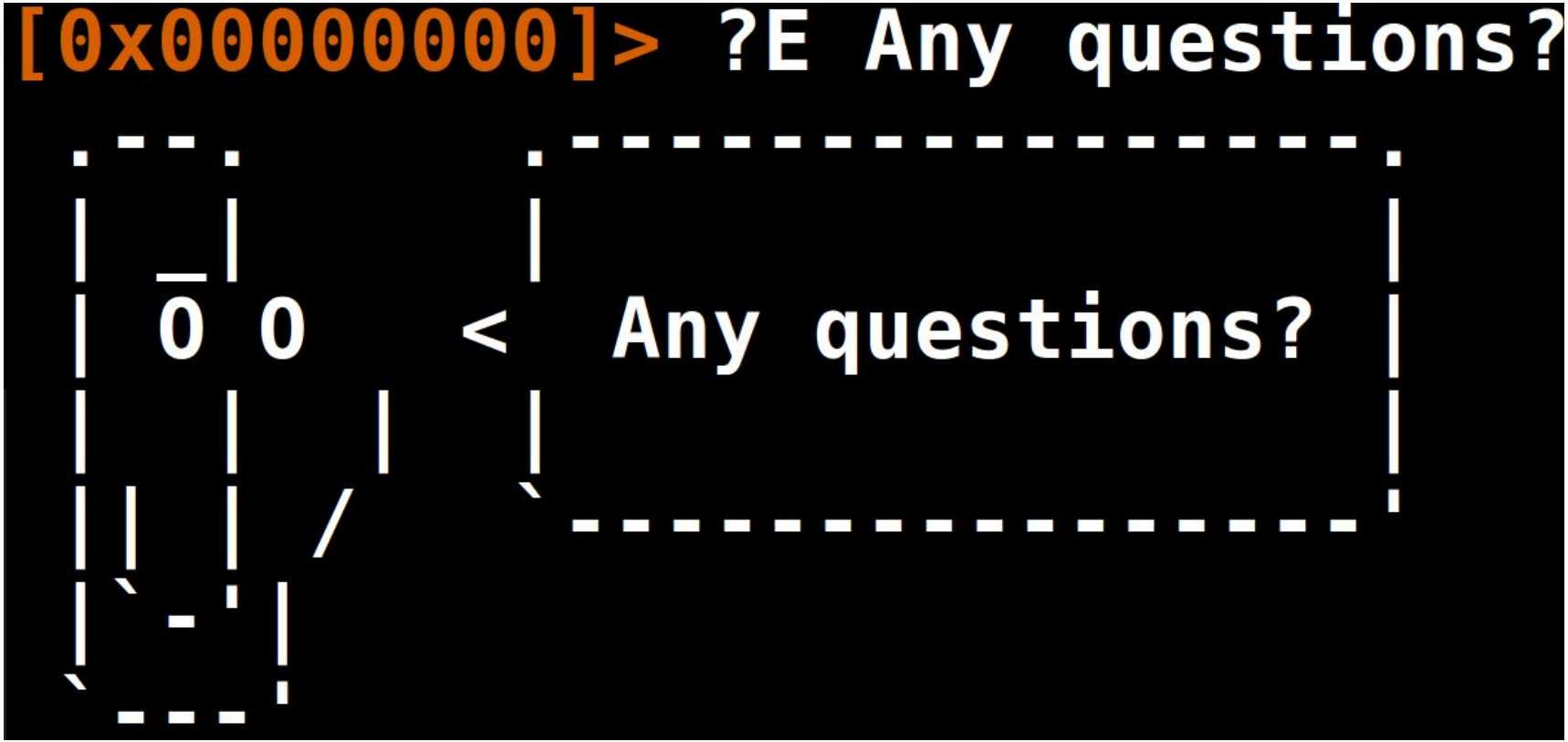
```python
#!/usr/bin/env python3
import r2pipe

#  open the binary with attached debugger
r2 = r2pipe.open('./runme', ['-d'])

while 'invalid' not in r2.cmd('s'):
    #  do one step + seek to rip register
    r2.cmd('ds;sr rip')

    #  dissamble one instruction
    json = r2.cmdj('pdj 1')

    if not json:
        continue

    json = json[0]
    opcode = json['opcode']


    #  identify an instruction with a password
    if 'cmp byte [rcx]' in opcode:
        #  extract next character of a password
        char = str(hex(json['ptr']))
        #  write the character to stack
        r2.cmd('wx {} @ rcx'.format(char))
        #  print the character
        print(chr(int(char, 16)), end='', flush=True)

print()
```

antecky.cz/r2

Useful links

- radare2 book
  - https://radare.gitbooks.io/radare2book/content
- radare2 exploration
  - https://monosource.gitbooks.io/radare2-explorations/content
- radare2 cheat sheet
  - https://github.com/radare/radare2/blob/master/doc/intro.md
- Reverse Engineering for Beginners
  - https://beginners.re
  - An open source book about reverse engineering (x86, ARM, MIPS)
- Compiler Explorer
  - https://godbolt.org
  - Shows an assembly output of compiled source code

Questions



antecky.cz/r2