Introduction to Binary Reversing
using Radare 2

Giovanni Lagorio

giovanni.lagorio@unige.it
https://zxgio.sarahah.com

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova
Italy

December 12, 2017
Outline

1. Introduction
2. Radare: The good, the bad and the ugly
3. ASLR
4. Suggested workflow
5. IOLI Crackmes
Tools of the trade (1/2)

- **Identification:**
  - `md5sum`, `sha*sum`, ..., `rahash2`, ...

- **Static Analysis**
  - `strings`, `rabin2`, `rafind2`
  - `readelf`, `rabin2`
  - `objdump`, `rasm2`, `radare2`
  - `radiff2`
  - **decompilers**
    - `RetDec`, `Snowman`, `HexRays ($$$)`, ...
Dynamic Analysis

- `ldd` / `lddtree`
  
  *Beware!* `ldd(1)`: ...`ldd` may attempt to obtain the dependency information by directly executing the program ...

- `/proc/pid/maps`; see `proc(5)`
- `strace`
- `ltrace`
- `gdb`
- `radare2`

**Spoiler alert...**

**Exploiting**

- `ragg2`
- `pwntools`
- `ropper`
- `libformatstr`
Outline

1. Introduction
2. Radare: The good, the bad and the ugly
3. ASLR
4. Suggested workflow
5. IOLI Crackmes
Radare (http://www.radare.org) is a portable open-source reversing framework

- that can...
  - disassemble (assemble for) many different architectures
  - debug/patch programs
  - run on Linux, *BSD, Windows, OSX, Android, iOS, ...
  - perform forensics on filesystems and data carving
  - be scripted in Python (2 at the moment), ...
  - use powerful analysis capabilities to speed up reversing ...

- Great and quite active community

  - https://twitter.com/radareorg
  - https://telegram.me/joinchat/ACR-FkEK2owJSzMUYjt_NQ
  - some great blog posts on how to perform many different tasks ...
The bad

- Usability issues
  - silent failures (no feedback or error messages in many cases)
  - poor command parsing
  - no GUI
- Scarce “official” documentation, especially about *internals*

So, extremely steep learning curve

https://it.pinterest.com/pin/461407924294118013/
The ugly

https://github.com/radare/radare2/issues on 29th October 2017
However... love is blind 😊

https://twitter.com/blackOwl/status/902525610173628416
Getting started

- e-books
  - R2 book
    https://radare.gitbooks.io/radare2book/content/
  - Radare2 Explorations
    https://monosource.gitbooks.io/radare2-explorations/content/

- blog posts
  - Defeating IOLI with radare2 in 2017
  - A journey into Radare 2
    - https://www.megabeets.net/a-journey-into-radare-2-part-1/
    - https://www.megabeets.net/a-journey-into-radare-2-part-2/
  - Radare 2 in 0x1E minutes
    blog.techorganic.com/2016/03/08/radare-2-in-0x1e-minutes/

- cheatsheet
  - strongly biased: a useful (the best, actually 😊) cheatsheet:
    https://github.com/zxgio/r2-cheatsheet
  - direct download:
    github.com/zxgio/r2-cheatsheet/raw/master/r2-cheatsheet.pdf
Installation on Ubuntu-derived distro

1. remove (outdated distro) versions, if any:
   sudo apt purge radare2 radare2-plugins libradare2-dev

2. install some tools:
   sudo apt install build-essential git python python-dev python-virtualenv xdot gcc-multilib

3. clone the official repository:
   git clone https://github.com/radare/radare2.git

4. build and install
   - cd radare2
   - git checkout tags/2.0.1 # checkout a stable release
   - sys/user.sh --without-pull
   - add `export PATH=$PATH:~/bin` to your .bashrc

5. install r2pipe (and other tools we’ll use later) inside some virtualenv, e.g. r2py:
   - cd && python -mvirtualenv r2py && . r2py/bin/activate
   - pip install --upgrade pip
   - pip install r2pipe pwntools ropper libformatstr
An example of ~/.radarerc

```
e asm.bytes=0
e asm.cmtright=true
e cmd.stack=px 32
e dbg.slow=true
e scr.wheel=false
e scr.utf8=true
e scr.utf8.curvy=true
eco solarized
```
Sanity check

To try it out:

. ~/r2py/bin/activate # activate Python virtual-env
python # start Python interpreter
import r2pipe
r2 = r2pipe.open("/bin/ls") # open /bin/ls through r2
print(r2.cmd('pd 10 @ main')) # print the first 10
    # instructions of main
In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

https://en.wikipedia.org/wiki/Address_space_layout_randomization
Checking ASLR

from proc(5); /proc/sys/kernel/randomize_va_space contains:

0  ASLR off
- default for architectures that don’t support ASLR, and
- when the kernel is booted with norandmaps parameter

1  mmap(2) allocations, stack, and vDSO (virtual Dynamic Shared Object) page are randomized
- shared libraries will be loaded at randomized addresses
- text segment of PIE-linked binaries will also be randomized
- default if the kernel was configured with CONFIG_COMPAT_BRK

2  also support heap randomization
- default if the kernel was not configured with CONFIG_COMPAT_BRK

rabin2

Use rabin2 -I to see if a binary has PIC or other mitigations enabled
The same informations can be obtained with i (yep, lowercase) inside r2
Disabling ASLR

Under Linux you can disable ASLR:

- writing 0 into /proc/sys/kernel/randomize_va_space
  - you need to be root
  - this change has (a non-permanent) effect on the whole system, which is something you probably don’t want
    - to make it permanent, see sysctl(8)
- using gdb: set disable-aslr on
- using rarun2: asrl=no
  - it writes 0 to /proc/sys/kernel/randomize_va_space
- using the command setarch; for instance:
  
  setarch $(uname --machine) --addr-no-randomize bash

Unless in VM, use this one

---

Let’s see this in action

- no ASLR
- ASLR, no PIE
- ASLR, PIE
IOLI crackmes

- As an example, we’ll reverse the IOLI crackmes
  - by Pau Oliva (https://twitter.com/pof)
    “I wrote IOLI crackmes to teach my wife a bit of reversing around 10 years ago, I’m amazed they are still useful nowadays...”
  - This part of tutorial is heavily influenced/inspired by:
    by Julien Voisin

Always debug/run inside VMs

IOLI crackmes are absolutely safe; however, whenever you reverse an unknown binary, do it inside a VM
(Very simple) Static analysis is enough 😊

- `rahash2 -a md5,sha1,sha256 crackme0x00`

| crackme0x00: 0x00000000-0x00001d70 md5: 99327411dd72a11d71... |
|--------|------------------|
| crackme0x00: 0x00000000-0x00001d70 sha1: f2bf1c7758c7b1e22... |
| crackme0x00: 0x00000000-0x00001d70 sha256: 3aed9a3821134a2... |

- `strings`
- `rabin2 -z`
- Of course, r2:
  - `iz` (same as `rabin -z`)
  - `aa; pdf @ main`
    - There are a lot of different analyses; see a?
Some information

- r2 gives name to interesting offset; these *bookmarks* are called **flags**
  - *fs, f*
  - in many places, flag names can be expanded by using *tab-completion*
- strings can be seen with *iz* and *izz*
  - to see where they are used *axt @@ str.*
- after the analysis, *afl* lists the functions
Crackme0x00 - dynamic analysis

Not actually needed in this example, obviously

- `rarun2 -h > crackme0x00.rr2`
- on another terminal `rarun2 -t`
- edit `crackme0x00.rr2`
- run with: `rarun crackme0x00.rr2`
- debug with:
  
  r2 -A -d -e dbg.profile=crackme0x00.rr2 ./crackme0x00
  
  the following should be equivalent (but broken at the time of writing):
  
  r2 -A -d -r crackme0x00.rr2 ./crackme0x00
Crackme0x00: a bit of refactoring

My approach to reversing is to *iteratively*:

- explore
- (re)name things / add comments
- restart

saving refactor commands into “a poor man’s project”; that is:

- write all refactoring commands inside a `.r2` file
- for each (interesting) function:
  - seek to beginning address: `s`
  - give the function a meaningful name: `afn`
  - when necessary, remove (wrongly) inferred locals: `afv-`
  - rename (some) locals: `afvn`
  - set some comments: `CCu`
  - set some flags (=bookmarks): `f`
  - go back to the previous seek: `s-`
- reload the “project” with `. filename . r2`
Crackme0x00: debugging

In visual mode:

- step with S (step-over; s for stepping-in)
- before calling sym.imp.strcmp you can inspect the stack:
  - : to get into normal mode (à la vi)
  - pxW 8 @ esp; ps @ eax; ps @ [esp]; ps @ str.250382
  - pxr 8 @ esp
  - pf ss @ esp
no luck with strings or rabin2

aa; s main; V
  - imp....stands for imports
  - str....stands for strings

to “decode” (from hex) from
  - visual mode
    - did or
    - ?
  - normal mode
    - ahi d @
    - ?
Crackme0x02 and Crackme0x03

- check them with rabin2; what is the difference?
- crack the first one
  - can you do it statically?
  - ESIL VM can be handy: aeim, aepe, aesu
  - then, ?vi
- for the second one
  - try ag sym.test | xdot -
  - try s sym.test; VV
  - reverse obfuscation algorithm of Crackme0x03, and write a small deobfuscator
I think that –A/aa do too much; do you remember afv–?