**RE lab 01 - Black-box binary analysis**

**Setup (for all labs):**

You will need a virtualization software with snapshot capabilities (consider VirtualBox or VMware Player)

* One Linux VM with graphical user interface: Ubuntu 18.04 is recommended!
* One Windows VM: Win7 or Win10 recommended. Download official free VMs [here](https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/)

**Why do we need a VM?**

Some lab tasks will require malware analysis. You definitely don’t want to have malware corrupting your files. Moreover, completely solving a task might require resetting the VM to the initial state. Hence the need for snapshotting.

**Lab files**

Download from [here](https://pwnthybytes.ro/unibuc_re/01-lab-files.zip) Password is infected

**Tool presentation: strace/trace (Linux)**

**Installation**

sudo apt-get install strace ltrace

**Technical information**

[ltrace(1)](http://man7.org/linux/man-pages/man1/ltrace.1.html%7C) is a utility that can list the calls made to library functions made by a program, or the [syscalls](http://man7.org/linux/man-pages/man2/syscalls.2.html) a program makes. A syscall is a function that uses services exposed by the kernel, not by some separate library.

The way strace works is with the aid of a special syscall, called [ptrace](http://man7.org/linux/man-pages/man2/ptrace.2.html). This single syscall forms the basis for most of the functionality provided by ltrace, strace, gdb and similar tools that debug programs. It can receive up to 4 arguments: the operation, the PID to act on, the address to read/write and the data to write. The functionality exposed by ptrace() is massive, but think of any functionality you’ve seen in a debugger:

* attach/detach to a process
* set breakpoints
* continue a stopped program
* read/write registers
* act on signals
* register syscalls

A tool like strace only traces syscalls and reads registers in order to provide some pretty printing strictly concerning the syscalls of the traced process. However, ltrace provides further functionality and gathers information about all library calls. Here’s how ltrace does its magic:

* it reads the tracee memory and parses it in order to find out about loaded symbols
* it makes a copy of the binary code pertaining to a symbol using a PTRACE\_PEEKTEXT directive of ptrace()
* it injects a breakpoint using a PTRACE\_POKETEXT directive of ptrace()
* it listens for a SIGTRAP which will be generated when the breakpoint is hit
* when the breakpoint is hit, ltrace can examine the stack of the tracee and print information such as function name, parameters, return codes, etc.

**Tutorial task 1**

Using strace, investigate which syscall is used in the listing utility ls to get the filenames in a directory.

$ ls -a

. .. obscure obscure.c

$ strace /bin/ls .

execve("/bin/ls", ["/bin/ls", "."], [/\* 57 vars \*/]) = 0

...

openat(AT\_FDCWD, "/proc/filesystems", O\_RDONLY) = 3

fstat(3, {st\_mode=S\_IFREG|0444, st\_size=0, ...}) = 0

read(3, "nodev\tsysfs\nnodev\trootfs\nnodev\tr"..., 1024) = 355

read(3, "", 1024) = 0

close(3) = 0

access("/etc/selinux/config", F\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/usr/lib/locale/locale-archive", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=3031632, ...}) = 0

mmap(NULL, 3031632, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f6b95a2c000

close(3) = 0

ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0

ioctl(1, TIOCGWINSZ, {ws\_row=52, ws\_col=211, ws\_xpixel=0, ws\_ypixel=0}) = 0

stat(".", {st\_mode=S\_IFDIR|0755, st\_size=4096, ...}) = 0

openat(AT\_FDCWD, ".", O\_RDONLY|O\_NONBLOCK|O\_DIRECTORY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFDIR|0755, st\_size=4096, ...}) = 0

getdents64(3, /\* 4 entries \*/, 32768) = 112 <<<<<<<<<<<<<<<<<<<<<<<<<<<

getdents64(3, /\* 0 entries \*/, 32768) = 0

close(3) = 0

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 0), ...}) = 0

write(1, "obscure obscure.c\n", 19obscure obscure.c

) = 19

close(1) = 0

close(2) = 0

exit\_group(0) = ?

+++ exited with 0 +++

**Tutorial task 2**

Using ltrace, find out what input is accepted by the obscure binary:

$ ./obscure

1234

Wrong

$ ltrace ./obscure

\_\_isoc99\_scanf(0x5595b33758d4, 0x7fff07c26700, 0x7fff07c26848, 0x7f9141a45718this\_is\_my\_input

) = 1

strcmp("this\_is\_my\_input", "super\_secret\_password123325") = 1

puts("Wrong"Wrong

) = 6

+++ exited (status 0) +++

$ ./obscure

super\_secret\_password123325

Correct!

**Tool presentation: ProcMon (Windows)**

[Process Monitor](https://docs.microsoft.com/en-us/sysinternals/downloads/procmon) is an advanced monitoring tool for Windows that shows real-time file system, Registry and process/thread activity. It combines the features of two legacy Sysinternals utilities, Filemon and Regmon, and adds an extensive list of enhancements including rich and non-destructive filtering, comprehensive event properties such session IDs and user names, reliable process information, full thread stacks with integrated symbol support for each operation, simultaneous logging to a file, and much more.

**Tool presentation: API Monitor (Windows)**

[API Monitor](http://www.rohitab.com/apimonitor) is a free software that lets you monitor and control API calls made by applications and services. Its a powerful tool for seeing how applications and services work or for tracking down problems that you have in your own applications.

**Task 1: Solving a Linux crackme (puzzle)**

The binary will ask for your input and apply some checks on it. Whenever a check is failed, it will print “WRONG”. Since the binary is heavily obfuscated, static analysis is out of the question.

Fortunately, the binary is not completely securely designed, all checks except for the last one can be iteratively derived using ltrace alone. Try to get some intuition into each check and pass them one by one:

* use pwntools (install it) and Python to programatically call the binary directly and get its output **(1p)**
* stop calling the binary directly; wrap it inside ltrace and get all the library functions called **(1p)**
* bypass the length check by tring various inputs **(2p)**
* pass all the other checks **(2p)**
* find the correct password **(2p)**

**Task 2: Investigating a Windows malware**

Using API Monitor and ProcMon, gather as much information as possible about the malware:

* Where does it connect to? **(2p)**
* What registry keys does it access and why? **(2p)**

Select/deselect API categories according to what you want to obtain, in order to avoid scrolling through endless API calls.

**Bonus task: Malware vaccine**

In real-world infection scenarios, malware authors try to make their “software” as robust as possible. To this end, various synchronization mechanisms must be put into place.

* A ransomware strain will want to encrypt a system only once and consistently such that decryption will be possible (otherwise, they get no money back from the victims)
* A trojan will want to be running in the background only once in order to leave as small a footprint as possible and to have correct accountability (victim count) on the attacker’s end

However, the reverse can be applied by defenders: the same synchronization mechanisms can be put into place or simulated in order to fool the malware into thinking that the machine is already infected.

You can get more info [here](https://www.sans.org/blog/looking-at-mutex-objects-for-malware-discovery-and-indicators-of-compromise/)

* To get the bonus points, figure out how this malware uses synchronization to avoid reinfection and then devise a way to “vaccinate” machines against this malware. **(4p)**