Exercise 3

Reverse Engineering (RE) and Binary Patching

Log into your VM (`user` / `1234`), open a terminal and type in `infosec pull 3`.

- When prompted, enter your username and password.
- Once the command completes, your exercise should be ready at `/home/user/3/`.

When you finish solving the assignment, submit your exercise with `infosec push 3`.

Reminder - IDA

In the recitation, we've seen the basics of using IDA for reverse engineering a binary, and for figuring out where and how to patch it. In this homework assignment **we will usa IDA a lot**. Make sure you re-review the slides/video if you need a refresher.

Luckily for you we've already installed it on your machines. After downloading this exercise, you'll be able to launch IDA from the terminal[1], by typing the command `ida path/to/your/binary` (with the path to the binary you wish to disassemble).

Question 1 (60 pt)

In this exercise you will reverse engineer an example program that validates files using some logic. You will unravel the dark magic of what's happening inside the binary, and experience how this information can be exploited in various ways.

- All files belonging to this question are under the `q1/` directory.
- The binary we will be working is the `q1/msgcheck` program; this program receives a path to a file to validate, and returns **0** if the file is valid and **1** if it's invalid.

`/home/user/3/q1$ ./msgcheck 01.msg`	Run the program on 01.msg
`valid message`	← File seems to be valid
`/home/user/3/q1$ echo $?`	Print the exit code
`0`	← It's indeed valid
`/home/user/3/q1$ ./msgcheck 02.msg`	Run the program on 02.msg
`invalid message`	← File seems to be invalid
`/home/user/3/q1$ echo $?`	Print the exit code
`1`	← It's indeed invalid

[1] You will need to close and then re-open the terminal, for this to work

Part A **(20 pt)**
First, reverse engineer the `msgcheck` program to understand which messages it considers valid and which messages it considers invalid.

Then, inside `q1/q1a.py`, implement the `check_message(path)` method, so that it receives a path to a `.msg` file and returns **True** on valid messages and **False** on invalid messages.

Document your solution (briefly!) inside `q1/q1a.txt`.

Part B **(10 pt)**
Now, we'll write a Python script to fix `.msg` files so that they become valid. Inside `q1/q1b.py`, implement the `fix_message(path)` method, so that it receives a path to a `.msg` file and creates a new file with a similar name with a `.fixed` suffix.

```
/home/user/3/q1$ ./msgcheck 02.msg
invalid message
/home/user/3/q1$ echo $?
1
/home/user/3/q1$ python q1b.py 02.msg
done
/home/user/3/q1$ ./msgcheck 02.msg.fixed
valid message
/home/user/3/q1$ echo $?
0
```

Document your solution (briefly!) inside `q1/q1b.txt`.

Part C **(10 pt)**
Find another way to fix `.msg` files, implement the `fix_message(path)` method in `q1/q1c.py`, and document your solution (briefly!) inside `q1/q1c.txt`.

Part D **(10 pt)**
This time, instead of fixing the message files, we will patch the program itself! We will do this by patching the program so that it always follows the valid code branch (regardless of whether the message is valid or not).

Inside `q1/q1d.py`, implement the `patch_program(path)` method, so that it receives a path to the `msgcheck` program and write the patched program to the path together with a `.patched` suffix.

(See screenshot on the next page)

```
/home/user/3/q1$ ./msgcheck 02.msg
invalid message
/home/user/3/q1$ echo $?
1
/home/user/3/q1$ python q1d.py msgcheck
done
/home/user/3/q1$ chmod +x msgcheck.patched
/home/user/3/q1$ ./msgcheck.patched 02.msg
valid message
/home/user/3/q1$ echo $?
0
```

← Make the result file executable

Document your solution (briefly!) inside `q1/q1d.txt`.

Part E (10 pt)
Find another way to patch the program, this time so that it returns **0** for all messages (whether valid or not), but without changing anything else (i.e. the output to the screen).

Inside `q1/q1e.py`, implement the `patch_program(path)` method, so that it receives a path to the `msgcheck` program and write the patched program to the path together with a `.patched` suffix.

```
/home/user/3/q1$ ./msgcheck 02.msg
invalid message
/home/user/3/q1$ echo $?
1
/home/user/3/q1$ python q1e.py msgcheck
done
/home/user/3/q1$ chmod +x msgcheck.patched
/home/user/3/q1$ ./msgcheck.patched 02.msg
invalid message
/home/user/3/q1$ echo $?
0
```

Document your solution (briefly!) inside `q1/q1e.txt`.

Question 2 (40 pt)

In this exercise you will patch a binary to implement more interesting logic than just changing a return value or print. The program we'll patch is `q2/readfile` - a program that reads files line by line. For example, for the file `q2/1.txt` the output will look as follows:

```
/home/user/3/q2$ ./readfile 1.txt
Line 1
Line 2
#!echo Victory
Line 3
/home/user/3/q2$
```

Our goal is to patch the program so that every line beginning with a **#!** will be executed (but not printed). For example, for `q2/1.txt` the result of patching would be:

```
/home/user/3/q2$ python q2.py readfile       ← Run our code to
done                                             patch the program
/home/user/3/q2$ chmod +x readfile.patched   ← Make the result
/home/user/3/q2$ ./readfile 1.txt                file executable
Line 1
Line 2
Victory                                      ← Here's the change
Line 3
```

Inside `q2/q2.py`, implement the `patch_program(path)` method, so that it receives a path to the `readfile` program and write the patched program to the path together with a `.patched` suffix.

Since this question is potentially challenging, try following the steps detailed below:

1. Reverse engineer the `readfile` program and find **dead zones** into which you can patch your code (we added these zones deliberately in this program, so they're going to be hard to miss :))

2. Out of the two dead zones, one is quite big (has plenty of space for your code) and one is very small (doesn't have enough space for our code, but we can use it to redirect to our code in the other dead zone). Identify which is which.

3. Use IDA to figure out the offset in the code (binary) of each patch, and also the virtual address of each patch.

4. Inside `q2/patch1.asm` write x86 assembly for what we'll patch into the small deadzone - code to perform the redirection from the small deadzone to the big deadzone.

5. Inside `q2/patch2.asm` write x86 assembly for what we'll patch into the big deadzone:
    a. The code will check if the string starts with **#!** or not.
    b. For lines not starting with **#!**, it will jump back to the original code, right **before** the call to `printf`.
    c. For lines starting with **#!**, it will first call `system` (a standard library function to execute a string as a shell command[2]) and then jump back to the original code, right **after** the call to `printf`.

6. To assemble the code from the `.asm` files, you can use the code we provided you inside `q2/assemble.py`:
    a. First do `import assemble`
    b. Then call `assemble.assemble_file` (to get the binary machine code the assembly in the file) or `assemble.assemble_data` (to get the machine code for instructions directly specified in a string)

Document your solution (briefly!) inside `q2/q2.txt`.

Final notes:

- This exercise is more challenging than the previous ones. It does not mean it's impossible, but please please don't leave it for the last minute.
- **Document your solutions.**
- Don't use any additional third party libraries that aren't already installed on your machine (i.e. don't install anything).
- If your answer takes an entire page, you probably misunderstood the question.

[2] How wonderful that it was "miraculously" included in the `readfile` program :)