Exercise 4

Buffer Overflow (BOF)

Log into your VM (`user` / `1234`), open a terminal and type in `infosec pull 4`.

- When prompted, enter your username and password.
- Once the command completes, your exercise should be ready at `/home/user/4/`.

When you finish solving the assignment, submit your exercise with `infosec push 4`. NOTE: After solving `q2/q2a.py`, you will get permission errors when submitting, unless you read and follow the instructions at the end of this exercise. Please read them carefully.

Background

In this exercise, you will exploit a real vulnerability in existing code on your machines, and leverage that to gain temporary root permissions. You'll then use these permissions to install a "backdoor", to allow yourselves to regain these root permissions. But more important, you'll be able to do this:

```
root@ubuntu:/home/user/4# echo "I am g`whoami`!"
I am groot!
```

The program we'll attack is `sudo` - a standard program included on Unix systems, and used to execute commands with root permissions. For full details see Wikipedia (or your favourite Operating Systems course), but the tl;dr is:

- Like most file systems, each file is owned by a user/group
- One of the permissions bits on a file is the setuid bit
    - When set, this bit causes a program to automatically run under the permissions of the user owning the file
- The `sudo` program is owned by the `root` user and has the setuid bit on
- When a user tries to use `sudo`, it checks the user belongs to the `sudo`'ers user group, and then executes the command

It should now be clear why it's attractive to attack the sudo program. With our goal set clearly, let's begin 😈.

Question 1 (40 pt)

Enter the `q1/` directory. Within this folder, you will find a `sudo` program, which receives a password and executes a command as root iff the password is correct. However, ~~you~~ we don't know the password…

1. Use both the source code and IDA to find a vulnerability in the `sudo` program.
2. Inside the `q1/q1.py` script, implement the `run_command(command)` function - the function receives a command string, and runs the command as root using the `sudo` program[1].
    a. Use `subprocess.call` or `os.execl` to run commands (instead of `os.system`), otherwise you'll have problems with commands containing certain characters
    b. To test if it works, run the `whoami` command - this command prints the name of the user running it. If you did everything right, this should be root :)
    c. If successful, it will look like so:

```
/home/user/4/q1$ python q1.py whoami
Running command…
root
```

3. Describe the vulnerability and your solution in `q1/q1.txt`.

Exploit the vulnerable sudo program

Use your solution to question 1, to add yourself to the `sudo`'ers list, by running the following command:

```
/home/user/4/q1$ python q1.py "adduser user sudo"
Running command…
```

Once done, **stop for a moment to appreciate what you just accomplished**. You just gained an unauthorized root access to a system, by hacking it, for realz.

Welcome to the world of information security, a cool world with endless possibilities, and a scary world once you realize how dangerous bad code can be.

[1] Don't try brute-forcing the password and then using a fixed password. No points will be given for that solution…

Restart your machine for the permission changes to take effect. From now on, you will be able to run commands using the real sudo program, not just the one we provided. You'll need this for the next question.

Question 2 (60 pt)

Enter the `q2/` directory. Within this folder, you will find yet another `sudo` program. While the vulnerability from the first question was fixed, a new and much more interesting vulnerability was introduced in the new program.

We will exploit this vulnerability to open an interactive shell (using `/bin/sh`) with root privileges.

1. Use both the source code and IDA to find a vulnerability in the `sudo` program.
2. Inside the `q2/q2a.py` script, implement the `crash_sudo` function - the function will invoke `sudo` in a way that will make it crash, and generate a core dump[2]. Describe your solution in `q2/q2a.txt`.

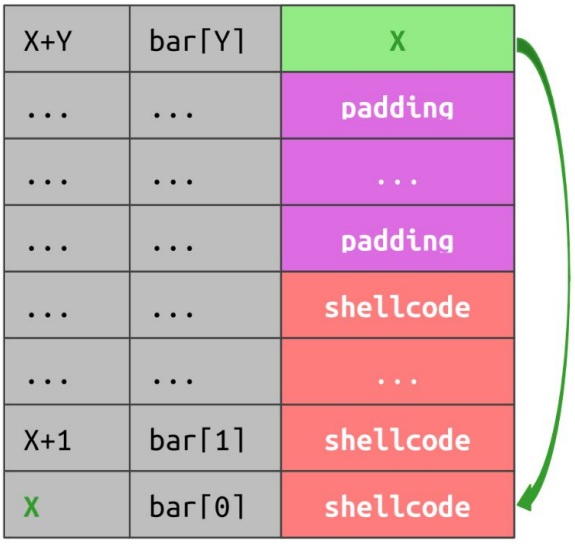
```
/home/user/4/q2$ python q2a.py
Segmentation fault (core dumped)
/home/user/4/q2$ ls -lah
Total 384K
drwxrwxr-x 2 user user 4.0K Mar 11 23:58 .
drwxrwxr-x 4 user user 4.0K Mar 11 23:56 ..
-rw-rw-r-- 1 user user  931 Mar 11 23:59 assemble.py
-rw------- 1 root root 340K Mar 12 00:05 core
...
```

3. The core dump of a program is a "snapshot" of the program memory and registers at the point in time right when it crashed. It would be really useful if we could look at it with GDB, however it is owned by root[3]...

4. Oh, wait. We are (g)root! Use (the real) sudo to open the core dump with GDB, to do the same sort of analysis we did in class!
    a. Find where the buffer begins (let's call this X)
    b. Find at which offset from the beginning of the buffer, we have the value we want to "update" (let's call this offset Y)
    c. Now, we know what we should do - create a buffer of size Y, beginning with the shellcode, followed by padding (if needed) until we reach a length of Y, and then finally add the address X.

[2] If you see a "Segmentation fault (core dumped)" message, specifically with the "core dumped" part, but not actual core file is created - restart your machine with a valid internet connection to receive an automatic update to fix this.
[3] If we could have accessed it, that would be a security problem!

d. **This process is illustrated on the next page.**



5. Now, the final part is writing your shellcode. We've seen the basic technique in class, including how to get the string `"/bin/sh"`, how to do a syscall, and more. Your job is to complete the missing parts, and connect it all together.
    a. Write your shellcode directly in x86 assembly inside **`q2/shellcode.asm`** and use the **`assemble.py`** script to assemble it.
        i. Optional: If you want to test your shellcode, write a C program to read the output of the **`assemble.py`** script into a buffer, and then "Call" this buffer to execute your shellcode. Check that you can really get a shell :)
        ii. Hint: The **`assemble.py`** script can help you check if your shellcode has zero bytes in it
    b. Inside the **`q2/q2b.py`** script, implement the `run_shell` function, which opens a shell with root permissions
        i. Again, don't use `os.system`
    c. As usual, document your solution in **`q2/q2b.txt`**.

```
/home/user/4/q2$ python q2.py
# whoami
root
#
```

Final notes:

- IMPORTANT: Your submission is going to (automatically) include a core dump in q2a, and the core dump is "owned" by root, so we can't access it directly.

This means you'll have to do the submission as root for submission to work: `sudo infosec push 3`

- Document your code (and your shellcode!).
- 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. This is true to all parts of this question - code, shellcode and documentation.
- Each student is getting slightly different binaries for this question - don't be surprised if your solution is different from those of other students.