Exercise 7

Code injection and Syscall interception with Ptrace

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

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

When you finish solving the assignment, submit your exercise with `infosec push ex-ptrace`.

Motivation

In this exercise, you will experiment with using the `ptrace` functionality. We'll have a C&C server controlling a malware, and an antivirus attempting to detect the malware. From the C&C server, you will send commands to the malware, to modify the execution of the antivirus, so the antivirus will no longer see our virus.

We will experiment with both code injection (modifying the code of the antivirus) and syscall interception (modifying the results returned from syscalls dynamically), both which can be used to sabotage the detection efforts of the antivirus.

Background

Let's go over the files and tools you'll use, starting with the `binaries/` directory:

- `binaries/`**`malware`** (and its source files) - a simple backdoor that:
  - Connects to a C&C server every few seconds to download a payload
  - Writes the payload to `/tmp/payload`, and executes it
    - A payload can be a shell command or a binary file; anything that runs with `chmod +x /tmp/payload; /tmp/payload`
  - Uploads back the STDOUT of the payload to the server
- `binaries/`**`antivirus`** (and its source files) - a simple antivirus that:
  - Receives a directory and scans files in it recursively
  - It compares the hashes of the files in it with static signatures of known malware (including the hash of our malware)
  - For each matching file, it emits a warning.
- `binaries/`**`libvalidator.so`** (and its source files) - a part of the antivirus that was factored out (to be easily upgradeable), and containing the

`check_if_virus(char* path)` function that the antivirus invokes on the files it scans to check if they're suspicious.

- `binaries/`**`stop_exercise.sh`** - this script will kill all the running binaries of the exercise (`malware`, `antivirus` and `/tmp/payload`) to allow you to start a "fresh" run.
    - Note that **this script must be executed as root** (i.e. with `sudo`)

Outside the `binaries/` directory, we have a new script, **`server.py`** - **a simple C&C server**. If you check out its code (which is pretty simple), you'll see it does the following:

- It listens on port 8000 for incoming connections from the malware
- When it accepts a connection, it sends a payload to the malware
- From the malware, it receives back the product (the payload's STDOUT), and invokes a handler function to handle the product
- The handler function can call the `add_payload` function to enqueue another payload and handler
    - By having each handler add more payloads and handlers, you can build arbitrarily complex logics with many steps

For example, we added an implementation that performs a directory listing on `/etc`, and if it contains the `shadow` file (see Wikipedia), it sends a followup payload to read it, and then extracts the root password out of it.

To see this for real time, run `python3 server.py`, and then `binaries/malware`; you can also run `/binaries/antivirus` to see it detect our malware. Look at the server logs. Now look at the malware logs. Now look at the antivirus logs. Back to the server logs. Back to the malware logs. I'm on a horse.

Question 1 (10 pt)

In this part, we'll get started by implementing a C&C server that, if the antivirus is running, finds its process ID (`pid`), and then simply kills it with `kill -9 <pid>`. This has to happen in two steps:

- First, send a command to find the process pid, and parse its STDOUT.
- Then, send a command to kill that process, using the pid you've just extracted.

Part A (5 pt)

To make this a little easier, we've implemented the `EvadeAntivirusServer` base class in `evasion.py`, so you only need to implement these methods:

- `payload_for_getting_antivirus_pid()`: this function returns the payload that's used to get the antivirus pid. We're good at naming stuff.
  - **Note**: You can assume the name of the antivirus process is `antivirus`.
  - Don't assume anything on the path from which it was invoked.
- `get_antivirus_pid(product)`: this function is called on the product of the previous payload (the STDOUT), to extract the `pid` from it.

Implement these methods in `evasion.py`.

Note: there's **no need to implement the `evade_antivirus(pid)` function**. This is automatically implemented for you in all questions (`q1.py`, `q2.py`, …).

Q: What if no antivirus process is running?
A: As written in the function docstring, return -1.

Part B (5 pt)

In `q1.py`, implement the `get_payload(pid)` function. This function returns the payload that's used to evade the antivirus's detection mechanism; in this case, this should be the command to kill the antivirus. As usual, document your solution in `q1.txt` (specifically, document how you extracted the process ID. We know how you generated the kill command).

Q: How does the malware interact with the C&C server?
A: As is implied by its name, the C&C **server** is a **remote server** that is managed by the malicious actor that controls the malware. In this entire exercise, we are going to assume that the server runs on a **different** machine, and is only interacting with the malware through the network socket. This means that the server must send the

payload over the network in order for the malware to be able to get it. The malware won't be able to access anything that the server didn't explicitly send it over the network (For example, **they can't share files on the filesystem**).

Question 2 (20 pt)

Killing the antivirus is pretty crude. This time, write a binary payload in `q2.c` that uses `ptrace` to **overwrite `check_if_virus`'s code with assembly that always returns 0**, and serve it via the `get_payload` function in `q2.py`.

Continuing the tradition from exercise 5, update the addresses you use in `addresses.py`. For this exercise, update `CHECK_IF_VIRUS_CODE` in `addresses.py` and fill it in the payload using the Python code (i.e. don't hardcode this address in `q2.c` or `q2.py` - instead always read the value from `addresses.py` and put it in the shellcode). As usual, document your solution in `q2.txt` (specifically, document how you found the address to override).

That would be a good point to move on to question 3.
However, you probably have some questions, so let's try answering those first:

Q: The first step is still finding the antivirus pid and extracting it; but now that you've got it, how do you "embed" it into your binary payload? And how do we "embed" the memory address to overwrite?
A: Lucky for us, we know how to patch binaries. Actually, we enjoy it. So we're going to add a global variable, for example `int pid = 0x12345678;` in `q2.c`, and then compile it into `q2.template` - that is, not the final payload, but a template for what we're going to send. In fact, you can use the `makefile` we provided to do just that.

Now, in `get_payload`, we don't simply read `q2.template` and serve it, but search for that unusual `0x12345678` sequence in it, and when we find it, we replace it with the 4 bytes of the actual pid. Neat, huh? And we can repeat this trick with another global variable, to pass the memory address we wish to overwrite.

To conclude, we send `q2.template` with the antivirus pid and memory address to overwrite/patch, resulting in a payload that `ptrace`'s the antivirus, overwrites `check_if_virus`'s code, and "blinds" the antivirus to our malware (and any other malware, actually).

Q: I'm getting a message that the target is 'up to date' when running `make` with the `makefile`?
Run `make clean` before, to delete the previously built version of the template files, and then run make again with the relevant target.

Q: Why does it take the payload a few seconds to run? And why does the antivirus still print a warning one last time in some of the runs?
A: You can't `ptrace` a sleeping process, so your payload may hang for several seconds until the antivirus wakes up. Additionally, depending on whether the OS runs `ptrace` or the scan first, the antivirus may emit a warning one last time. If it does, don't worry; just make sure no warning appears in any of its next scans.

Question 3 (35 pt)

Overwriting `check_if_virus`'s code is not ideal, as `libvalidator.so` might get updated, and the opcodes' address might change. This time, write a binary payload in `q3.c` that uses `ptrace` to overwrite `check_if_virus`'s GOT entry with some other function with a similar signature, that will return 0 on our malware, and serve it via the `get_payload` function in `q3.py`. As usual, document your solution in `q3.txt` (specifically, document how you found the GOT address, and document which alternative you're using).

As before, document addresses you use - this time update `addresses.py` with `CHECK_IF_VIRUS_GOT` - the address of the GOT entry to override, and `CHECK_IF_VIRUS_ALTERNATIVE` - the address of the similar function to write.

Again, more questions that might trouble you are answered below.

Q: How do we get the pid and addresses into the payload
A: Similar to the previous question: use the `makefile` to create `q3.template`, patch it with the actual pid and addresses, and pat yourself on the back when the warning disappears from the antivirus's logs.

Q: Wait, patching the GOT?
A: Yes. In class, we mentioned that the GOT is the table storing the addresses for symbols we use from shared libraries. Just find another function with the same signature, that would return 0, and replace the addresses in the GOT.

Q: How do we know which address to patch in the GOT?
A: Re-read **recitation 7** and **recitation 5** to find the GOT addresses.

Question 4 (35 pt)

Overwriting `check_if_virus`'s code is a good idea, but future versions of the antivirus may have several similar functions, called from several places in the code, and we're going to have to patch all of them.

So instead of doing that, write a binary payload in `q4.c` that uses `ptrace` to intercept the antivirus's system calls and **make all the read syscalls fail**. This will cause every file to look empty to the antivirus, and as a result, our malware will not match its signature.

Note the important difference - while the previous payloads rewired the antivirus memory and then quit, **this payload needs to keep running**, in order to keep intercepting and sabotaging system calls.

Therefore, when you're debugging, to make sure there's only one `/tmp/payload` running, **use the `stop_exercise.sh` script to kill all previously running payloads/viruses/etc. and start with a fresh run**.

Implement your solution in `q4.py` and `q4.c` and document it in `q4.txt`.

Final notes:

- Document your code (and your payloads!).
- Remember to document all addresses in `addresses.py`, and don't hardcode addresses anywhere else (so we can update `addresses.py` ourselves).
- Don't use any additional third party libraries that aren't already installed on your machine (i.e. don't install anything).
- None of the coding parts (both C and Python) is supposed to be very long. If it's way more than 60-70 lines per question, it may be that you misunderstood the question.