# Lab 3

Create a virus using Python / Malware Analysis

## [Creating a self-replicable Virus]

Python provides a powerful tool to automate routine command line operations. You will create a (**self-replicable) virus** using the following python codes.

1. Run Ubuntu VM for today's lab and do the following.
    - Create folders:
      ```
      $ mkdir -p ~/Desktop/lab3/virus1
      $ mkdir -p ~/Desktop/lab3/virus2
      ```
    - Create two files *test1.virus* in `~/Desktop/lab3/virus1` and *test2.virus* in `~/Desktop/lab3/virus2`.
    - `$ cd ~/Desktop/lab3` and create *lab3.py* for today's lab.

2. "subprocess" is a useful python module that allows you to execute new processes, connect to their input/output/error pipes, and obtain their return codes. Write a simple code to print the list of files in the current folder using a subprocess module*:*

*lab3.py*

```
#!/usr/bin/env python3
import subprocess

subprocess.call(["ls", "-l"])
```

- Compile and run lab3.py by typing:
  ```
  $python3 lab3.py
  ```
  What's happening?

- Note that `#!/usr/bin/env python3` is defined at the beginning of the python program to make it executable. Change the property of lab3.py to an executable program using the following command:
  ```
  $ chmod 777 lab3.py
  ```
  Then, execute the program by typing:
  ```
  $ ./lab3.py
  ```

- "subprocess.call" executes "ls -l" and takes ["ls", "-l"] which is a list variable having two entities such as "ls" and "-l". You need to parse your Linix command as a list to execute *subprocess.call* since *this function* takes input Linix commands as <u>a list format</u>. In order to get more information about a list variable, see [https://docs.python.org/3/tutorial/introduction.html#lists](https://docs.python.org/3/tutorial/introduction.html#lists).

3. You can use *subprocess.call* without manually changing a Linux command into a list variable by setting "*shell*" property true. But for code safety, the recommended way is to use *this option only for a fixed command.*
   Execute the following:

```
#!/usr/bin/env  python3
import subprocess

subprocess.call("ls  -l", shell=True)
```

   What's the result?

   Of course, you may try to use pipeline "|". Try to execute the following modified code, see the results:

```
#!/usr/bin/env  python3
import subprocess

p1 = subprocess.call("ls  -l | grep py", shell=True)
```

4. *Popen* supports more flexible management of commandline processes. For example, you can execute "ls -l | grep py" using *Popen* without *"shell=true"*. Run the following code and execute other various possible commands by yourself.

```
#!/usr/bin/env  python3
import subprocess
from subprocess  import Popen, PIPE

p1 = Popen(['ls', '-l'], stdout=PIPE)
p2 = Popen(['grep', 'py'], stdin=p1.stdout,  stdout=PIPE)
p1.stdout.close()
output = p2.communicate()[0]
p2.stdout.close()
print(output.decode('ascii'))
```

5. To avoid damage in your file system, select files only having "*.virus" in a current folder as target host files. You can get the list of this file using *glob.glob* ("*.virus"). For example, execute the following code and observe the output:

```
#!/usr/bin/env  python3
import glob

for item in glob.glob("*.virus"):
   print(item)
```

**[Tips]** Alternatively, you can use *subprocess* with the command "ls *.virus" to get the list of files. In this case, you have to convert the output of *subprocess*, which is just a single string, to a list format having multiple entities to get the same result.

6. 1) Opening a file, 2) reading all lines in the opened file and 3) writing new lines on the file can be permitted with the following command.

```
#!/usr/bin/env  python3
import sys
import subprocess

Filein = open("lab3.py", 'r')  #open file to read
all_contents = Filein.readlines()
print(all_contents)
Filein.close()

Fileout = open("lab3_copy.py", 'w') #open file to write
Fileout.writelines(all_contents)
Fileout.close()
```

   See *lab3_copy.py* to check that all operations are successfully done:

7. You also can read only a few numbers of lines and store each line of the file to a list. One easy way is using the following:

```
some_contents = [line for (i,line) in enumerate(Filein)  if i < 2]
```

   The above command will read only two lines from Filein and save it to some_contents as a list. Replace all_contents in the above example to some_contents and see *lab3_copy.py* again to check the output.

**[Task]**

T1. Make a simple python virus program performing the following:

- Read the list of all target host files.
- Comment out the original contents of the host files (using '#').
- Attach itself (python codes of the original virus) in the host file.
- Make the infected host file executable.

T2. The virus program attaches itself to host files so that it propagates the virus. Test it using the following sequence:

1) Create *virus.py* in ~/Desktop/lab3/virus1 and execute it. If *virus.py* works, *test1.virus* is infected and executable.
2) Copy *test1.virus* to ~/Desktop/lab3/virus2 and execute *test1.virus*.
3) Check whether *test2.virus* is also infected.

**[TIP]** A simple virus program (say, virus.py) can be implemented in the following sequences:

1) Read the self-replication codes from *virus.py* or the infected file.

   i. The name of the file running is automatically assigned in "*sys.argv[0]*" when it is executed (e.g. *sys.argv[0]* will be "*virus.py*" if you run *./virus.py*)

2) Read all file names which having "*virus*" as an extension.
3) For each target file,
   i. Open the file to read.
   ii. Read all contents from the file and store the contents to a variable.
   iii. Close the file and open the file again to write.
   iv. Write self-replication codes at the beginning of the file.
   v. Comment out all lines of the original contents (i.e. prepend "#" to each line) and write them to the file.
   vi. Close the file.
   vii. Make the target file executable.

**[Malware Analysis]**

1. Open a web browser on Windows and go to https://www.winitor.com/download/
2. Download the zip file and extract the file somewhere. You will need to run pestudio.exe.
3. Now go to https://bitbucket.org/jongkil/ethereum_lab/downloads/Magnify.exe and download "Magnify.exe".
4. Run pestudio.exe and drag and drop Magnify.exe to the main window of the pestudio program. (The loading will take some time.)
5. After Magnify.exe is fully loaded, answer the following questions.
   1) What are md5, sha1 and sha256 fingerprints of this binary?
   2) What is compiler-stamp (time stamp)?
   3) How can we view the APIs imported by this binary?
   4) What DLLs this binary depends on?
   5) Can you spot some blacklisted DLLs?
   6) List the API functions that dwmapi.dll imports.
   7) What are the sections that this binary consists of?
   8) Can you find section hash values?
   9) What are the resources this binary uses?
   10) Check the extracted strings by clicking "strings" button.

Note that Magnify.exe is a system file helping people with visual impairment to log in Windows systems. As it can be accessed even before users log in, it often becomes a target for hackers to gain an access to a Windows system. The attached file is the normal Magnify.exe file, but one can check whether it is compromised by checking the above questions.