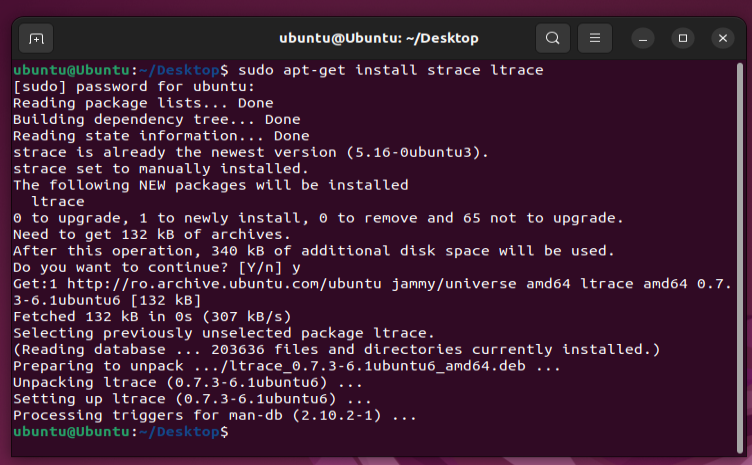
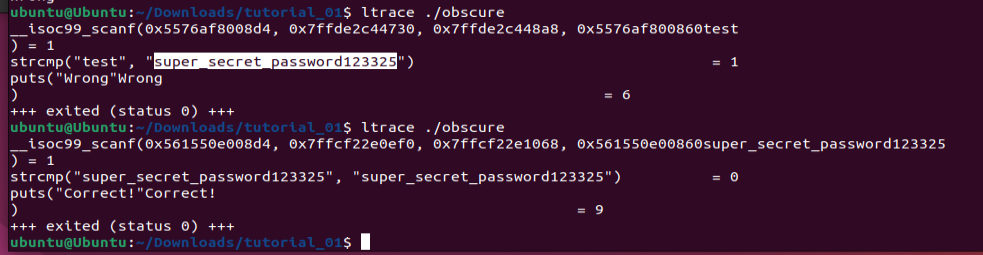
Lab Session 0x01

First step (Linux): **$ sudo apt-get install strace ltrace**



We use ***ltrace*** to understand what is a good input for the *obscure* binary from the zip file.



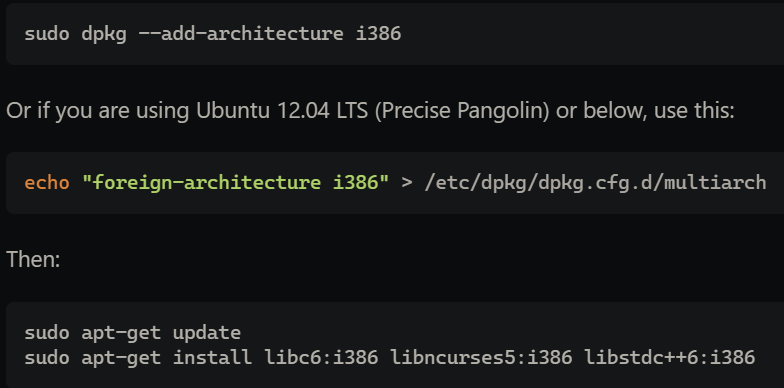
Task 1: for Linux

* use pwntools and Python to automatically call the binary directly and get its output; (1p)

We need to install ***pwntools***. Following the [documentation](https://docs.pwntools.com/en/stable/install.html):



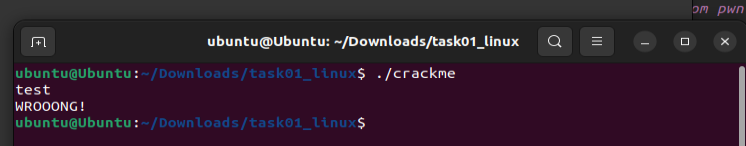
If we try to run *./crackme*, we get an error: ***No such file or directory***. We need to run a 32-bit executable file on a 64-bit multi-architecture Ubuntu system, so we have to add the i386 architecture and install the three library packages *libc6:i386*, *libncurses5:i386*, and *libstdc++6:i386*. Following the steps from [here](https://askubuntu.com/questions/454253/how-to-run-32-bit-app-in-ubuntu-64-bit/454254#454254):



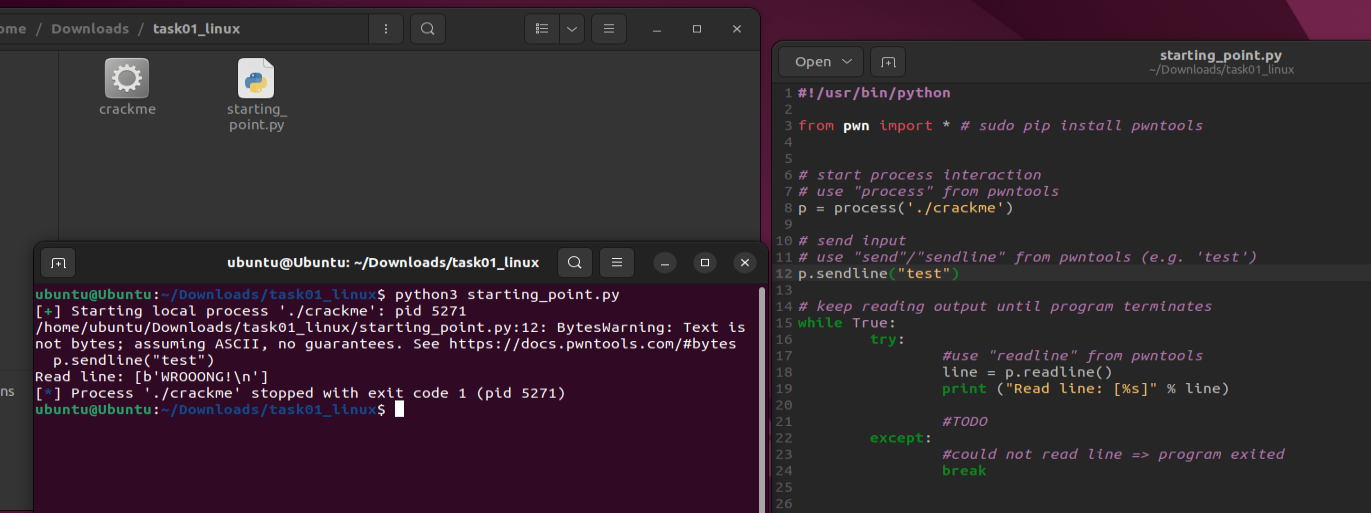
Proof:



Now we run ***./crackme***:

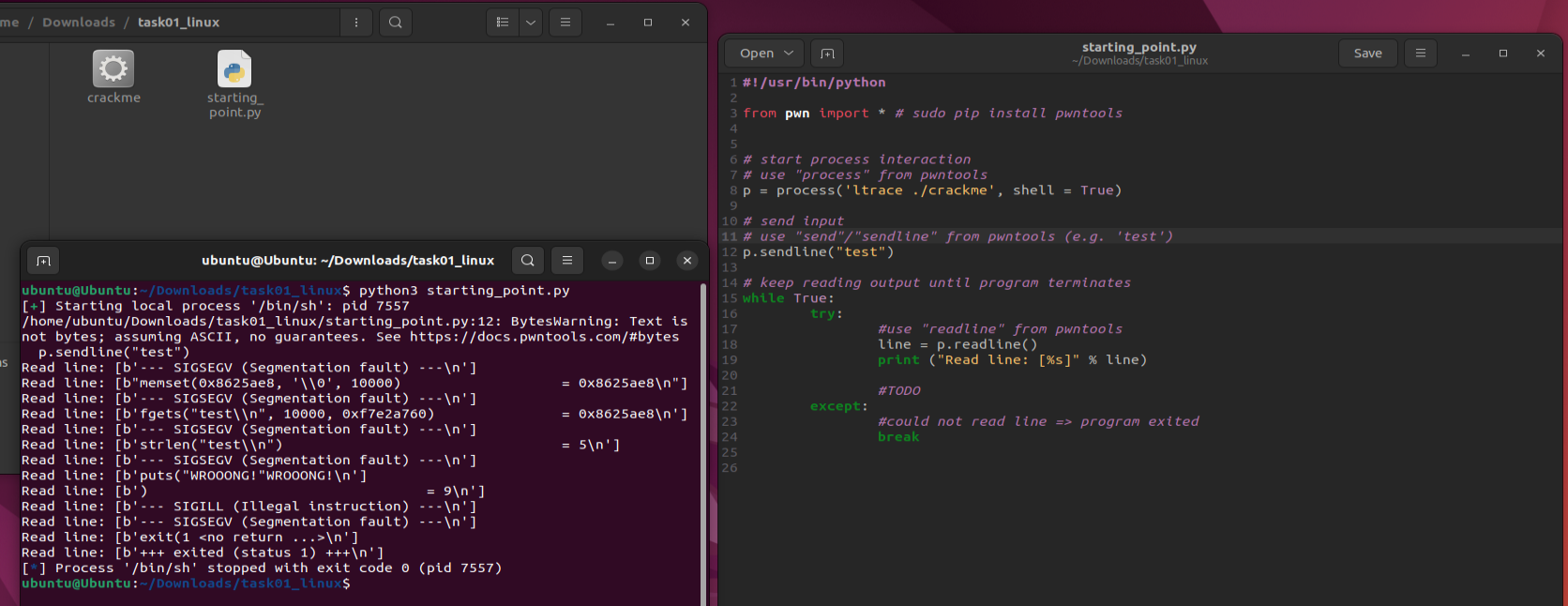


We modify the *.py* file and run it:



The code is in ***task1.py***. The output of the script is printed in the console.

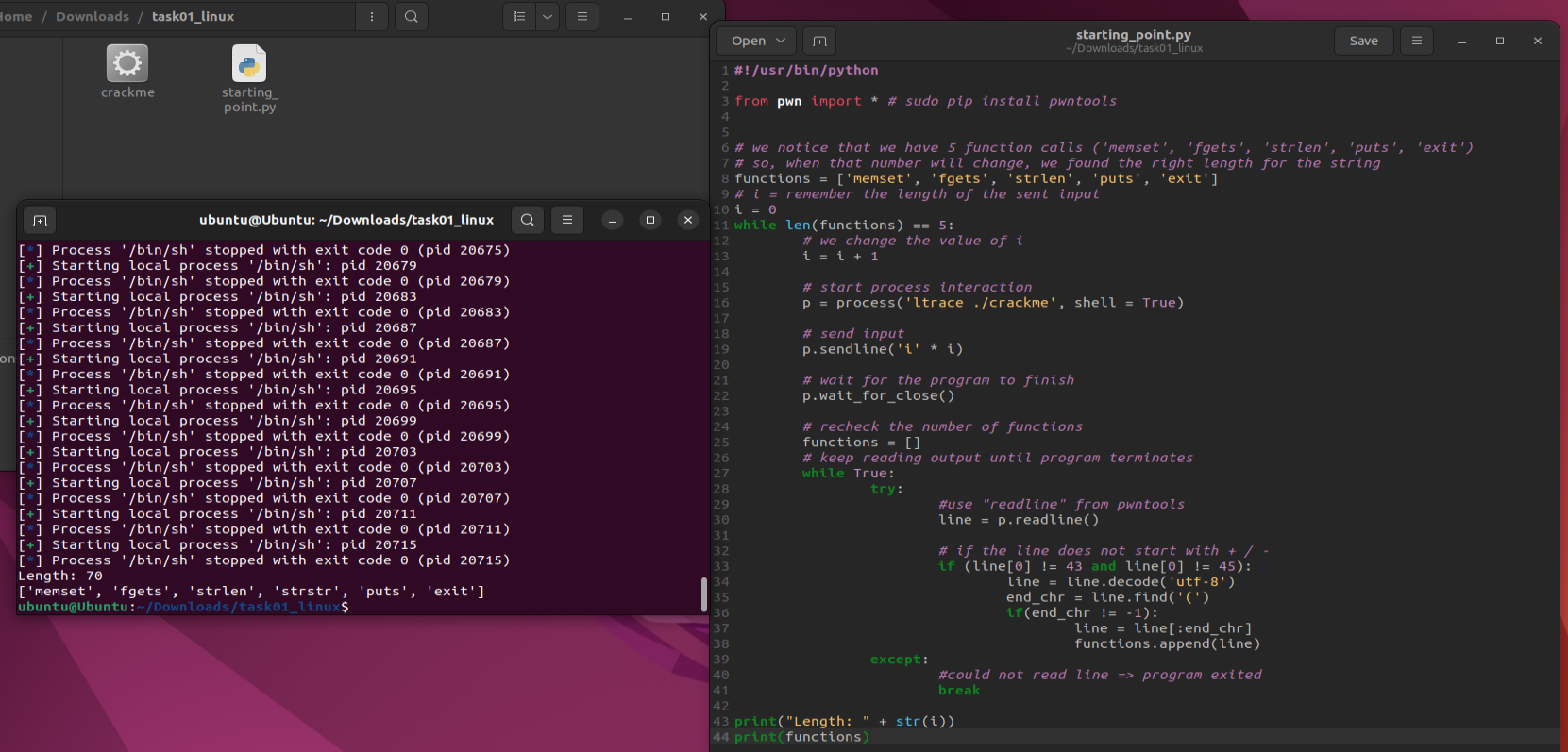
* stop calling the binary directly; wrap it inside ***ltrace*** and get all the library functions called; (1p)



The code is in ***task2.py***. The output of the script is printed in the console (we can see the library function calls → we see a ***strlen***, so we need to bypass the length).

* bypass the length check by trying various inputs; (2p)

We can notice that we have 5 function calls ('memset', 'fgets', 'strlen', 'puts', 'exit'), so, when the number of functions will be different, then we found the proper length.



The code is in ***task3.py***. The length is 70 (71 if we count the final **\n** added auto) and we can notice another function called was made: ***strstr***.

* pass all the other checks; (2p)

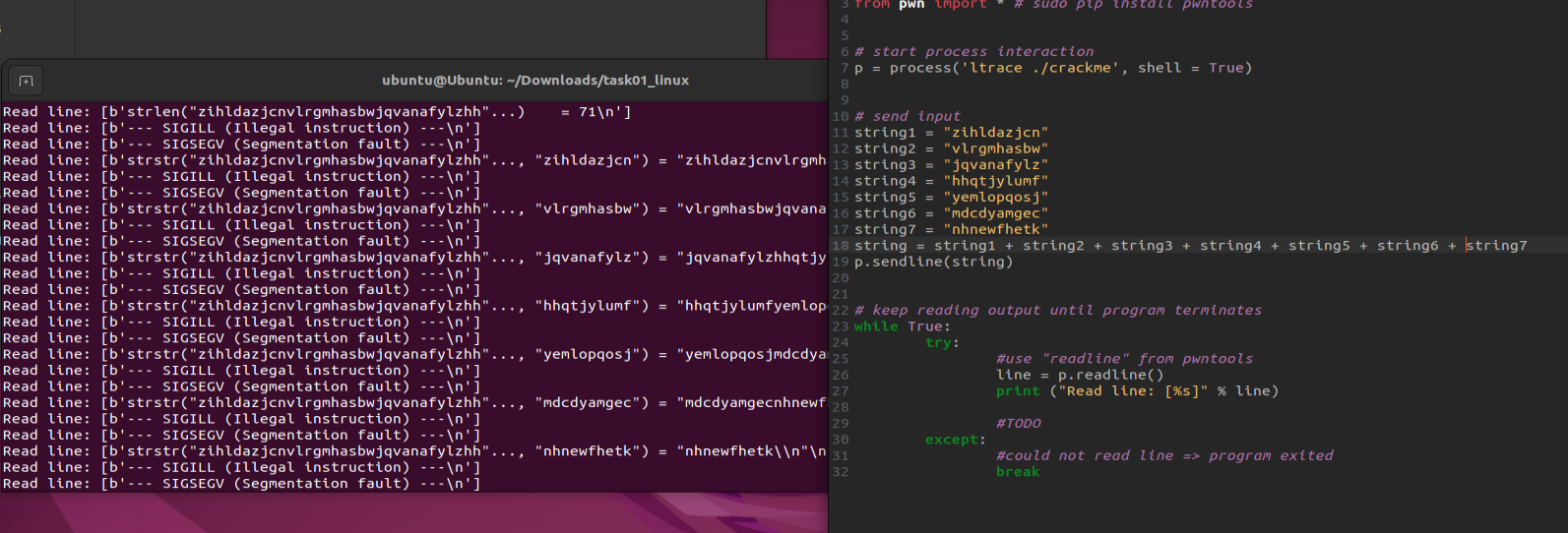
By running ***task2.py***, but with 70 characters, we get:



So we found the first substring, let’s see what happens if we append this substring at the beginning of our input:



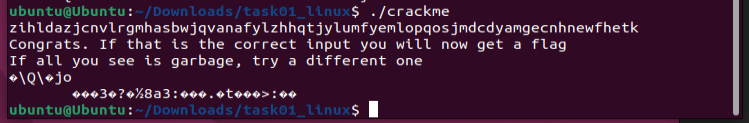
We found another substring. We add this string after the first one and we repeat the operation until there are no substrings left. We find a total of **7 substrings**:



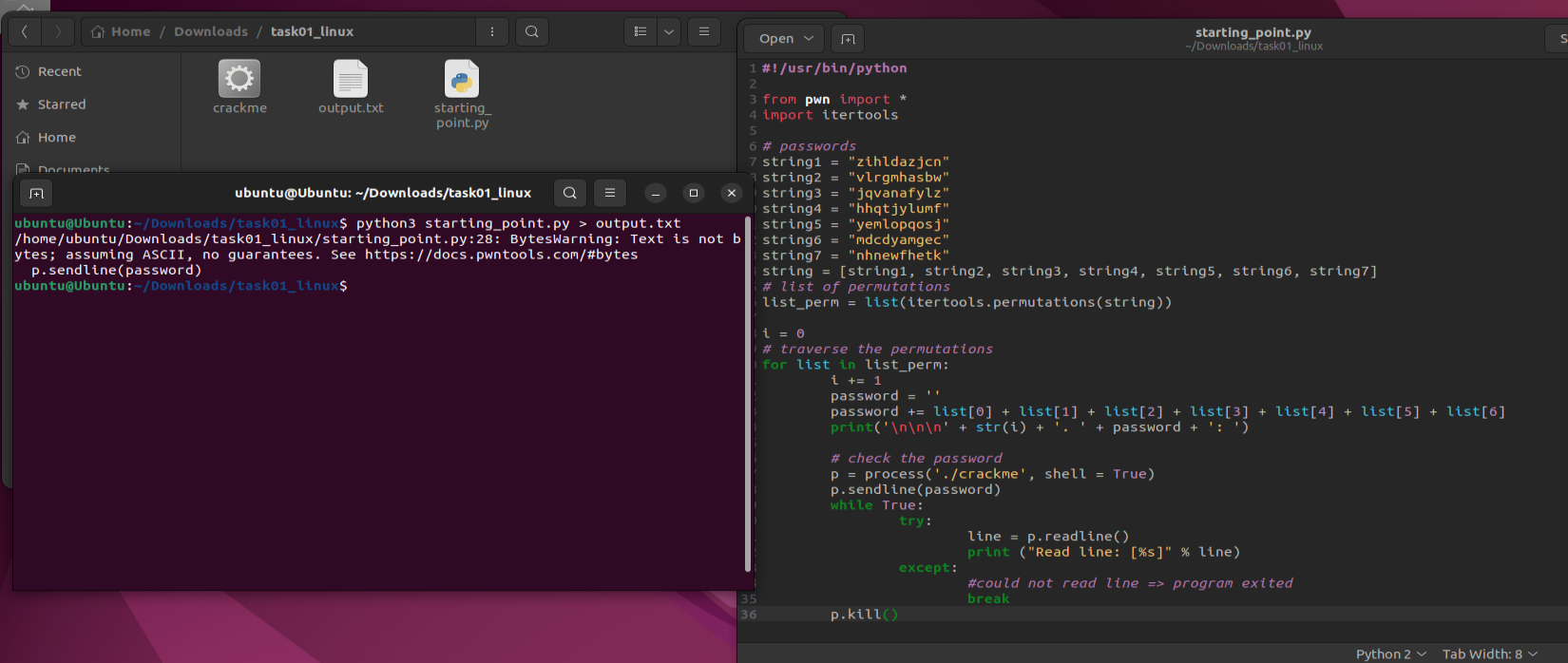
This code is in ***task4.py***.

* find the correct password. (2p)

If we try to insert the password as the 7 substrings in the order we found them, this is what we get:

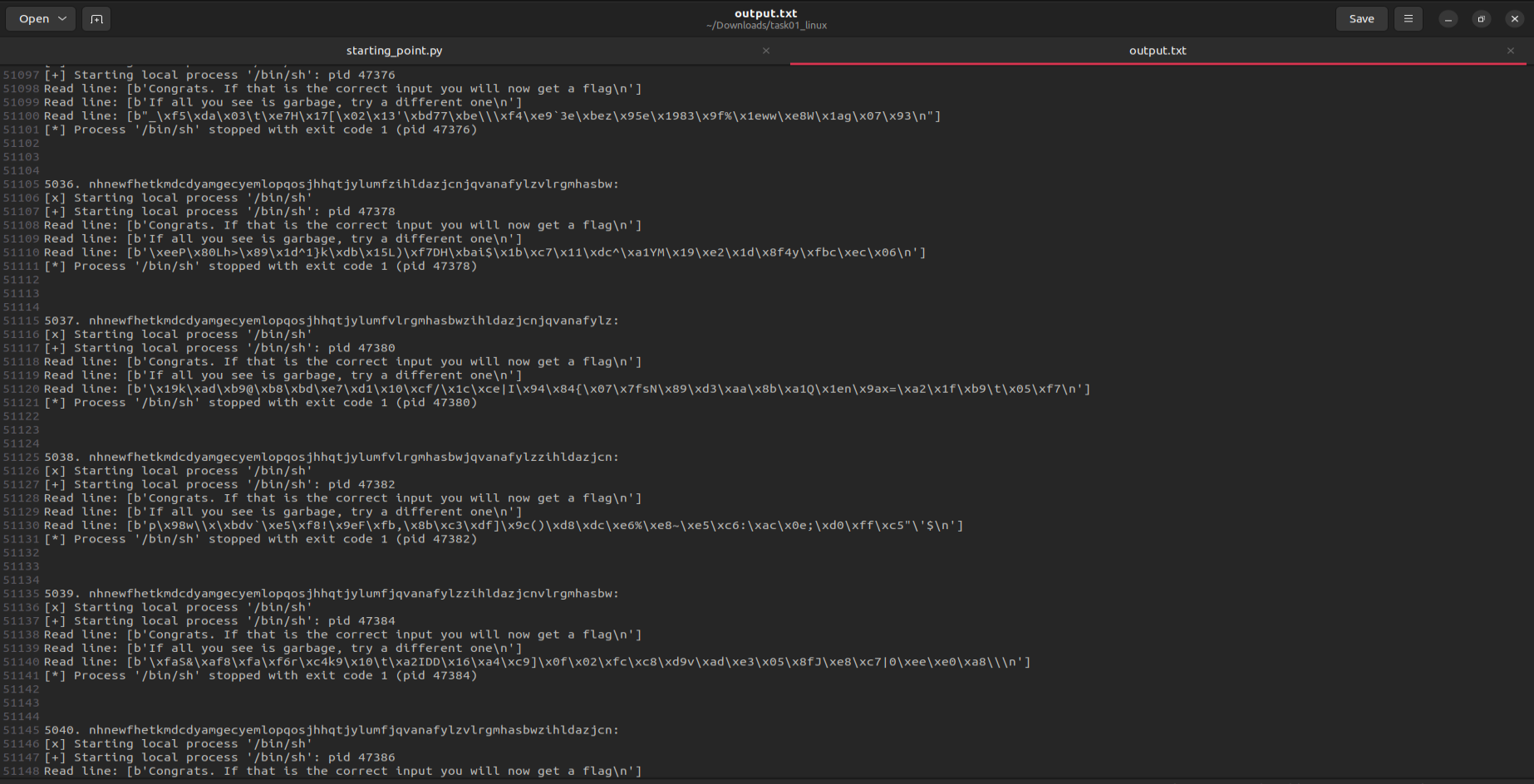


We will generate all the permutations from the 7 substrings in order to get the correct password:

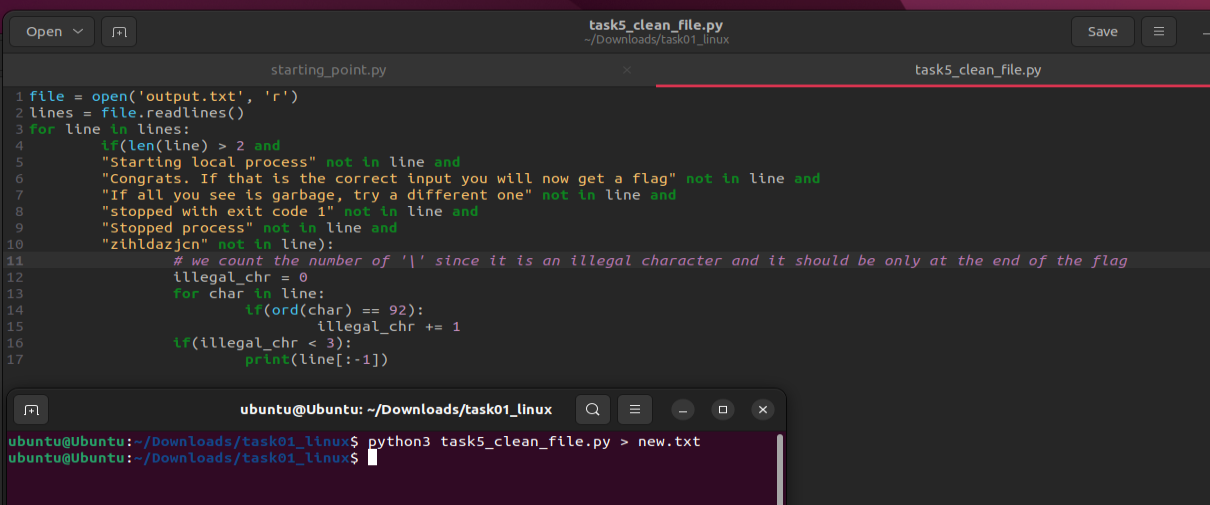


The code is in ***task5.py***.

This is a segemnt of the output (***output.txt***):



We clean the file in order to find the correct password. The code is in ***task5\_clean\_file.py*** (we ignore all the lines that cannot be potential flags and for the remaining ones, we count how many „*illegal*” characters are in the line – illegal as in ‚\’ that has ASCII code = 92; if we have more than 3 illegal characters it can’t be our flag because that means it contain strange characters).

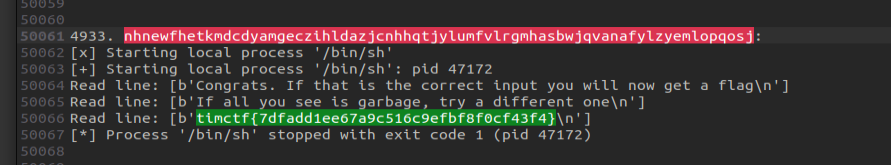


And the result:



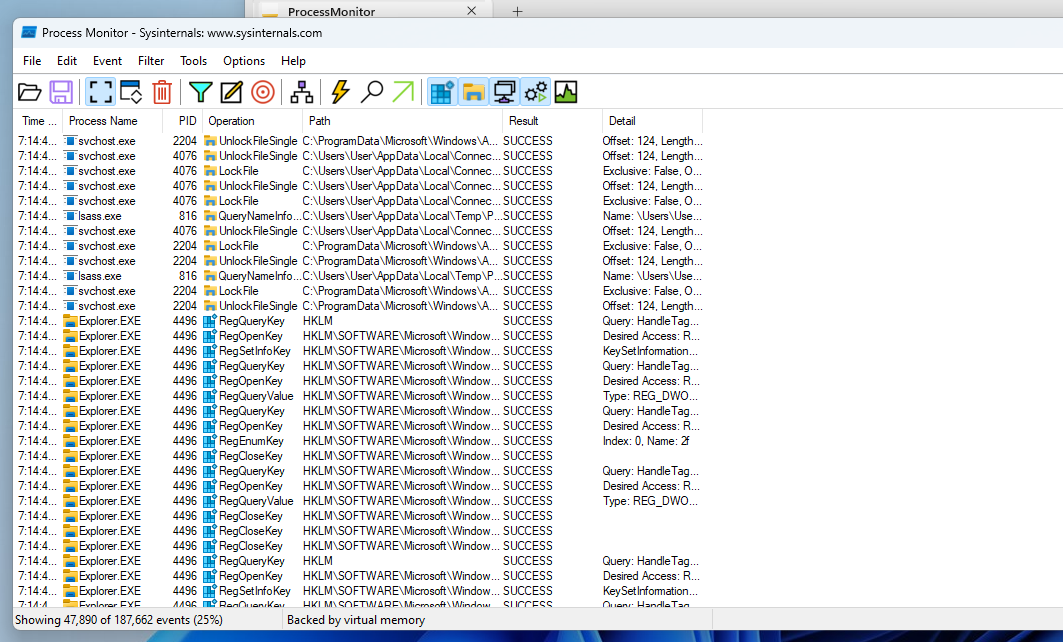
We notice an interesting string, that is our flag: **timctf{7dfadd1ee67a9c516c9efbf8f0cf43f4}**

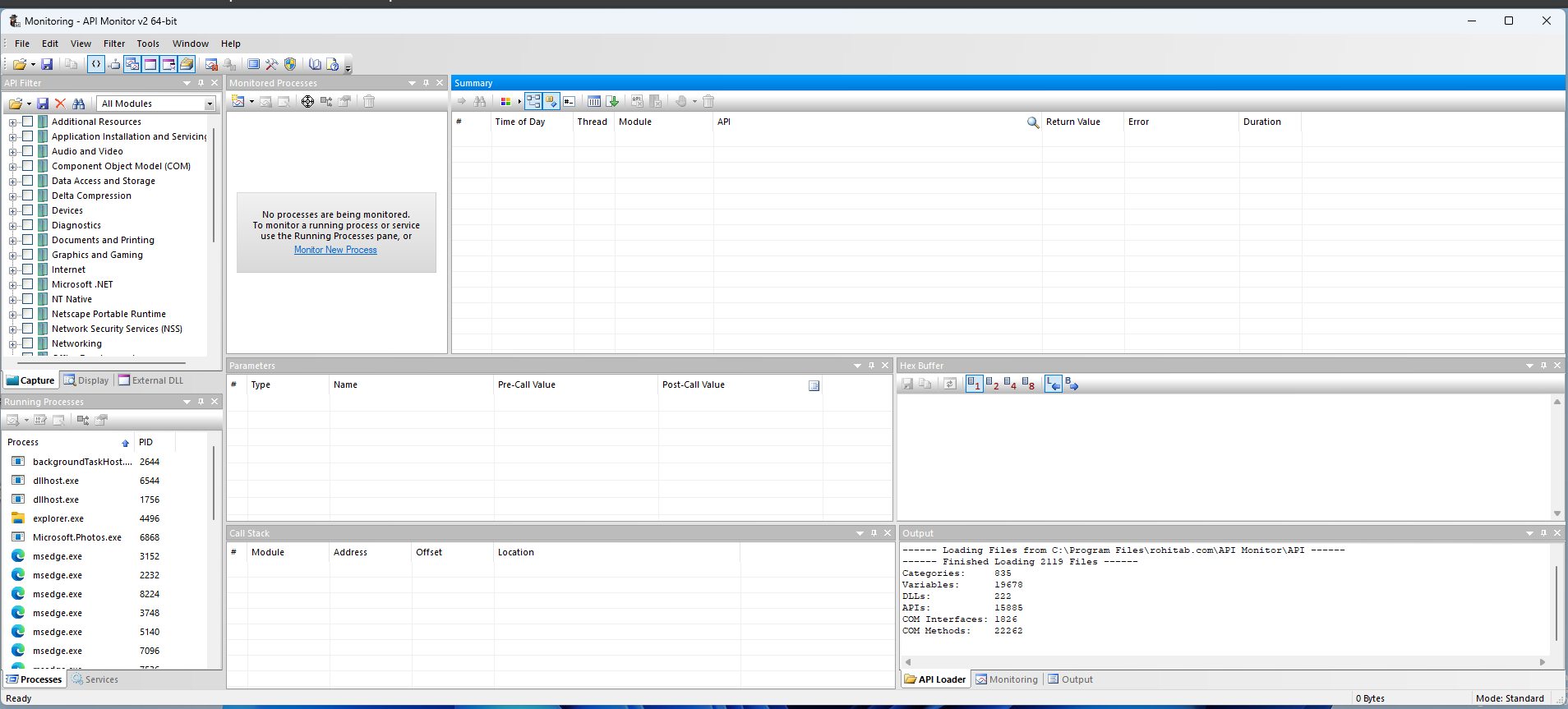
If we go in the original file and search for this flag, we get the correct password: **nhnewfhetkmdcdyamgeczihldazjcnhhqtjylumfvlrgmhasbwjqvanafylzyemlopqosj**



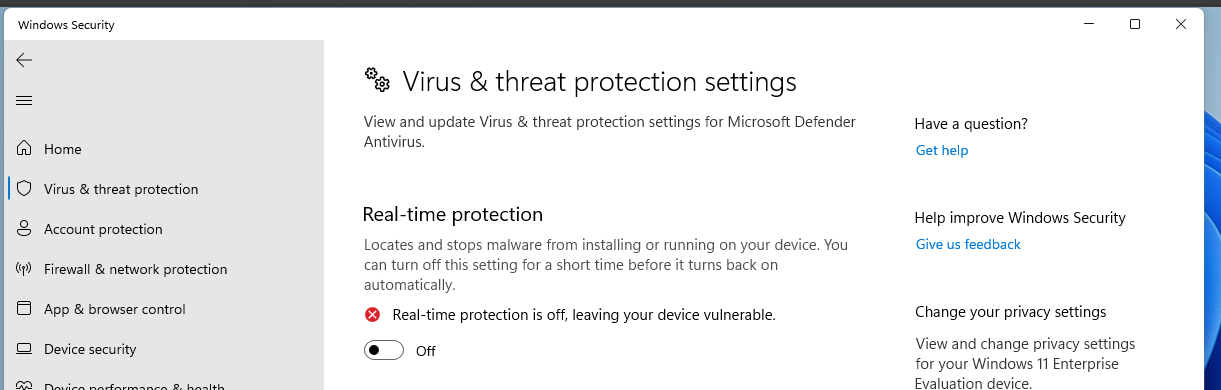
Task 2: for Windows

We first install the programs that we will use: [Process Monitor](https://docs.microsoft.com/en-us/sysinternals/downloads/procmon) & [API Monitor](http://www.rohitab.com/apimonitor).

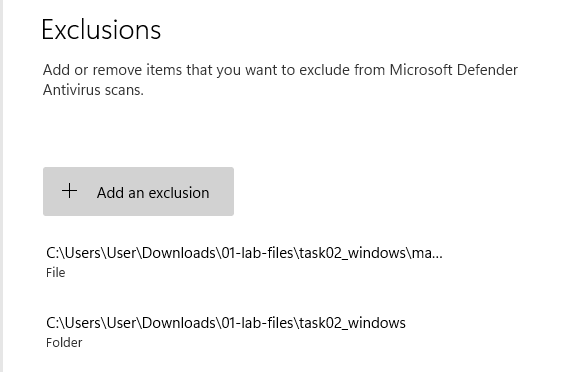




For starters, we close real-time protection for Windows...



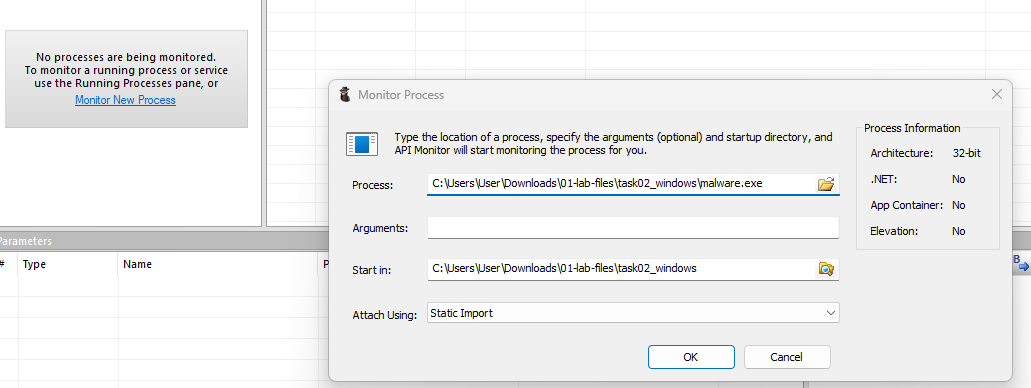
and we add to exclusions (one for the folder where we have the malware and one for malware.exe) – in order to ensure that the malware won’t be deleted by the Windows antivirus.



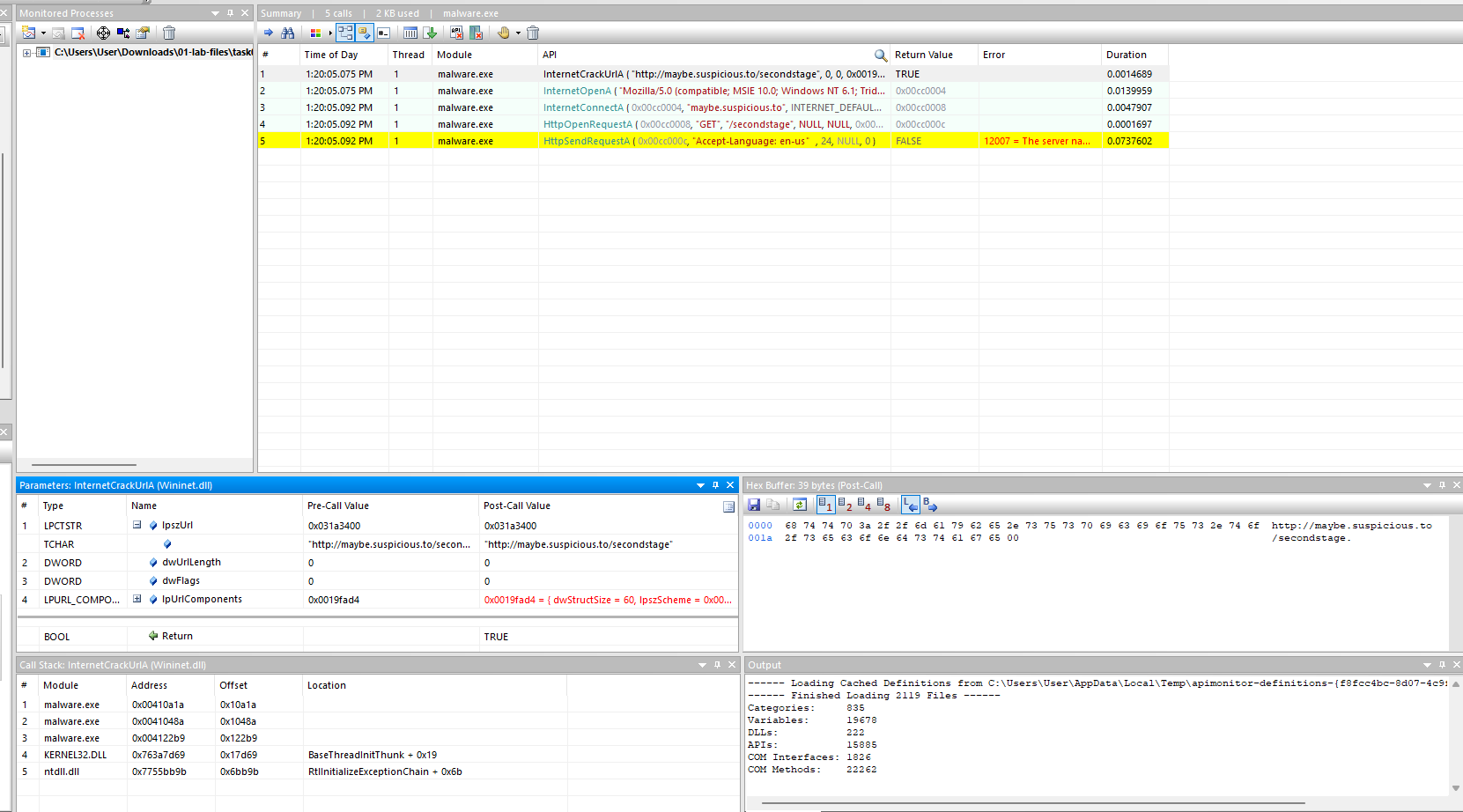
Now we can proceed with the tasks.

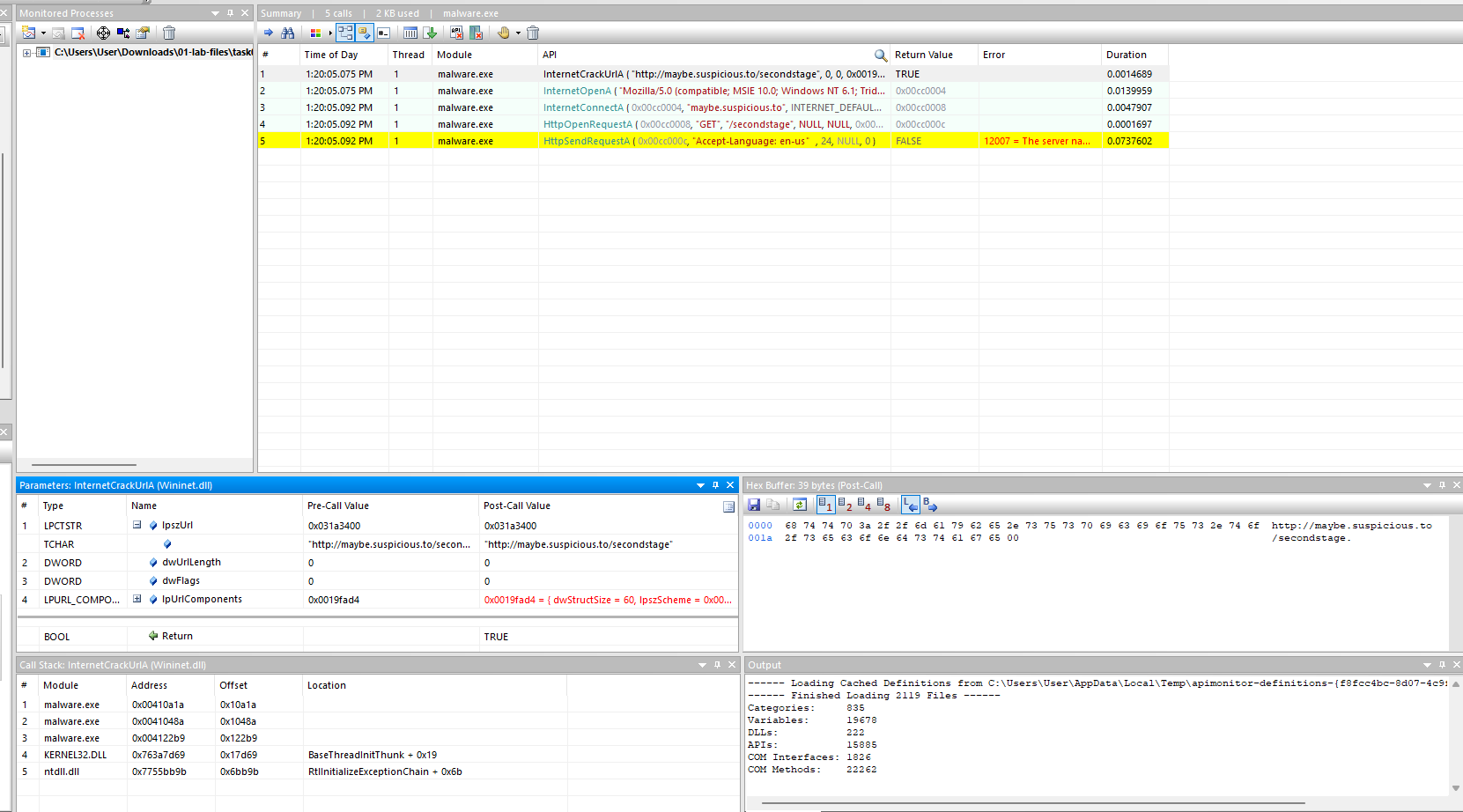
* where does it connect to? (2p)

First, we connect to **API Monitor** for 32-bit (the malware is 32-bit; otherwise we get an error) and from the left panel, we select ***Internet***.



This is what we get:

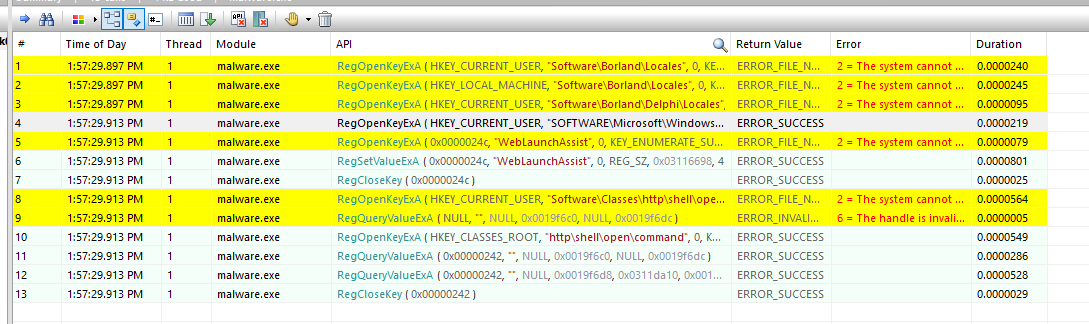


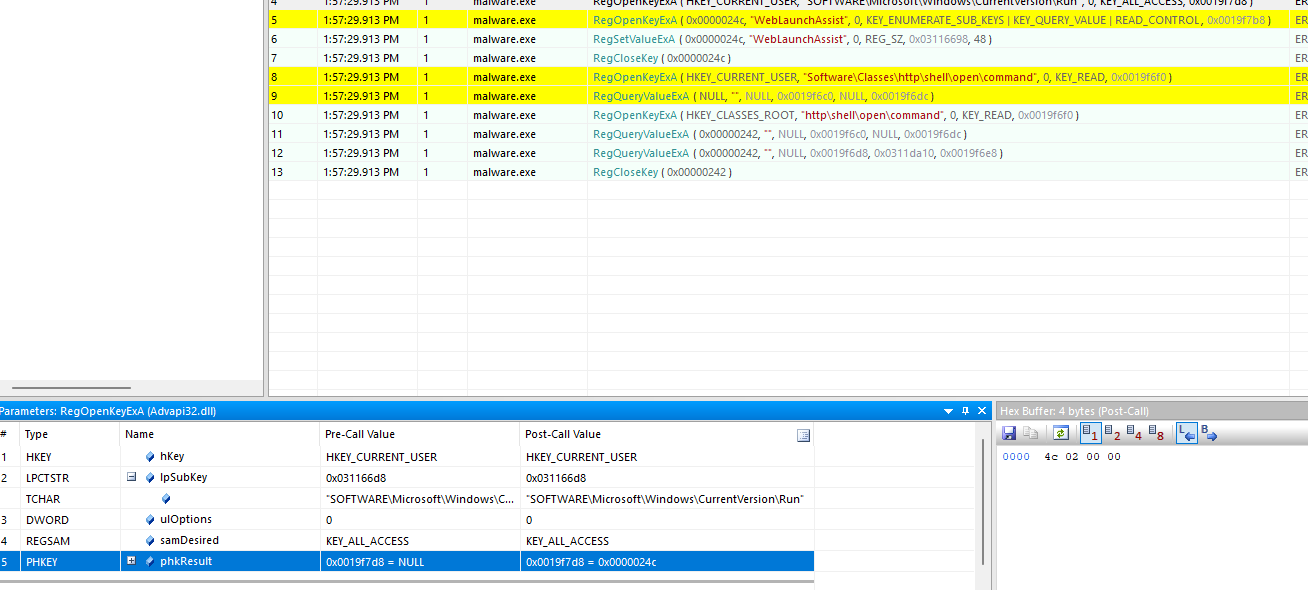


As we get observe, the malware is making a connection to "***http://maybe.suspicious.to***" URL and is trying to **GET** "***/secondstage***".

* what registry keys does it access and why? (2p)

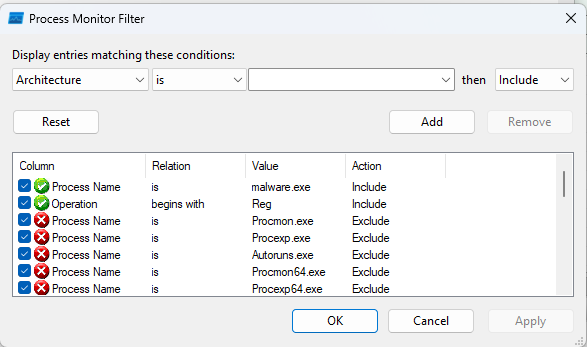
If we choose ***System Services/Windows System Information/Registry*** instead of ***Internet***, we get this:



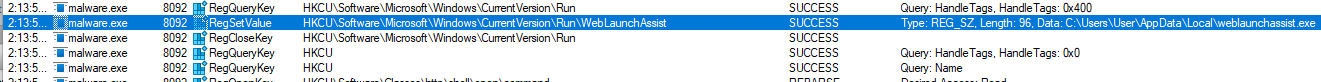


These are the keys being accessed:

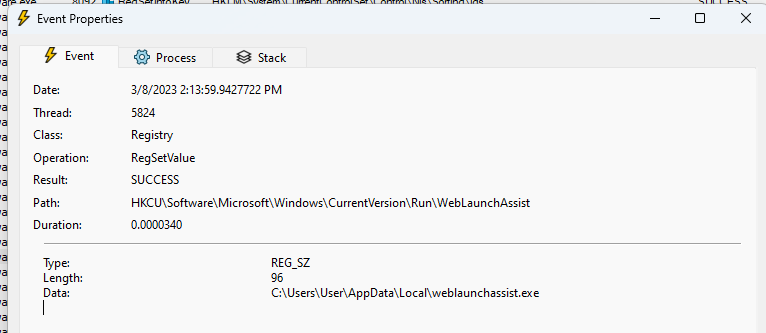
* HKEY\_CURRENT\_USER:
  + *1."Software\Borland\Locales"*
  + *2."Software\Borland\Delphi\Locales"*
  + **3."SOFTWARE\Microsoft\Windows\CurrentVersion\Run"**
  + **4."Software\Microsoft\Windows\CurrentVersion\Run\WebLaunchAssist"** (by looking at the Keys)
  + **5."Software\Microsoft\Windows\CurrentVersion\Run\WebLaunchAssist"**
  + 6."Software\Classes\http\shell\open\command"
* HKEY\_LOCAL\_MACHINE:
  + *7."Software\Borland\Locales"*
* HKEY\_CURRENT\_USER:
  + 8."Software\Classes\http\shell\open\command"
* ***1, 2 and 7*** → Malware.exe is binary compiled, so when the application starts up, it automatically makes some checks of the locale of the system ([source](https://www.delphipower.xyz/guide_5/using_resource_dlls.html)).
* ***3, 4 and 5*** → „Use Run or RunOnce registry keys to make a program run when a user logs on. The Run key makes the program run every time the user logs on” ([source](https://learn.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys)). Using ProcMon with the following configuration...



... we can notice:



The value of WebLaunchAssist is set to ***C:\Users\User\AppData\Local\weblaunchassist.exe*** (a file created by the malware previously).





* ***6 and 8*** → The malware attempts to change the default browser to Internet Explorer ([source](https://www.dostips.com/forum/viewtopic.php?t=8210)).

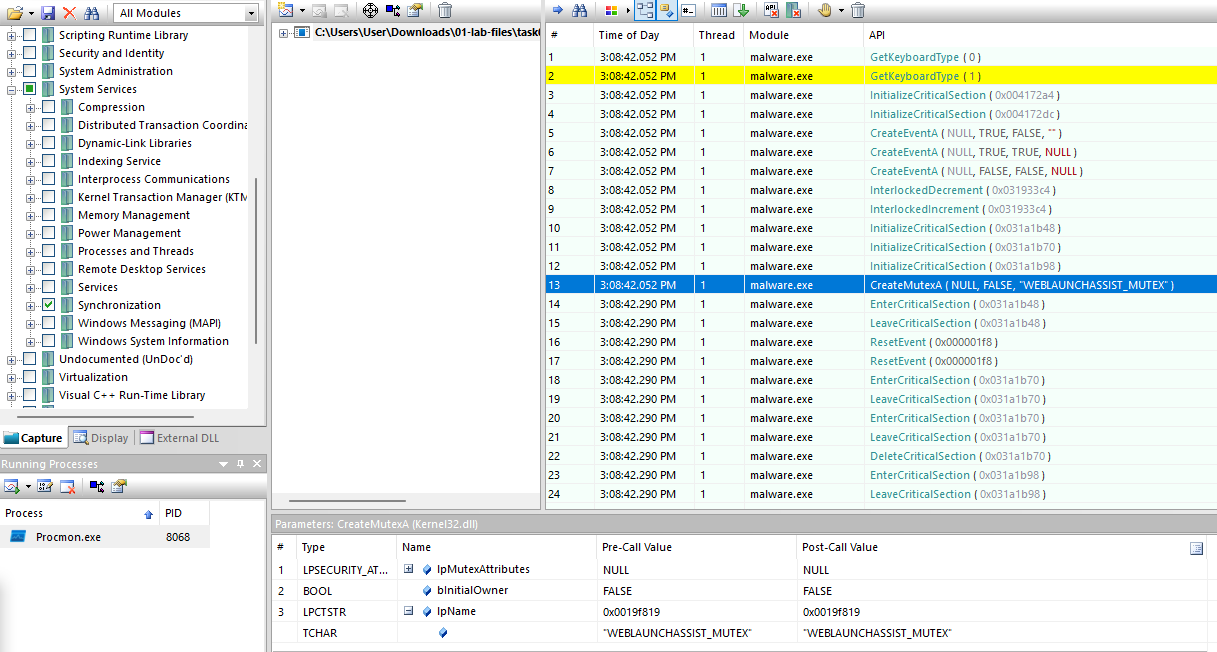


On the topic of registry keys, using ***ProcMon*** we can notice more keys than what we initially found using ***API Monitor***, but they seem to gather information about the system.

Task 3: Bonus Task

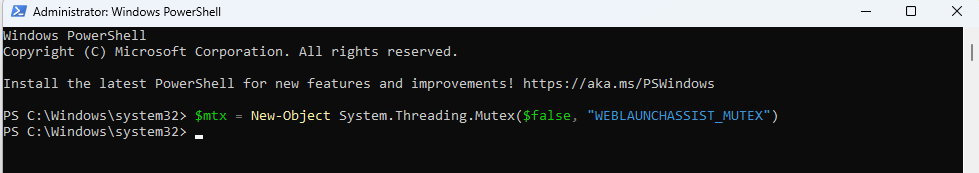
* To get the bonus points, figure out how this malware uses synchronization to avoid reinfection and then devise a way to “vaccinate” machines against this malware. (4p) ([source](https://www.sans.org/blog/looking-at-mutex-objects-for-malware-discovery-and-indicators-of-compromise/))([source](https://learn-powershell.net/2014/09/30/using-mutexes-to-write-data-to-the-same-logfile-across-processes-with-powershell/))

Using ***API Monitor***, we filter the results with ***System Services/Synchronization*** and run *malware.exe*:

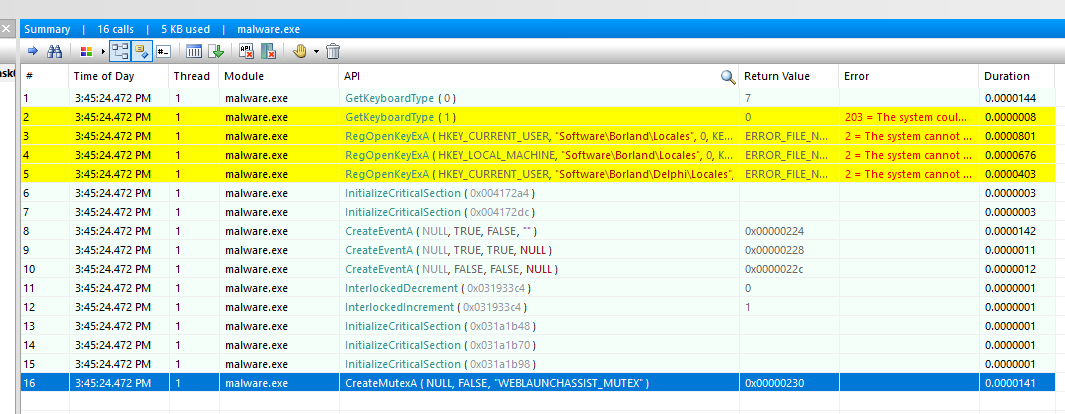


We can notice that the malware creates a mutex "WEBLAUNCHASSIST\_MUTEX".

We restore the VM to a state before we run the malware. We create a new mutex (with administrator rights) with the same name (***$mtx = New-Object System.Threading.Mutex($false, "WEBLAUNCHASSIST\_MUTEX")***):



If we run API Monitor again (filtering on for ***Internet***, ***System Services/Windows System Information/Registry*** and ***System Services/Synchronization***):



And with ***ProcMon***:



It is apparent that the process quickly exits after attempting to create a mutex that already exists, so that means that the vaccine is working.