Lebanese American University

School of Arts and Sciences

Department of Computer Science and Mathematics

CSC435 (Computer Security)

Lab #01 (Malware Analysis)

A Report Done By

Ahmad Hussein ahmad.hussein03@lau.edu

Mahmoud Joumaa mahmoud.joumaa@lau.edu

And presented to Dr. Ayman Tajeddine

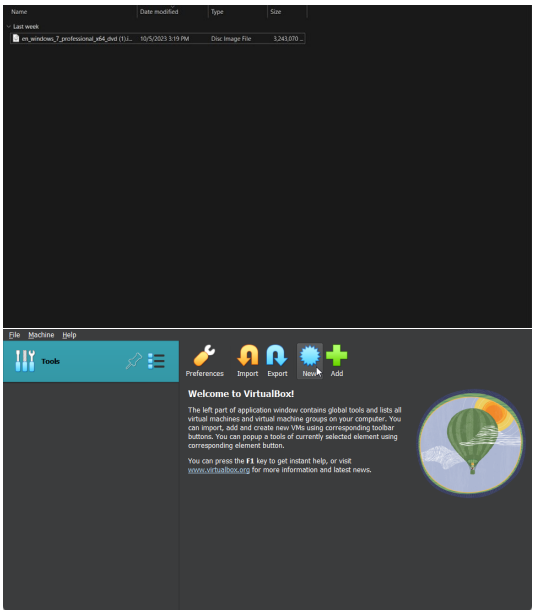
October 2023

Table of Contents

I. Prelab

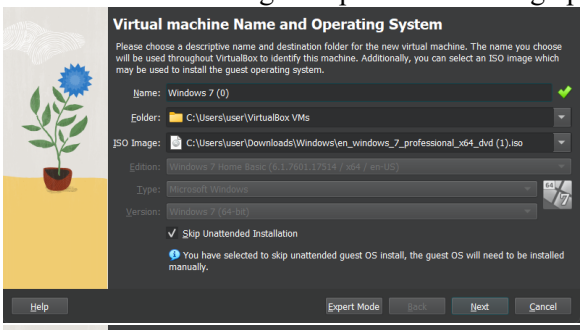
Setting Up the Virtual Machine

The first step of setting up the environment would be to choose the operating system that would run on the virtual machine. For this lab, we opted for *Windows 7*.

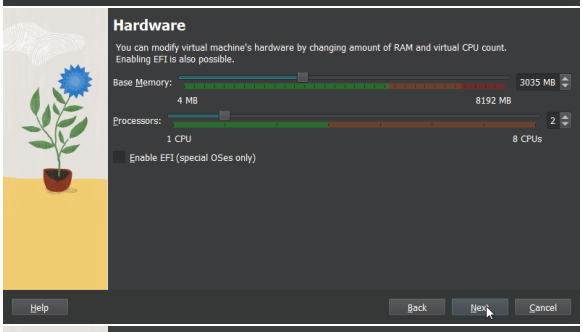


Once the *Windows 7 iso* file is downloaded, we created a new virtual machine inside of virtual box.

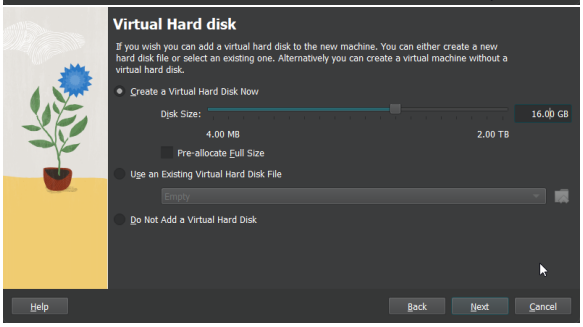
We then went through the process of setting up the virtual machine following virtual box pop ups.



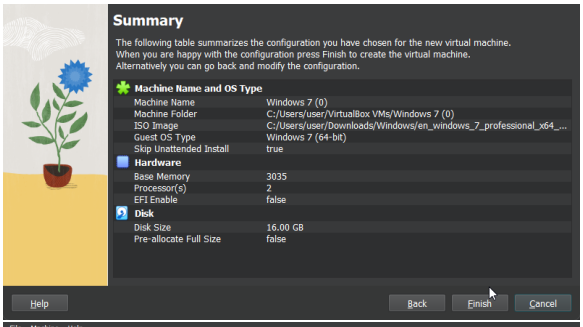
First, we chose the *Windows 7 iso* file.



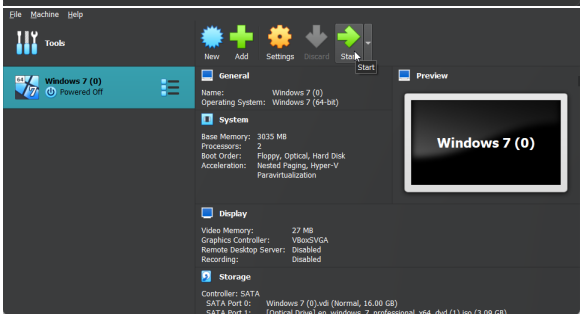
Next, we allocated enough RAM and CPU for the virtual machine.



Then, we provided the virtual machine with sufficient disk space.

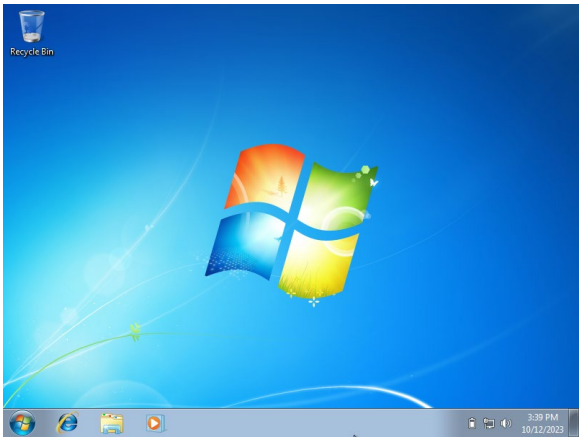


Finally, we finalized the setting up our *Windows 7* virtual machine.

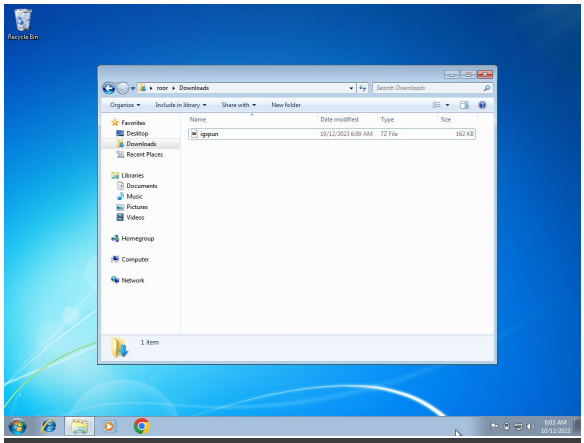


Our virtual machine could now be started.

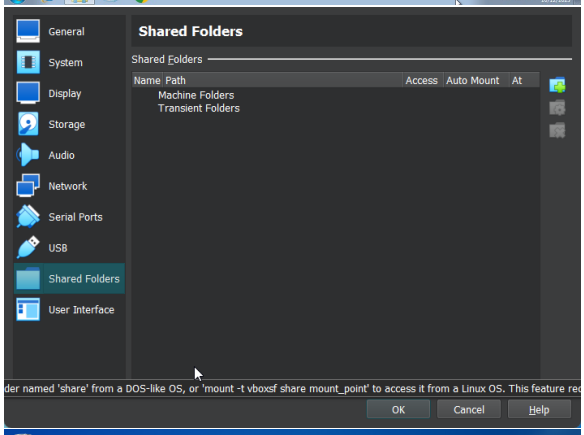
We, then, finalized setting up *Windows 7* on the machine.



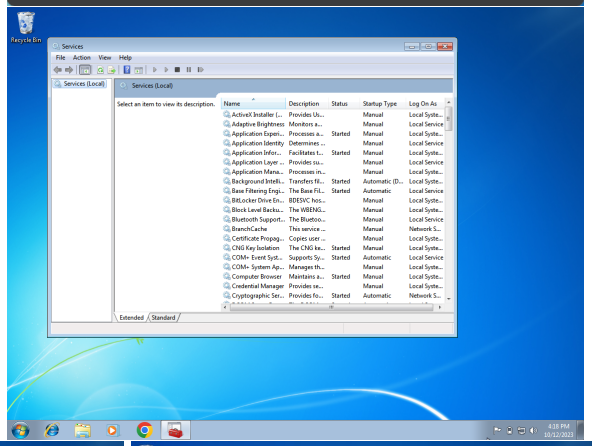
We then resumed to download the virus file and 7-zip to unzip the virus file when needed.



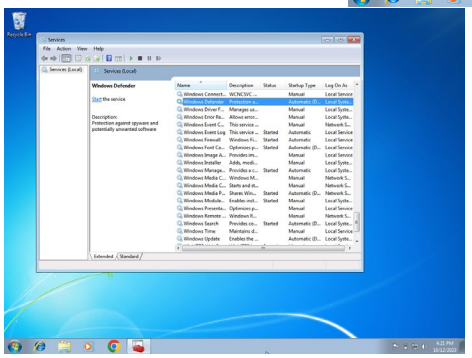
We made sure there are no shared folders.

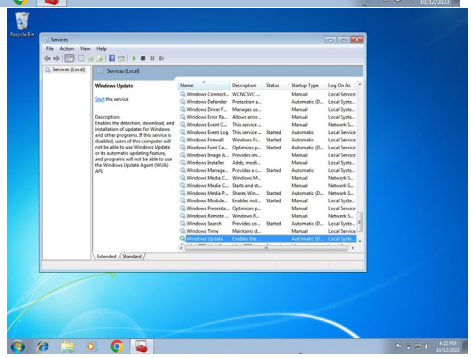


We opened *servies.msc*

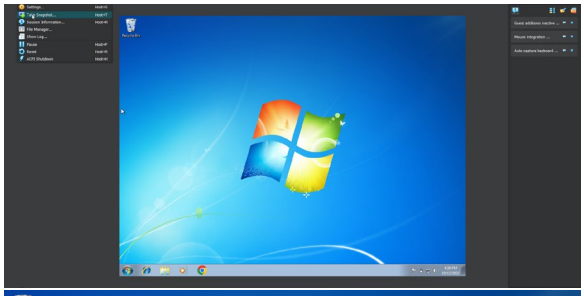


We disabled *windows update* and *windows defender*.

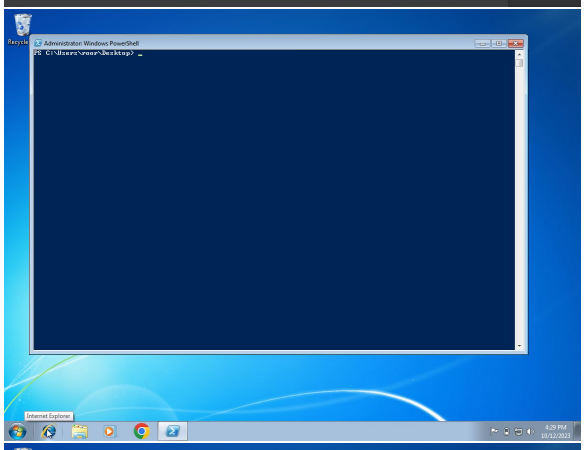




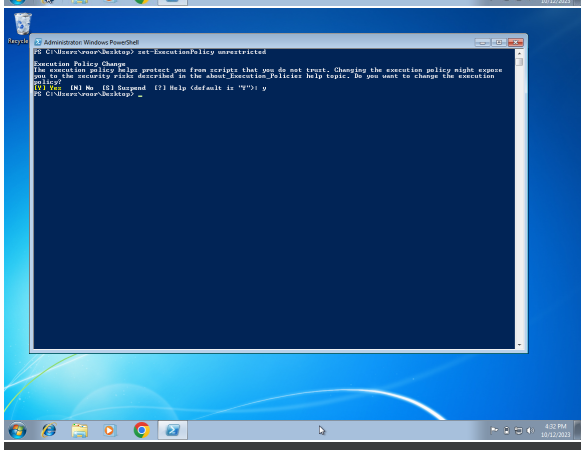
We saved a snapshot of our current environment.



We opened *powershell* and navigated to the desktop path.

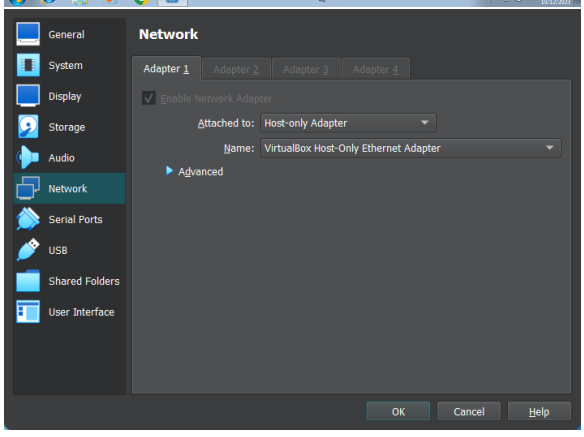


We ran the command *Set-ExecutionPolicy unrestricted*.



We changed the network adapter to *Host-only* adapter to disconnect the virtual machine from the internet.

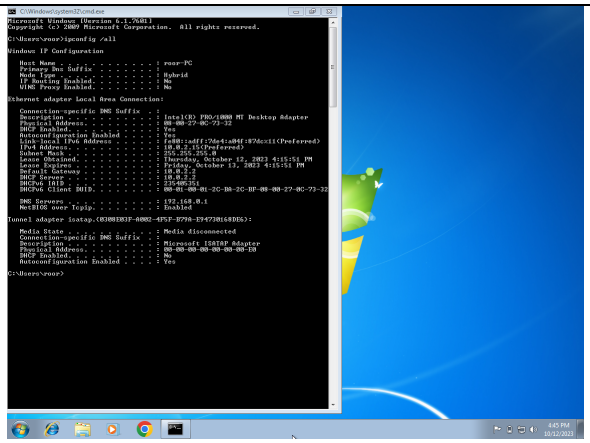
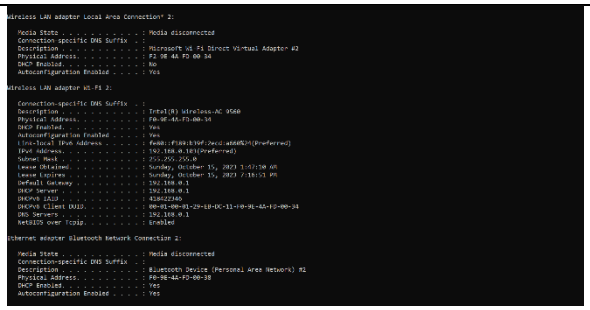
(Note that all the software used for this lab was downloaded from the internet BEFORE changing the network adapter back to *Host-only*. Details of the software used are found in following sections of this report. A '*' icon next to a reported step indicates that it was completed before changing the adapter back to *Host-only*.)



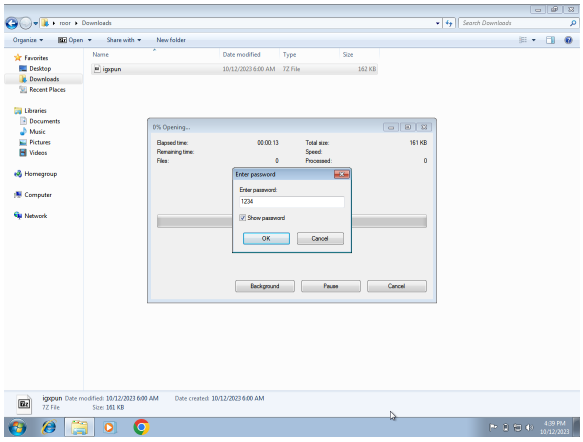
No difficulties were faced when setting up the environment. The pre-lab was straightforward for the most part.

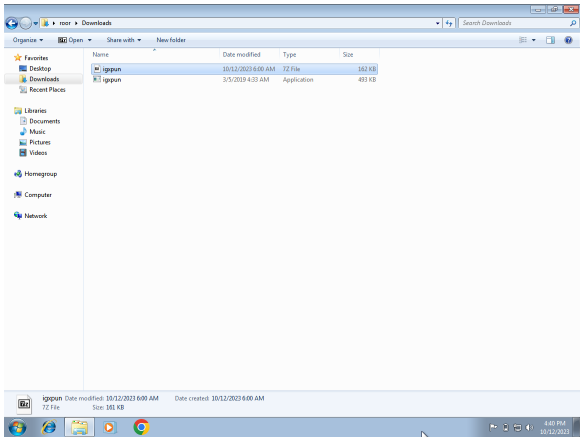
II. Static Analysis

Before proceeding any further with the report, we made sure to check both of our host machine's and virtual machine's MAC (physical) and IP (IP4) addresses using the *ipconfig/all* command. The results are charted in the following table:

	Virtual Machine MAC Address: 08-00-27-0C-73-32 IP Address: 10.0.2.15
	Host Machine MAC Address: F0-9E-4A-FD-00-354 IP Address: 192.168.0.103

1. Downloading the Malware





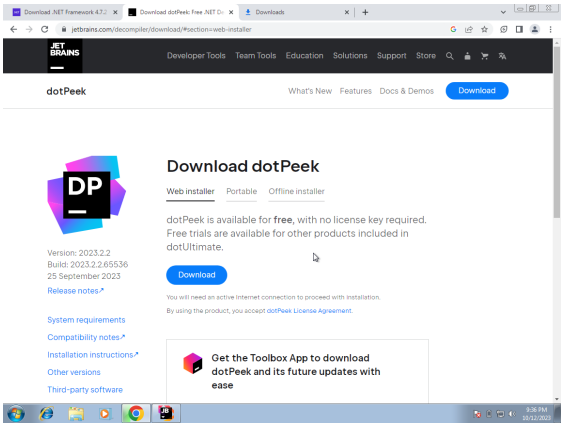
* We downloaded the compressed *igxpun.exe* from blackboard on our virtual machine using *Google Chrome* as *Internet Explorer* does not support *Javascript* which rendered us unable to log in to the LAU portal. We then decompressed the zipped file using the password *1234*.

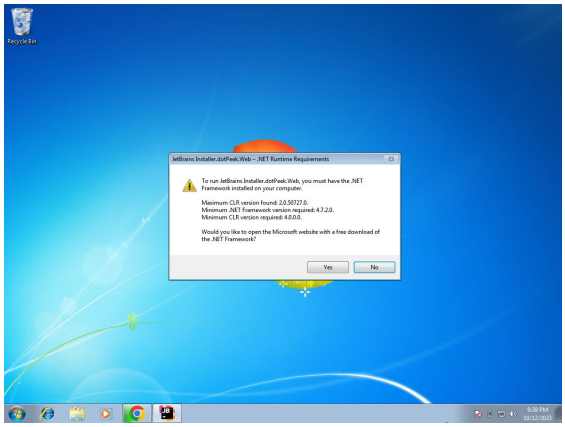
2. Choosing a Decompiler

a. A decompiler is a programming tool that is used in reverse engineering applications to 'reverse' the operations of a compiler. It typically transforms the executables (low-level) for machines into source code (high-level) for programmers.

b. Both DNSpy and Dotpeek are decompilers that perform the same basic function. They decompile the lower-level code into intermediate-level or high-level code. However, we opted for Dotpeek for the main reason of having the open-source DNSpy project archived and deprecated. Dotpeek is still maintained and updated while DNSpy no longer receives any updates.

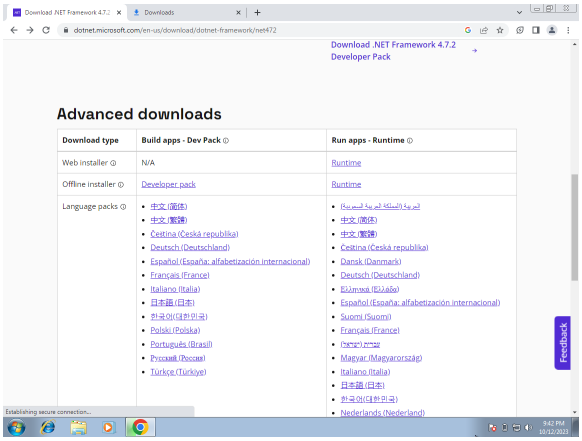
* Following are screenshots illustrating the installation process of *Dotpeek*.



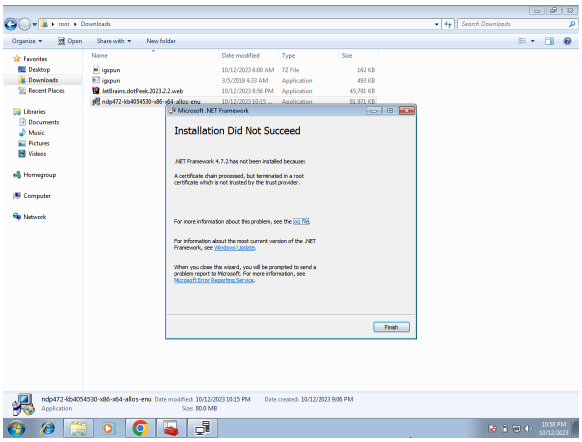


* Trying to install *Dotpeek* pops up the above error, so we followed the instructions provided by Elie Hanna to fix it.

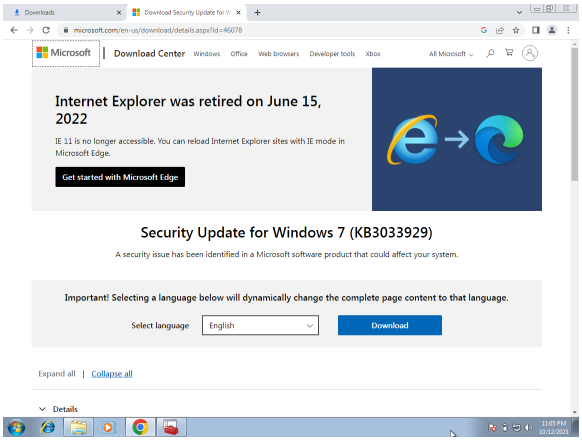
We first downloaded the offline installer for .NET version 4.7.2



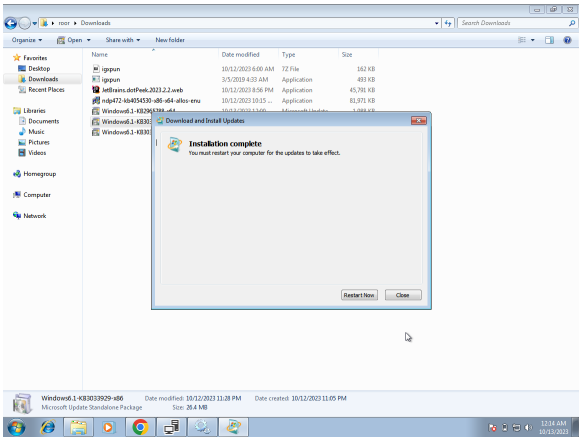
After the successful of .NET, however, an "*Installation Did Not Succeed*" error message pops up when we tried to install *Dotpeek*.



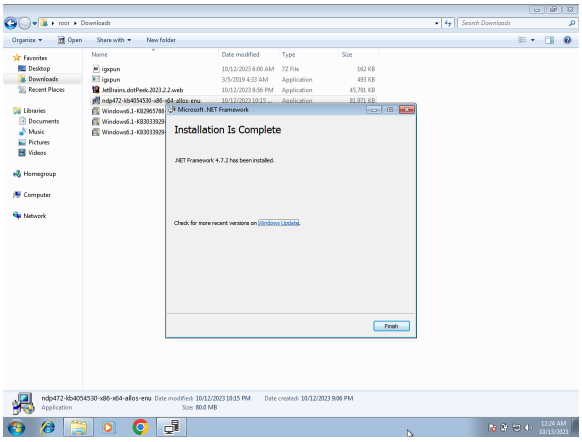
To fix this error, we downloaded the *KB3033929 Windows Security Update*. (We had to restart Windows Installer, Windows Module, and Windows Update services to run the security update).



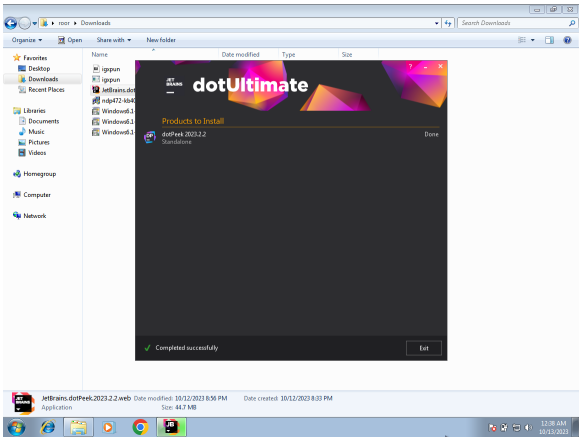
However, when the update downloaded, we tried installing it but the error the "*Update is not applicable on your computer*" popped up, so we found a 64-bit version of the update instead of the 32-bit version suggested in the instructions, and that resulted in a successful installation.



We restarted our virtual machine to allow the update to complete before installing the offline .NET version 4.7.2



Once .NET was installed, we were ready to install *Dotpeek*.

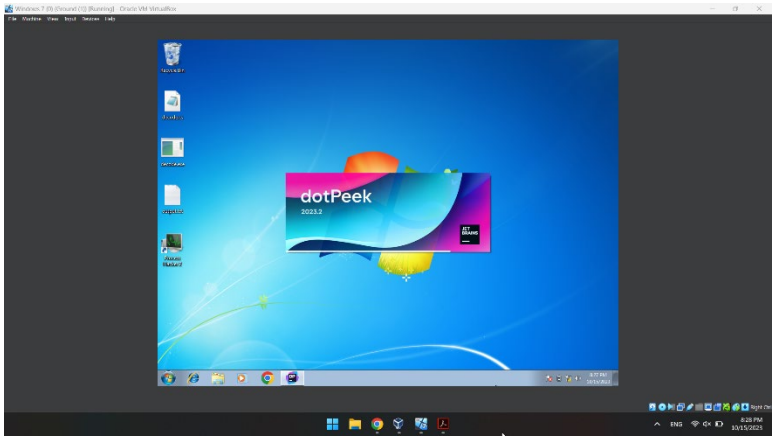


3. Resetting the Connection

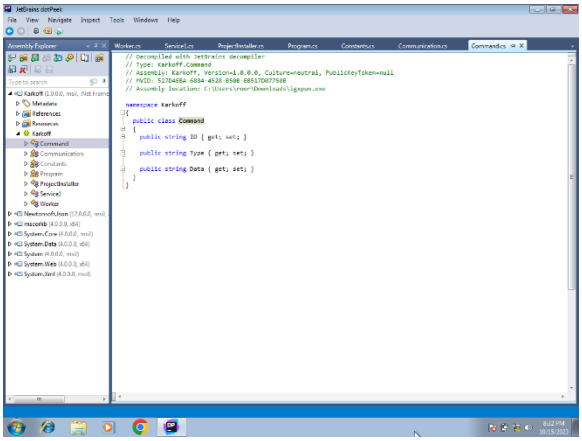
We then stopped the windows services *Windows Installer*, *Windows Module*, and *Windows Update* before making sure to switch the network back to *Host-only*.

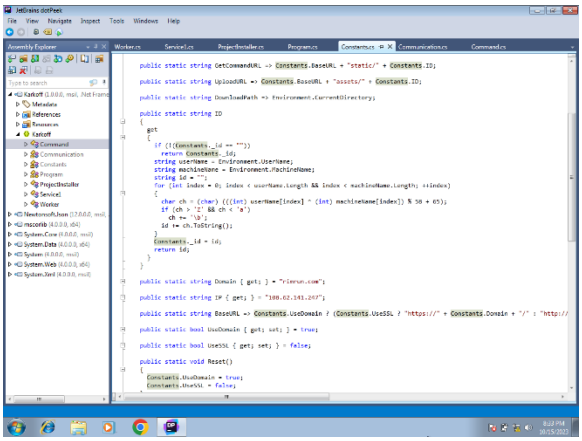
Next, we took a new snapshot of our virtual machine before starting the static analysis of the malware.

4. Decompiling the Malware



a. The code is written in C#. We could tell because the file extensions are '*.cs*' (as shown in the open tabs of the *Dotpeek* interface in the below screenshot). The code also utilizes the *.net* C# framework which further validates our hypothesis.

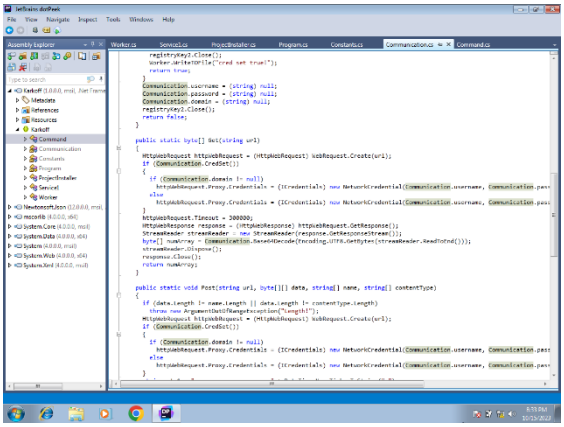


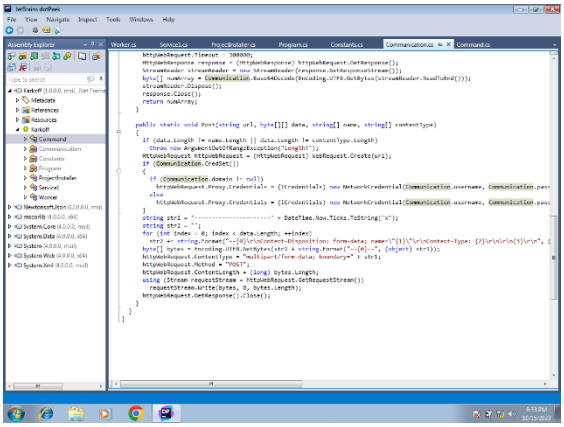
b. 

As visible in the above screenshot, the malware is communicating with domain (*rimrun.com*) and IP (*108.62.141.247*) which are hardcoded in *Constants*.

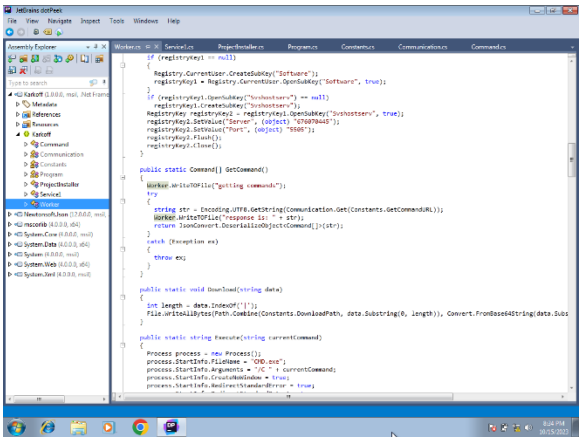
c. The malware communicates with the C&C via the two methods *GET()* and *POST()* that are defined in *Communication*.

Each of the two methods creates a web request (*http* or *https* protocol) and checks whether the credentials are set via the *credSet()* method. The response is then retrieved and saved in a decoded array. The *Get()* method is mainly used to request from the server while the *Post()* method is mainly used to send information to the server.

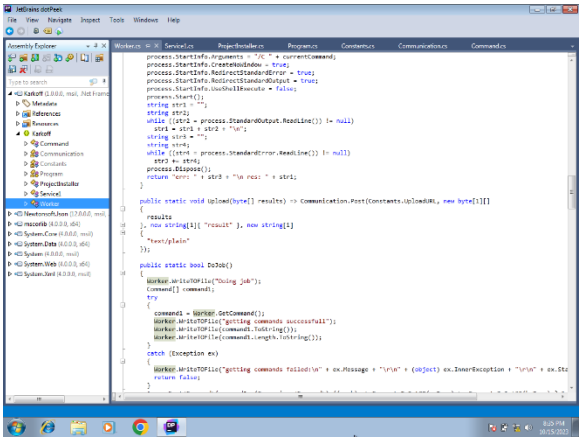




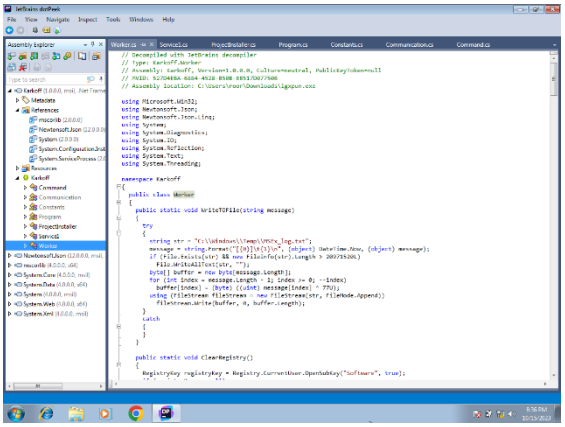
The *Get()* method is called in the *GetCommand()* method to fetch commands from the *commandURL* defined in *Constants*.

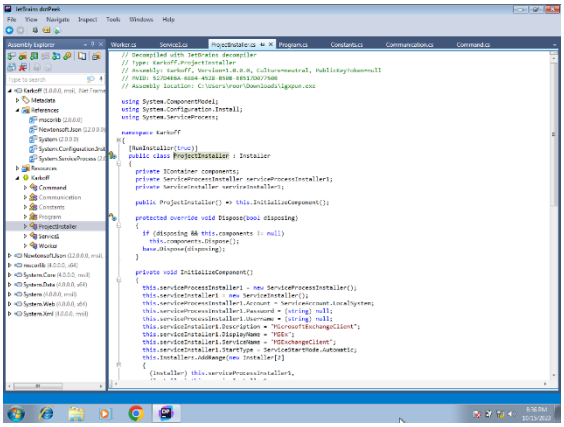


The *POST()* method is called in the *Upload()* method to send the results to the *UploadURL* defined in *Constants*.



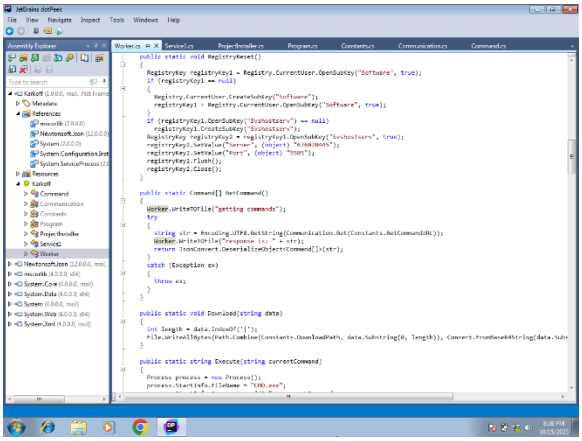
d. Yes, there are libraries embedded in the malware, including *Microsoft.Win32* (handles events raised by the operating system and manipulates system registry), *System.Threading* (enables multithreading), and *System.ServiceProcess* (controls windows services applications).



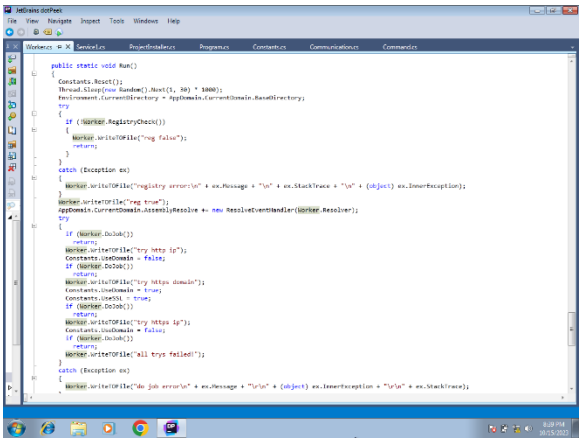


The malware communicates with the *commandURL* defined in *Constants* via the *GetCommand()* method to retrieve the commands. The *GetCommand*() calls the *Get(commandURL)* method to

retrieve the array of commands as formatted json strings, and returns those strings as deserialized (i.e. decoded) *.net command* objects.

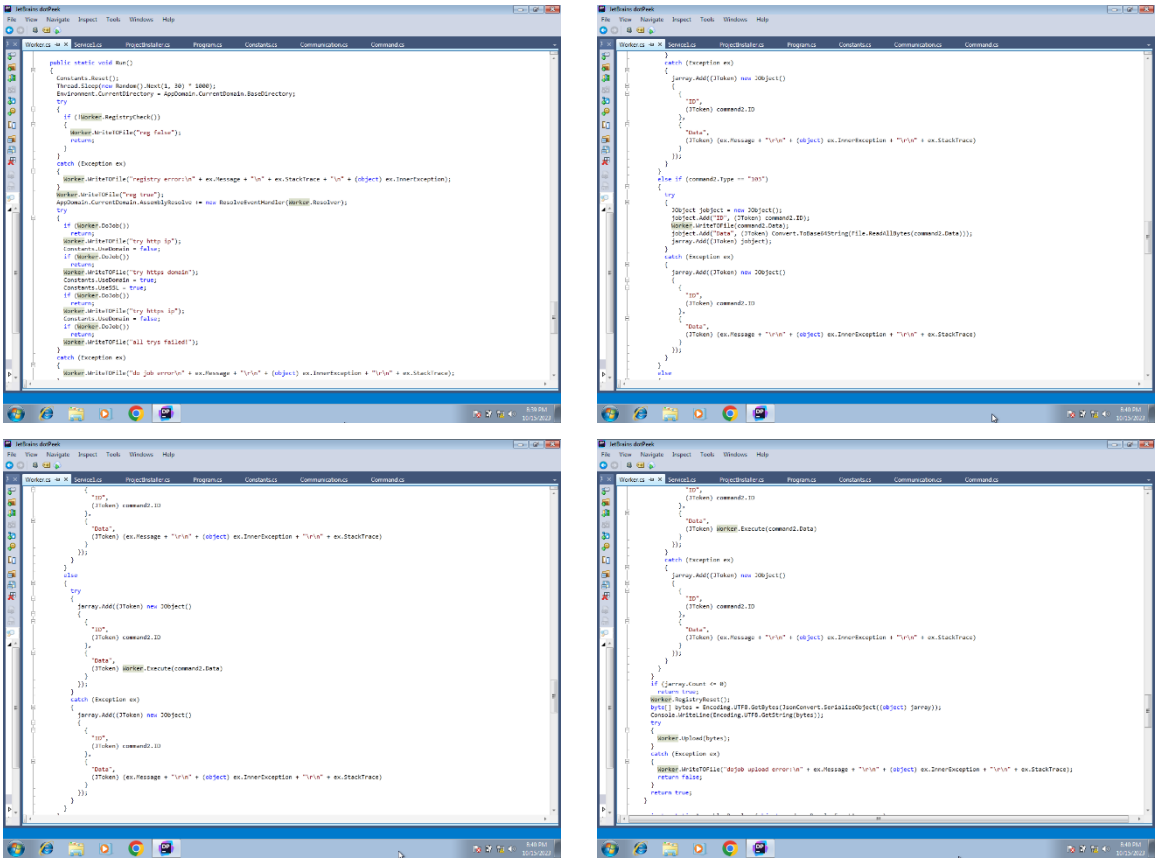


The commands are executed via the *DoJob()* method whenever it's called in *Run()* when a new thread is started.



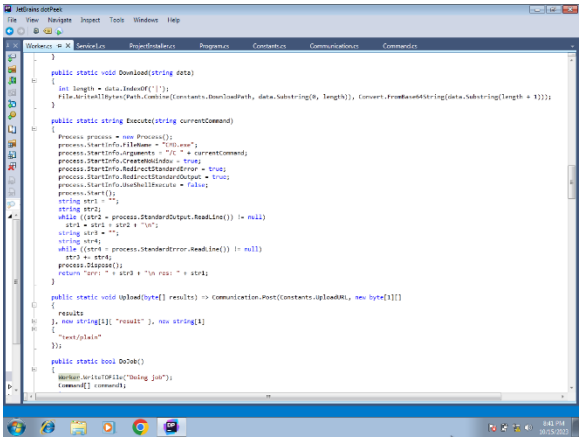
The *DoJob()* method first loops over all commands and inspects the type of the command at the current index, as shown in the below screenshots.

If the command's type is *101*, it calls *Download()* to download the command's data and adds a new *JToken* including the command's ID and data "*Done*" to the *JArray*. If the command's type is *103*, the malware adds a new *JObject* to the *JArray* with the command's ID and data converted to *Base64String*. Otherwise, a new *JToken* with the command's ID and *Execute()*'s return is added to the *JArray*.

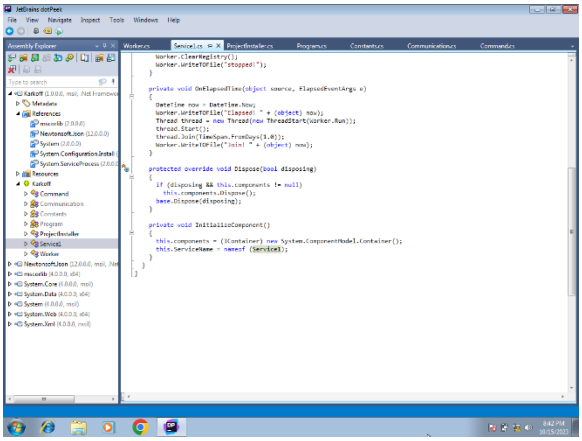


Upon calling the *Execute()* method, a new process starts and opens *cmd.exe* with the configurations as shown in the below screenshot. These configurations render the running process more difficult to detect by the user:

- *process.StartInfo.CreateNoWindow=true* (the process starts without a window to contain it)
- *process.StartInfo.RedirectStandardOutput=true* (output is not written to the StandardOutput stream)
- *process.StartInfo.RedirectStandardError=true* (errors are not written to the StandardError stream)
- *process.StartInfo.UseShellExecute=false* (the shell is not used but rather a process is created directly from the executable)



Once the process finishes running, the *Dispose()* method clears current components.

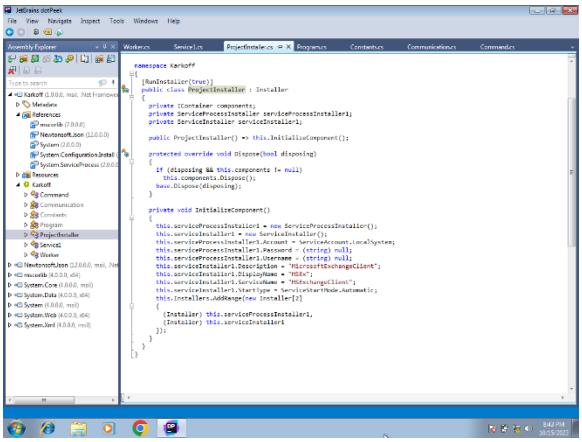


e. The malware utilizes the *InitializeComponents()* method defined in *ProjectInstaller* to install itself.

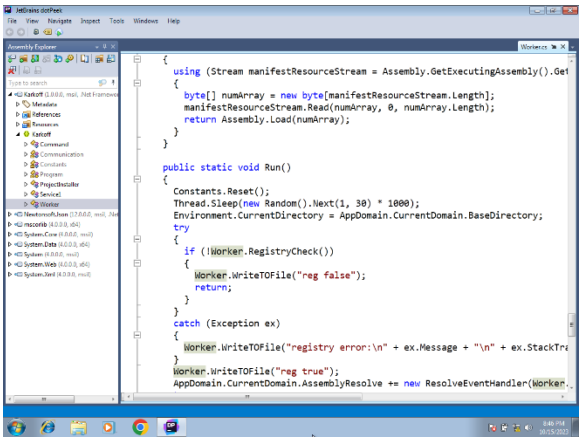
The *ServiceProcessInstaller* installs components common to all services during an installation.

The *ServiceInstaller* installs the service-specific functionality.

The *ServiceProcessInstaller* has its account set to *LocalSystem* which is a predefined local account used by the service control manager. This account is typically not recognized by security subsystems and has extensive administrative privileges, which allows for the malware to operate freely without being detected by typical security countermeasures.

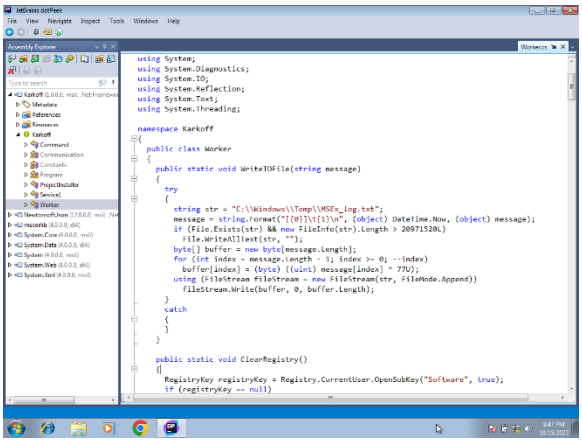


The *Run()* method also implements a random sleep timer (between 1 second and 30 seconds) for the created thread which may also help in hiding this malware as its execution times become less consistent.

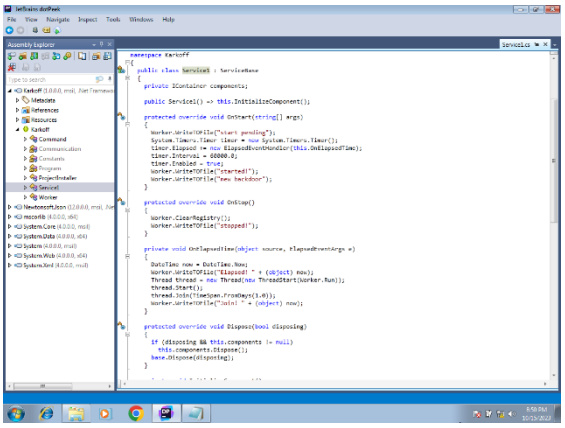


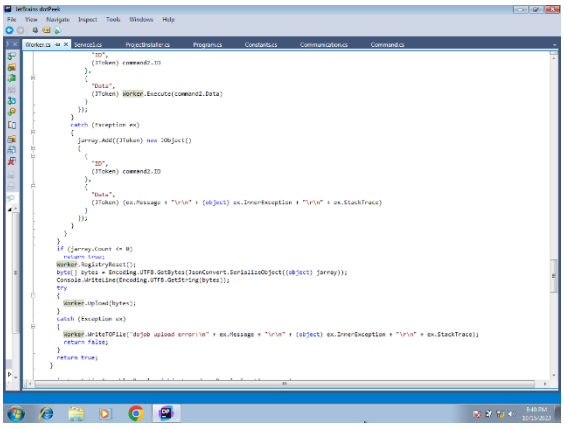
The name of the *Windows service* that the malware disguises itself as is "*MSExchangeClient*". This malware, thus, installs itself as a windows service but appears as MSExchange, deceiving the user upon shallow inspection.

f. The malware stores its activity locally by calling the *WriteToFile()* method defined in *Worker*. This method keeps track of the malware's activity and the corresponding dates of each activity in the *MSEx_log.txt* located in *C:\Windows\Temp* (shown in the next screenshot) as it is being called during the malware's different stages.



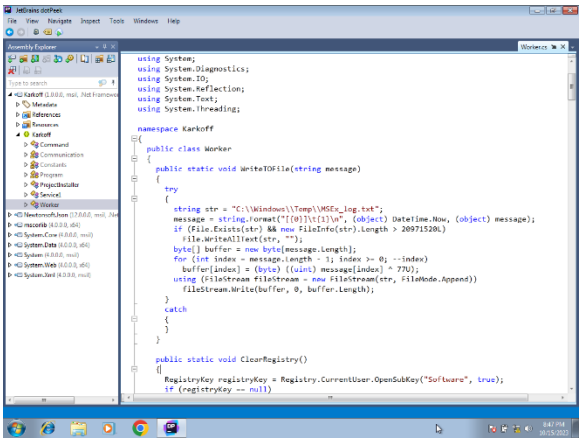
For example, the following screenshots show the *WriteToFile()* method being called in *OnStart()*, *OnStop()*, *OnElapsedTime()*, and *DoJob()* respectively.



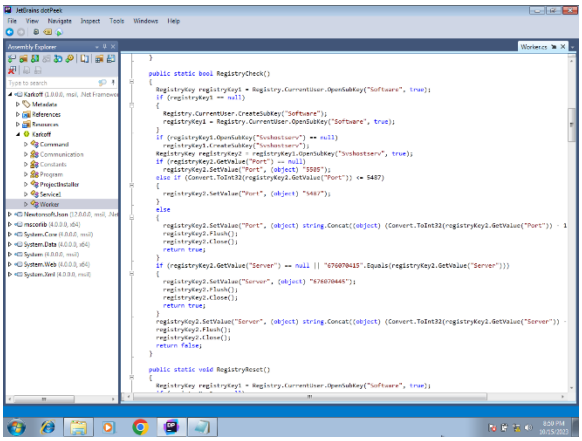


The malware could be storing its activity locally for several reasons. One of which is logging its activity even when the infected device is not connected to the internet. Another could be to avoid getting detected by security measures since it avoids provoking unusual network traffic.

g. Yes, the malware encrypts its log file by *XORing* the contents of the file with *77U* (acting as the encryption key) before writing to the file. (Note that *77U* is Unicode for the character '*M*'). The now encrypted content written in the file can then be decrypted by applying *XOR* again with the same '*M*' key. This encryption technique is thus symmetric as it uses the same key (*M*) to encrypt and decrypt (property of the *XOR cipher*).

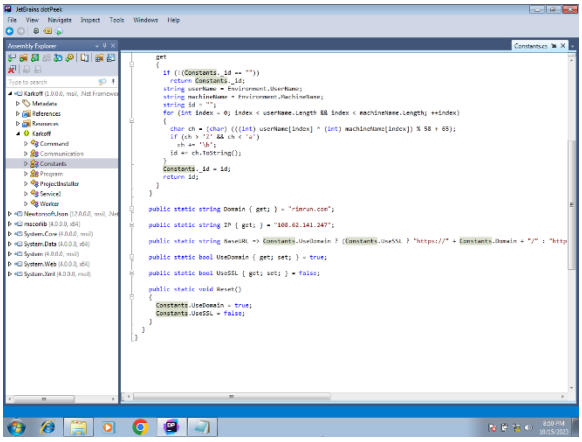


h. The *RegistryCheck()* method first attempts to open the *Software* subkey from the *Current User* registry. If it does not exist (i.e. *registrykey1 == null*), it creates it then opens it. Similarly, it then attempts to open the *Svshostserv* registry subkey and save the result in *registrykey2*.



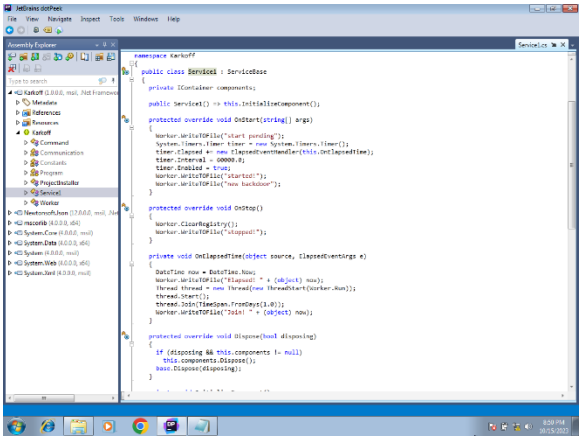
The method then tries to access the value of the port of the *registrykey2*. If the value is null, it creates a port with value of *5505*. If the value of the port is less than or equal to *5487*, it sets it to *5487*. Otherwise, it sets the port value to 1 less than the current port value (i.e. *value = value - 1*). It then flushes and closes *registrykey2* to make sure that the registry configurations are saved. Note that the method directly returns *true* in the last case (i.e. when setting *value = value - 1*). The method checks the server of *registrykey2* otherwise. If it does not exist (i.e. *server == null*), the method sets it to *676070415* then flushes and closes the registry before returning *true*. Otherwise, the method sets the server to 1 less than its current value. (i.e. *server = server - 1*). The method then flushes and closes the registry before returning *false*.

The return value of the *RegistryCheck()* function is used in the *Run()* method that is executed whenever a new thread is started. If it's *false*, the thread won't execute the *DoJob()* method. Otherwise, the thread will attempt to execute the *DoJob()* using different combinations of *UseDomain* and *UseSSL* booleans that are defined in *Constants*.

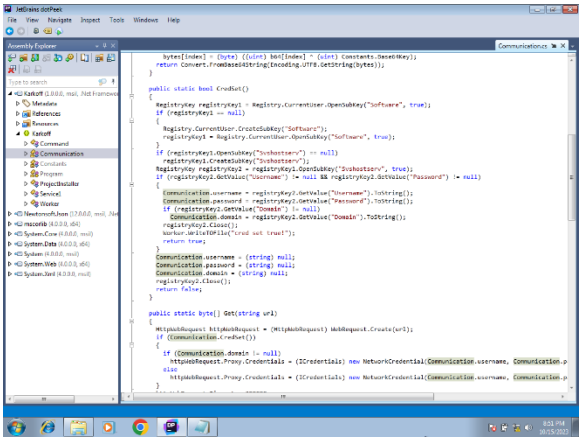


These changes in *Port* and *Server* may serve as a way to introduce some randomness to the malware's infection as this avoids a spike of unusual network traffic.

i. The timer in the *OnStart()* method serves the purpose of a time bomb. Its duration is *60000 ms = 60 seconds = 1 minute* as defined in *timer.Interval* (shown in the screenshot below). This timer triggers the *ElapsedTime()* method that creates a worker thread which waits for the previous join (for a maximum of 1 day or when the previous thread is terminated) before executing the *DoJob()* method. This is triggered every 1 minute.



j. The *Svshostsrv* registry subkey serves as a container that holds all the credentials needed by the *credSet()* function to evaluate its boolean return, which then impacts whether the started thread will reach the *DoJob()* method called in *Run()*.



5. Malware Kind(s)

The malware is a dropper since it acts as a trojan that installs itself on an infected machine. It is also a virus since it replicates various threads that run when the executable starts running triggered by the encoded time bomb. As for its type based on its payload, it can vary widely depending on the commands it receives from the server. If the command is to encrypt certain files, then it's ransomware. If the command is to access the user's camera, then it's spyware. If the command is to mine cryptocurrency in the background, then it's a cryptominer. Basically, it can be anything the attacker wants it to be as long as it is receiving the correct command.

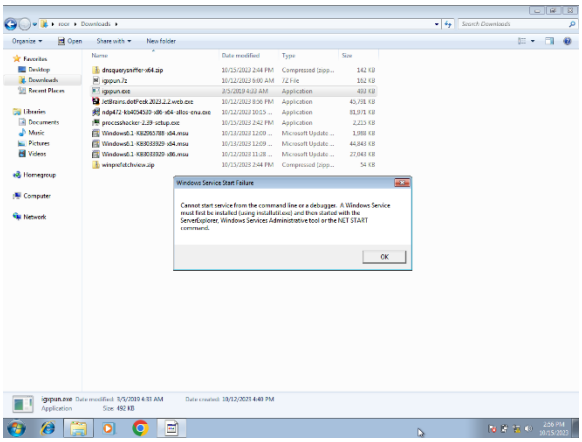
6. Malware Briefing

Karkoff was first discovered April 2019. It is .NET based malware believed to have been created by the same group behind DNSpionage (first discovered in late 2018 by Cisco Talos) for use as a remote execution tool during their attacks. These attacks targeted Lebanon as well as the UAE using various methods, including macros in Microsoft Word and Excel files or redirecting .gov websites to the malicious server.

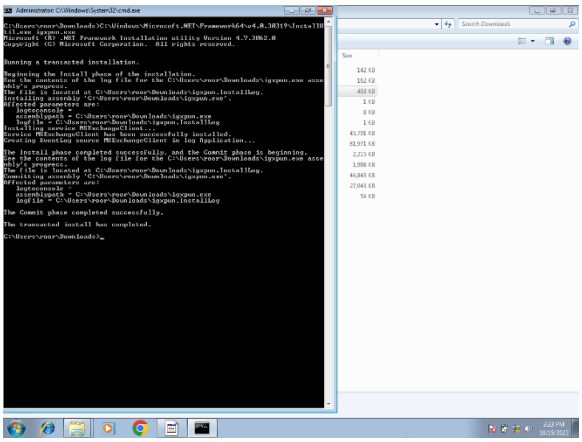
III. Dynamic Analysis

1. Running the Malware

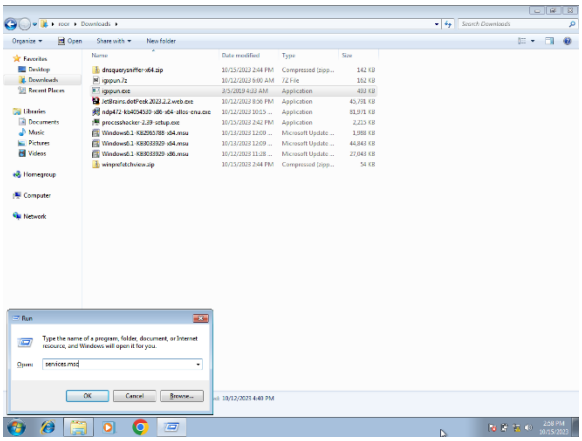
The *igxpun* file is a '*.exe*' file, but when we tried running it by double clicking, this error popped up.

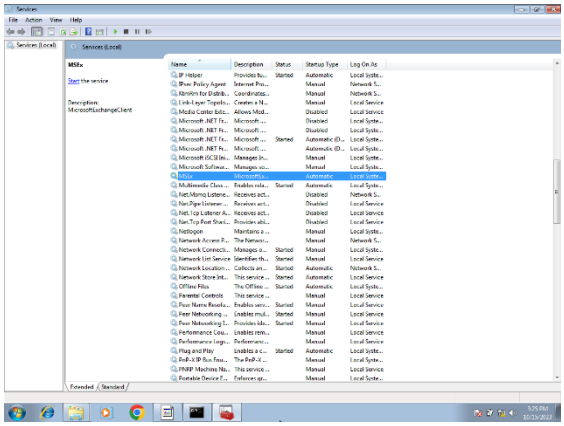


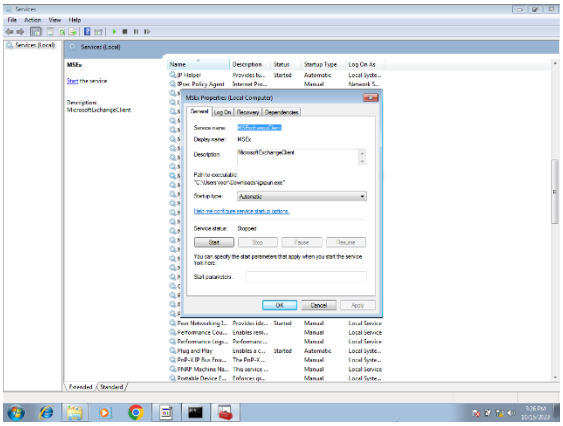
That is why is opted to run the service using the *InstallUtil.exe* command via the command prompt terminal as follows:



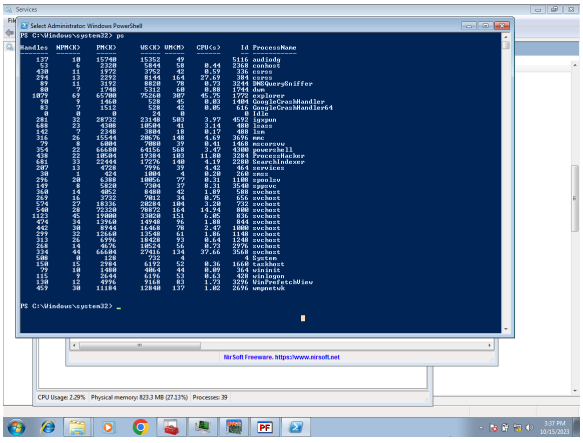
We were then able to confirm that the service has been installed by navigating to the *services.msc* using and searching for *MSEx.*

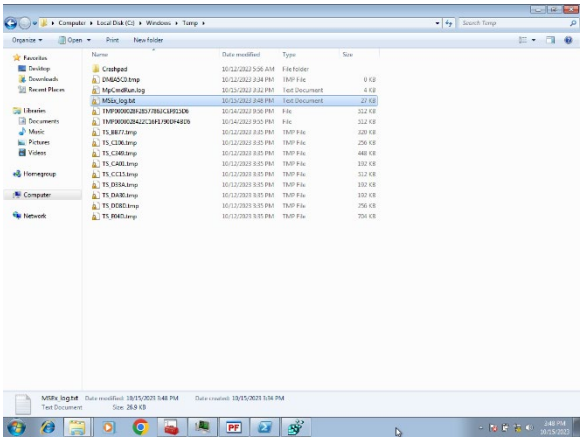






We were also able to pinpoint the service when listing all running processes using the *ps* command in *powershell* as well as the malware's corresponding *MSEx_log.txt file in C:\Windows\Temp*.





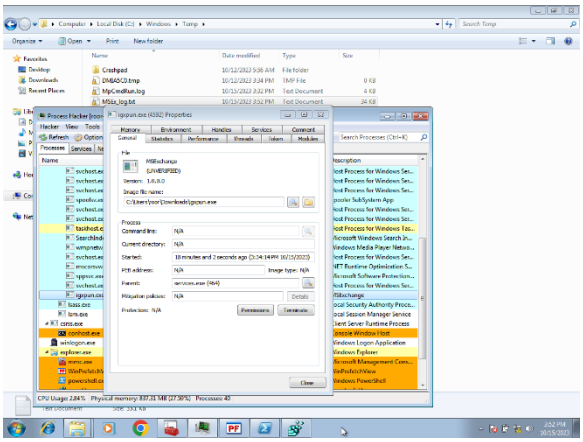
2. Malware Behavior

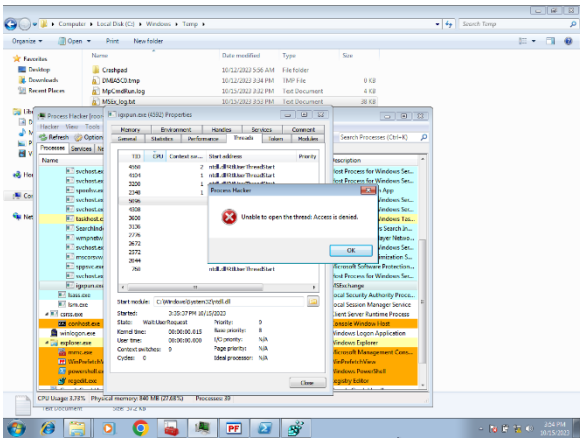
To study the behavior of the malware, we examined it via three different tools run with administrative privileges (these tools were installed prior to changing the network back to *Host-only* *):

a. Process Hacker

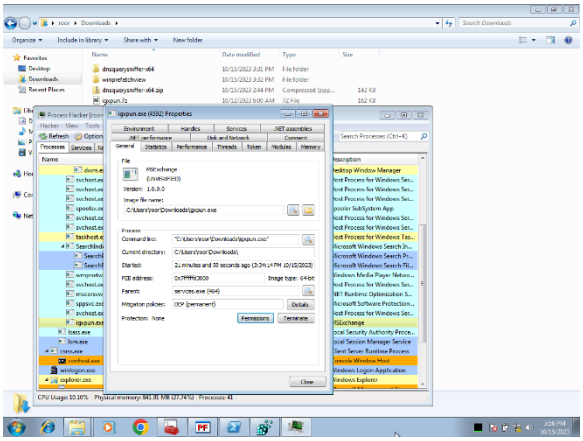
This tool provided a comprehensive view of all running processes and allowed for a detailed view of MSExchange.

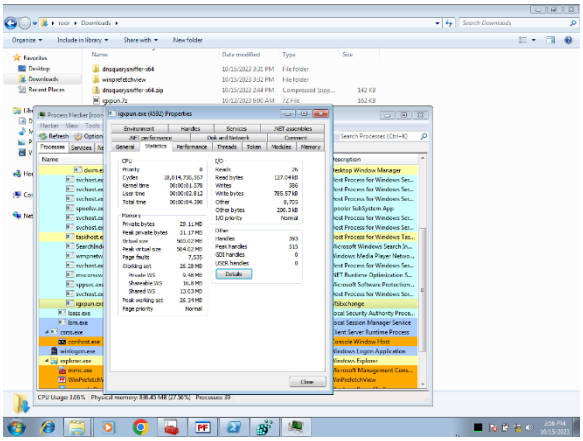
Here are samples of when we ran *Process Hacker* without administrative privileges:

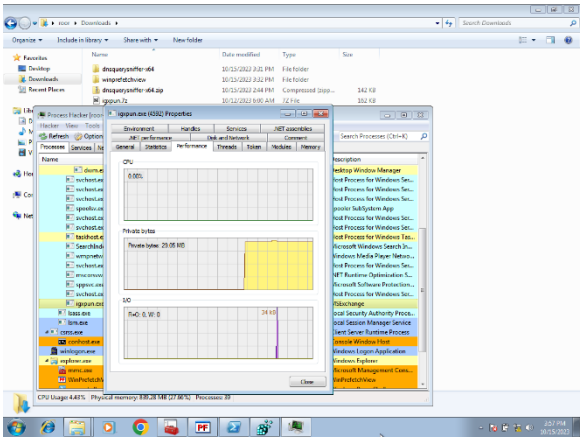


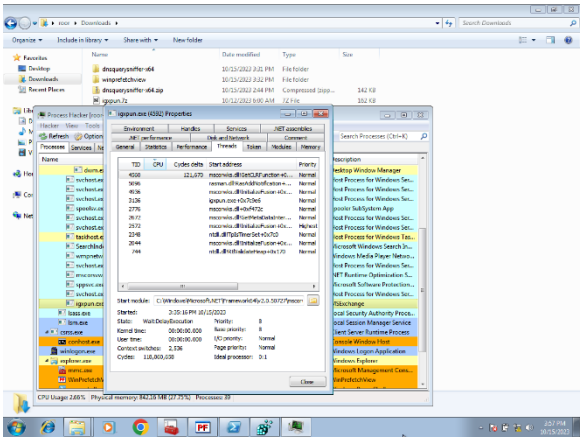


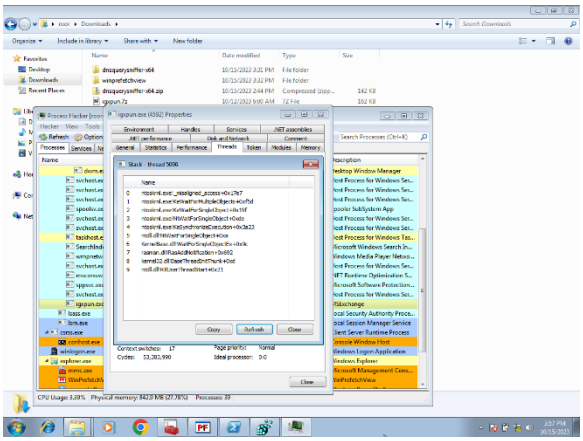
The following screenshots show the different information provided when run as administrator:

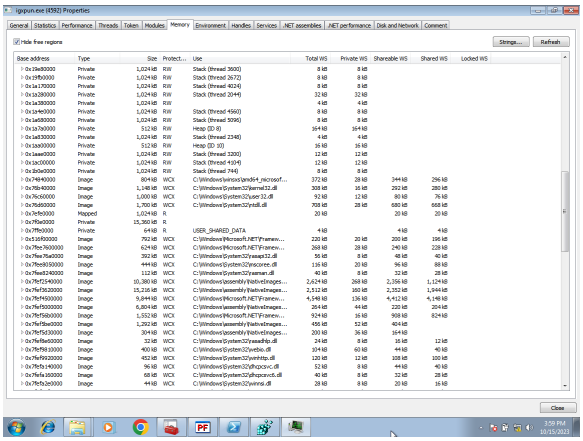


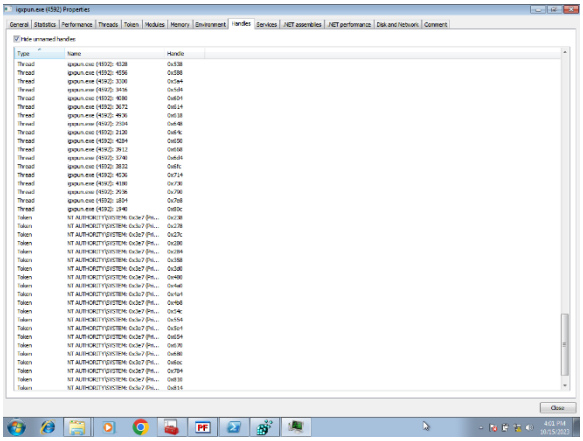


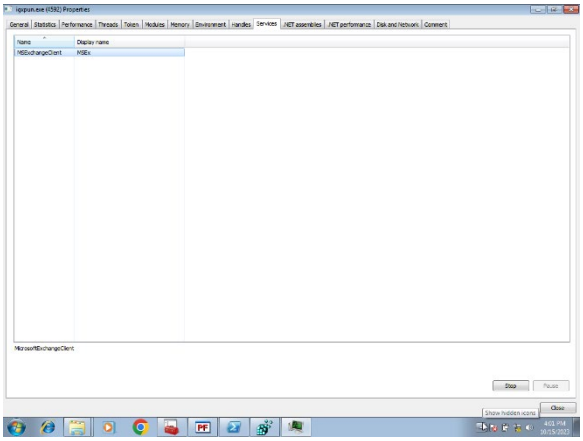


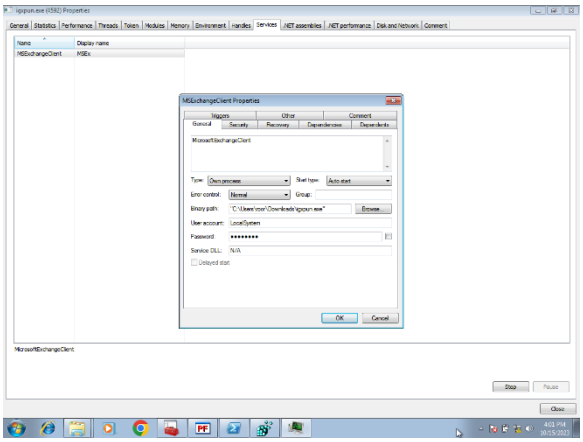


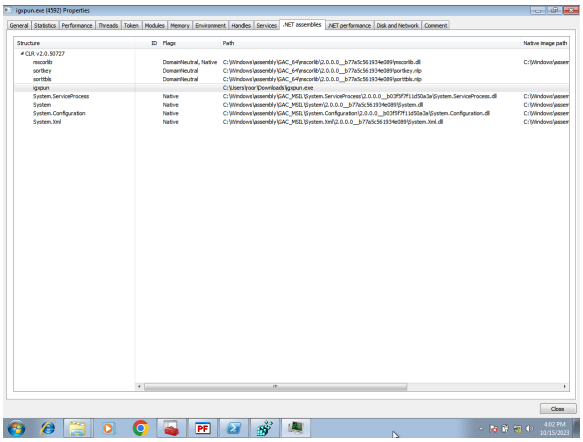


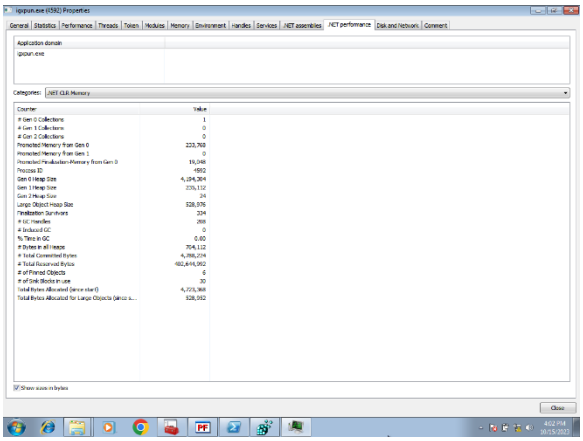


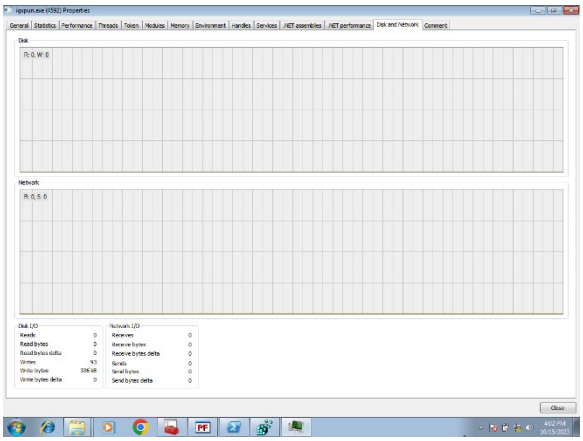










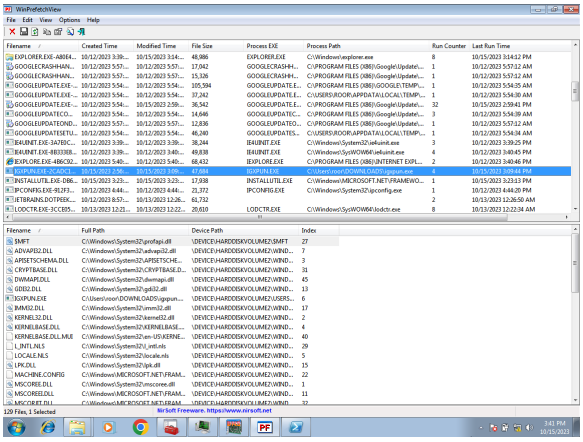


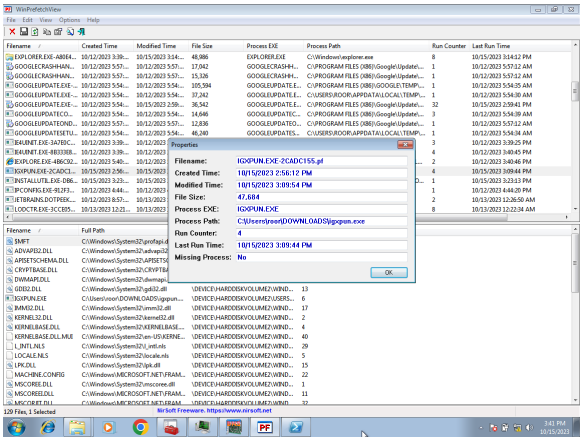
Important notes to keep track of:

- The service name is indeed *MSExchangeClient* and the display name is *MSEx*
- Various threads are being created
- The *igxpun assembler* was detected
- No network traffic is recorded (which is expected since the virtual machine is not connected to the internet)

b. WinPrefetchView

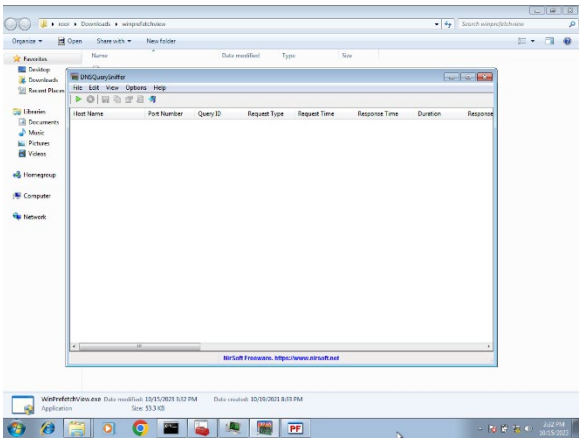
This tool provided a detailed view of all running processes, as well as for the *igxpun* in specific. It did not provide as much detail as *Process Hacker*.





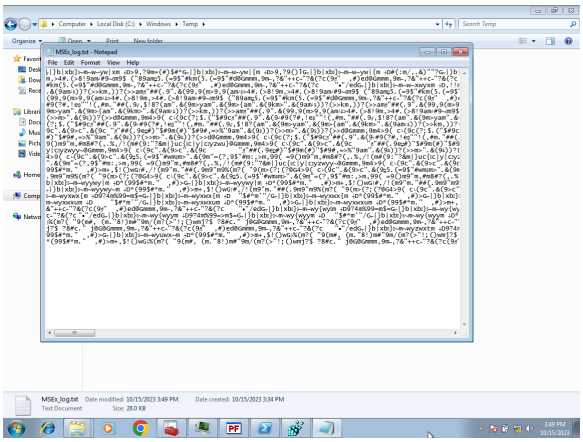
c. DNSQuerySniffer

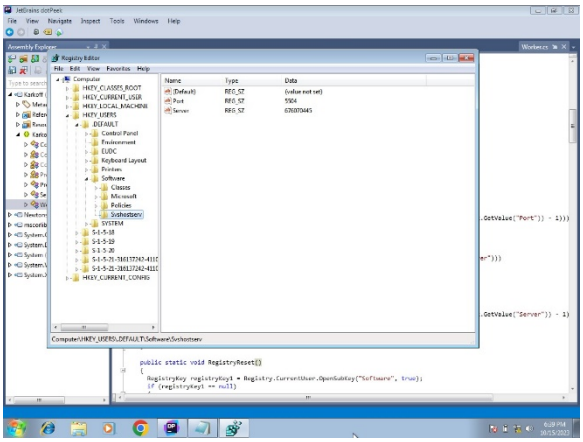
This tool did not record any output for the entire duration it was kept on. This is expected since it monitors the queries sent to the DNS server, but no queries were getting through as the virtual machine is disconnected from the internet. We then stopped the tool.



3. Comparison: Static vs Dynamic Analysis Results

Yes, the dynamic analysis results support those of the static analysis. As noted in *Process Hacker*, the service is creating threads in a timely manner. Other indicators of compatibility between the two analyses are the appearance of its encrypted *MSEx_log.txt* file and the *svshostsrv* registry subkey, as shown in the following two screenshots respectively.

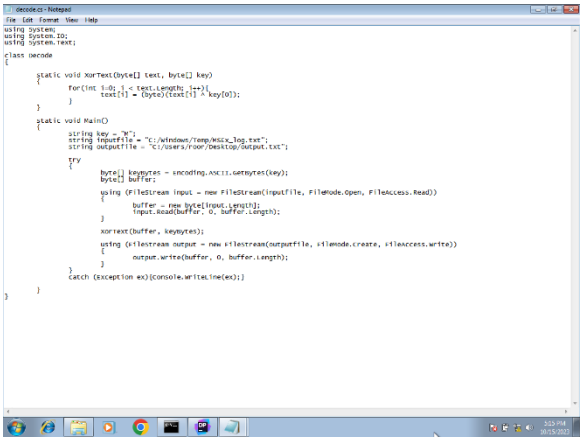


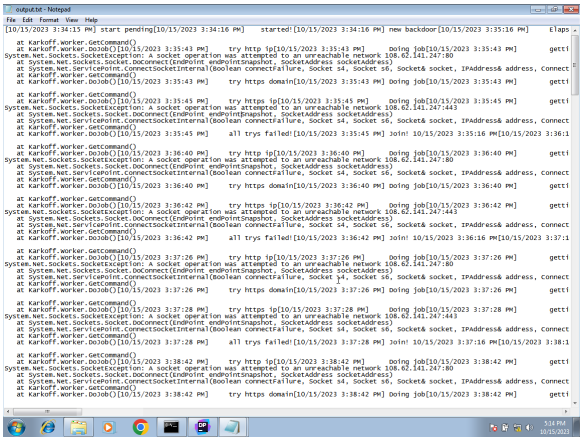


4. Accessing the Log File

As the log file is encrypted using *XOR cipher* with the key '*M*', we can decrypt it by applying that same operation (*XOR*) again on the encrypted contents using the same key '*M*'.

We have compile a C# code to read the content of the file, decrypt it, and then save the output to a file. The screenshot to the left shows the code while that to the right shows the output file contents.

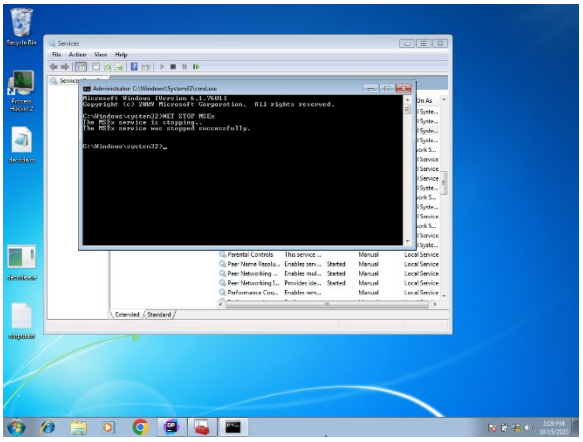




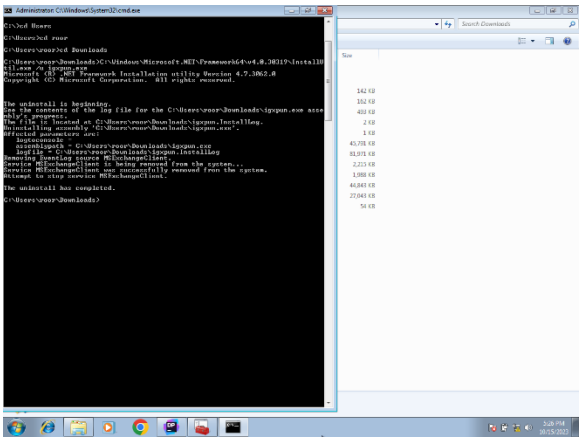
Both the content (failing tries) and the structure (tabs and new lines) of the output file suggest that the decryption is a success.

5. Terminating the Malware

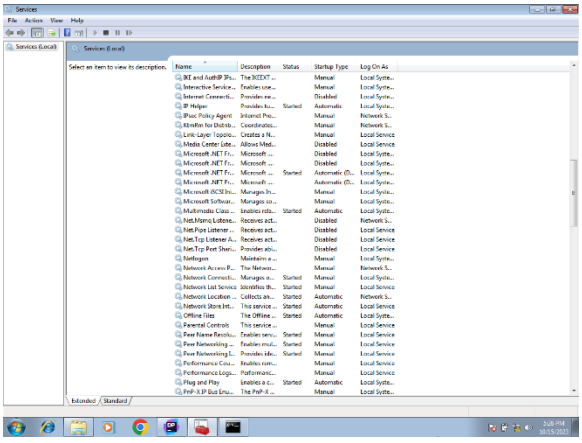
To stop the MSEx service, we ran the *NET STOP MSEx* command on the command prompt. The service was stopped successfully



However, since this malware would restart itself (the *MSEx service*) upon system reboot, we opted to delete the service entirely by running the *InstallUtil.exe* command with the */u* flag. The deletion was a success.



We confirmed the success of the deletion when navigating back to the *services.msc* to look for the service but couldn't locate it.



IV. Conclusion

Throughout this lab report, we both worked on all aspects of the lab together. However, we opted to work on one machine only (MAC and IP address of which are presented in the second section of the report).

This lab was generally fun to work on. The main challenge was getting started as it could was a little overwhelming at first, but when breaking the work, and the code, into smaller manageable pieces, we eventually started to make sense of what was going on.

V. References

Windows 7 Download https://www.softlay.com/downloads/windows-7-ultimate

What is a decompiler https://study.com/academy/lesson/what-is-a-decompiler-definition-uses.html

DNSpy https://github.com/dnSpy/dnSpy Dotpeek https://www.jetbrains.com/decompiler/

Microsoft.Win32 https://learn.microsoft.com/en-us/dotnet/api/microsoft.win32?view=net-7.0

System.Threading https://learn.microsoft.com/en-us/dotnet/api/system.threading?view=net-7.0

System.ServiceProcess https://learn.microsoft.com/en-us/dotnet/api/system.serviceprocess?view=dotnet-plat-ext-7.0

Process.StartInfo.CreateNoWindow https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.createnowindow?view=net-7.0

Process.StartInfo.RedirectStandardOutput https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.redirectstandardoutput?view=net-7.0

Process.StartInfo.RedirectStandardError https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.redirectstandarderror?view=net-7.0

Process.StartInfo.UseShellExecute https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.useshellexecute?view=net-7.0

Dispose https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose#dispose-and-disposebool

Local System Account https://learn.microsoft.com/en-us/windows/win32/services/localsystem-account

XOR Cipher https://www.wikiwand.com/en/XOR_cipher

List of Unicode Characters https://www.wikiwand.com/en/List_of_Unicode_characters

C# Json Deserialize Object https://www.newtonsoft.com/json/help/html/Overload_Newtonsoft_Json_JsonConvert_DeserializeObject.htm

RegistryKey.Flush https://learn.microsoft.com/en-us/dotnet/api/microsoft.win32.registrykey.flush?view=net-7.0

RegistryKey.Close https://learn.microsoft.com/en-us/dotnet/api/microsoft.win32.registrykey.close?view=net-7.0

Thread.Join https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread.join?view=net-7.0

Dropper https://www.wikiwand.com/en/Dropper_(malware)

Karkoff

https://blog.talosintelligence.com/dnspionage-brings-out-karkoff/

https://www.zdnet.com/article/dnspionage-campaign-releases-new-karkoff-malware-into-the-wild/

https://digital.nhs.uk/cyber-alerts/2019/cc-3037#:~:text=Karkoff%20is%20a%20newly%20observed,execution%20tool%20during%20these%20campaigns

Starting or stopping a windows service https://learn.microsoft.com/en-us/biztalk/core/how-to-start-stop-pause-resume-or-restart-biztalk-server-services

Deleting a windows service https://www.lifewire.com/deleting-service-in-windows-7-vista-and-xp-153356#:~:text=Right%2Dclick%20the%20service%20you,name%2C%20and%20then%20press%20Enter