

COS20030

**Malware Analysis**

*Lab 9*

**Android APK analysis, .NET analysis**

# Purpose

In the first part of this lab, we are going to learn how to reverse-engineer Android programs.

In the second part of this lab, we are going to analyse a .NET executable file.

# APK Analysis

The Android Package with the file extension APK is the file format used by the Android operating system, and some other Android-based operating systems for the distribution and installation of mobile apps, mobile games and middleware.

An APK file is an archive file containing different components of the Android application. Android applications are primarily written in Java. To run on Android devices, the Java code is compiled into Java bytecode and then translated into Dalvik bytecode. The Dalvik bytecode can be found in DEX (Dalvik Executable) files in the APK package.

To analyse an APK file we are going to extract its components and decompile the DEX file to inspect its code.

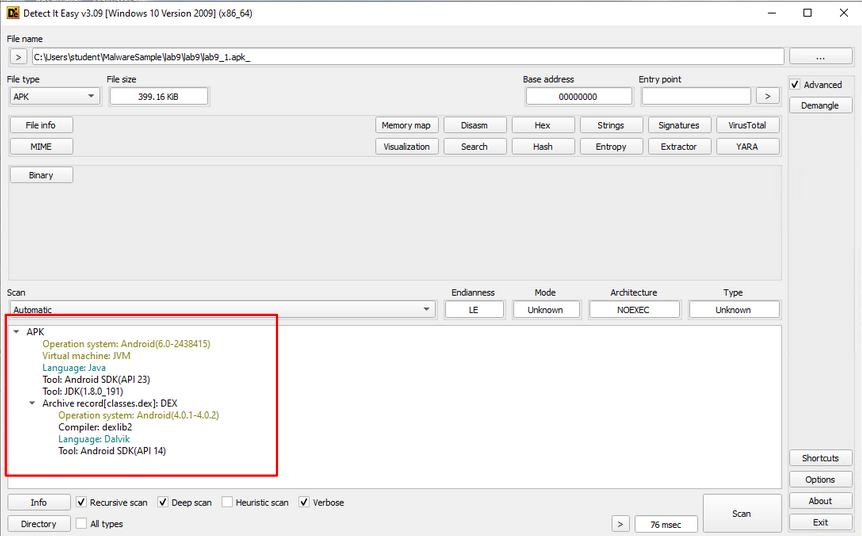
## Exercise 1: Android malware

Purpose: To learn how to analyse an APK file with an encrypted payload to understand its behaviour and also extract the final payload.

In this exercise, we have a malicious APK file. This file is a “loader” a.k.a “stager” malware with an encrypted payload inside. A loader malware(first stage) loads a new module(second stage) from embedded data in memory. While the first stage has a simple logic and doesn’t contain the main malicious behaviour, the payload(second stage) of the attack has the actual malicious capabilities. The purpose is to keep the initial stage light and undetectable.

We are going to analyse this file and understand how it loads the second stage of the attack in memory.

1. Extract the contents of lab9.zip with the password “infected”.
2. Open lab9\_1.apk in DiE to confirm its file format. (See the example screenshot below)



1. Then click on “File info” and go to the “Hex” view to confirm that the file is an archive file by inspecting the file format signature at offset 0. You can learn about file signatures from the Wikipedia page below:

<https://en.wikipedia.org/wiki/List_of_file_signatures>



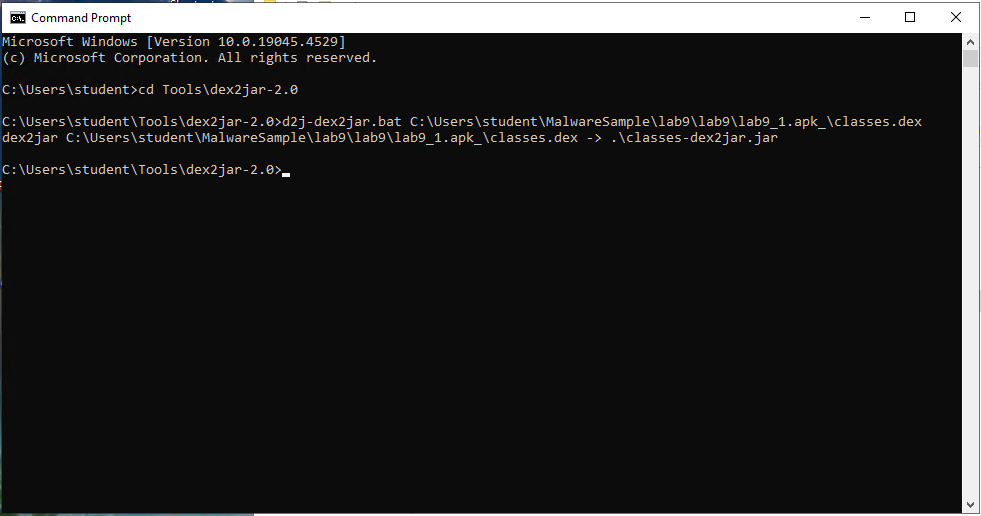
1. Rename the file to lab9\_1.apk\_.zip and extract the contents to see the components of the APK file.
2. Every file or directory inside an APK file has a specific purpose in the Android program. Explore the extracted files and try to find what appears to be the encrypted payload.
3. Inside the assets directory, you can find a file called “bin”. This file can be the encrypted payload of this malware. But we need to confirm this theory.
4. Open the file in DiE to inspect this file. Go through different sections and answer the following questions about this file.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the file type recognised by DiE | Binary |
| 2 | What is the md5 hash of this file? | 6a6406409136f54c112ef4c220a4f537 |
| 3 | Does the file contain a known file format signature at offset 0? |  |
| 4 | Does this file have any readable strings inside? | Nope, cant find any |
| 5 | What is the entropy of this file? Does it suggest that it's an encrypted file? | 7.91721, yes it suggest that the file is packed |

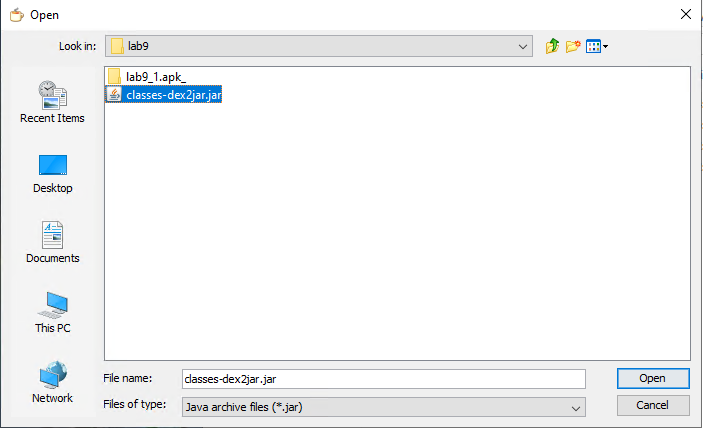
1. The extracted components of the APK file also contain a file named classes. dex. This file contains the compiled code for the Android application.
2. Open this file in DiE. Confirm the file type recognised by DiE and also check the file format signature in the “Hex” section.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the signature of the DEX file? | 64 65 78 0A 30 33 35 00 |

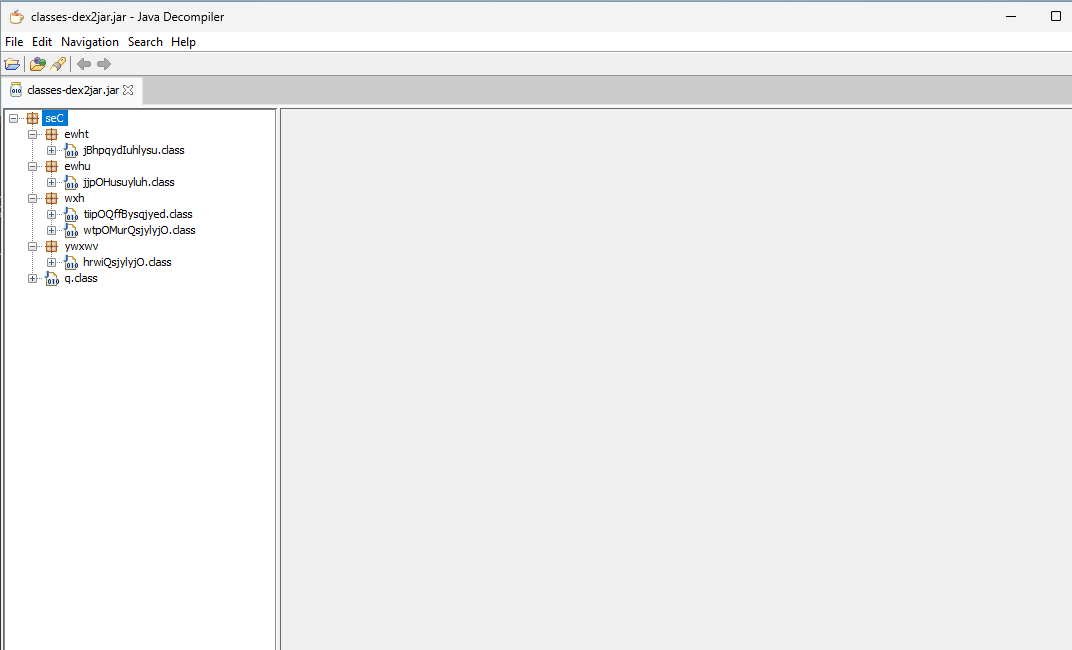
1. The DEX file is a compiled file and contains bytecode. We need to convert it to a JAR file to see the JAVA classes. To do that we are going to use a commandline tool called Dex2Jar.
2. Open a command prompt and navigate to the Tools\dex2jar-2.0 directory. (See the screenshot)
3. Execute the d2j-dex2jar.bat file and pass the classes.dex file extracted from the APK file to this tool. (See the example screenshot below)



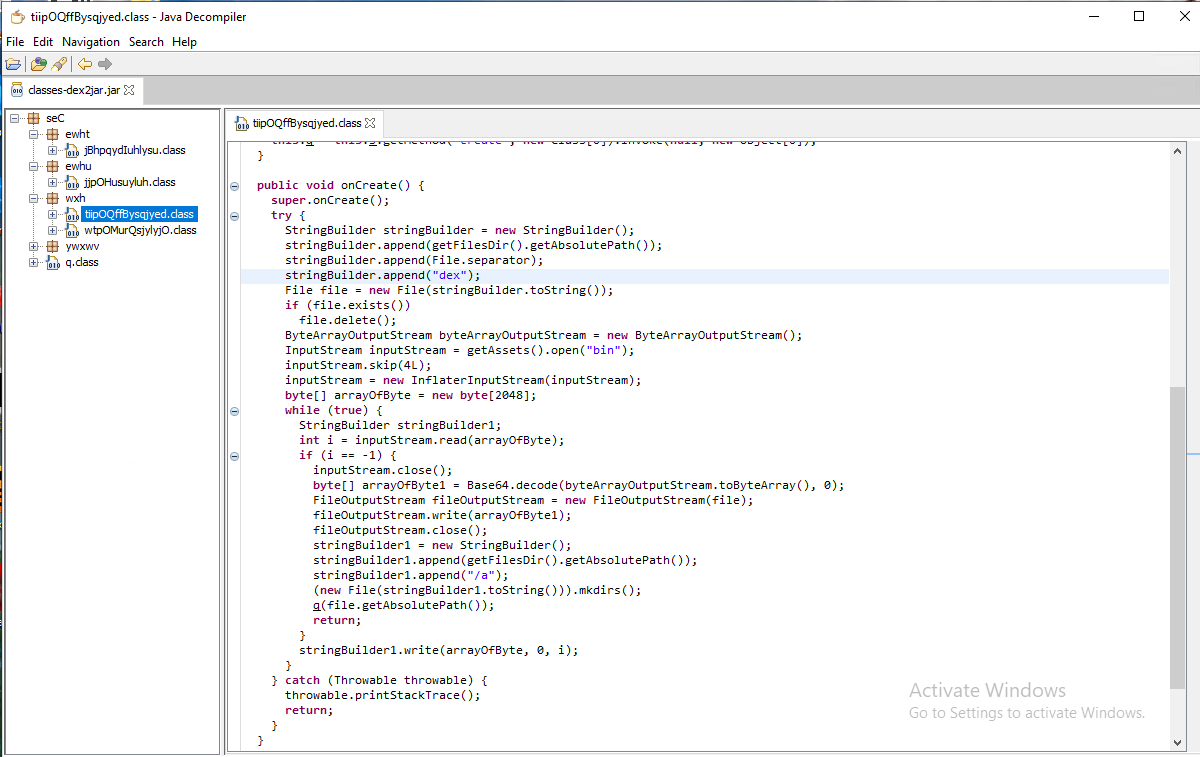
1. This tool creates a file inside the dex2jar-2.0 directory named “classes-dex2jar.jar”. This file contains the decompiled JAVA classes. Copy this file into your lab9 directory.
2. The next step is to load this file inside the jd-gui tool to inspect the Java code. Execute jd-gui.exe from its desktop shortcut and load the “classes-dex2jar.jar”. Choose “Java archive files (\*.jar)” as the “Files of type:” field when navigating to the path of the JAR file. (See the example screenshot below)



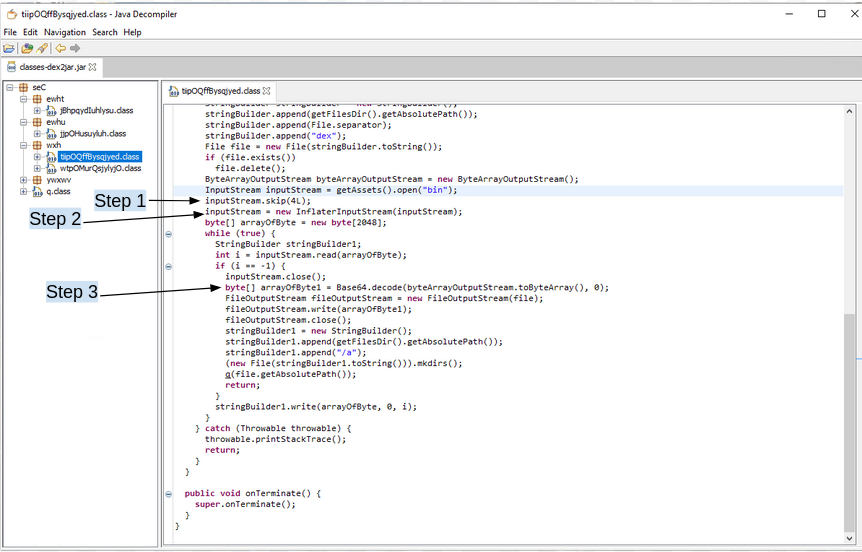
1. From the left menu, expand the decompiled code to see all the classes like below.



1. Click on each class to see the decompiled Java code. Skim through the code of the classes to find where it attempts to access the “bin” asset which we earlier suspected to be the payload of the malware.
2. In the function shown in the screenshot below, you can find a reference to the “bin” asset.



1. Read the function’s code to figure out how the asset is being used.
2. The malware creates a new dex file from the data inside the “bin” asset. To create the dex file’s content, the data from the “bin” asset is manipulated in 3 steps. (See the screenshot below)



1. Examine each step to understand what it does. Answer the questions with your findings.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the operation in Step 1? | Skips 4 characters |
| 2 | What is the purpose of Step 2? What decompression algorithm is used in this step? | Zip delete |
| 3 | What is the purpose of Step 3? What decoding algorithm is used in this step? | Base64 |

### Extension exercise 1

Write a short Python script to create the final DEX file from the “bin” asset.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | Python script to create the DEX file | Import base64  Importy zlib  Fp = open(‘bin’)  Data = fp.read()  Fp.close()  Data2 = data[4:]  Data3 = zlib.decompress(data2)  Data3 = base64.b64decode(data3)  Fp2 = open(‘data.dex’.’wb’)  Fp2.write(data4)  Fp2.close() |
| 2 | Md5 hash of the final DEX file | af2890a472b85d473faee501337564a9 |

# .NET Analysis

In this part of the lab, we are going to learn how to analyse a malicious PE(Portable Executable) file written in VisualBasic for the .NET environment.

dnSpy is a powerful tool to decompile, debug and edit .NET applications.

Open the dnsPy tool from its desktop shortcut. Once it is loaded, you can see there are a few items in the left-hand side pane which is called “Assembly Explorer”. Simply go to the File menu and click on “Close All”. Now the tool is clean of any other applications and ready to be used for our analysis.

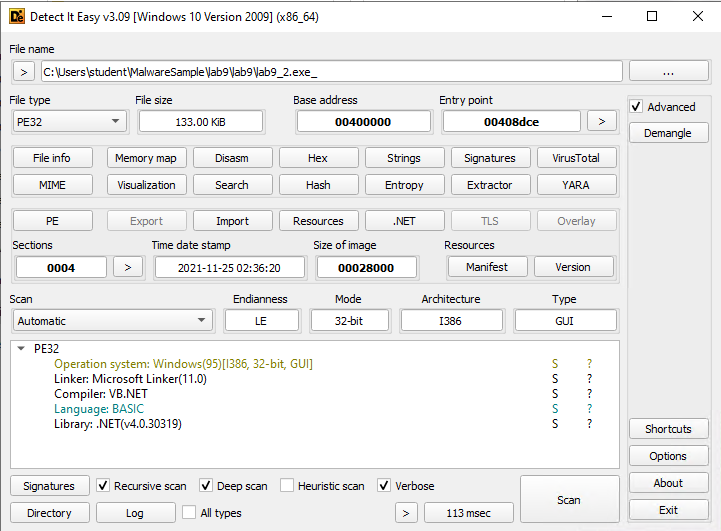
## Exercise 2: .NET malware

Purpose: To learn how to decompile a .NET executable file and analyse it to learn about its malicious behaviour.

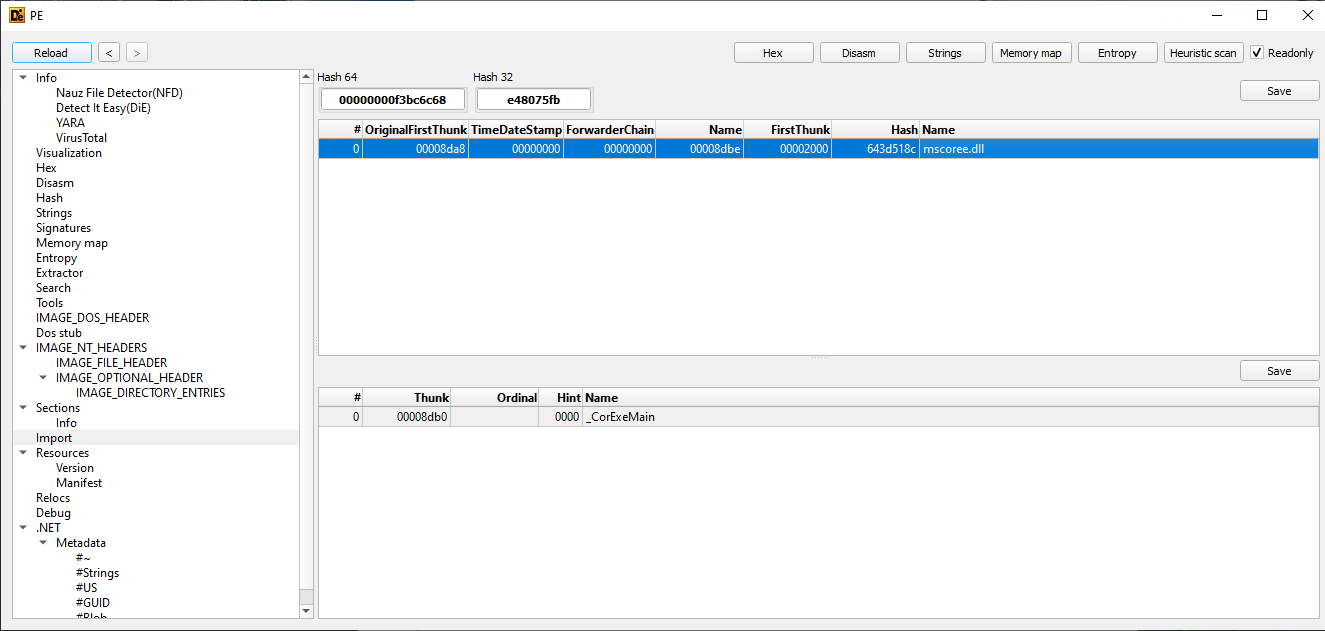
The malicious file we are going to analyse in this exercise is a backdoor malware compiled for the .NET framework. This file is dropped by a malicious document file on a victim’s machine. We have analysed the malicious document file in lab7(lab7\_1.doc - md5: 02bfa75cc9a8fe54828b39440a86fca7). During the analysis of lab7\_1.doc, we learned that the VBA Macro script embedded into the MS DOC file extracts the final payload from an object inside an XML component of the document file. The VBS Macro then creates an executable file and finally executes it. We are going to analyse that payload in this exercise.

The malware is a backdoor module and our end goal is to find the IP address of the command and control server that this malware communicates with as well as the port numbers used by this backdoor.

1. Open lab9\_2.exe\_ in Die and confirm its file format. (See the example screenshot below)



1. Look at the IAT(Import Address Table) of this file. All the modules compiled for the .NET framework have the same import table. Only the “\_CorExeMain” function is imported from the “mscoree.dll” Library. (See the example screenshot below)



1. Open the “.NET” section from DiE to find metadata and streams of the .NET file.
2. To decompile .NET modules we use the “dnSpy” decompiler. Execute “dnSpy.exe” via its desktop shortcut and open the “lab9\_2.exe\_” file inside the tool.
3. You will see the decompiled malware among other .NET modules present on the system in the left pane and some metadata information in the right pane. (See the example screenshot below)

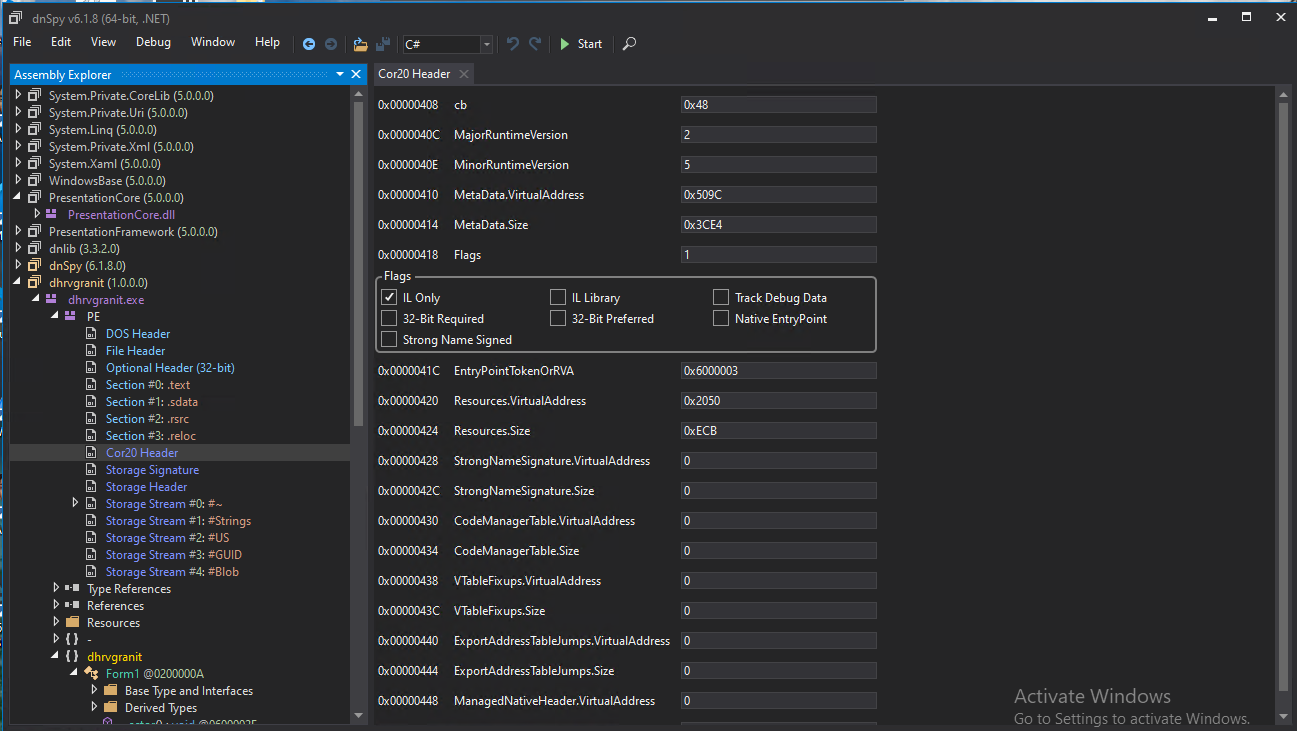


The “dnSpy” window has two main panes. The left-hand side pane is the Assembly Explorer, which shows the libraries used in the code. The right-hand side pane extends the information provided in the Assembly Explorer pane, such as representing the source code or the Metadata information of the library you select from the Assembly Explorer.

1. Read the information provided in the right pane and answer the following questions about this file.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the Timestamp of this program? | Timestamp: 619E6A04 (11/25/2021 2:36:20 AM) |
| 2 | What is the path of the Entry point function? | Dhrvgranit.My.MyApplication.Main |
| 3 | What is the value shown as “AssemblyProduct”? | AssemblyProduct(“dhrvgranit”) |
| 4 | What is the TargetFramework value? | (“.NETFramework.Version=v4.6”, FrameworkDisplayName = “.NETFramework 4.6”) |

1. Expand the details of the PE file and navigate to “Cor20 Header”. Here you can find basic information about the .NET file such as its EntryPoint. (See the example screenshot below)



1. Now, let’s inspect the code of the malware and look for the IP address and port numbers inside the file.
2. You can right-click on the name of the program in the left pane and choose “Go to Entry Point”. However, the main malicious behaviour of this program is implemented into objects from other classes. (See the example screenshot below)



1. Skim through these classes to familiarise yourself with the code of this program.
2. You can find two fields under the “MAIMN” class with the names “ports” and “ips”. Click on each of them to see their definition.



1. The values in the array of “ips” show decimal values which should be converted into ASCII characters to see the IP address of the command and control server.

Fill out the table below with the IP address and port numbers.

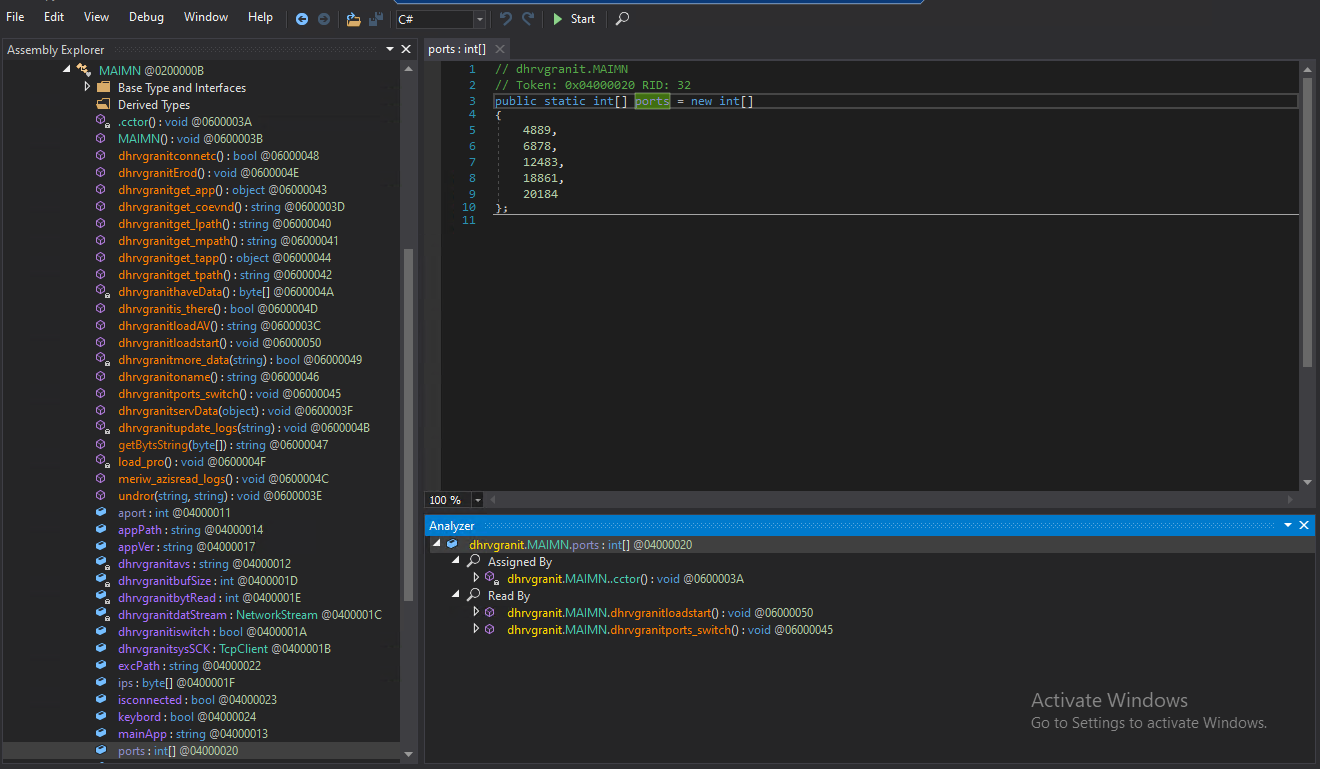
|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the IP address that this backdoor communicates with? |  |
| 2 | What port numbers are used by this malware for its communication with the command and control server? |  |

1. Now, let’s find out how the IP address and the port numbers are used by navigating through the code of the malware.

To help better navigate through the source code, if you right-click on any global variable, API or function name and click on Analyze, a window called Analyser will open up. (See the example screenshot below)



The analyzer lists all the references to this object throughout the whole application code, including the list of functions that read this object. (See the example screenshot below)



You can double-click on each function listed to see its code in the top window as well as references to this function in the expanded view inside the Analyzer window.

Answer the following questions with the findings from your analysis.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the name of the function that handles the connection to the command and control server?  Explain what this function does. |  |
| 2 | When and how does the malware decide to choose a new port number for the connection? |  |

## Extension exercise 2

One of the main functionalities of this malware is to collect some information about the victim’s machine and send it to its server.

Look for where it collects system information, and fill out the table below.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the name of the function that collects the system information and then creates a string based on the information it collected? |  |
| 2 | What system information does it collect? |  |

Another piece of information that the file collects, is the list of running processes. Find where it does that and fill out the table below.

|  |  |  |
| --- | --- | --- |
| # | Question | Answer |
| 1 | What is the name of the function that collects the list of running processes? |  |
| 2 | What is the format of the string it creates from the process names? |  |

*End of Lab*