**SWINBURNE UNIVERSITY OF TECHNOLOGY**

School of Science, Computing, and Engineering Technologies

Text

Description automatically generated

Malware Analysis

COS20030

**Assignment 2A**

Group Project: Technical Report Writing

**TUTOR**

Harry Ouaida

**STUDENTS**

Tran Anh Thu Pham - 103818400

Dang Khoa Lam – 103843431

Word count:

**HAWTHORN – FEBRUARY 2025**

Table of Contents

I. Summary of the Attack 2

[I. Summary of the attack 2](#_Toc190263329)

[II. Execution Diagram 2](#_Toc190263330)

[1. Summary of Execution Flow 2](#_Toc190263331)

[2. Diagrams 2](#_Toc190263332)

[III. Stage 1 6](#_Toc190263333)

[1. Technical Details 6](#_Toc190263334)

[2. Descriptions 7](#_Toc190263335)

[IV. Stage 2 8](#_Toc190263336)

[1. Technical Details 8](#_Toc190263337)

[2. Descriptions 9](#_Toc190263338)

[3. Communication with the server 11](#_Toc190263339)

[4. How malware does reconnaissance on the victim’s system 13](#_Toc190263340)

[5. Download and execute another malware from the remote server 15](#_Toc190263341)

[V. Analysis methods 17](#_Toc190263342)

[1. Static Analysis 18](#_Toc190263343)

[2. Threat Intelligence and Reputation Analysis 19](#_Toc190263344)

[VI. Conclusion 19](#_Toc190263345)

[VII. Scripts or automated techniques used for analysis 20](#_Toc190263346)

[VIII. Yara rules 22](#_Toc190263347)

[1. Stage 1 22](#_Toc190263348)

[2. Stage 2 23](#_Toc190263349)

[IX. Indicators of Compromise (IoC) 24](#_Toc190263350)

[1. md5 hashes 24](#_Toc190263351)

[2. sha1 hashes 25](#_Toc190263352)

[3. sha256 hashes 26](#_Toc190263353)

[4. Malicious IP addresses or URL 27](#_Toc190263354)

[X. MITTRE ATT&CK 28](#_Toc190263355)

[XI. Advanced Yara: Writing a YARA rule for heuristic malware detection 30](#_Toc190263356)

[XII. Python Scripting 35](#_Toc190263357)

[XIII. References 40](#_Toc190263358)

[XIV. Reflection 40](#_Toc190263359)

**Analysis of a Multi-Stage VBA Macro Backdoor: Execution, Payload, and Attribution**

# Summary of the attack

This report provides a detailed examination of embedded macro malware discovered in a Microsoft Office file. It primarily explores the malware’s execution process, command-and-control (C2) communication, and payload deployment. The goal is to provide a thorough understanding of how the malware establishes a remote connection, executes its operations, and interacts with the attacker’s server.

A visual diagram of the malware’s execution flow is included to illustrate its behaviour. Key functions such as dhrvgranitconnetc(), dhrvgranitErod(), and load\_pro() are analyzed to determine their roles establishing connections, retrieving payloads, and interacting with other components. The relationships between these functions are also explored to gain deeper insights into the malware’s overall operation.

Additionally, the malware’s tactics and techniques are mapped to the MITRE ATT&CK framework. The report also covers the methodologies and tools used for analysis, DiE.exe, dnSpy.exe, and VirusTotal. Lastly, Indicators of Compromise (IoCs) are used to identify potential threats.

# Execution Diagram

## Summary of Execution Flow

The attack starts when a user opens a malicious Office document and enables macros, triggering an embedded VBA script. This script obfuscates its code to evade detection and executes key functions to connect to the attacker’s C2 server, download and decrypt the payload, and execute it in memory. The malware then establishes persistence by modifying system settings and communicates with the C2 server, allowing the attacker to remotely execute commands and exfiltrate data. Finally, the malware enables keylogging, data theft, lateral movement and maintaining control.

## Diagrams

The diagrams below represent the execution flow and the stages of the malware and their connection with each other.

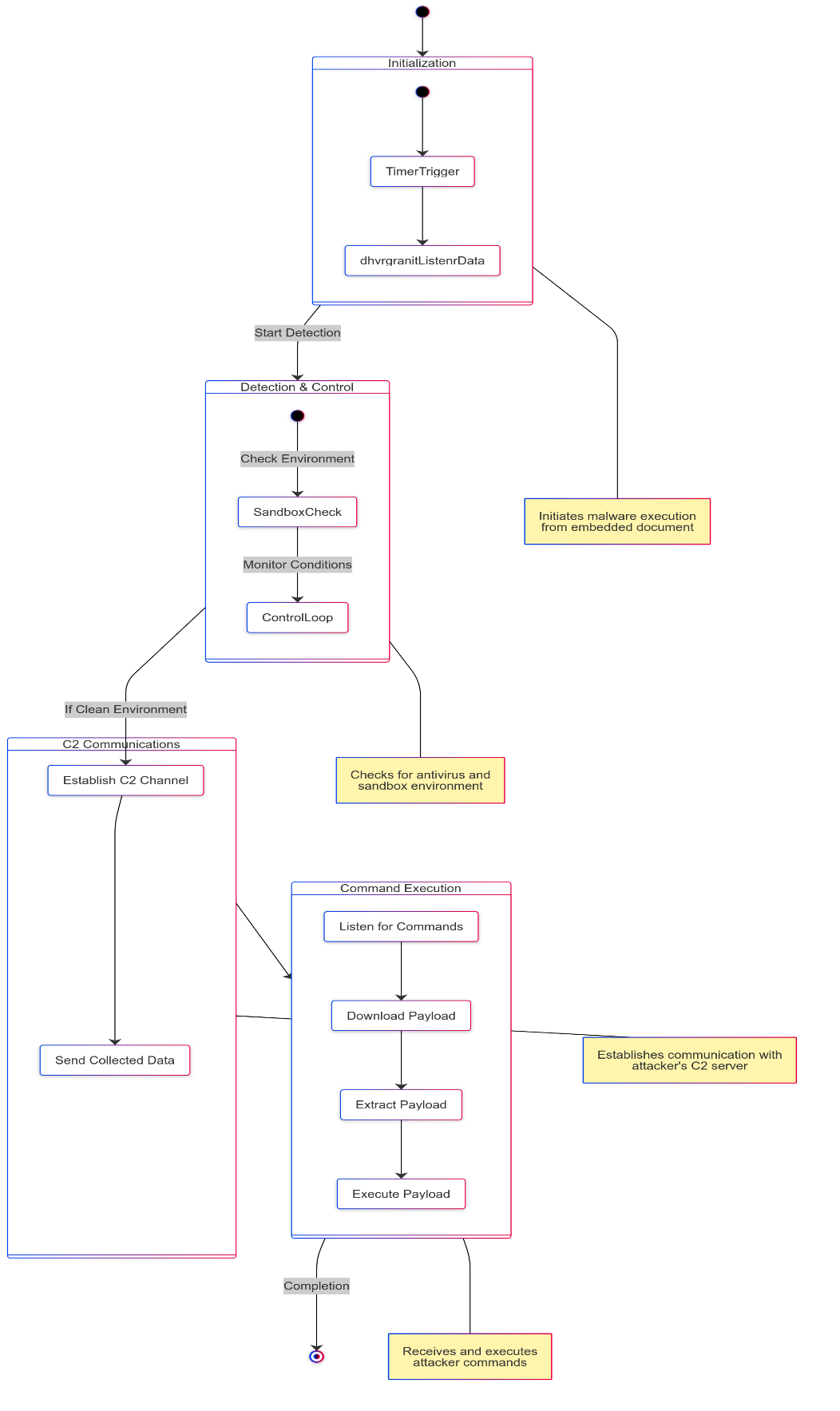


Fig 1. Malware Execution and C2 Communication Flowchart

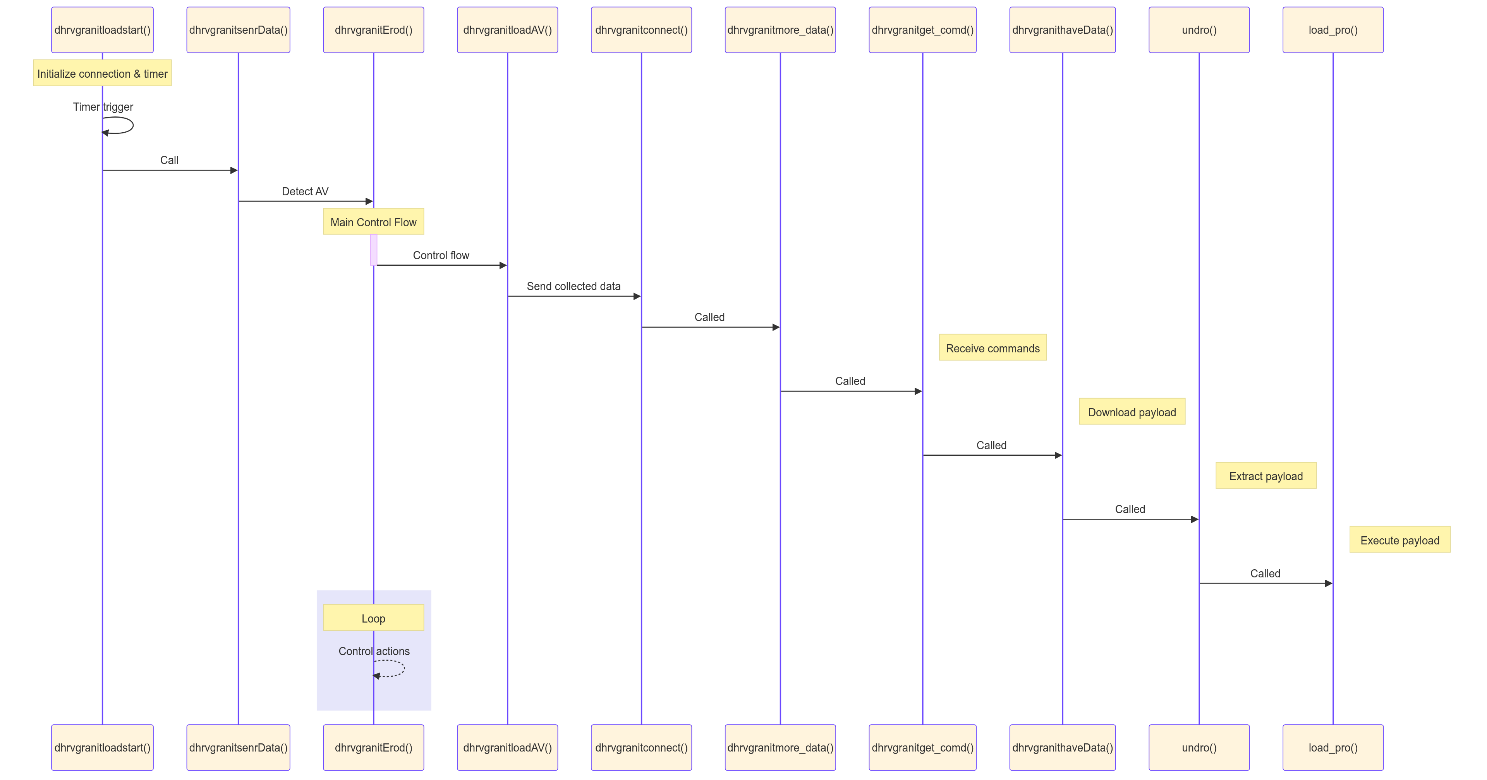


Fig 2. Malware Execution Sequence Diagram

These two diagrams describe a multi-stage malware attack, each with a defined function that performs specific tasks. Here's a detailed explanation:

|  |  |  |
| --- | --- | --- |
| **Stage** | **Function** | **Description** |
| Initialization and Timer Trigger | dhrvgranitloadstart() | Initiates the malware execution. It’s embedded within a document, and once executed, it starts the attack process |
| Timer Trigger | Ensures that the malware doesn’t immediately execute, which helps avoid detection |
| dhvrgranitListenrData() | Be responsible for beginning the data collection process, monitoring system activity, network traffic, or user inputs to gather information for future exploitation |
| Detection and Control Flow | dhrvgranitErod() | Checks whether the system is running antivirus software or in a sandbox environment. It’s a common tactic of malware to avoid execution in an environment where it might be analyzed or detected |
| Control Flow Loop | If the malware by antivirus or sandbox detection, it enters a loop where it constantly checks for certain conditions and executes functions based on those conditions |
| Command and Control (C2) Communications | dhrvgranitconnetc() | Establishes a communication channel between the infected system and the attacker’s Command and Control (C2) server. This allows the attacker to issue commands to the infected machine remotely |
| dhvrgranitmore\_data() | Once the connection is established, this function sends the gathered data back to the attacker’s server |
| Receiving and Executing Commands | dhvrgranitgetr\_Coemd() | Listens for commands sent by the attacker from the C2 server. These commands could be to perform actions like further data collection, file uploads/downloads, or to execute malicious payloads |
| dhrvgranitNewObData() | Downloads the payload that the attacker wants to execute on the infected system |
| undro() | Extracts the payload from its encrypted or compressed state so that it can be executed |
| load\_pro() | Finally, this function executes the payload, giving the attacker full control over the infected system. |

# Stage 1

In the first stage, a malicious document named **lab7\_1.doc** contains a VBA macro that extracts a Portable Executable (PE) file from an embedded .xml component, writes it to disk, and executes it. Using DiE.exe, we analyze the document’s hashes, type, and timestamp. Since it's an MS Word file (a zip archive), we extract its contents, revealing **vbaProject.bin**, which contains the compiled VBA macro code.

To examine the macro, we use olevba.py, which extracts a VBA script designed to retrieve a malicious executable from a hidden Shape object ("**Text Box 2**"), reconstruct it from split text data, and write it to a new file (**dhrvgranit.exe**). The macro then executes the file using the Shell command and displays a fake error message, "Not Supported format!", to deceive the user.

Next, we identify the data used to generate the executable by searching for "**Text Box 2**" using **findstr**, and locating it in document.xml and **vbaProject.bin**. The extracted data is verified as a PE file by detecting the "MZ" signature. The data that we found in the **document.xml** file is used to create an executable file, confirming its malicious intent.

## Technical Details

md5 - **02bfa75cc9a8fe54828b39440a86fca7**

sha1 - **03fc473eb4bfa10a5844862160acc491bf432a8c**

sha256– **f72272d5be19156af6f1d15b5d61356dd0f0fe3ccc5c1b6ca1cc781b6e141b4f**

File name - **lab7\_1.doc\_**

File type - **Microsoft Office Word**

Timestamps - **2021-11-25 20:00:00** (creation time)

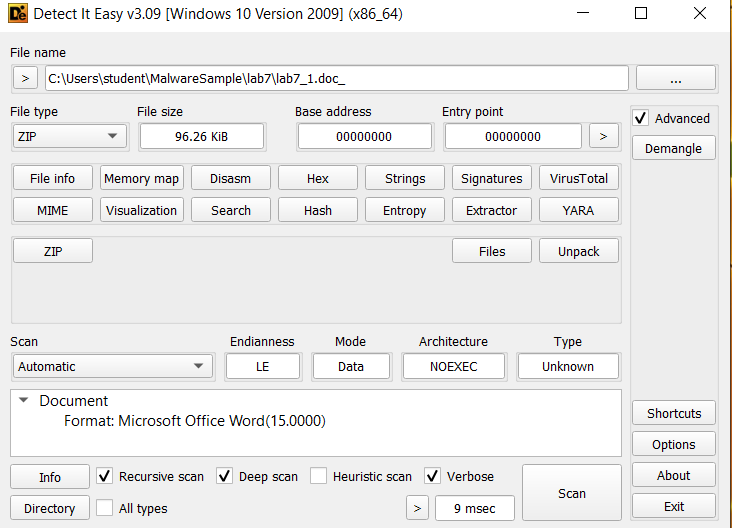


Fig 3. Using the DiE.exe tool to determine technical details of the lab7\_1.doc file

## Descriptions

The primary responsibility of this stage is to deploy and execute a malicious payload through an embedded VBA macro in a Microsoft Office document. When the document is opened, the macro extracts a PE file from an embedded .xml component writes it to disk as dhrvgranit.exe, and executes it.

This process is triggered by winword.exe, which spawns a child process (dhrvgranit.exe) to facilitate further stages of the malware. To deceive the user, the macro displays a fake error message after execution. The dropped payload remains hidden within the system, enabling potential further malicious activities.

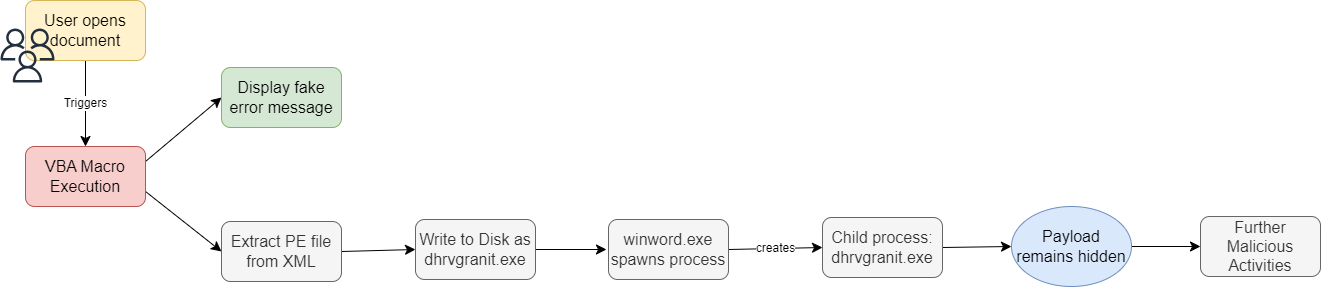


Fig 4. Stage 1 VBA Macro-Based Malware Execution Flow

# Stage 2

In this stage, we analyzed a backdoor malware compiled for the .NET framework, which is dropped by the **lab7\_1.doc** file examined in Stage 1.

First, we identified the command-and-control (C2) server and port numbers the malware uses. Using DiE.exe, we examined its hashes, type, and timestamp. The Import Address Table (IAT) showed that **\_CorExeMain** was the only imported function from mscoree.dll, confirming it as a .NET executable.

Next, we decompiled the malware using dnSpy, revealing that the entry point is d**hrvgranit.My.MyApplication.Main**, with **dhrvgranit** as the AssemblyProduct and **.NETFramework,Version=v4.6** as the Target Framework. Within the MAIN class, we found two fields—"ports" and "ips"—containing network details. The "ips" array, when converted from decimal to ASCII, revealed the C2 server’s IP address.

Using an **Analyzer** tool, we traced how the malware handles C2 communication. The function **dhrvgranitconnetc()** attempts to connect to **104.129.27.131**, switching between ports 4**889, 6878, 12483, 18861, 20184** if a connection fails. Once connected, it establishes a network stream for data exchange.

The malware also collects system information through **collectSystemInfo(),** gathering details like computer name, user name, OS, architecture, system uptime, locale, and domain name. It further retrieves a list of running processes via **listRunningProcesses()**, formatting them as "process1.exe" before transmission.

## Technical Details

md5 - **4ad5b2369a9f8dfaf32c3a31e92a35d7**

sha1 - **4ad5b2369a9f8dfaf32c3a31e92a35d7**

sha256- **2a03ba547e97964ba9780fe7a6f80b3c07c811f4c9f770359ae61941a219411b**

File name – **lab9\_2.exe\_**

File type - **PE Executable (PE32)**

Timestamp - **2021-11-25 02:36:20** (creation time)

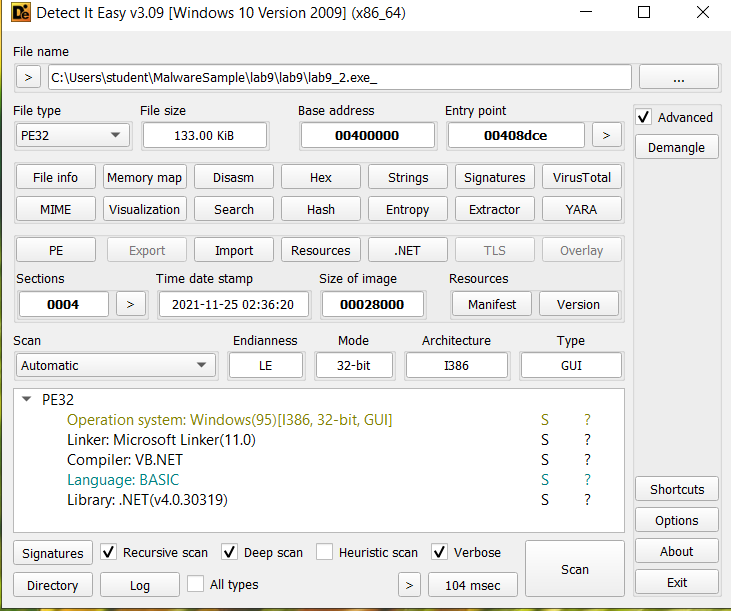


Fig 5. Using the DiE.exe tool to determine technical details of the lab9\_2.exe\_ file

## Descriptions

In Stage 2, the malware is a .NET-based backdoor designed to provide remote access and control over the infected system. It establishes a connection with the attacker's C2 server, enabling command execution, data exfiltration, and system reconnaissance. The malware gathers system information, lists running processes, and transmits this data to the attacker. Additionally, it ensures persistence by maintaining its network connection and adapting to different ports if the primary one fails.

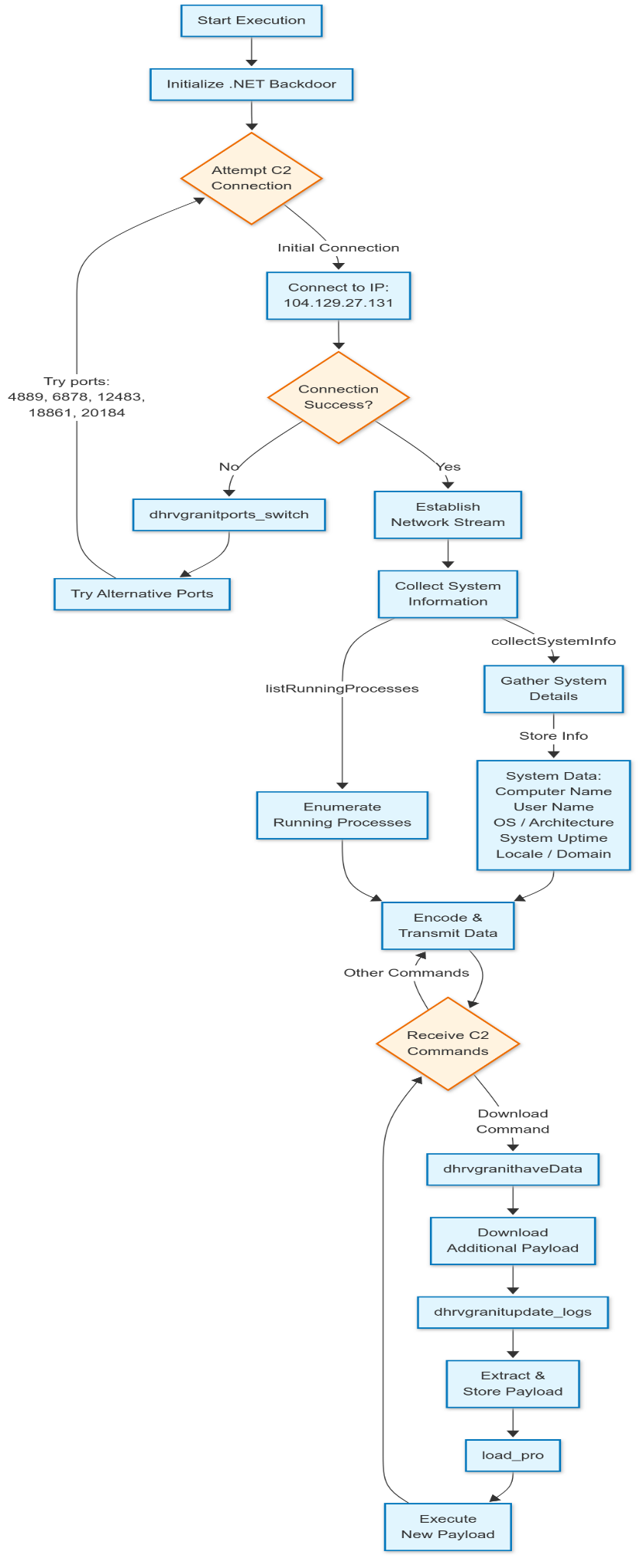


Fig 6. .NET Backdoor Command & Control Flow Diagram

## Communication with the server

Upon execution, the malware attempts to connect to the attacker's remote C2 server. The malware uses a TCP-based communication **mechanism** to allow the attacker to execute commands and exfiltrate data from the victim's machine. The communication process includes the following steps.

The remote IP address used by the malware is stored in the ips variable as a byte array. Once decoded using ASCII values, the IP address resolves to **104.129.27.131**.

A screenshot of a computer

Description automatically generated

Fig 7. Find the remote IP address used by the malware using the dnSpy tool

The port numbers, which are stored in the **ports** variable, include **4889, 6878, 12483, 18861,** and **20184**. These ports are used for network communication, and the malware can switch between them to maintain a connection if one becomes unavailable.

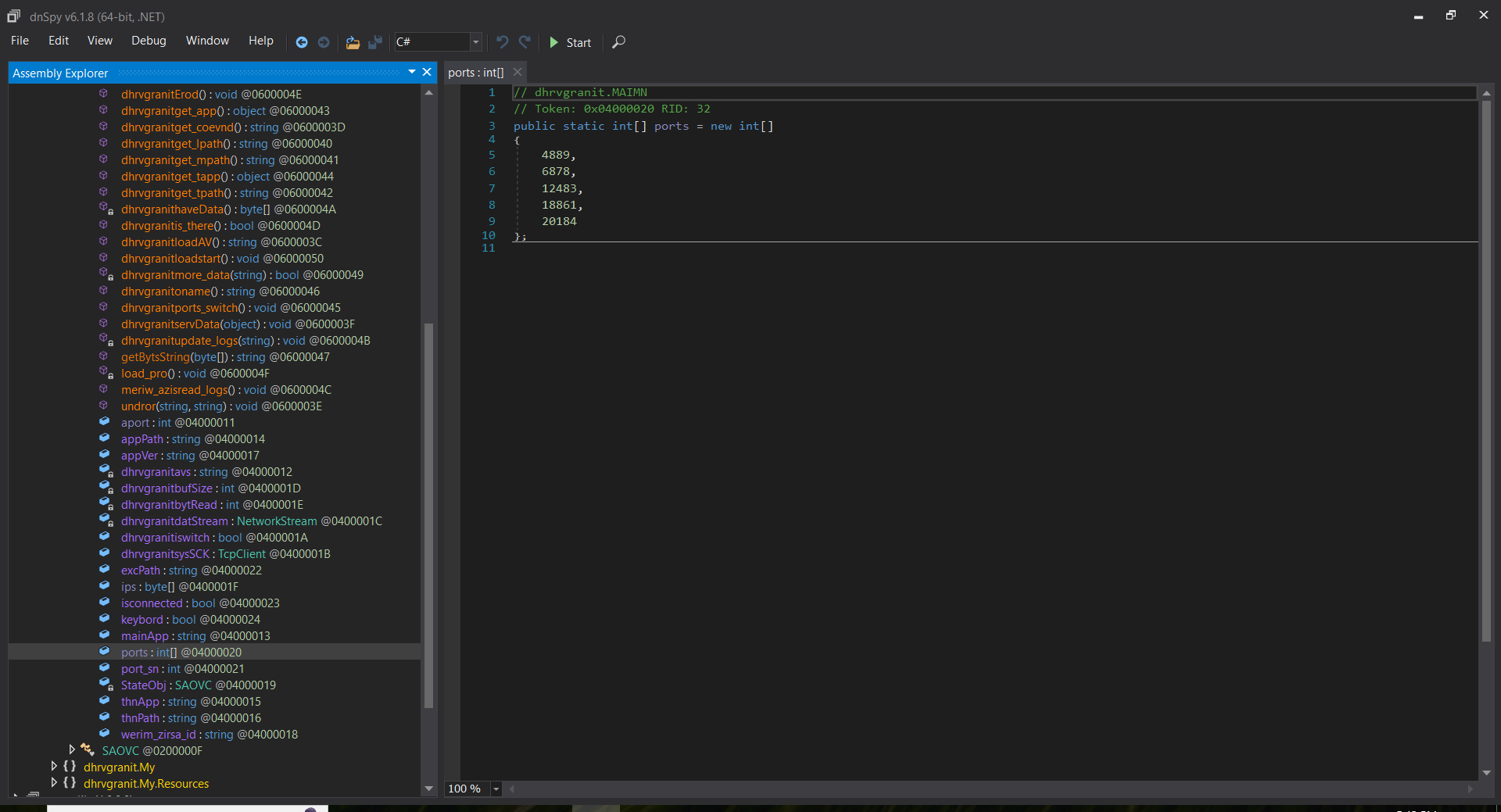


Fig 8. Find ports number used by the malware using the dnSpy tool

The primary function responsible for establishing the remote connection is **dhrvgranitconnetc()**. This function first checks whether an active connection already exists. If no connection is detected, it attempts to create a new one using a TcpClient object. It then retrieves the IP address from the **ips** variable and iterates through the list of available ports to establish a successful connection. If the initial attempt fails, the function invokes **dhrvgranitports\_switch()** to handle port switching.

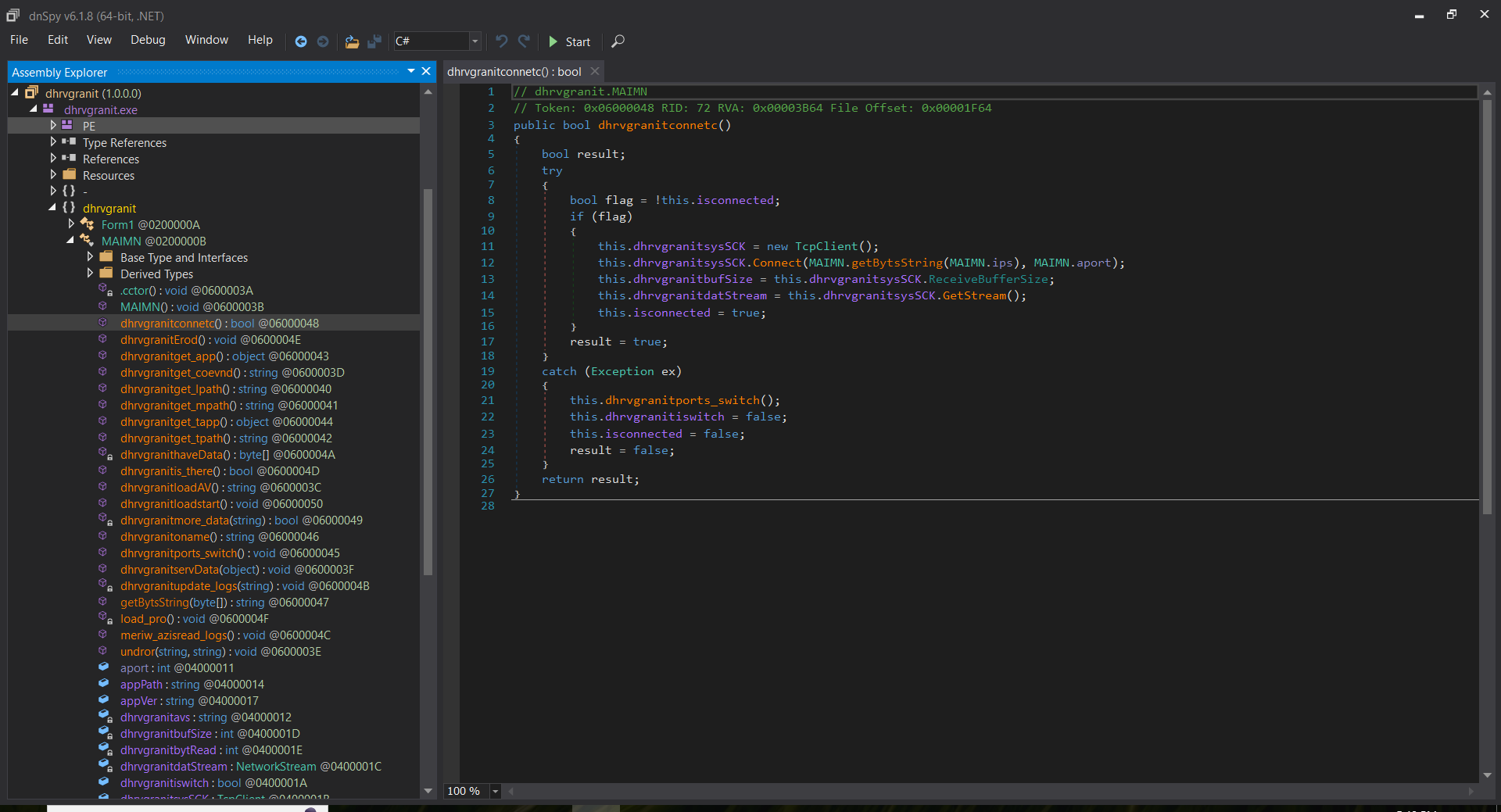


Fig 9. dhrvgranitconnetc() function for establishing the remote connection

The **dhrvgranitports\_switch()** function is crucial in ensuring continuous communication with the C2 server. When a connection attempt fails, this function iterates through the available ports and attempts to reconnect using a different one. This mechanism enhances the malware’s resilience by allowing it to bypass network restrictions or firewall rules that may block specific ports. By dynamically switching between ports, the malware increases its chances of maintaining a persistent connection to the attacker’s server.

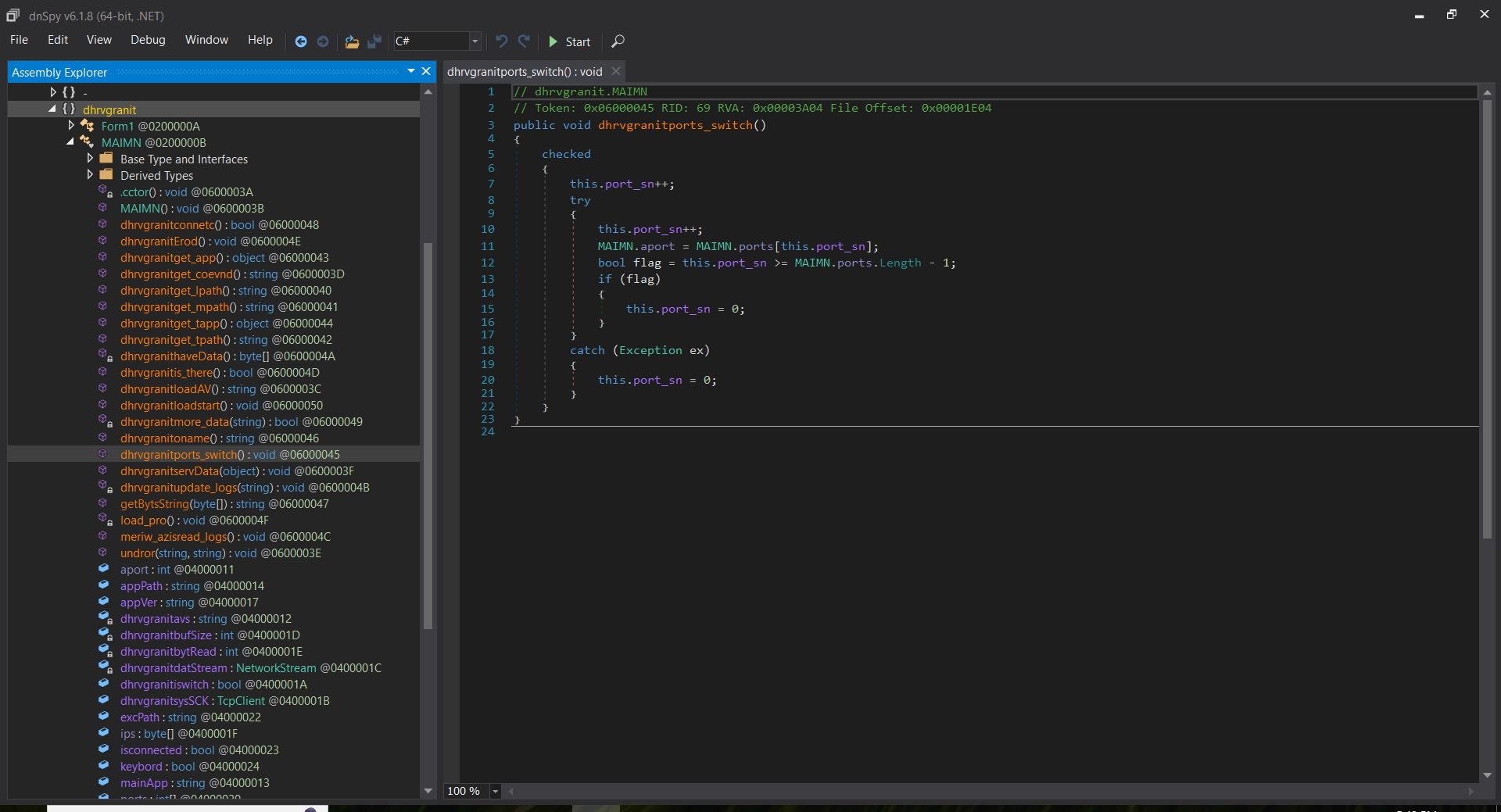


Fig 10. dhrvgranitports\_switch() function for continuous communication with the C2 server

## How malware does reconnaissance on the victim’s system

The malware collects system information by scanning active processes and transmitting the data to a remote server. The function **dhrvgranitloadAV()** is responsible for retrieving a list of running processes on the victim’s machine using **Process.GetProcesses()**. These process names are stored as a string, with each entry separated by **#** to facilitate data parsing.

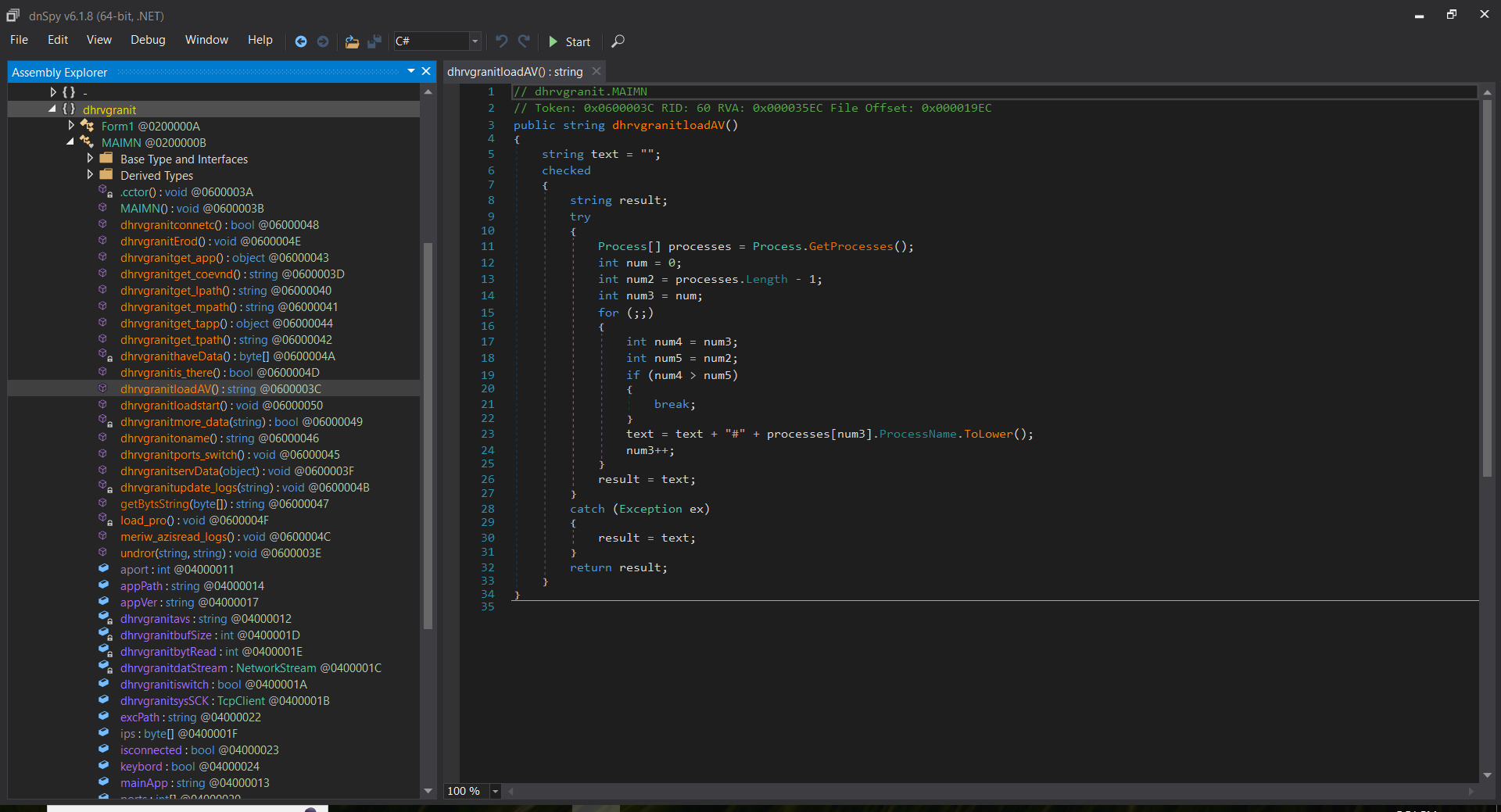


Fig 11. dhrvgranitloadAV() function for retrieving a list of running processes

After gathering this information, the malware uses the **dhrvgranitmore\_data()** function to encode the collected data into bytes before sending it to the attacker’s server. This function utilizes a **NetworkStream** object (**dhrvgranitdatStream**) to establish communication and transmit the data. By continuously monitoring and reporting system activity, the malware helps the attacker analyze the victim’s system, evade security measures, and maintain control.

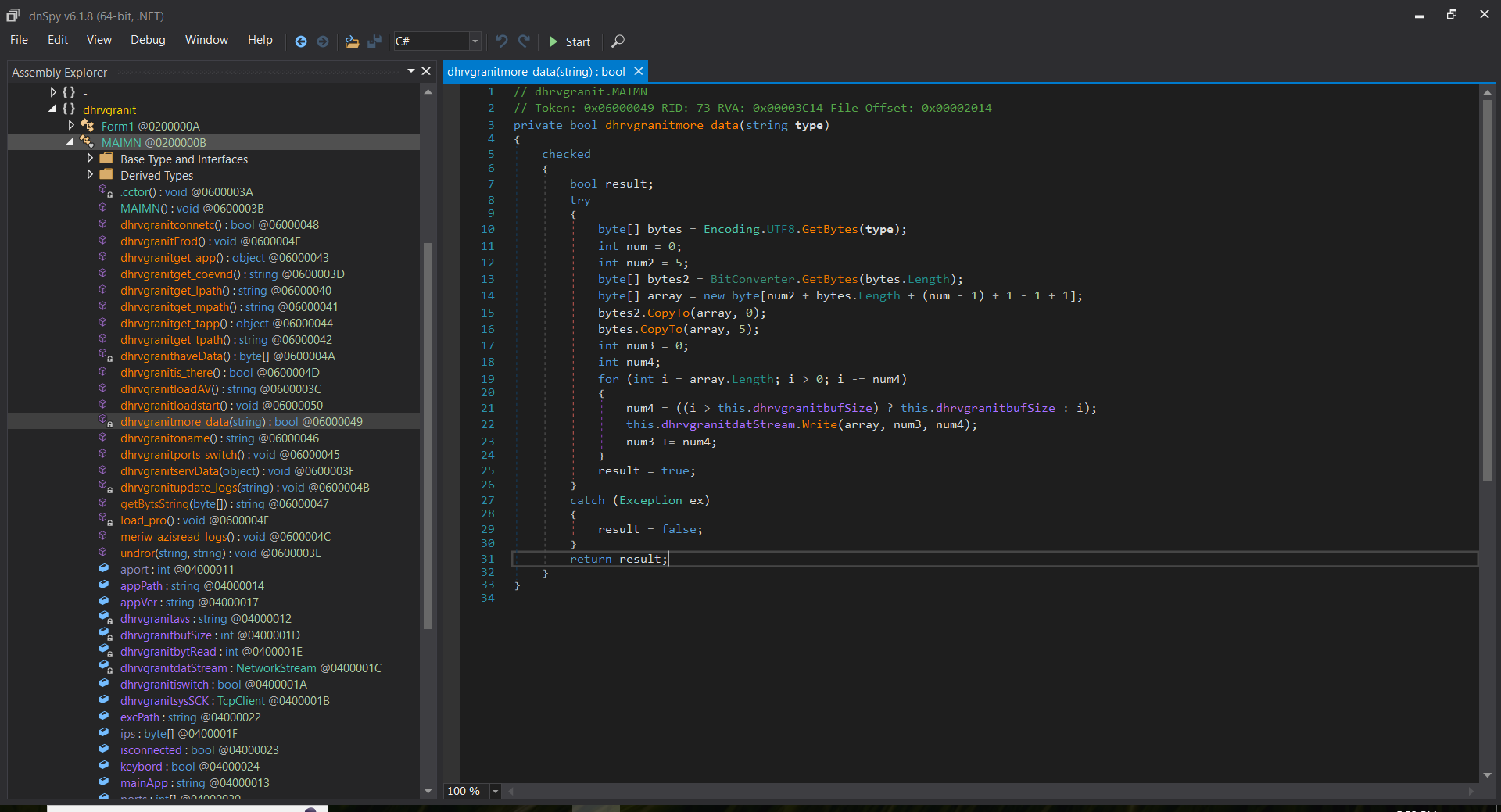


Fig 12. dhrvgranitmore\_data() function to encode the collected data into bytes

## Download and execute another malware from the remote server

The malware downloads and executes additional malicious payloads from its remote server, allowing the attacker to deploy further threats. Several functions handle the retrieval, storage, extraction, and execution of these payloads.

The function **dhrvgranithaveData()** downloads the payload from the remote server. It first determines the size of the payload by reading the first 5 bytes, then allocates and stores the entire payload in a byte array for further exploitation.

A screenshot of a computer program

Description automatically generated

Fig 13. dhrvgranithaveData() function for downloading the payload from the remote server

Once the malware successfully downloads the payload, **dhrvgranitupdate\_logs()** ensures the file is properly stored on the disk by creating a specific directory if it does not exist. If the payload is compressed, **undror()** is responsible for extracting its contents before deleting the source archive.

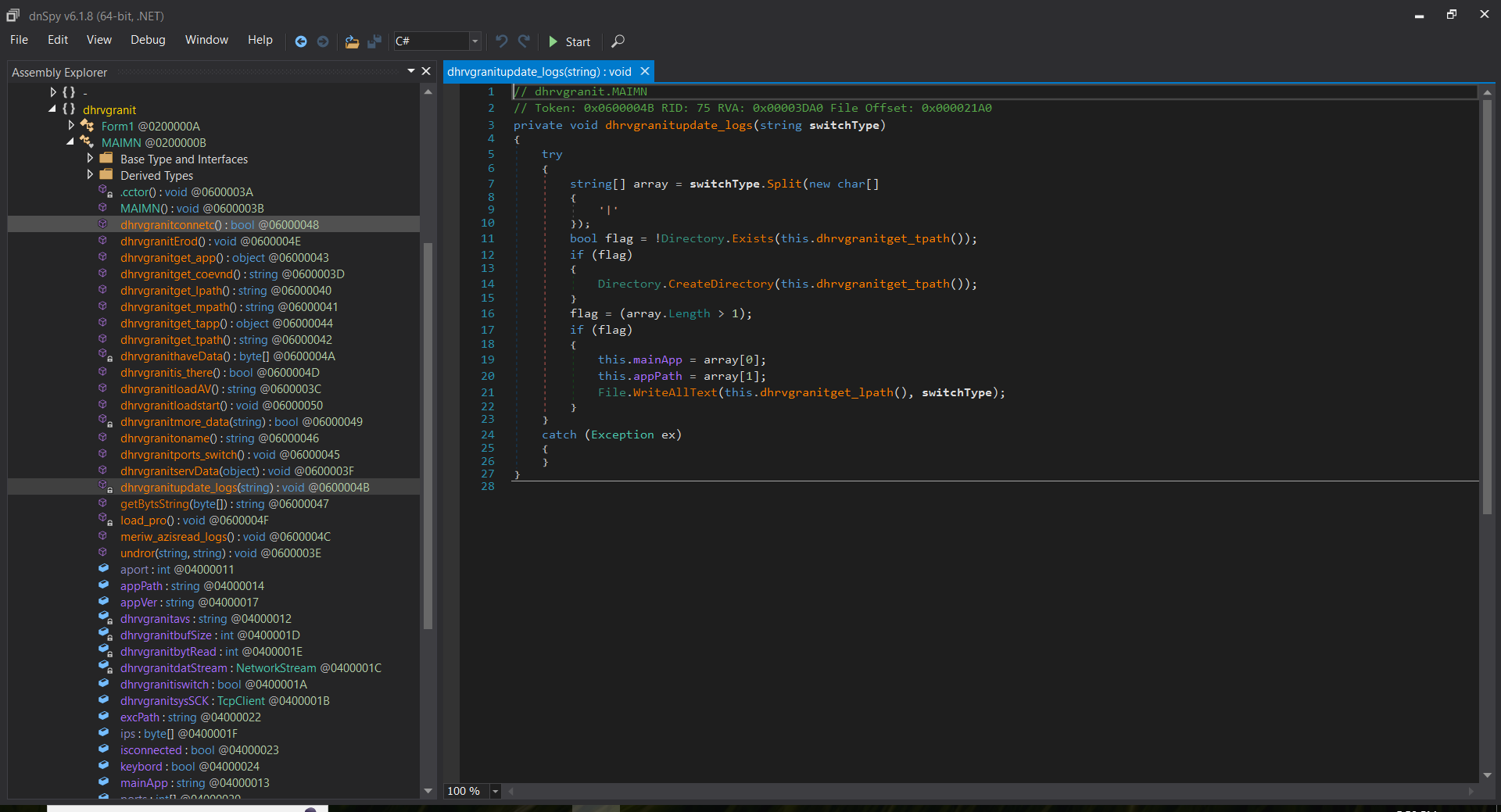


Fig 14. dhrvgranitupdate\_logs() function for ensuring the file is properly stored on the disk

A screenshot of a computer program

Description automatically generated

Fig 15. undror() function for extracting its contents before deleting the source archive

To execute the payload, the malware calls **load\_pro()**, which creates a new process to run the malicious file. The entire process of downloading, storing, extracting, and executing the malicious payload is coordinated by **dhrvgranitErod()**. This function processes commands from the remote server, downloads additional payloads when instructed, and executes them using **load\_pro().**

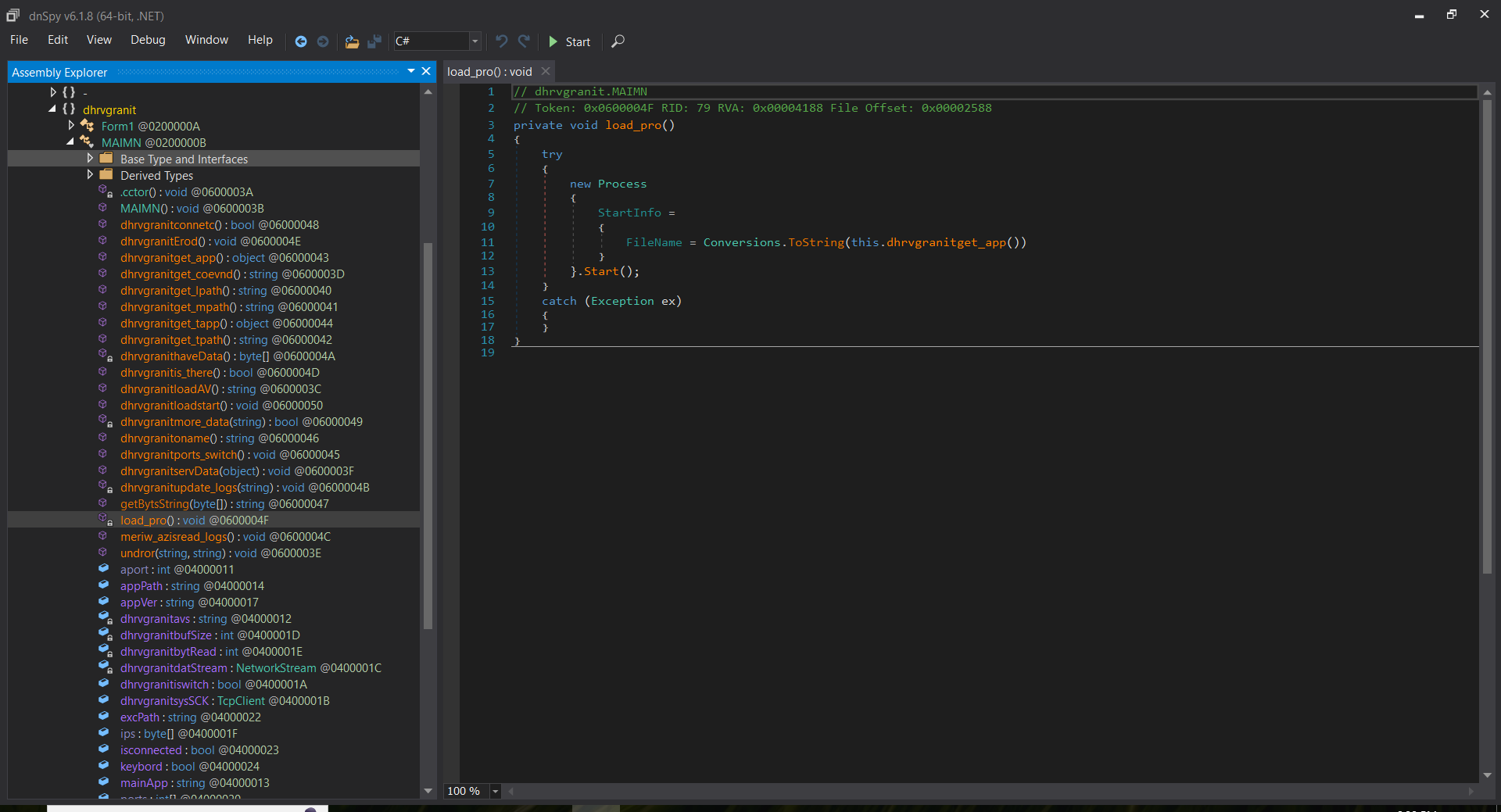


Fig 16. load\_pro() function for creating a new process to run the malicious file

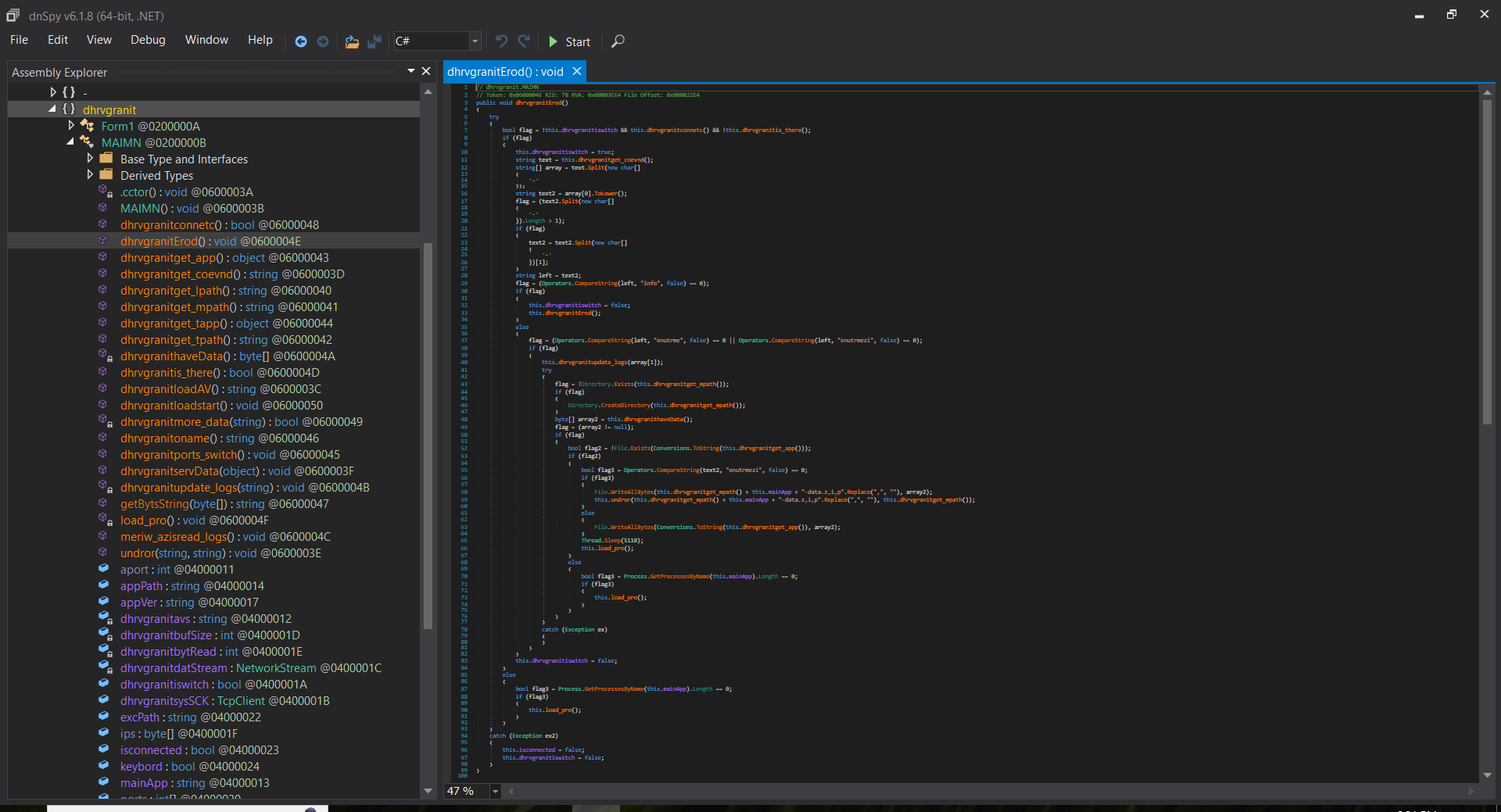


Fig 17. dhrvgranitErod() function for downloading, storing, extracting, and executing the malicious payload

# Analysis methods

To thoroughly examine the multi-stage malware, we utilized a combination of Static Analysis, and Threat Intelligence and Reputation Analysis. The following tools and techniques were applied to analyze the different stages of the attack:

## Static Analysis

Static analysis was performed to inspect the malware without execution, using tools such as **DiE.exe** to determine hashes, file types, entropy, and obfuscation techniques. This confirmed that lab7\_1.doc contained embedded VBA macros and that lab9\_2.exe was a .NET-based Portable Executable (PE32) file. *(Please Refer to Figures 3 and 5 for a more detailed analysis)*

The **olevba.py** tool from oletools was used to extract and deobfuscate the VBA macro code from **vbaProject.bin**, revealing functions responsible for extracting and executing dhrvgranit.exe and hiding payloads within a shape object (**Text Box 2**).

A screenshot of a computer program

Description automatically generated

Fig 18. Macro code (VBA script) that extracts the contents of a malicious executable file from an object in the document

Additionally, **dnSpy.exe** was employed to decompile the .NET-based malware (lab9\_2.exe), exposing key functions such as **dhrvgranitconnetc()** and **collectSystemInfo(),** which established connections to the C2 server at 104.129.27.131. *(Please Refer to Figure 7 for a more detailed analysis)*

## Threat Intelligence and Reputation Analysis

VirusTotal was used to assess the reputation of the malware by analyzing the hash values of **lab7\_1.doc** and **lab9\_2.exe**, which were extracted using the DiE.exe tool. This tool provided detailed behavioural reports and indicators of compromise (IOCs), including associated IP addresses and file hashes, helping to identify the malware’s potential threats and reach.

The figure below shows the result that the file is malicious when I enter the sha256 hash of the **lab9\_2.exe** file into VirusTotal.

A screenshot of a computer

Description automatically generated

Fig 19. Analyzing the hash value of lab9\_2.exe using VirusTotal

# Conclusion

This report provides a comprehensive analysis of the malware by examining both Stage 1 and Stage 2 to uncover its behavior and functionalities. The Stage 1 analysis details the initial execution process, explaining how the dropped payload is delivered and executed to set the next stage for further exploitation. The Stage 2 analysis provides deeper insights into the malware’s capabilities, particularly its ability to communicate with a remote server, conduct reconnaissance, and extract sensitive information from the victim’s system. Additionally, it explores the key functions the malware employs to download and execute additional malicious components from its remote server. These findings highlight the malware’s ability to expand its reach and maintain persistent communication. Furthermore, by leveraging multiple malware analysis techniques – including static analysis with DiE.exe tool and dnSpy tool, as well as threat intelligence from VirusTotal – this report offers an in-depth understanding of the malware’s functionality, behavior, and overall impact.

# Scripts or automated techniques used for analysis

In this section, we documented the Python script used to extract the payload from the malicious Microsoft Word document.

***import re***

***import os***

***print("Converting strings to PE...")***

***try:***

***# Open the XML document containing embedded malicious payload***

***with open("lab7\_1/word/document.xml", "r") as file:***

***content = file.read()***

***print("Opened lab7\_1/word/document.xml")***

***# Extract hexadecimal-encoded PE file content***

***match = re.search(r'77 90[0-9 ]+', content) # Looking for PE magic number "MZ"***

***if match:***

***hex\_string = match.group(0)***

***hex\_values = hex\_string.split(" ")***

***# Convert extracted hex values to a byte array***

***byte\_data = bytearray(int(value, 16) for value in hex\_values)***

***# Convert byte array to bytes***

***payload\_bytes = bytes(byte\_data)***

***print("Converted from strings to PE successfully, writing to file...")***

***# Write the extracted payload to a file***

***output\_path = os.path.join(os.getcwd(), "payload")***

***with open(output\_path, 'wb') as writer:***

***writer.write(payload\_bytes)***

***print(f"Payload successfully written at: {output\_path}")***

***else:***

***raise Exception("Hexadecimal pattern not found in document.xml.")***

***except Exception as e:***

***print(f"Error: {e}")***

The figure below shows the result of the payload file extracted from the malicious document:

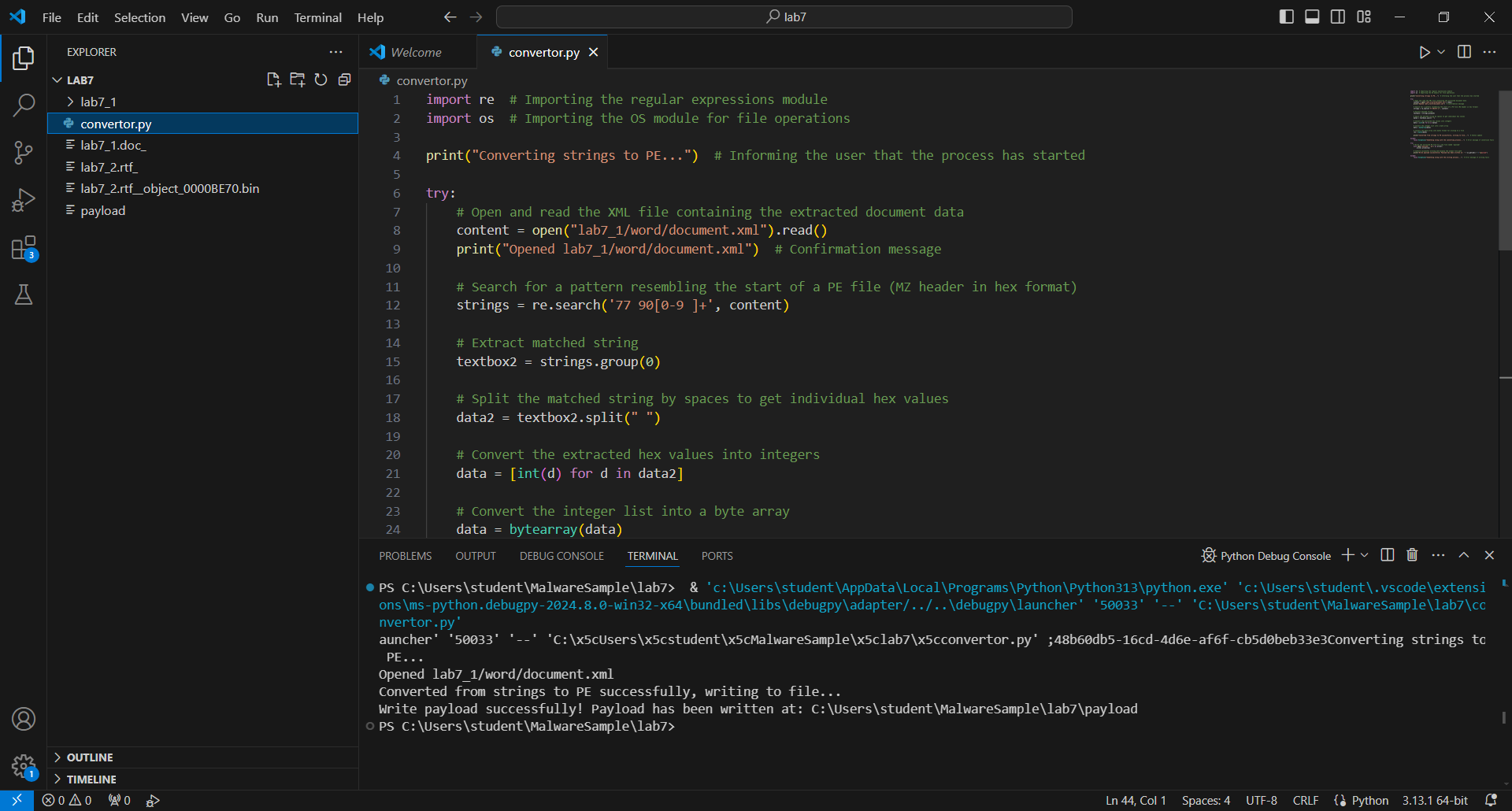




Fig 20. The result of the executed Python script to extract the payload from the malicious document

We used the DiE.exe tool to find the md5 hash of this payload: **4ad5b2369a9f8dfaf32c3a31e92a35d7**

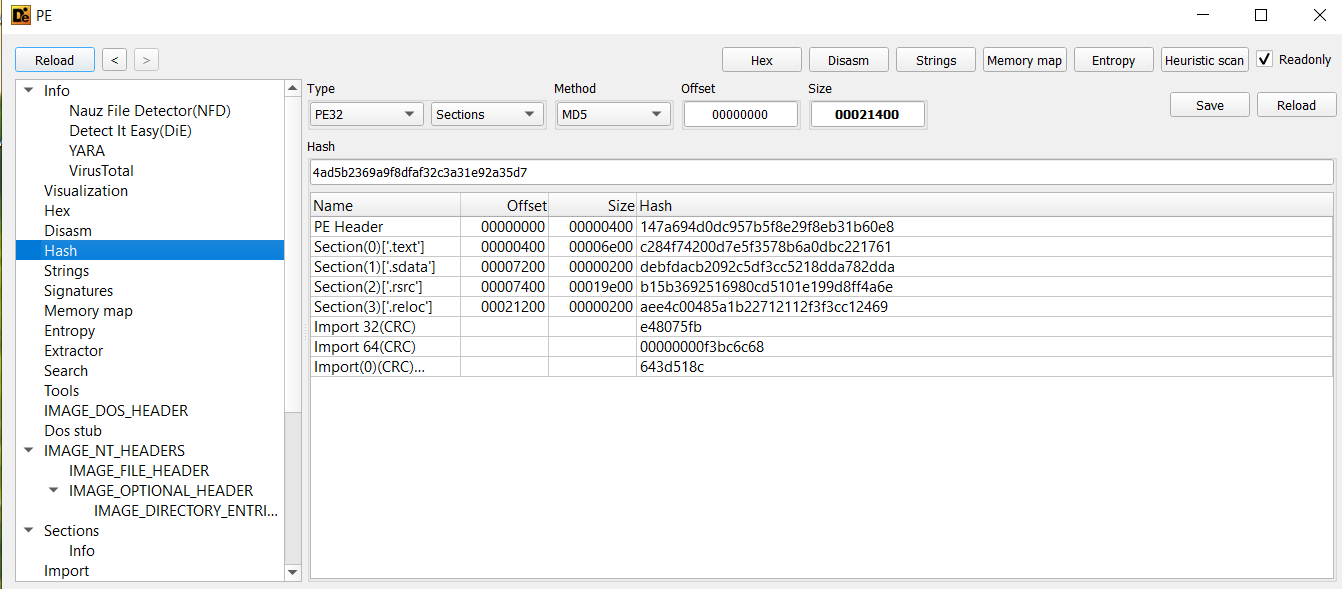


Fig 21. Using the DiE.exe tool to determine the hash of the executed payload

# Yara rules

The following Yara rules below are matching each stage of the attack [1]:

## Stage 1

***rule Stage1\_VBA\_Macro\_Execution {***

***meta:***

***description = “Detects suspicious VBA macro execution in Microsoft Office document”***

***author = “Tran Anh Thu Pham”***

***date = “2025-2-4”***

***reference = “Stage 1”***

***category = “Initial Access”***

***strings:***

***// Common VBA function names used in macro-based malware***

***$macro1 = “AutoOpen” nocase wide***

***$macro2 = “Document\_Open” nocase wide***

***$macro3 = “Workbook\_Open” nocase wide***

***$macro4 = “Shell.Application” nocase wide***

***$macro5 = “WScript.Shell” nocase wide***

***$macro6 = “CreateObject” nocase wide***

***$macro7 = “cmd.exe /c” nocase wide***

***$macro8 = “powershell” nocase wide***

***// Suspicious encoded PowerShell commands***

***$encoded\_ps1 = “powershell -e” nocase wide***

***$encoded\_ps1 = “Invoke-Expression” nocase wide***

***$encoded\_ps1 = “FromBase64String” nocase wide***

***// Potential payload delivery URLs***

***$url = /https?:\/\/[a-zA-Z0-9\_\/\.\-]+/ nocase wide***

***condition:***

***Any of ($macro\*) or any of ($encoded\_ps\*) or $url***

***}***

## Stage 2

***import “pe”***

***import “math”***

***rule Stage2\_Payload\_Execution {***

***meta:***

***description = “Detect malicious PE payload execution and C2 communication”***

***author = “Tran Anh Thu Pham”***

***date = “2025-2-4”***

***reference = “Stage 2”***

***category = “Execution and Persistence”***

***strings:***

***// Detect suspicious function names or variables used in malware***

***// Since most of the function for stage 2 has “dhrvgranit” strings, we use this string to filter out the malware***

***$func1 = “dhrvgranit” nocase wide***

***$func2 = “load\_pro” nocase wide***

***$func3 = “new\_ob\_data” nocase wide***

***$func4 = “run\_malware” nocase wide***

***$func5 = “execute\_payload” nocase wide***

***// Detect command execution patterns***

***cmd\_exec1 = “cmd.exe /c” nocase wide***

***cmd\_exec2 = “powershell” nocase wide***

***cmd\_exec3 = “rundll32.xe” nocase wide***

***cmd\_exec4 = “schtasks.exe /create” nocase wide***

***cmd\_exec5 = “reg add” nocase wide***

***// Detects C2 communication patterns***

***$c2\_url1 = /https?:\/\/[a-zA-Z0-9\_\/\.\-]+/ nocase wide***

***$c2\_url2 = "POST /gate.php" nocase wide***

***$c2\_url3 = "GET /payload.bin" nocase wide***

***condition:***

***pe.is\_pe and (any of ($func\*) or any of ($cmd\_exec\*) or any of ($c2\_url\*))***

***}***

# Indicators of Compromise (IoC)

Here are the hashes of the malicious files and the malicious IP address we analyzed in the sections above using the DiE.exe tool and dnSpy tool:

## md5 hashes

* lab7\_1.doc file: **02bfa75cc9a8fe54828b39440a86fca7**

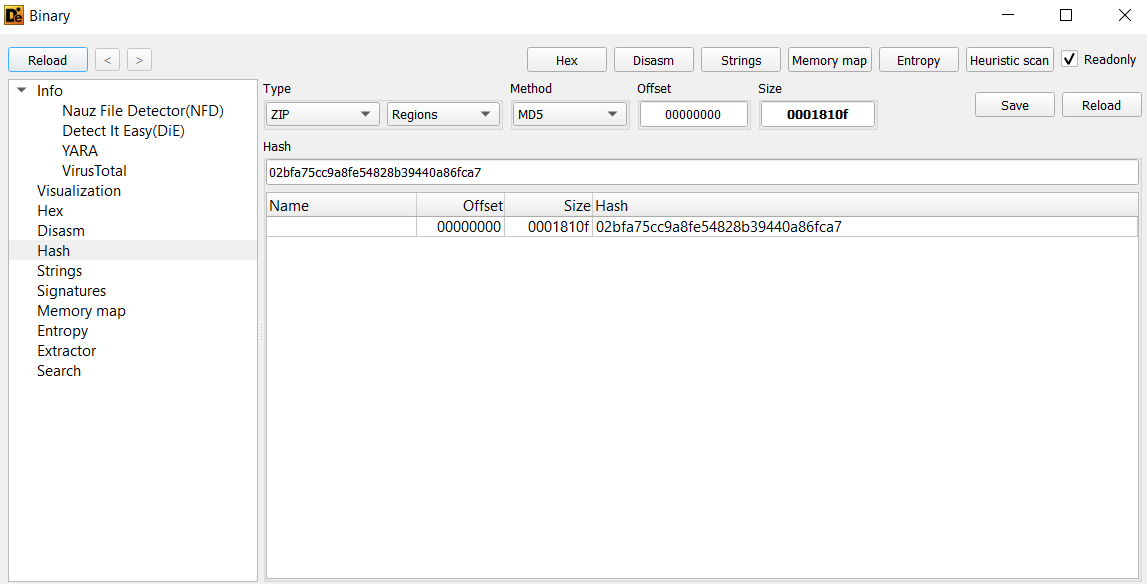


Fig 22. Md5 hash of lab7\_1.doc file

* lab9\_2.exe\_ file: **4ad5b2369a9f8dfaf32c3a31e92a35d7**

A screenshot of a computer

Description automatically generated

Fig 23. Md5 hash of lab9\_2.exe\_ file

## sha1 hashes

* lab7\_1.doc file: **03fc473eb4bfa10a5844862160acc491bf432a8c**

A screenshot of a computer

Description automatically generated

Fig 24. Sha1 hash of lab7\_1.doc file

* lab9\_2.exe\_ file: 04dc3b29b463467535a69bf2deea8edc779bfc0a

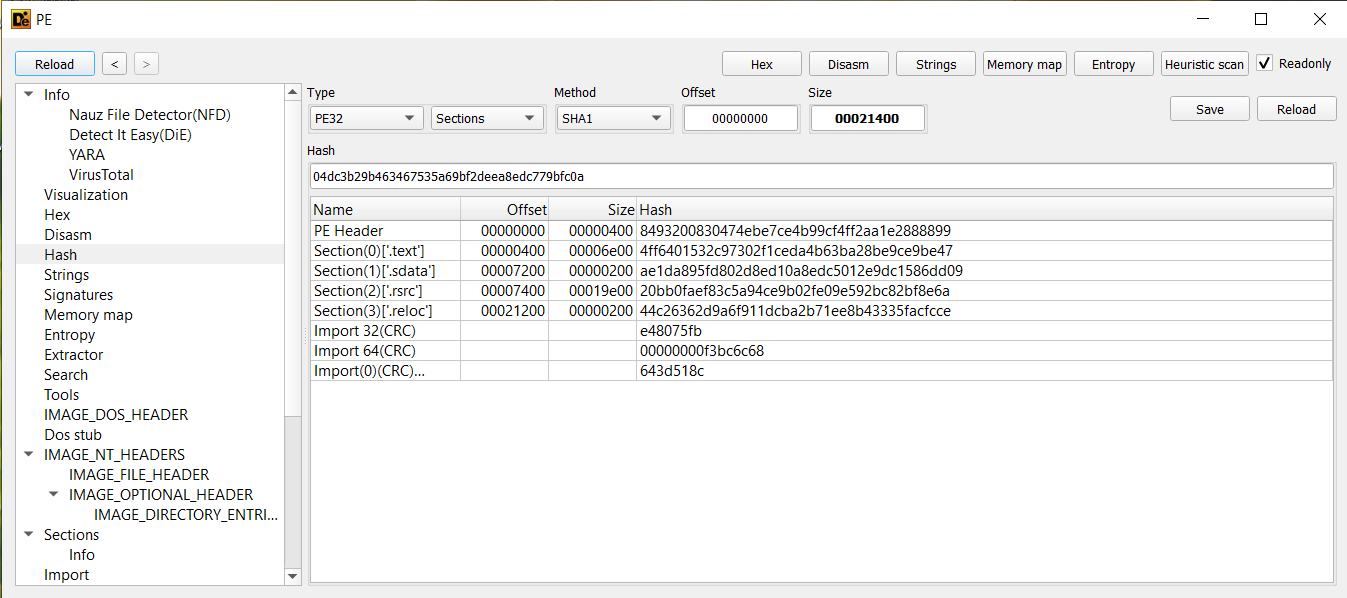


Fig 25. Sha1 hash of lab9\_2.exe\_ file

## sha256 hashes

* lab7\_1.doc file: **f72272d5be19156af6f1d15b5d61356dd0f0fe3ccc5c1b6ca1cc781b6e141b4f**

A screenshot of a computer

Description automatically generated

Fig 26. Sha256 hash of lab7\_1.doc file

* lab9\_2.exe\_ file: **2a03ba547e97964ba9780fe7a6f80b3c07c811f4c9f770359ae61941a219411b**

A screenshot of a computer

Description automatically generated

Fig 27. Sha256 hash of lab9\_2.exe\_ file

## Malicious IP addresses or URL

**104.129.27.131:** The malware stores this IP address in the ips variable as a byte array to evade detection. Once decoded using ASCII values, it resolves to this address, likely used for command and control (C2) or data exfiltration. *(Please Refer to Figure 7 for a more detailed analysis)*

# MITTRE ATT&CK

The following table details the MITRE ATT&CK techniques leveraged in this attack, along with explanations of how they were utilized [4].

|  |  |  |
| --- | --- | --- |
| **Technique ID** | **Technique Name** | **Notes** |
| T1566.001 | Phishing: Spearphishing Attachment | The adversary initiated the attack through a targeted spear phishing email containing a malicious Microsoft Office document. This document was crafted to appear legitimate, often mimicking an official communication to increase the likelihood of execution. Once the user opened the document, malicious macros were executed, initiating the attack chain. |
| T1204.002 | User Execution: Malicious File | The attack relied on social engineering to convince the user to open the malicious Microsoft Office document. The document contained embedded VBA macros, which, when enabled by the user, triggered the execution of the payload. This allowed the adversary to establish initial access and execute further malicious commands on the victim's system. |
| T1059.005 | Command and Scripting Interpreter: Visual Basic | The VBA macro embedded within the document executed Visual Basic commands, which were used to automate the initial execution of malicious scripts. These scripts connected to a remote command-and-control (C2) server, retrieved additional payloads, and executed system commands, effectively escalating the attack. |
| T1137.001 | Office Application Startup: Office Template Macros | To maintain persistence, the adversary used malicious macros embedded within Office templates (.dotm, .xltm, .potm files). Once the user opened an affected Office application, the macro would execute automatically, ensuring continued control over the compromised system even after a reboot. |
| T1564.007 | Hide Artifacts: VBA Stomping | To evade detection by antivirus solutions and security monitoring tools, the attacker employed VBA stomping. This technique involves replacing the human-readable VBA source code with non-readable compiled p-code, allowing the document to bypass static analysis while still executing malicious macros when opened. |
| T1057 | Process Discovery | After execution, the malware utilized the dhrvgranitloadAV() function to enumerate running processes on the victim's system. This function was used to identify security software, antivirus programs, or monitoring tools that could detect or block the attack. Based on the results, the malware could adapt its behavior to avoid detection. |
| T1571 | Non-Standard Port | To communicate with the adversary’s command-and-control (C2) server, the malware used non-standard TCP ports (e.g., 4889, 6878, 12483, 18861, 20184). By avoiding commonly monitored ports like 80 (HTTP) or 443 (HTTPS), the malware aimed to bypass traditional network security defenses and remain undetected. |
| T1027 | Obfuscated Files or Information | To avoid detection by security tools, the malicious payloads were compressed into ZIP files. Once extracted, the malware was executed using the dhrvgranitErod() function, which unpacked and deployed the malicious executable. This obfuscation technique helped the attacker evade antivirus scanning and file integrity checks.  *(To be more specific, please view Figure 28 below)* |

A screenshot of a computer program

Description automatically generated



Fig 28. The malicious payloads were compressed into ZIP files To avoid detection

# Advanced Yara: Writing a YARA rule for heuristic malware detection

The following YARA rules are below for detecting potentially malicious PE (Portable Executable) files. The ruleset contains specialized rules that analyze various aspects of PE files, including File header characteristics, Version information in optional headers, Size relationships between PE components, Memory layout configurations, Miscellaneous PE header values, Section characteristics, Section entropy analysis, and Resource section attributes [2], [3].

***// advanced.yara***

***// Created: 2025-02-04***

***// Created by Tran Anh Thu Pham***

***import "pe"***

***import "math"***

***// Rule: FileHeaderIndicators***

***// Purpose: Check for suspicious PE file header characteristics that may indicate malware***

***// Checks:***

***// - Suspicious timestamp ranges***

***// - Unusual number of sections (less than 1 or more than 8)***

***// - Presence of symbol table (uncommon in modern PE files)***

***// - Suspicious byte order flags***

***// - Stripped relocation information***

***rule FileHeaderIndicators {***

***meta:***

***description = "Analyzes PE file headers for potential malware indicators and suspicious modifications"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.is\_pe and (***

***(pe.timestamp < 694224000) or (pe.timestamp > 1325376000) or  // Check for suspicious timestamps***

***(pe.number\_of\_sections < 1) or (pe.number\_of\_sections > 8) or  // Check for abnormal section counts***

***(pe.pointer\_to\_symbol\_table > 0) or                           // Check for symbol table presence***

***(pe.characteristics & pe.BYTES\_REVERSED\_HI != 0) or           // Check for suspicious byte ordering***

***(pe.characteristics & pe.BYTES\_REVERSED\_LO != 0) or***

***(pe.characteristics & pe.RELOCS\_STRIPPED == 1)                // Check for stripped relocations***

***)***

***}***

***// Rule: PEOptionalHeaderVersionAttributes***

***// Purpose: Verify version information in PE optional header against known good values***

***// Checks:***

***// - Linker version combinations***

***// - OS version combinations***

***// - Image version combinations***

***// Triggers if versions don't match common legitimate values***

***rule PEOptionalHeaderVersionAttributes {***

***meta:***

***description = "Identifies potentially malicious PE files by analyzing version information in the optional header"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.is\_pe and***

***// Check for invalid linker versions***

***not (***

***pe.linker\_version.major == 8 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 9 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 7 and pe.linker\_version.minor == 10 or***

***pe.linker\_version.major == 10 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 6 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 5 and pe.linker\_version.minor == 10 or***

***pe.linker\_version.major == 5 and pe.linker\_version.minor == 12 or***

***pe.linker\_version.major == 5 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 2 and pe.linker\_version.minor == 25 or***

***pe.linker\_version.major == 2 and pe.linker\_version.minor == 0 or***

***pe.linker\_version.major == 3 and pe.linker\_version.minor == 10***

***)***

***and***

***// Check for invalid OS versions***

***not (***

***pe.os\_version.major == 4 and pe.os\_version.minor == 0 or***

***pe.os\_version.major == 6 and pe.os\_version.minor == 1 or***

***pe.os\_version.major == 5 and pe.os\_version.minor == 0 or***

***pe.os\_version.major == 5 and pe.os\_version.minor == 1 or***

***pe.os\_version.major == 5 and pe.os\_version.minor == 2 or***

***pe.os\_version.major == 6 and pe.os\_version.minor == 0***

***)***

***and***

***// Check for invalid image versions***

***not (***

***pe.image\_version.major == 0 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 6 and pe.image\_version.minor == 1 or***

***pe.image\_version.major == 5 and pe.image\_version.minor == 1 or***

***pe.image\_version.major == 5 and pe.image\_version.minor == 2 or***

***pe.image\_version.major == 6 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 8 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 5 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 1 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 9 and pe.image\_version.minor == 0 or***

***pe.image\_version.major == 4 and pe.image\_version.minor == 0***

***)***

***}***

***// Rule: PEOptionalHeaderSizeAttributes***

***// Purpose: Detect suspicious size relationships between PE components***

***// Checks for unrealistic ratios between:***

***// - Code size vs file size***

***// - Initialized data size vs file size***

***// - Uninitialized data size vs file size***

***// - Image size vs file size***

***// - Headers size vs file size***

***rule PEOptionalHeaderSizeAttributes {***

***meta:***

***description = "Evaluates PE file size attributes to detect anomalies and potential tampering"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.is\_pe and (***

***(pe.size\_of\_code \ filesize > 1) or                          // Code size shouldn't exceed file size***

***(pe.size\_of\_initialized\_data \ filesize > 3) or              // Initialized data size check***

***(pe.size\_of\_uninitialized\_data \ filesize > 1) or           // Uninitialized data size check***

***(pe.size\_of\_image \ filesize > 8) or                        // Image size ratio check***

***(pe.size\_of\_headers \ filesize > 0)                         // Headers size check***

***)***

***}***

***// Rule: PEOptionalHeaderLocationAttributes***

***// Purpose: Check for suspicious memory layout configurations***

***// Examines ratios between:***

***// - Code base address vs file size***

***// - Data base address vs file size***

***// - Entry point vs file size***

***rule PEOptionalHeaderLocationAttributes {***

***meta:***

***description = "Examines PE file memory layout attributes for suspicious configurations"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.is\_pe and (***

***(pe.base\_of\_code \ filesize > 2) or                         // Suspicious code base location***

***(pe.base\_of\_data \ filesize > 3) or                         // Suspicious data base location***

***(pe.entry\_point \ filesize > 2)                             // Suspicious entry point location***

***)***

***}***

***// Rule: PEOptionalHeaderMiscellaneousAttributes***

***// Purpose: Check for unusual PE header values that are typically zero in legitimate files***

***// Examines:***

***// - Loader flags (should be 0)***

***// - Number of RVA and sizes (should be 16)***

***// - Win32 version value (should be 0)***

***rule PEOptionalHeaderMiscellaneousAttributes {***

***meta:***

***description = "Detects unusual PE header configurations that may indicate malware"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.is\_pe and (***

***pe.loader\_flags != 0 or                                      // Loader flags should be zero***

***pe.number\_of\_rva\_and\_sizes != 16 or                         // Should always be 16 for normal PE files***

***pe.win32\_version\_value != 0                                 // Should be zero in normal PE files***

***)***

***}***

***// Rule: PESectionRules***

***// Purpose: Identify suspicious section characteristics***

***// Checks for:***

***// - Empty sections***

***// - Unrealistic virtual vs raw size ratios***

***// - Invalid line number pointers***

***// - Suspicious section characteristics***

***// - Abnormal entropy values***

***rule PESectionRules {***

***meta:***

***description = "Identifies suspicious PE section characteristics often associated with malware"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***for any i in ( 0 .. pe.number\_of\_sections - 1 ) : (***

***(pe.sections[i].raw\_data\_size == 0) or                      // Check for empty sections***

***(pe.sections[i].virtual\_size \ pe.sections[i].raw\_data\_size) > 10 or  // Check for suspicious size ratios***

***(pe.sections[i].virtual\_size < pe.sections[i].raw\_data\_size) or      // Virtual size should be >= raw size***

***(pe.sections[i].pointer\_to\_line\_numbers != 0) or            // Should be zero in modern PE files***

***(pe.sections[i].characteristics & 0x00000080 != 0) or       // Check for suspicious characteristics***

***(pe.sections[i].characteristics & 0x10000000 != 0) or***

***(math.entropy(0, filesize) < 1) or (math.entropy(0, filesize) > 7)  // Check for abnormal entropy***

***)***

***}***

***// Rule: PESectionEntropyRules***

***// Purpose: Detect potential encryption or packing through entropy analysis***

***// Checks:***

***// - High entropy virtual address flag***

***// - Section-specific entropy values (> 6.9 indicates potential encryption/packing)***

***rule PESectionEntropyRules {***

***meta:***

***description = "Analyzes PE section entropy to detect potential encryption or packing"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***pe.HIGH\_ENTROPY\_VA != 0 or for any i in (0..pe.number\_of\_sections - 1) : (***

***math.entropy(pe.sections[i].raw\_data\_offset, pe.sections[i].raw\_data\_size) > 6.9  // Check for high entropy sections***

***)***

***}***

***// Rule: RSRCRules***

***// Purpose: Check for suspicious resource section characteristics***

***// Examines:***

***// - Resource language settings that may indicate automated malware creation***

***rule RSRCRules {***

***meta:***

***description = "Examines PE resource attributes for anomalies typical in malware"***

***author = "Tran Anh Thu Pham"***

***date = "2025-04-02"***

***reference = "https://sansorg.egnyte.com/dl/zdmLYMKnP1"***

***condition:***

***for any resource in pe.resources: (***

***(resource.language & 0xF000) >> 12 == 0                     // Check for suspicious resource language settings***

***)***

***}***

The figures below show the saved results of the executed Advanced Yara Rules file (**advanced.yara**) to a new file named **res.txt**:

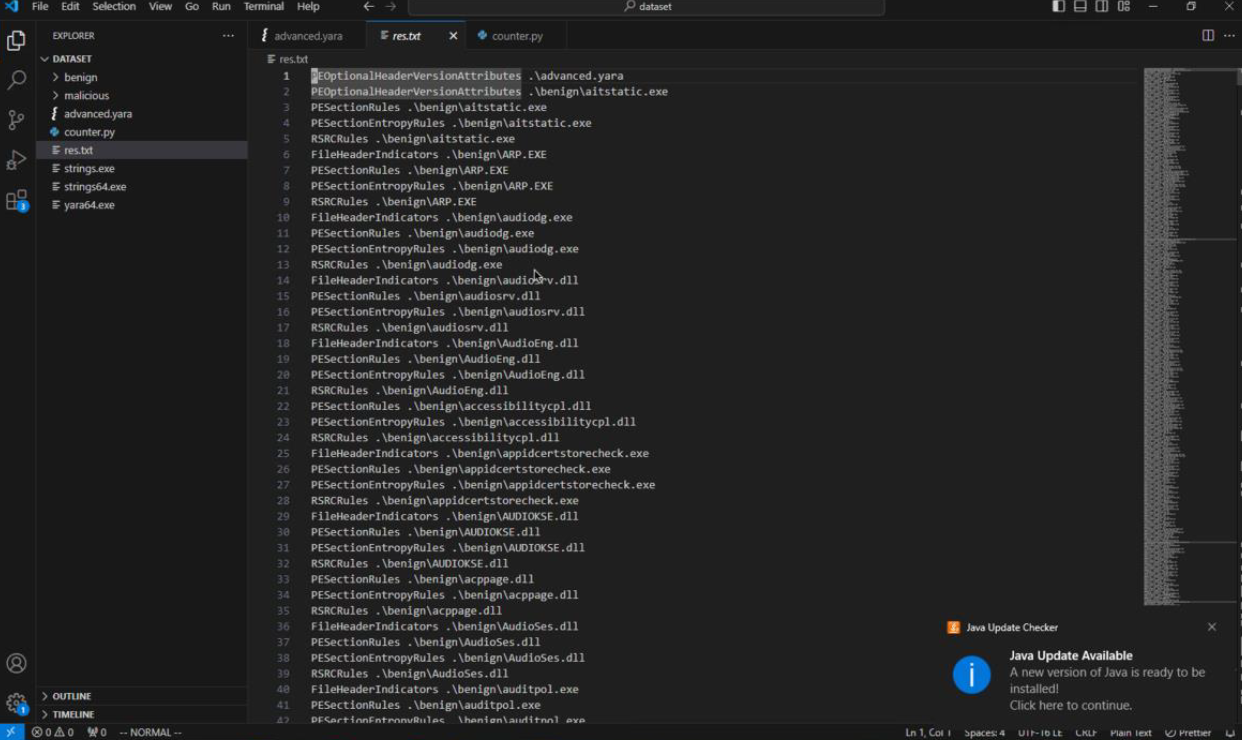


Fig 29. The saved results of the executed Advanced Yara Rules file

# Python Scripting

The following Python script extracts and converts a DEX file from a .bin file found in Android APK malware samples.

***from abc import ABC, abstractmethod # Importing abstract base classes for defining an interface***

***import os # Importing OS module for file operations***

***import base64 # Importing Base64 module for decoding***

***import zlib # Importing zlib module for decompression***

***class IDEXHandler(ABC):***

***"""***

***Abstract class defining the interface for DEX file handling.***

***Any class inheriting from this must implement the handleConvertingToDEX method.***

***"""***

***@abstractmethod***

***def handleConvertingToDEX(self, dex\_name: str) -> str:***

***pass***

***class DEXHandler(IDEXHandler):***

***"""***

***Concrete class that implements DEX file extraction and conversion from a binary file.***

***"""***

***def handleConvertingToDEX(self, dex\_name: str) -> str:***

***try:***

***# Create file path for the output .dex file in the current working directory***

***file\_path = os.path.join(os.path.abspath(os.getcwd()), f"{dex\_name}.dex")***

***print("DEX file path:", file\_path)***

***# Remove existing .dex file if it exists***

***if os.path.exists(file\_path):***

***os.remove(file\_path)***

***# Locate the 'bin' file in the directory and subdirectories***

***bin\_file = None***

***for root, \_, files in os.walk(os.getcwd()):***

***if "bin" in files: # Look for a file named 'bin'***

***bin\_file = os.path.join(root, "bin")***

***break # Stop searching once found***

***# Raise an error if 'bin' file is not found***

***if bin\_file is None:***

***raise FileNotFoundError("Error: 'bin' file not found in current directory or subdirectories.")***

***print("Found bin file:", bin\_file)***

***# Read the 'bin' file and extract its content***

***with open(bin\_file, "rb") as f:***

***f.seek(4) # Skip the first 4 bytes (header data)***

***decompressed = zlib.decompress(f.read()) # Decompress the binary data using zlib***

***decoded = base64.b64decode(decompressed) # Decode the extracted data from Base64***

***# Write the extracted and decoded data to a new .dex file***

***with open(file\_path, "wb") as f:***

***f.write(decoded)***

***# Create a directory named 'a' (possibly for additional extracted files)***

***os.makedirs(os.path.join(os.path.abspath(os.getcwd()), "a"), exist\_ok=True)***

***print("DEX file successfully extracted and saved as:", file\_path)***

***return "Success"***

***except Exception as e:***

***print(f"An error occurred: {str(e)}")***

***return "Error"***

***def main():***

***"""***

***Main function to execute the DEX extraction script.***

***"""***

***print("Starting the transformation of bin file to DEX format...")***

***# Create an instance of DEXHandler***

***dex\_handler: IDEXHandler = DEXHandler()***

***# Convert the 'bin' file into a 'decoded.dex' file***

***result = dex\_handler.handleConvertingToDEX("decoded")***

***# Print the result of the conversion process***

***if result == "Success":***

***print("Successfully transformed the file to DEX!")***

***else:***

***print("An error occurred during the transformation. Please check the process.")***

***# Entry point of the script***

***if \_\_name\_\_ == "\_\_main\_\_":***

***main()***

Here is the result after running the script, a new file named **decoded.dex** is transformed from a bin file.

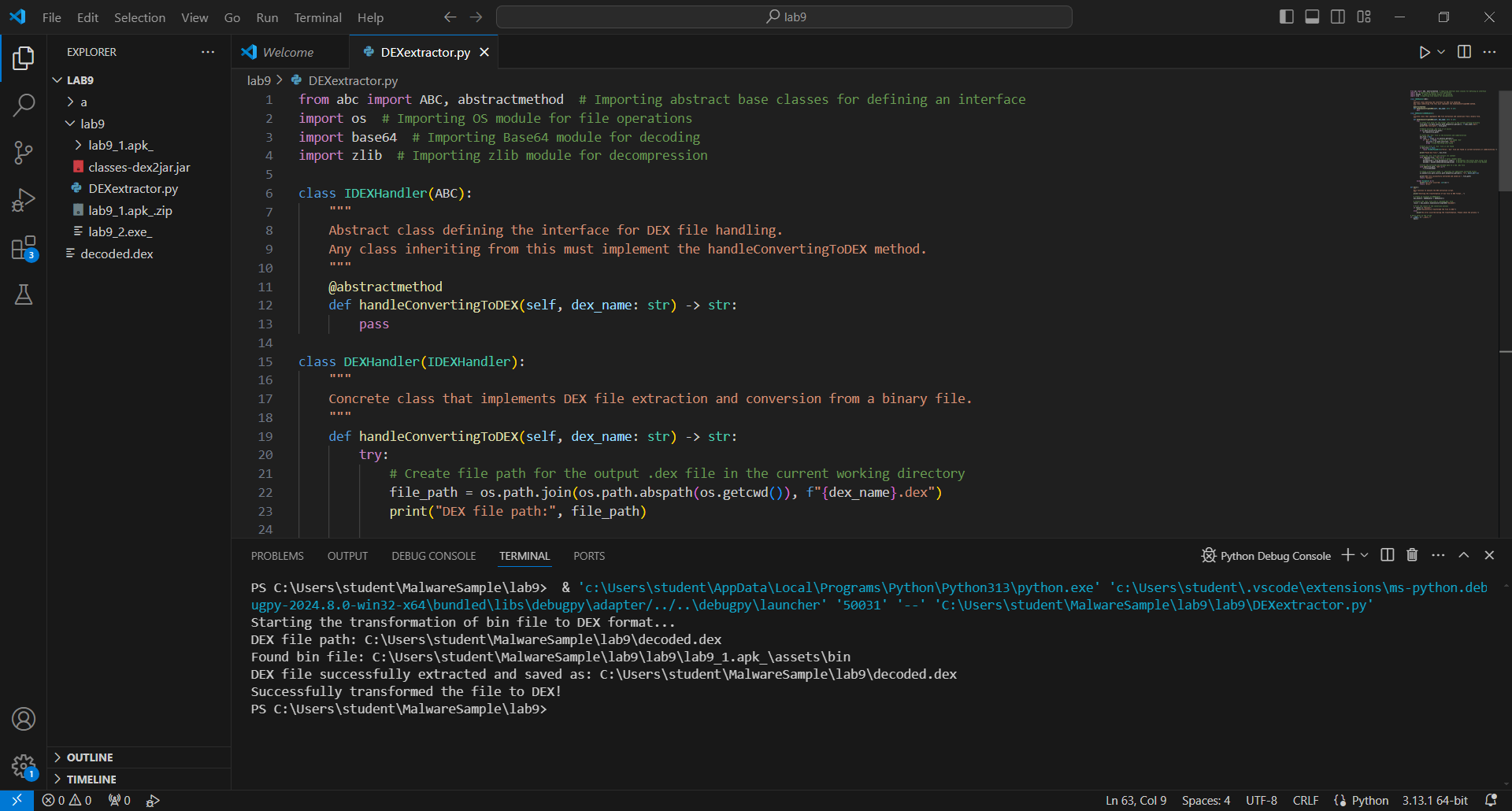


Fig 30. Convert a DEX file from a .bin file found in Android APK malware samples

After running the script, we convert the DEX file to JAR using the dex2jar tool. As a result, a new JAR file is converted with the name **decoded-dex2jar.jar**. The transform command is shown in Figure 31 below.

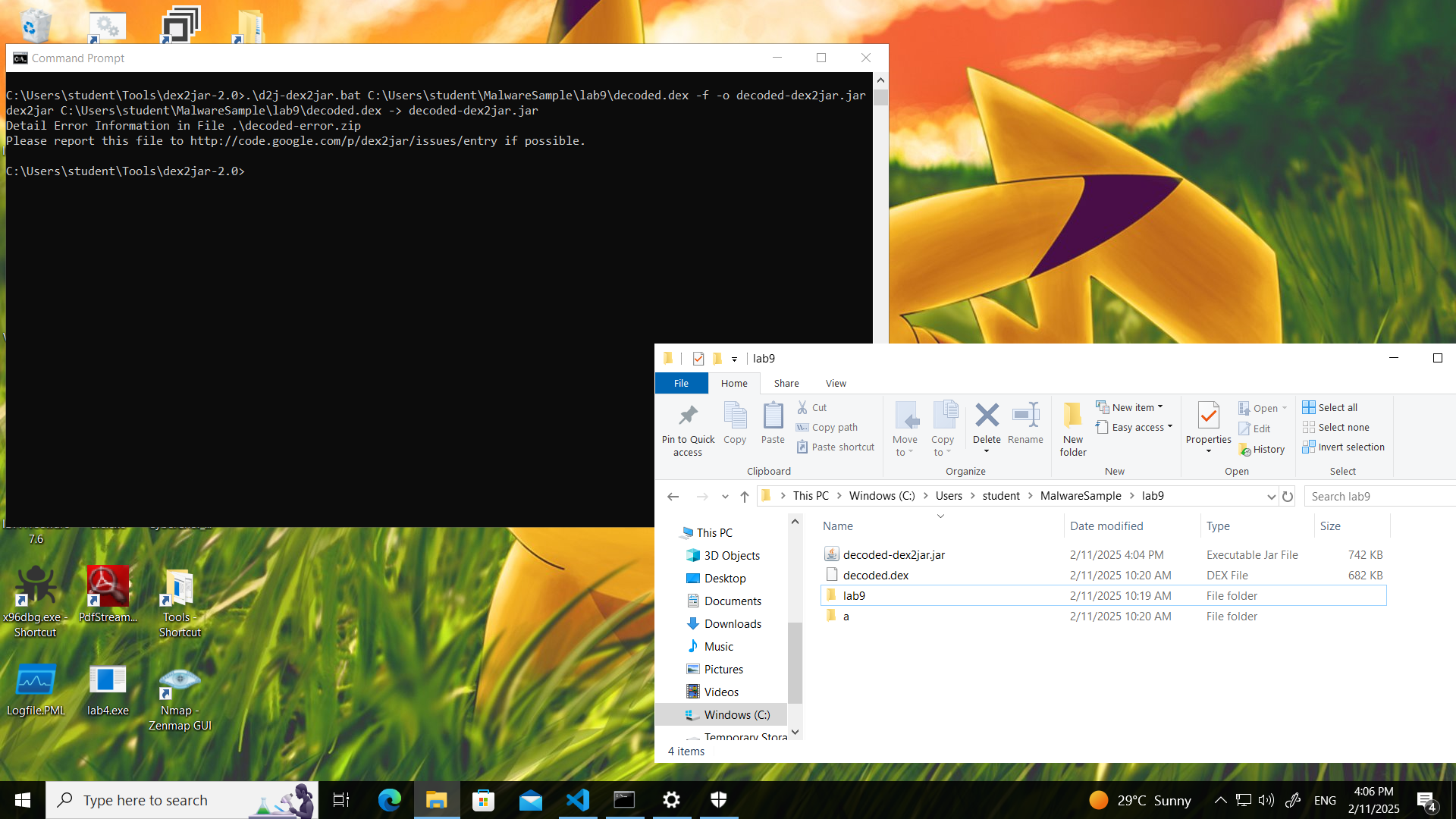


Fig 31. Convert the DEX file to JAR using the dex2jar tool

Finally, the DiE.exe tool was utilized to verify the file type and MD5 hash of the extracted DEX file:

Md5 hash: **af2890a472b85d473faee501337564a9**

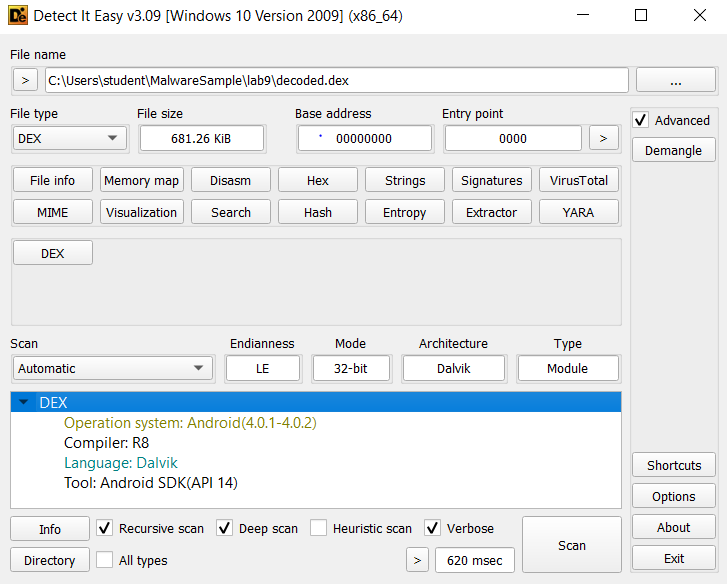


Fig 32. Verify the file type of the extracted DEX file

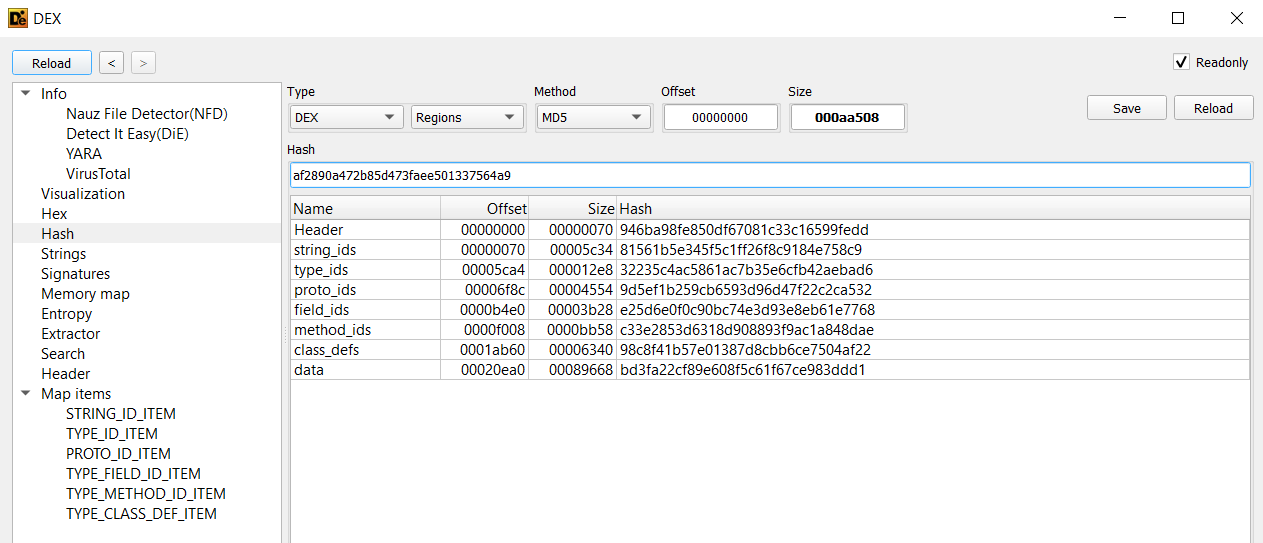


Fig 33. Find the file type and MD5 hash of the extracted DEX file

# References

1. Fox, N. (2021, May 17). *YARA Rules Guide: Learning this Malware Research Tool*. Varonis.com; Varonis. <https://www.varonis.com/blog/yara-rules?utm_source=chatgpt.com>
2. Project. (2022, April 29). GitHub. <https://github.com/Yara-Rules/rules>
3. 262588213843476. (2022, January 17). *Yara file that contains useful rules to detect malicious files*. Gist.

<https://gist.github.com/RachidAZ/d3c469cde5cf2498a451a7b9ba251b2d>

1. MITRE. (2024). MITRE ATT&CKTM. Mitre.org. <https://attack.mitre.org/>

# Reflection

Throughout this assignment, we gained valuable insights into malware analysis using a variety of techniques and tools. We learned how to conduct static analysis with tools like Detect It Easy (DIE), which allowed us to extract essential details such as file types, hash values, and timestamps, which are crucial for threat intelligence analysis. We also explored the functionality of the malware by using dnSpy for reverse engineering .NET components. This provided a deeper understanding of the C2 communication, payload delivery, and the execution flow of the malware. Additionally, by mapping the malware's behavior to the MITRE ATT&CK framework, we were able to strengthen our understanding of adversary tactics and techniques used in real-world cyber attacks. Through writing YARA rules, we learned how to identify patterns and behaviors associated with malicious activities in files and memory, helping to detect malware.

One of the biggest challenges we faced was writing YARA rules, especially the advanced YARA rules. We spent a significant amount of time researching how to structure these rules and ensure they were both effective. Debugging the rules proved difficult as small mistakes in logic or conditions could lead to false results, making it challenging. For instance, one of the most common difficulties when writing YARA rules is failing to account for different character encodings. YARA provides several string modifiers such as ascii, wide, nocase, and fullword to handle variations in the malicious text representation. If these are not used correctly, the rule may fail to detect malicious files.

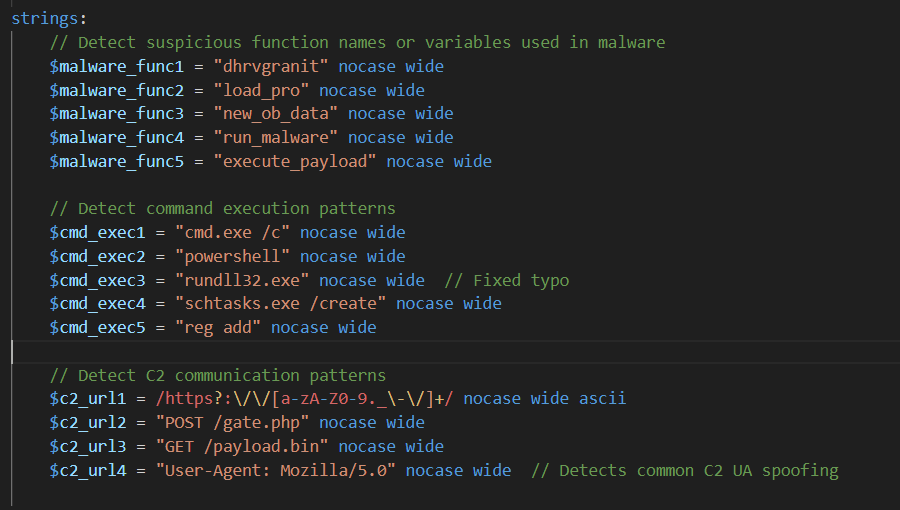


Fig 34. Choosing appropriate string modifiers issue

Additionally, interpreting technical details from the MITRE ATT&CK framework and translating them into a coherent narrative was complex. It required us to carefully map each technique used by the malware to its corresponding real-world actions.

To address these challenges, we conducted thorough research, including reviewing GitHub repositories and documentation, to better understand various YARA rule structures and obfuscation techniques commonly used in malware. All sources are specified in the reference section above. To solve string modifiers, we used **nocase wide** modifier to catches both ASCII and Unicode versions of strings, matches regardless of letter case, For example, the rule $cmd\_exec1 = "cmd.exe /c" nocase wide would match:

* + "cmd.exe /c" (ASCII)
  + "CMD.EXE /C" (ASCII)
  + "Cmd.Exe /c" (wide)
  + "cMd.ExE /C" (wide)

For mapping the MITRE ATT&CK framework, we meticulously verified each technique’s real-world matrix, cross-referencing multiple sources to ensure the accuracy of our findings and to match the analyzed malware in this assignment.

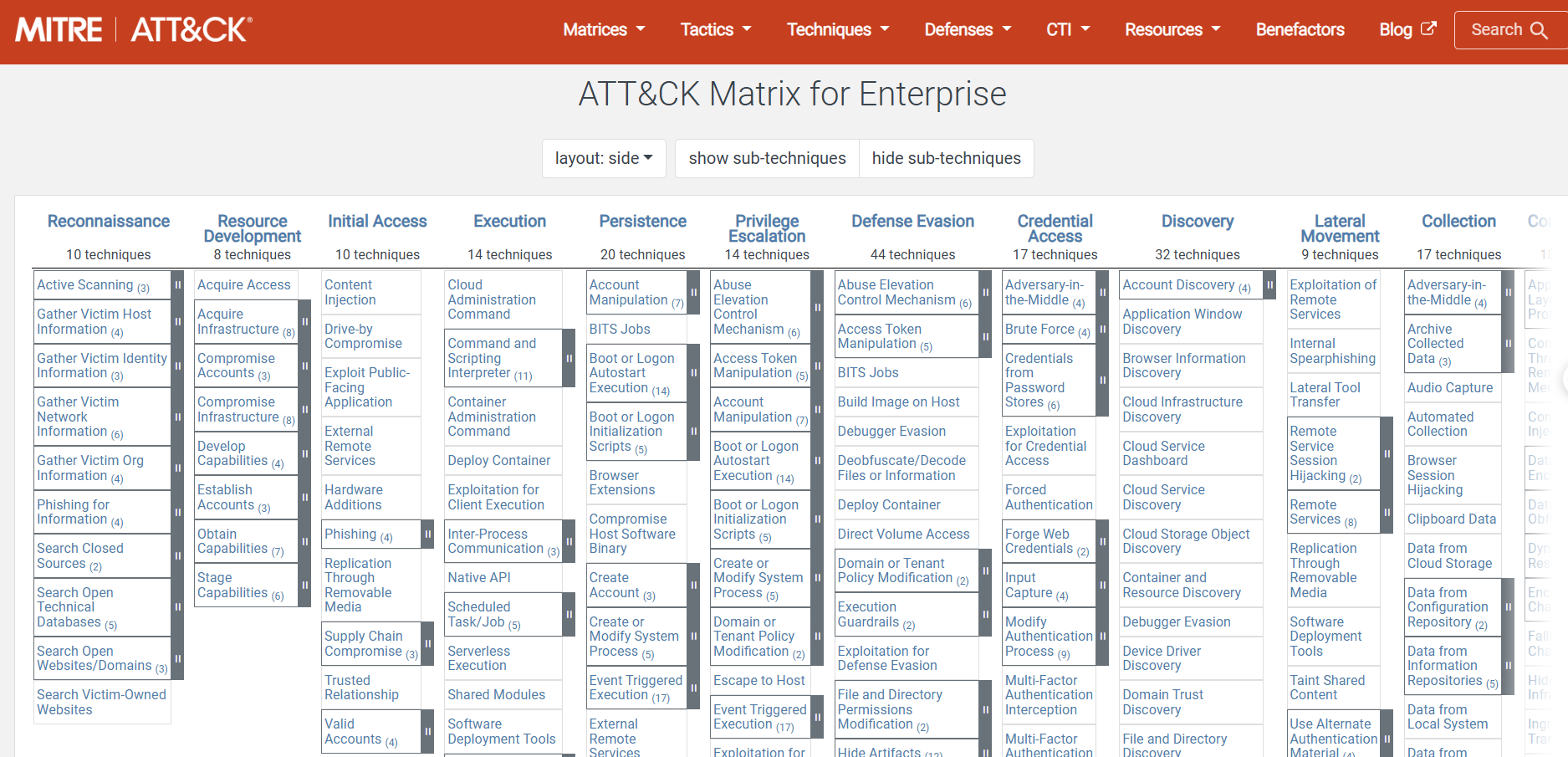


Fig 35. Finding the appropriate attack matrix to match the analyzed malware