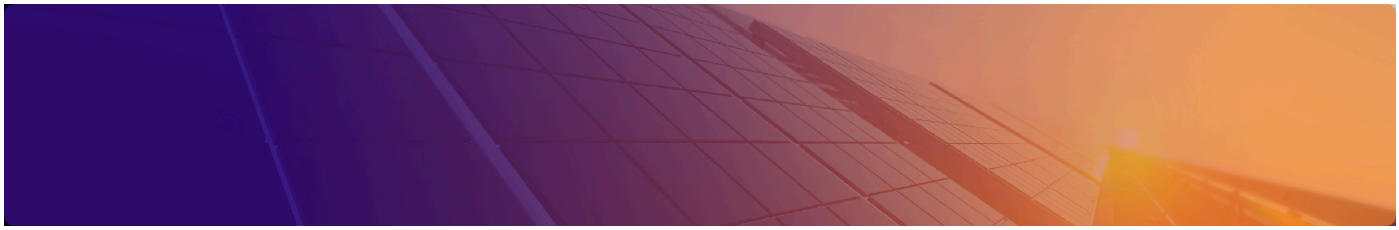
CHAMELEON#NET: A Deep Dive into Multi-Stage .NET Malware Leveraging Reflective Loading and Custom Decryption for Stealthy Operations

securonix.com/blog/chameleonnet-a-deep-dive-into-multi-stage-net-malware-leveraging-reflective-loading-and-custom-decryption-for-stealthy-operations

November 4, 2025



Threat Research

Threat Research

Securonix Threat Research Security Advisory

By Securonix Threat Research: Shikha Sangwan

October 08, 2025

tldr:

The Securonix Threat Research team has analyzed a sophisticated malspam campaign distributing RAT through **DarkTortilla** malware.

This campaign begins with a phishing email that tricks users into downloading a .BZ2 archive, initiating a multi-stage infection chain. The initial payload is a heavily obfuscated JavaScript file that acts as a dropper, leading to the execution of a complex VB.NET loader. This loader uses advanced reflection and a custom conditional XOR cipher to decrypt and execute its final payload, the **FormBook RAT**, entirely in memory. The campaign showcases layered evasion, from script obfuscation to fileless execution, ultimately aiming to establish long-term persistence and control over the compromised system.

Introduction

We've been tracking a malspam campaign delivering DarkTortilla, a highly modular and evasive .NET malware that has been active since 2015. Known for its ability to deploy a wide range of commodity malware such as AgentTesla, AsyncRAT, and various information stealers, DarkTortilla remains a persistent threat.

This particular campaign starts with a targeted phishing email, luring victims in the National Social Security Sector into downloading what appears to be a benign archive. However, this action kicks off a complex, multi-stage infection process designed to meticulously unpack and execute its final payload while evading detection. The attack chain leverages obfuscated JavaScript droppers, which in turn deploys a sophisticated VB.NET executable. This executable is the core loader, responsible for decrypting an embedded DLL using a unique, index-based XOR cipher and then reflectively loading it into memory.

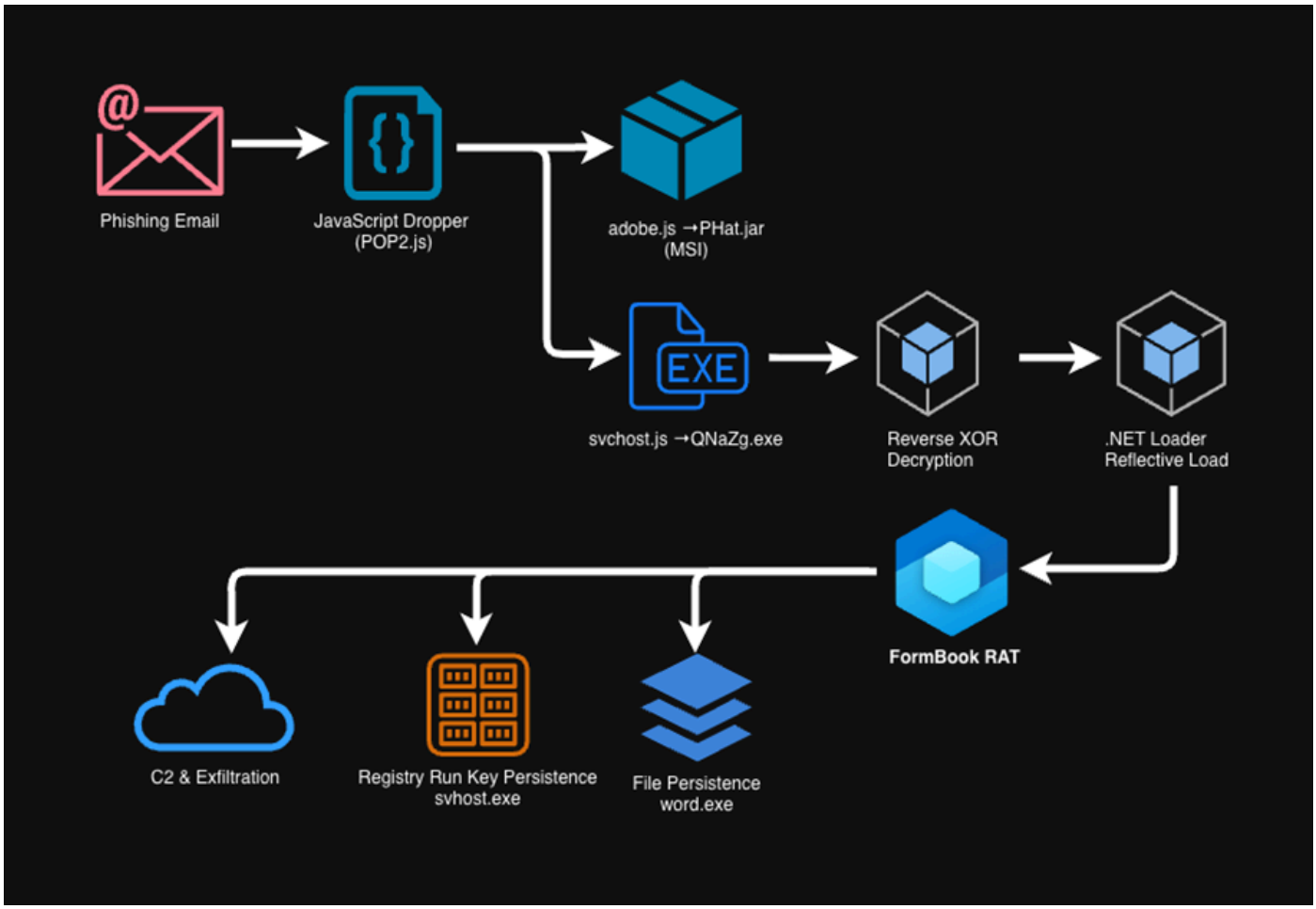
The final payload, identified as the **FormBook RAT**, establishes persistence through registry run keys and startup folder modifications, disables security measures, and grants the attacker full remote access to the victim's machine. This analysis will break down each stage of the attack, from the initial JavaScript dropper to the final in-memory execution of the FormBook RAT.

Key Findings

- **Initial Access via Malspam:** The campaign begins with phishing emails that lead to the automatic download of a .BZ2 archive containing the initial dropper.
- **Multi-Stage JavaScript Droppers:** The initial POP2.js script is heavily obfuscated and acts as a dropper for two subsequent JavaScript files, adobe.js and svchost.js.
- **Complex .NET Loader:** The second-stage svchost.js script drops a 32-bit VB.NET executable (QNaZg.exe), which serves as the main loader for the final payload. This loader is packed and exhibits high entropy.
- **Custom Decryption Routine:** The loader uses a specific decryption algorithm to unpack its embedded payload. The routine involves reversing a byte array and applying a conditional XOR operation (XORing with 0xC1) only on bytes at even indices.
- **Fileless Execution via Reflection:** The loader makes extensive use of .NET reflection and late-binding to load the decrypted payload (a DLL) directly into memory using AppDomain.Load(byte[]), avoiding dropping the final payload to disk.
- **Modular FormBook RAT:** The final in-memory payload is the FormBook RAT. It decrypts its own AES-encrypted configuration from its resources to guide its behavior.
- **Stealthy Persistence:** The malware achieves persistence by copying itself to AppData\Roaming\word\word.exe and creating a registry Run key (HKCU\…\Run) disguised as svchost.exe.
- **Defense Evasion:** The attack chain uses multiple evasion techniques, including script obfuscation, anti-analysis checks, and time-delay tactics using the ping command.

The Infection Chain

Initial Access: The Malspam Lure



The attack chain begins with a carefully crafted malspam email. These emails direct victims to a fraudulent webmail portal that mimics a legitimate service, prompting them to enter their Social Security (SSA or SSI) institutional credentials. This serves the dual purpose of credential harvesting and acting as a gatekeeper for the payload.

Figure 1 Fake webmail portal used for credential harvesting and payload delivery

Once the user submits their credentials, they trigger the download of a compressed archive of *81__POP1.BZ2*. When the archive is downloaded, the user is already in a task-completion mindset, expecting to see a document (like a PDF or Word file) related to the grant. The file inside the archive, POP2.js, almost certainly continues the deception looking like a text file. The Victim, believing they are simply opening the grant information they were promised, is thus enticed to double-click the file, initiating the infection chain.

This is not just random spam; this is a message that appears to be critical and time sensitive. Once the victim is invested enough to visit the fake portal and enter their credentials, they have been psychologically primed to believe the process is legitimate.

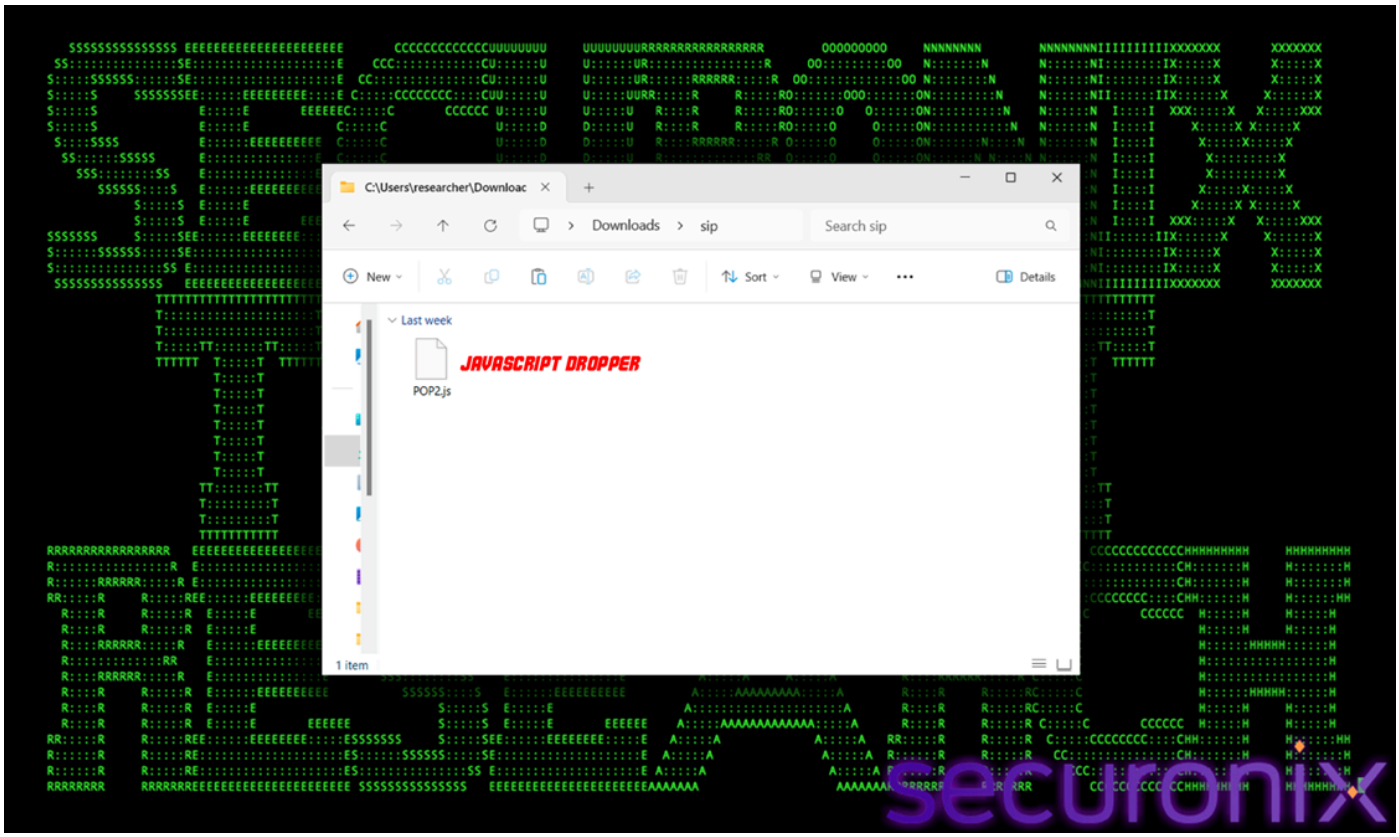


Figure 2 Extracted JavaScript Dropper

Stage 1: The Multi-Stage JavaScript Dropper

When executed, POP2.js acts as a multi-stage dropper. Its code is made nearly unreadable by several obfuscation layers designed to bypass static AV signatures. The script has a lot of noise, variables initialized but never used and meaningless string assignments. It uses String.fromCharCode and arithmetic operations for character generation, which is a classic sign of runtime decoding. Critical strings, like ActiveXObject names, are built character-by-character at runtime. The next-stage payloads are embedded inside the script as large, encoded Base64 strings.

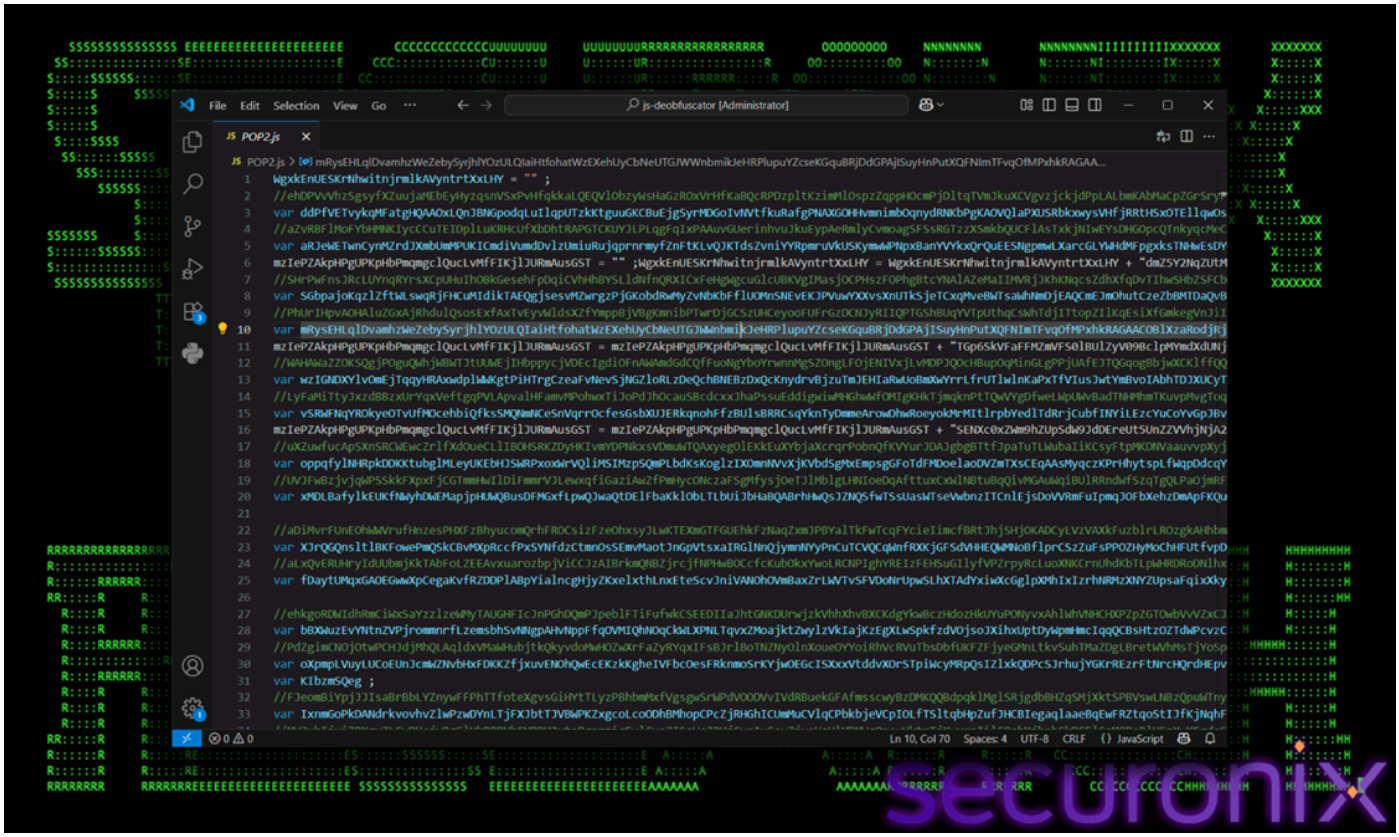


Figure 3 Heavily obfuscated dropper code

After deobfuscation, the script's functionality becomes clear. It uses a series of ActiveXObjects to decode and launch the next stage:

1. **Scripting.FileSystemObject**: Used to get the path to the user's temporary directory (%TEMP%).
2. **Microsoft.XMLDOM**: Used to decode the embedded Base64 strings. It creates an XML element, sets its dataType to **"bin.base64"**, and assigns the Base64 string to its text property. The decoded binary data is then retrieved from the nodeTypedValue property of this XML element. This is a common technique for Base64 decoding in older WSH environments.

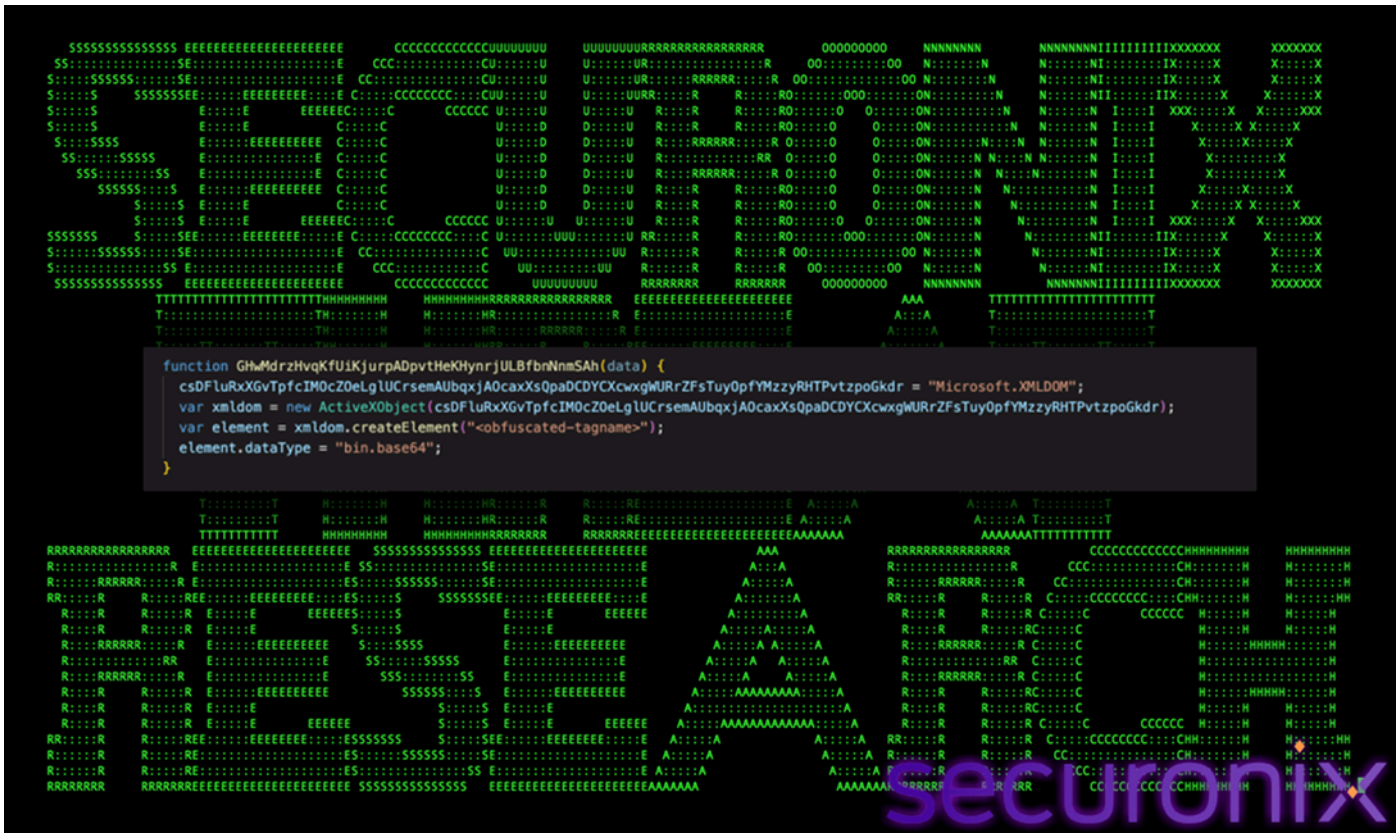


Figure 4 Payload decoding

1. **ADODB.Stream**: This object is used to write the decoded binary data to files on disk.

The JavaScript file functions as a multi-stage dropper. Its primary purpose is to decode two distinct, larger payloads embedded within its own code as Base64 strings. These decoded payloads are then written to the **%TEMP%** directory as **adobe.js** and **svchost.js**, respectively.

Finally, the script leverages **WScript.Shell** to execute both dropped JavaScript files, initiating the next stage of the infection chain.

Figure 5 Second-stage Javascript files dropped into the %TEMP% directory.

Stage 2: Dropping the .NET Loader

The second-stage scripts, adobe.js and svchost.js, function similarly to the first. They each contain another Base64-encoded payload.

> *svchost.js* drops the core component of the next stage: a .NET executable named *QNaZg.exe*, which is the **Darktortilla** strain. It firstly checks the Temp folder path using FileSystemObject and then decodes the embedded payload using MSXML DOM techniques. The decoded binary data is extracted via the “nodeTypedValue” property. Once the binary payload is accessible in memory, the script uses an ADODB.Stream object to write this data out to disk. The stream is initialized, configured for binary operations, and the decoded content is written directly to the stream. Before saving, the stream’s position is reset to ensure the full binary is written from the beginning of the file. The file is then saved to the temporary directory, with the overwrite option enabled to prevent errors if a file with the same name already exists. Finally, the stream is closed to release resources. Then it runs the dropped binary file using WScript.Shell object.
>
> *adobe.js* drops a file named *PHat.jar*, which our analysis revealed to be an MSI installer package. It behaves similarly as svchost.js.

Figure 6 QNaZg.exe and PHat.jar dropped by the secondary scripts

Stage 3: The .NET Loader – Decryption and Reflective Loading

DarkTortilla (QNaZg.exe) is a 32-bit VB.NET executable that serves as the main loader for the final payload. Its high entropy suggests it is either packed or heavily obfuscated. It turns out to be heavily obfuscated (all classes and method names).

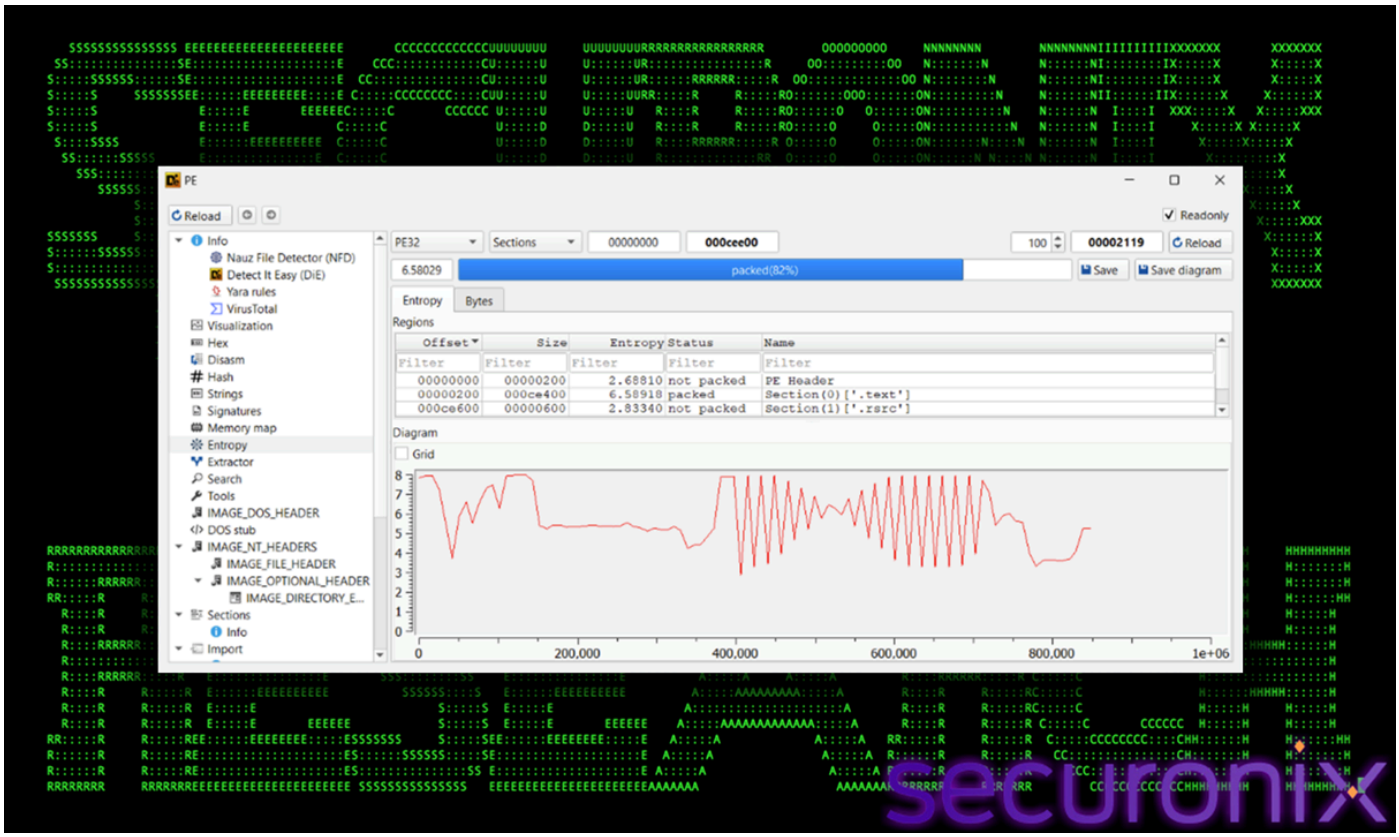


Figure 7 PE analysis of the .NET loader

This loader's primary responsibility is to decrypt and execute an embedded DLL stored within its own resources. This is accomplished through a multi-step process designed to evade analysis. The loader contains a large, hardcoded byte array, which is the encrypted final payload.

To decrypt it, the malware employs a custom, multi-step algorithm:

1. **Reverse:** The entire byte array is reversed using a 'LINQ .Reverse<byte>()' operation.
2. **Conditional XOR:** A custom function is applied to each byte of the reversed array. This function performs a conditional XOR operation: if the byte's index in the new, reversed array is **even**, it is XORed with the key 193 (0xC1). If the index is **odd**, the byte is left unchanged

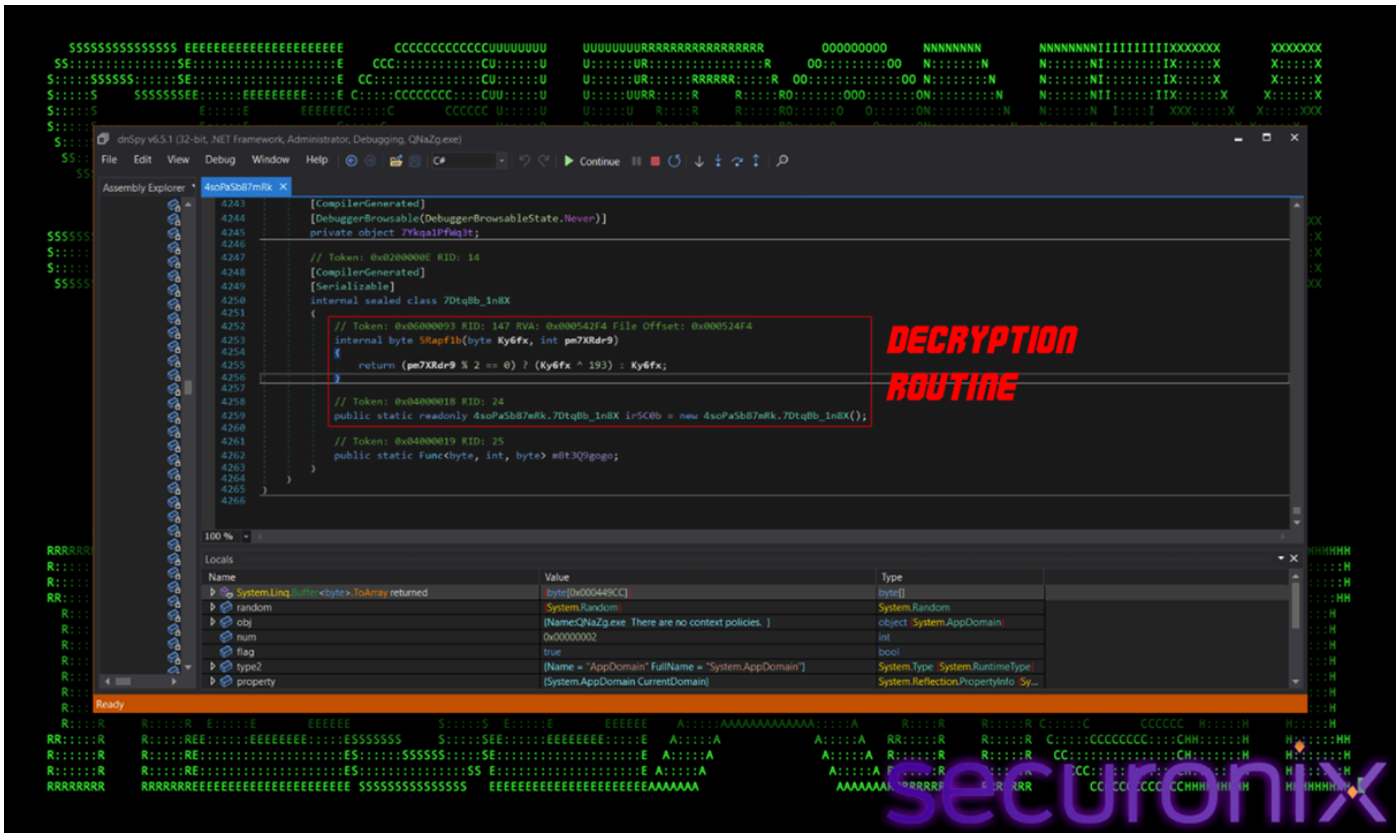


Figure 8 The simple, index-based conditional XOR decryption routine

This straightforward routine effectively decrypts the byte array, revealing a valid PE file in memory, identifiable by the “MZ” header.

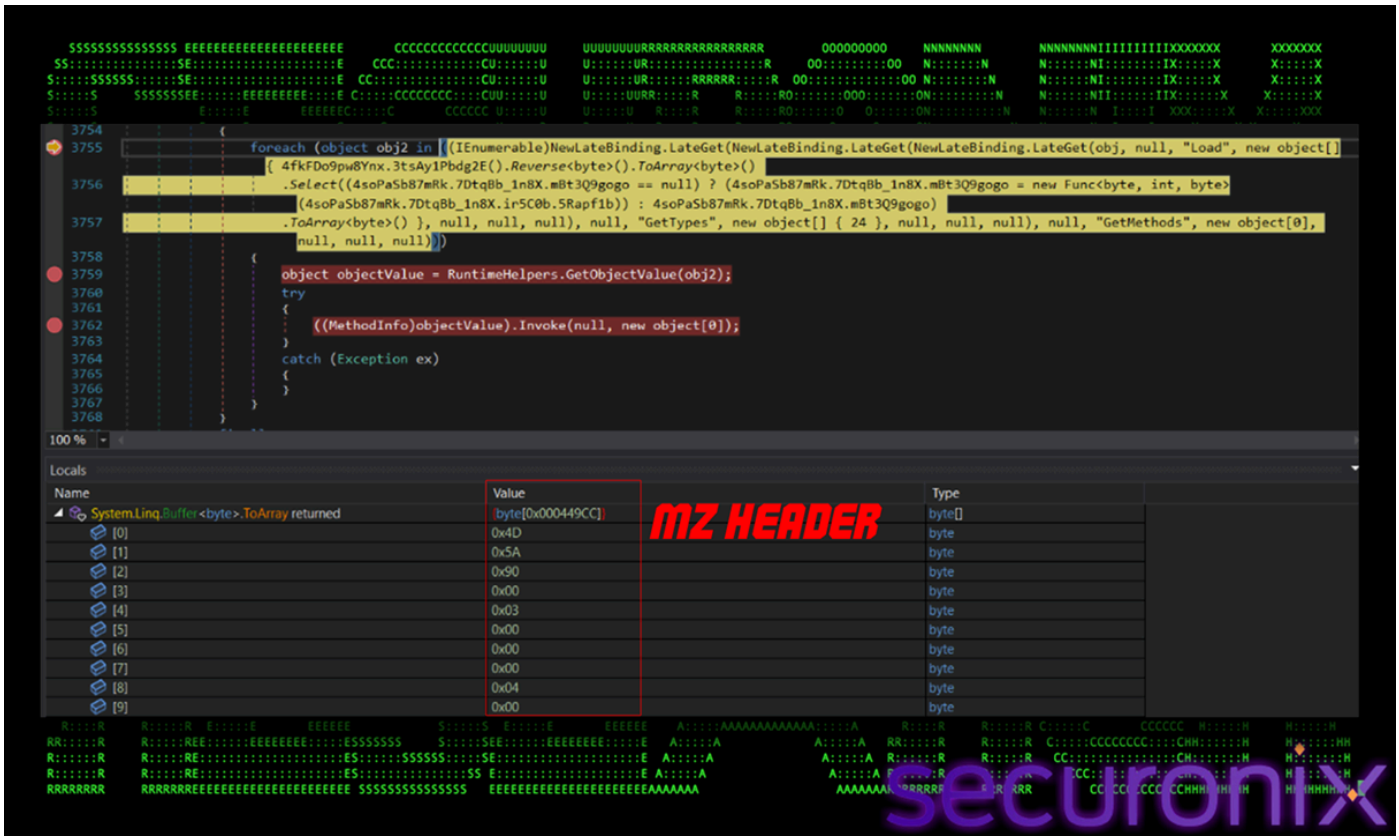


Figure 9 Decoded DLL

Reflective Loading and Late-Binding

With the final payload assembly now fully decrypted in memory, the DARKTORTILLA loader orchestrates its execution in a manner designed for maximum stealth: reflective loading. This is a technique that completely bypasses the traditional Windows Portable Executable (PE) loader. Instead of writing the payload to disk as a physical file, the malware leverages the **System.AppDomain.Load(byte[])** method. This instructs the .NET Common Language Runtime (CLR) to load the assembly directly from its byte array representation in the process's virtual memory.

Think of an AppDomain as a lightweight container or sandbox within a single operating system process. A single OS process can host multiple AppDomains. Each AppDomain provides a level of isolation for code executed within it. When you launch a .NET application, it typically runs in a default AppDomain. All assemblies (like your EXE and its referenced DLLs) are loaded into this domain. Malware like DARKTORTILLA leverages AppDomain.Load(byte[]) because it allows them to load a malicious assembly (their final payload DLL) directly from a byte array (which they've decrypted in memory). It's loaded purely from the process's memory space. This is a classic "fileless" technique that helps evade signature-based antivirus solutions that primarily scan files on disk.

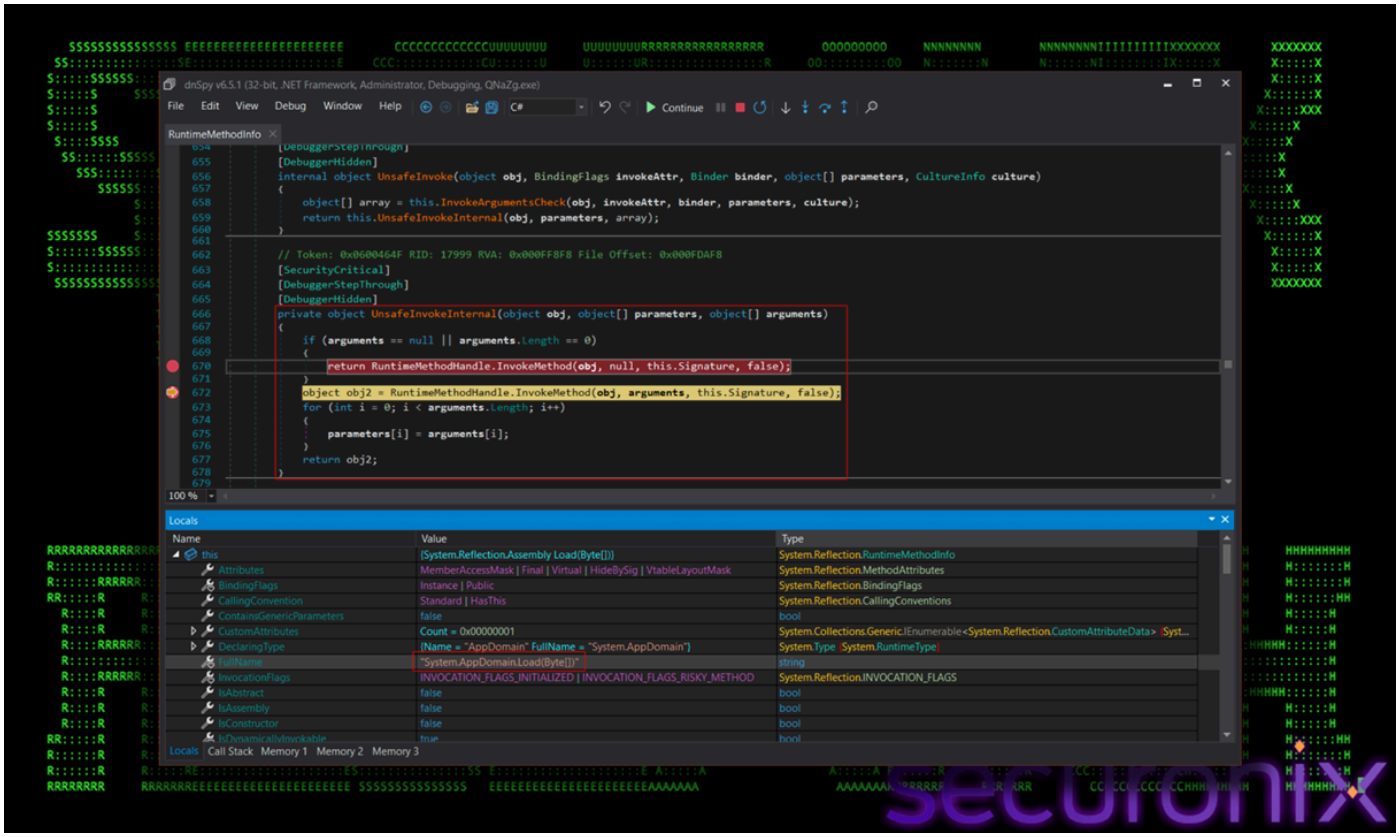


Figure 10 Reflectively loading the decrypted PE from memory

Once the payload is mapped into the application domain, the loader employs late binding and extensive .NET reflection to activate it. Rather than hardcoding direct calls to the payload's methods at compile time, the loader specifically utilizes the Microsoft.VisualBasic.CompilerServices.NewLateBinding.LateGet() method. As seen in Figure 11, **NewLateBinding.LateGet()** is invoked by passing the reflectively loaded payload as an argument, allowing the loader to dynamically retrieve and execute the payload's designated entry point (Segwenservice.Class1_PreStart.Method0()), which is the loaded module of the final payload. This late-binding mechanism is a sophisticated defence evasion tactic; it effectively obfuscates the true control flow and capabilities of the malware. By resolving and invoking method calls dynamically at runtime, it forces analysts into dynamic execution environments, adding layers of complexity to detection and reverse engineering efforts. This combination of reflective loading and dynamic invocation ensures the FormBook RAT achieves its malicious objectives with minimal footprint and maximum stealth.

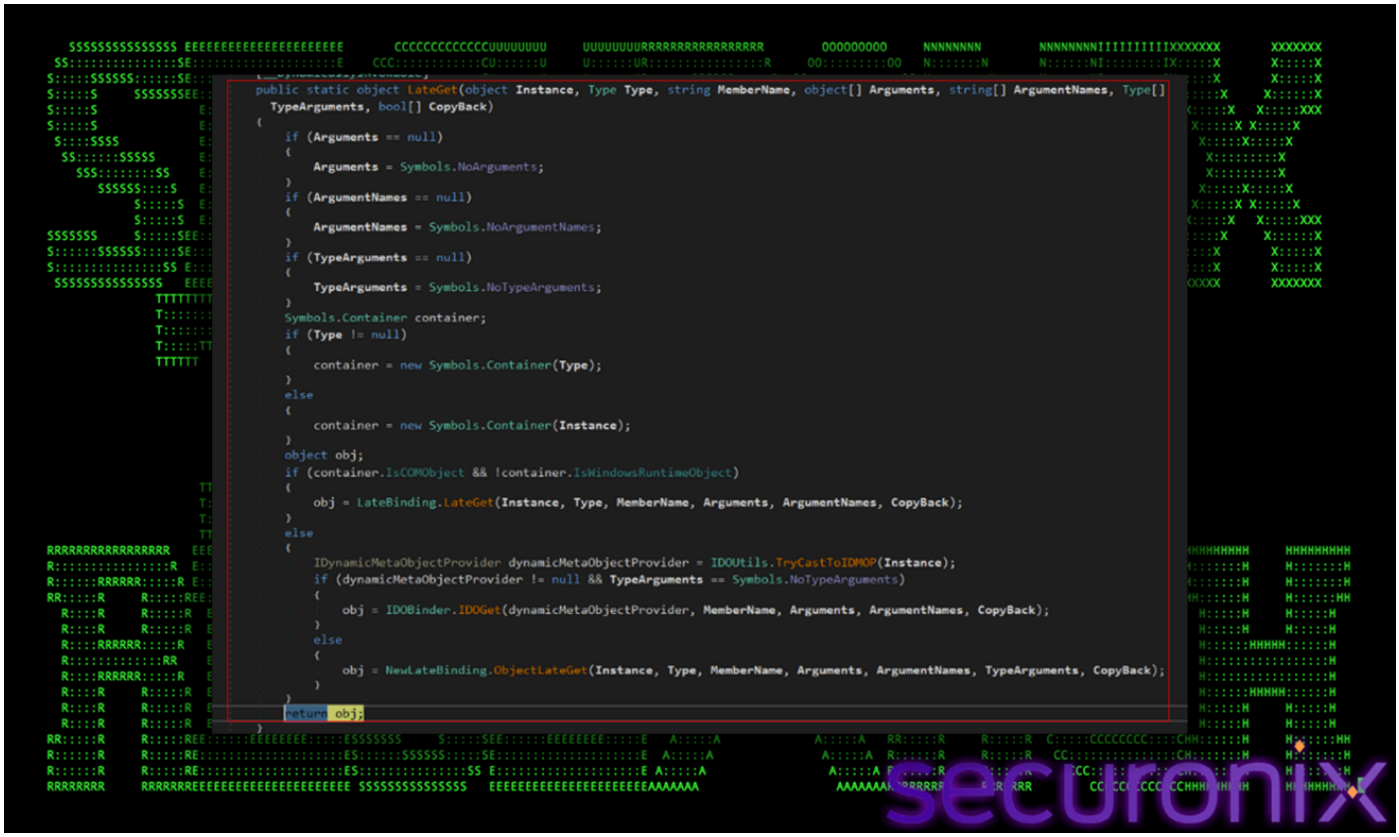


Figure 11 Using reflection to invoke the entry point of the in-memory payload

Stage 4: The Final Payload – FormBook RAT

The in-memory assembly, Segwenservice.dll, is the final payload, which we identified as a variant of the **FormBook RAT**. It is highly modular, with different classes responsible for specific tasks such as installation (Class11_Install), persistence (Class12_Startup, Class16_Persist), anti-VM

checks (Class8_AntiVMs), and decryption (Class5_Decrypter). The malware behavior is dictated by a configuration file stored as an encrypted resource. This resource is decrypted at runtime using **AES**, making it difficult to extract its configuration without dynamic analysis.

Its first action is to decrypt its own configuration, which is the encrypted resource.

Figure 12 Decrypting configuration data containing persistence and behavior flag

The encryption routine that processes the encoded resource is a standard AES implementation. It decrypts via memory stream with derived key IV. The decrypted AES byte array creates a binary reader over a memory stream from data, using UTF-8 encoding, and creates a list as a container. The list turned out to be a config which contains objects/key-value pairs.

This configuration dictionary dictates the malware behavior, including persistence, installation paths, and anti-analysis features, which we will discuss in a moment.

Persistence and Evasion

Based on its configuration, the RAT establishes persistence:

1. **File Placement:** A command is constructed to copy the loader (QNaZg.exe) to a new location: $APPDATA\Roaming\word\word.exe, which is the startup folder, masquerading as a Microsoft Word component. It first checks for hidden startup/startup folder. It's a common technique for persistence. By this, malware ensures it will run again on reboot/login. Names like "word.exe" in a "word" folder in AppData are easily overlooked by users and many basic security controls.

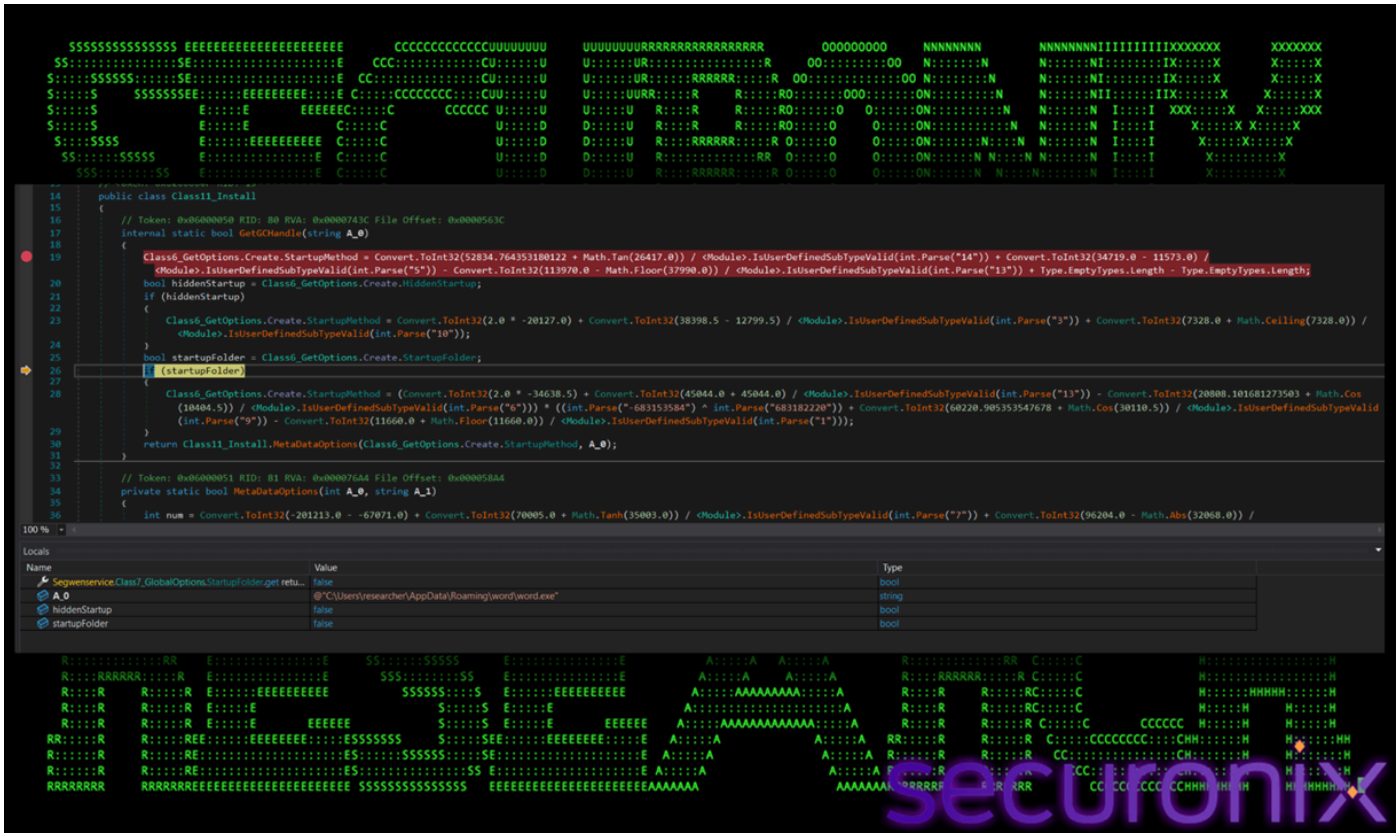


Figure 13 Malware drops itself into the startup folder

1. **Registry Run Key:** It also manages persistence through the Windows Registry. It opens the registry run key under HKCU (HKCU\Software\Microsoft\Windows\CurrentVersion\Run) and checks if a value with the name "svchost.exe" exists. Checks for the "word.exe" in "AppData\Roaming\word". Adds itself under Run, ensuring the dropped malware "word.exe" is launched at user login. Registry value name mimics "svchost.exe ", which is a legit Windows system process.

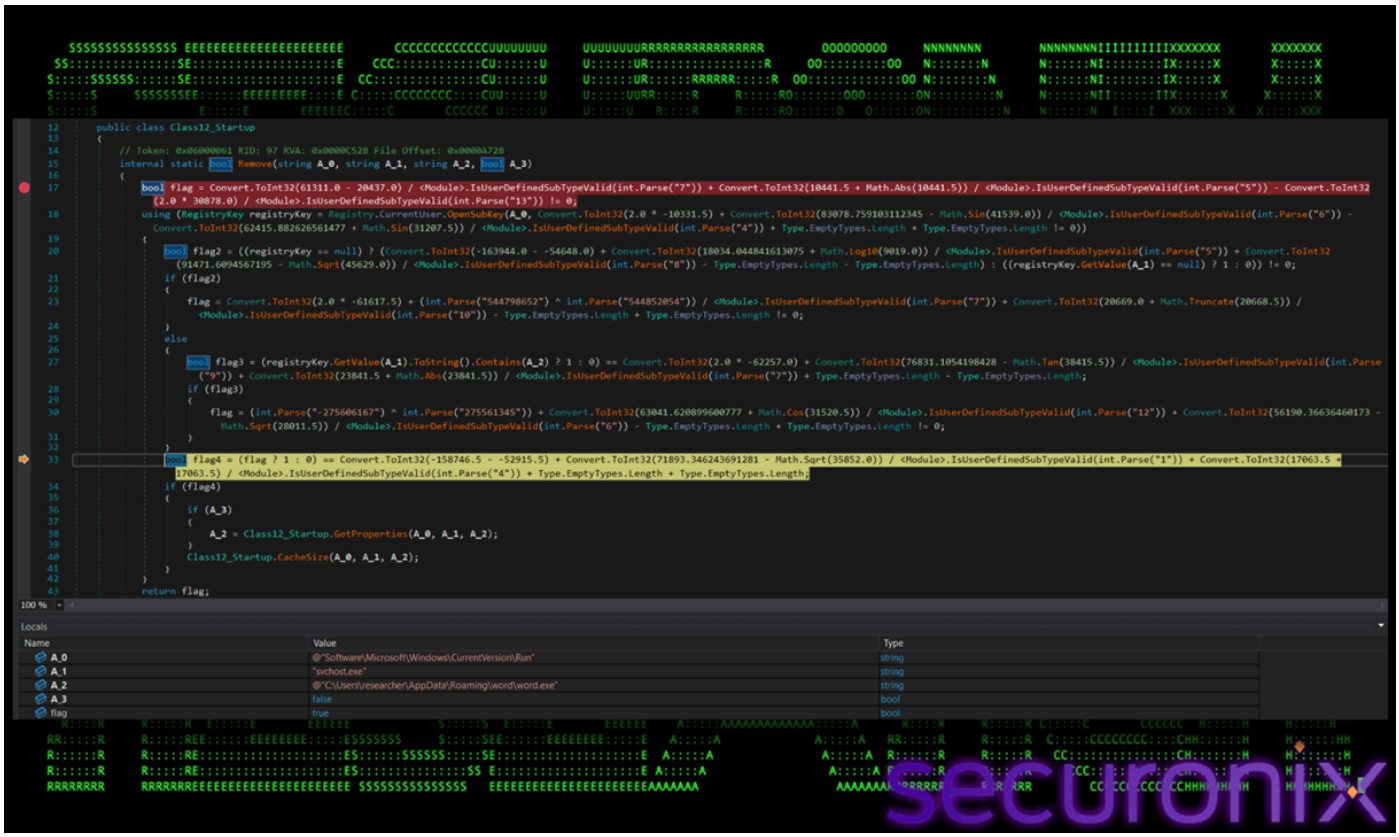


Figure 14 Persistence using registry run key

Anti-Analysis: Bypassing Automated Sandboxes

The malware anticipates being run in automated analysis environments and employs a classic technique to defeat them. It constructs and executes a cmd.exe command that includes two ping 127.0.0.1 -n 39 > nul commands.

cmd” /c ping 127.0.0.1 –n 39 > nul && copy
“C:\Users\RESEARCH~1\AppData\Local\Temp\QNaZg.exe”
“C:\Users\researcher\AppData\Roaming\word\word.exe” && ping 127.0.0.1 –n 39 > nul &&
“C:\Users\researcher\AppData\Roaming\word\word.exe”

Each ping command introduces a delay of approximately 38 seconds. Many automated sandboxes have a limited execution timeout; they terminate analysis if a sample appears to be inactive for too long. By introducing these significant delays, the malware can out-wait the sandbox, preventing its full behavior from being observed and analyzed.

Our analysis observed the RAT attempting to modify registry keys associated with Windows Defender to disable its real-time monitoring. This reduces the likelihood of detection by built-in endpoint protection.

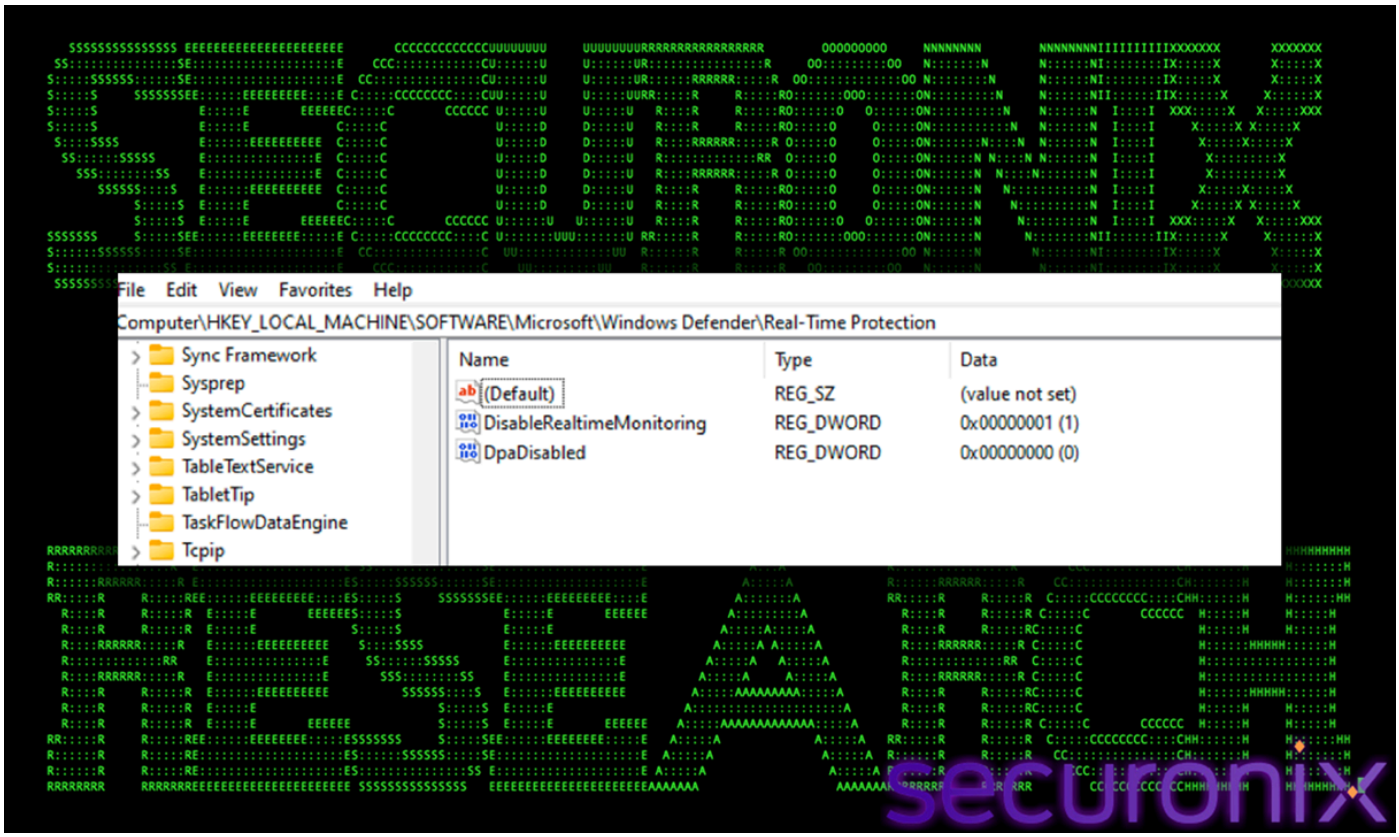


Figure 15 Windows defender disablement

Keylogging Functionality and Data Exfiltration

A primary function of the FormBook RAT in this campaign is pervasive keylogging. The malware implements a low-level keyboard hook, likely using Windows API, to intercept keystrokes across all applications. This allows it to capture every single key pressed by the user, regardless of the active window.

The captured keystrokes are not immediately exfiltrated but are buffered and written to local files for later transmission. During our analysis, we observed the RAT creating files with names matching the pattern KB_*.dat (e.g., KB_2025-10-07.dat) within its persistence directory (%APPDATA%\Roaming\). This batching approach minimizes network traffic, making detection harder, and ensures data is not lost if C2 connectivity is temporarily unavailable.

A typical keylogger log file from FormBook follows a structured format, enabling easy parsing by the attacker. Special keys (like Shift, Ctrl, Alt, Enter, Backspace) are often logged with specific tags or bracketed descriptions to distinguish them from standard character input.

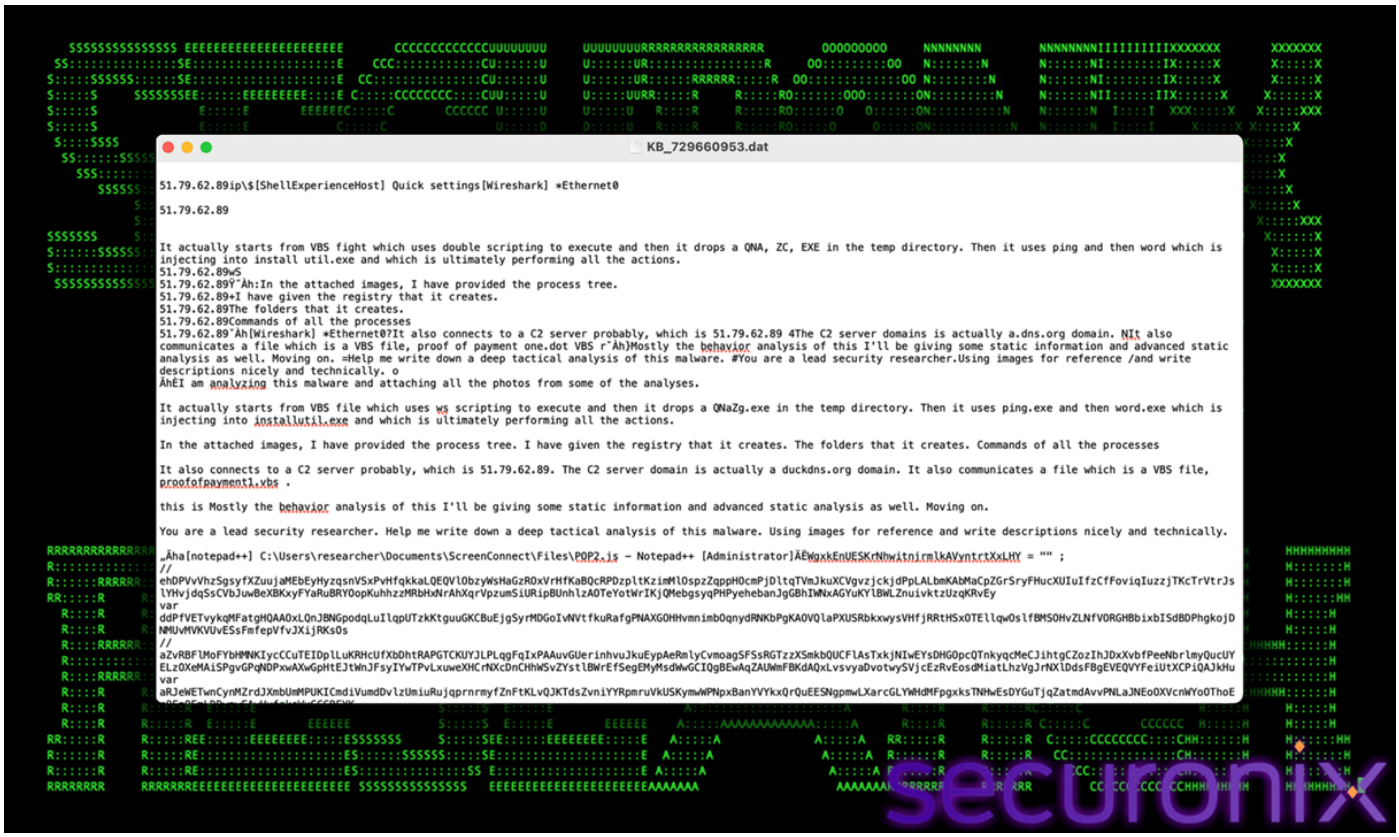


Figure 16 Log file created by RAT

C2 Infrastructure

The FormBook RAT, once established, initiates communication with its command and control (C2) infrastructure to receive commands and exfiltrate collected data. During our analysis, we observed the RAT attempting to connect to a duckdns.org dynamic DNS domain. Dynamic DNS services are a common tactic for malware operators as they allow the C2 IP address to change frequently while the domain name remains constant, making it harder for security teams to block.

Specifically, network captures revealed outgoing TCP connections to **51.79.62.89**, which resolves to the duckdns.org domain specified in the malware's configuration. Specifically, we observe multiple attempts to establish a connection on **destination port 57652**.

Wrapping up…

The DARKTORTILLA campaign demonstrates a sophisticated, multi-layered approach to malware delivery. The threat actors combine social engineering, heavy script obfuscation, and advanced .NET evasion techniques to successfully compromise targets. The use of a custom decryption routine followed by reflective loading allows the final payload to be executed in a fileless manner, significantly complicating detection and forensic analysis. By masquerading its files and registry keys with legitimate-sounding names, the malware attempts to blend in with normal system activity to maintain long-term persistence.

Victimology and Attribution

This campaign appears to primarily target organizations in the **National Social Security Sector**. The social engineering lures are tailored to this specific vertical. Currently, there is not enough information to definitively attribute this campaign to a specific country or threat group.

Securonix Recommendations

- **Maintain vigilance** against social engineering attacks and educate users to never download or execute files from untrusted web portals or emails.
- **Always verify** that software downloads come from legitimate and official websites.
- **Endpoint Security:** Deploy a strong defensive solution capable of monitoring for suspicious script execution, reflective loading of .NET assemblies, and unusual parent-child process relationships (e.g., word.exe spawning InstallUtil.exe).
- **Script Execution Policies**: Enforce strict script execution policies, such as blocking .js and .vbs files from the internet via email gateway rules or application control.
- **Enhanced Logging**: Enable enhanced command-line and PowerShell logging to capture obfuscated commands and fileless attack stages for investigation. Implement keyboard event logging and monitor file creation in %APPDATA% for suspicious KB_*.dat patterns.
- Securonix customers can scan endpoints using the Securonix hunting queries below.

MITRE ATT&CK Matrix

Tactics	Techniques
Initial Access	T1566.002: Spearphishing Link
Execution	T1059.007: JavaScript T1059.003: Windows Command Shell
Persistence	T1547.001: Registry Run Keys / Startup Folder
Defense Evasion	T1021.007: Remote Services: Cloud Services
Execution	T1027: Obfuscated Files or Information T1140: Deobfuscate/Decode Files or Information T1620: Reflective Code Loading T1036.005: Match Legitimate Name or Location T1562.001: Disable or Modify Tools (Windows Defender) T1497.003: Time Based Evasion (Ping Delay)
Credential Access	T1056.001: Input Capture: Keylogging

Command and Control	T1571: Non-Standard Port T1568.002: Dynamic Resolution: DGA (duckdns.org)

Relevant Securonix Detections

- Suspicious Javascript Execution From Temp Location Analytic
- Startup Run Registry Key Created to Suspicious Directory Analytic
- Potential Windows Defender Modification Registry Analytic

Relevant hunting Queries

(remove square brackets “[ ]” for IP addresses or URLs)

- index = activity AND rg_functionality = “Next Generation Firewall” AND destinationaddress = “51.79.62[.]89”
- index = activity AND rg_functionality = “Endpoint Management Systems” AND deviceaction = “File created” AND filename IN (“QNaZg.exe”, “adobe.js”, “svchost.js”, “word.exe”)
- index = activity AND rg_functionality = “Endpoint Management Systems” AND deviceaction = “Process Create” AND ChildProcessCommandLine CONTAINS “ping 127.0.0.1 -n 37 > nul &&”
- index = activity AND rg_functionality = “Endpoint Management Systems” AND deviceaction CONTAINS “Registry value set” AND eventtype = “SetValue” AND targetobject CONTAINS “SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run\\svchost.exe”
- index = activity AND rg_functionality = “Endpoint Management Systems” AND SourceProcessName IN (“wscript.exe”, “cscript.exe”) AND deviceaction = “Process Create” AND childprocesscommandline CONTAINS “AppData\\Local\\” AND childprocesscommandline CONTAINS “.exe”
- index = activity AND rg_functionality = “Endpoint Management Systems” AND image CONTAINS “InstallUtil.exe” AND parentimage CONTAINS “word.exe”

C2 and Infrastructure

C2 Address
51.79.62[.]89 Associated duckdns[.]org domain

Analyzed files/hashes

File Name	SHA256/MD5
part1	eba24c92b51d8fb24697952135a7d7bdf4a7511ab94be850fda1fc512675f6ad

81__POP1.BZ2	67c00ede3964cb78c64575b65b301f808958311b99779b71597f6282b1a4e9f2
PROOF OF PAYMENT1.vbs	4ebef5d23ce0fe6c2940ba7a2f6bfc512b1ec5f01458284d2ce0e71ee8787b81
POP2.js	d4c097412ab05630e6cb97b544dc7c0a0e238a4bdf5c79da679c7545face2dad
svchost.js	aab2b9cd5a946739bbb41ae2234adaf34ba9761445315c2b5ba270a7b931a2e2
adobe.js	56d627adc6e6e8967ade649f707134a501cfea5ec66322514536ee8ace3053fb
PHat.jar	7c9128d197301fcd89d6fd1b0077d2a35f2a98c6219386900d7e8c89e4799a86
QNaZg.exe	a428d2602ad3bad2d590ed68b17a308cff8ab7ff61da2a51acb83fd202b5358d
Segwenservice.dll [embedded payload]	8bcfc6dd444f3f577f026d465d826194db45cd205b24f77b9080debba96e3b7b