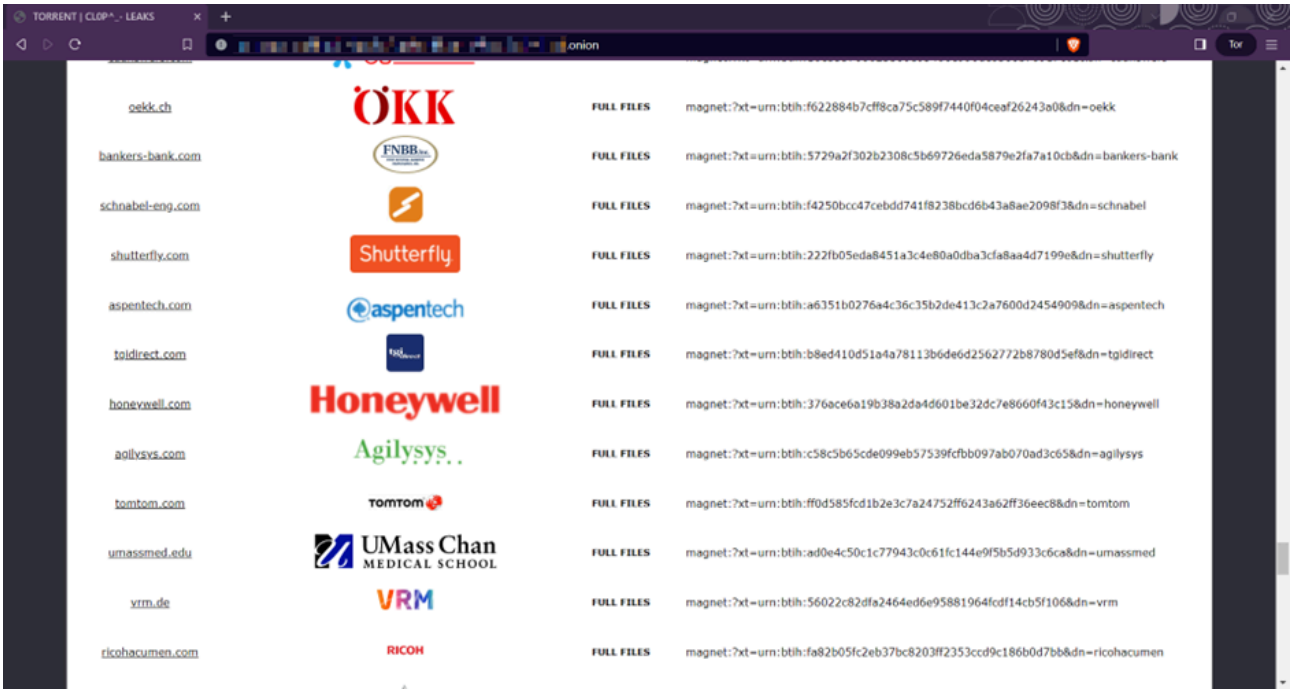
Inside a .NET Stealer: AgentTesla

By Chris Campbell

Published: 2020-12-04 · Archived: 2026-04-05 14:02:15 UTC

First seen in 2014, AgentTesla (S0331) is a .NET platformed stealer that has recently surpassed Emotet and Trickbot to become one of the most prevalent malware threats. At present it is the #2 most submitted malware family submitted to the ANY.RUN sandbox service, mostly thanks to the Emotet crew appearing to have taken an early Christmas vacation. From pray and spray spam runs through to more resourced campaigns targeting critical infrastructure sectors, AgentTesla appears to have a wide variety of operators. Up until 2019 it was available through the website of the developer, www.agenttesla[.]com, as moreorless a SaaS subscription that included 24x7 support, web management, delivery and packing services, and regular updates:



Predictably, the service was sold with the disclaimer *"Agent Tesla is a software for monitoring your personal computer. It is not a malware. Please, don't use for computers which is not access permission."*, in much the same fashion as open-source RAT's and ransomware are cautioned on GitHub as being *"for educational purposes only"*. While cracked and leaked copies of the tool were always available through forums and marketplaces, their availability has naturally exploded following closure of the official service.

Features

As a SaaS offering with a reported 6300+ customers (source: Krebs on Security), it was fair to expect that the feature set and reliability of the tool would continually improve to remain competitive, meet customer requirements and stay ahead of defenses. Current features include:

- Keylogging, clipboard scraping and screenshot capture.
- Credential theft from a wide selection of browsers, VPN, FTP and email clients, and Windows credential stores.
- FTP, HTTP and SMTP exfiltration.
- Tor proxying.

Occasionally custom modules have been seen in samples, such as the WiFi credential stealer.

Delivery

Like many varieties of malware, delivery is primarily via email. Compromised email credentials are frequently used for sending, and messages utilise your garden variety logistics, financial and current event templates. Most often we observe the AgentTesla payload attached to messages in an archive, but maldoc delivery is also commonplace. A full spectrum of payload delivery mechanisms are seen being employed by the maldocs, including links, macros, DDE commands and Office exploits (e.g. CVE-2017-11882 and CVE-2017-8570).

Loaders employ obfuscators/crypters for source protection and almost always .NET reflection to load the AgentTesla stealer, as will be illustrated in the following samples.

Email

The sample that we will first investigate in this post begins with a payment themed message that leverages the branding of a Turkish garment company:

Attached to the email is a zip file containing the payload: “swift copy.exe” (sha256: 9d626bb9d442d3762e5366f0fbefae41708936b9c254141fcf3b0a1b80291ebb). The sample is detected by 44 of 71 engines on VirusTotal (report) – so it’s not exactly low key.

Test Environment

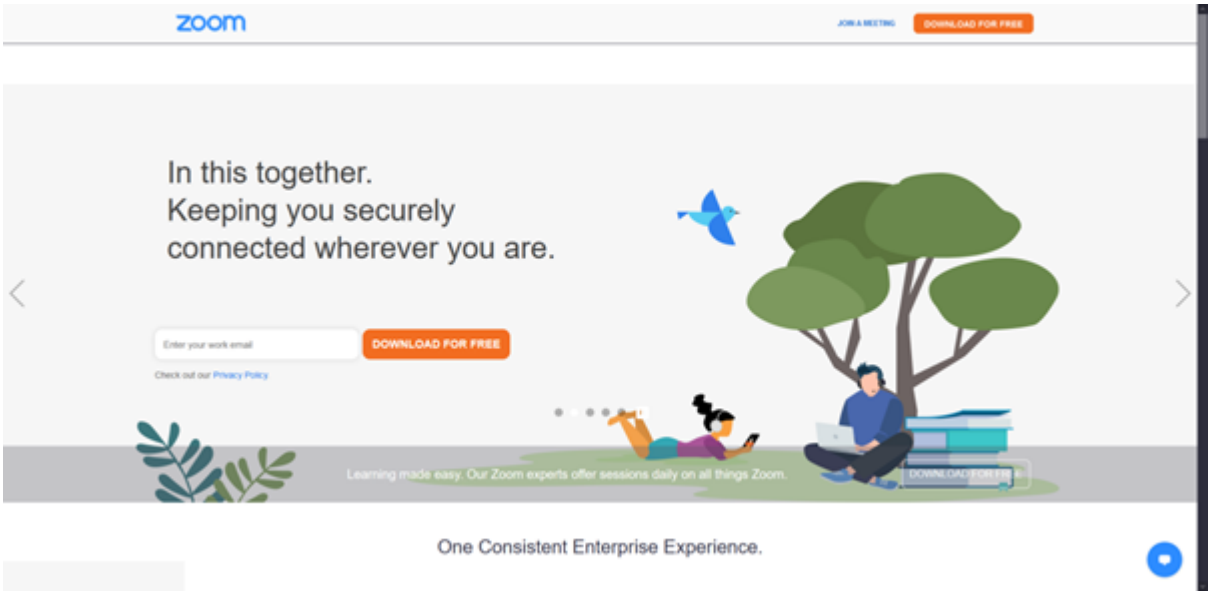
The sample is copied to a 64-bit Windows 7 analysis VM that is running a FileZilla FTP server and has a handful of dummy accounts set up, including for CoreFTP:

We know that this is one of the tools that AgentTesla is capable of stealing credentials from, so it is expected that this will prompt the sample to attempt exfiltration.

The debugger used is dnSpy (https://github.com/dnSpy/dnSpy).

Loader

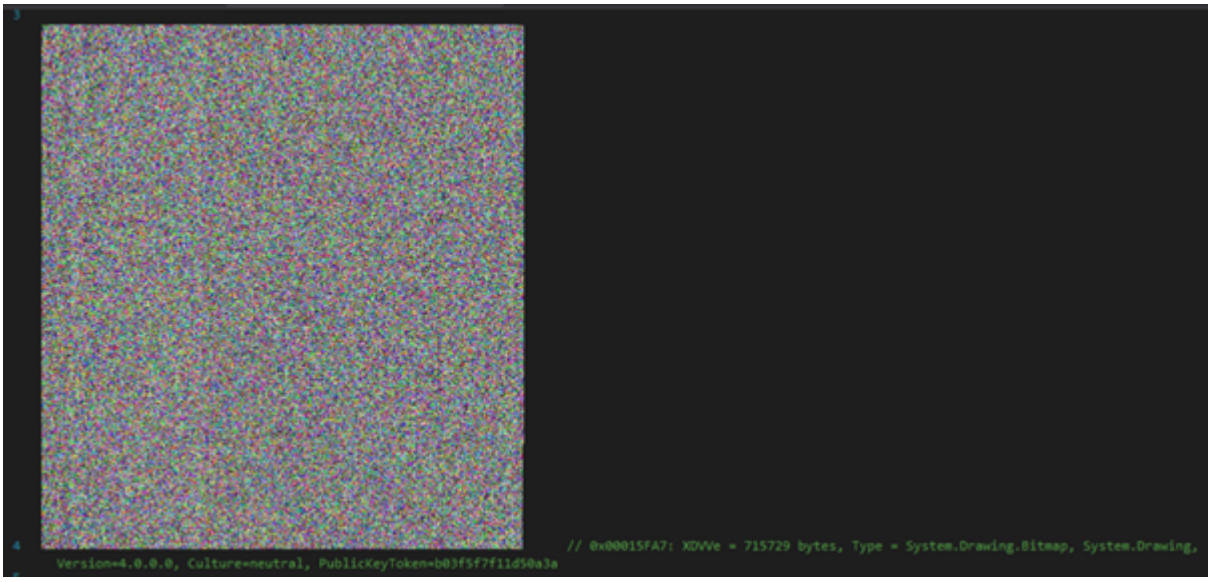
“PongGame” may seem like an odd choice of namespace for a loader, but this isn’t at all abnormal for the obfuscators used with AgentTesla. Naming conventions of legitimate programs are often adopted and applied across metadata, namespaces, classes, methods and objects.



The loader imports System.Reflection, indicating .NET reflection is likely used to load additional modules during unpacking:

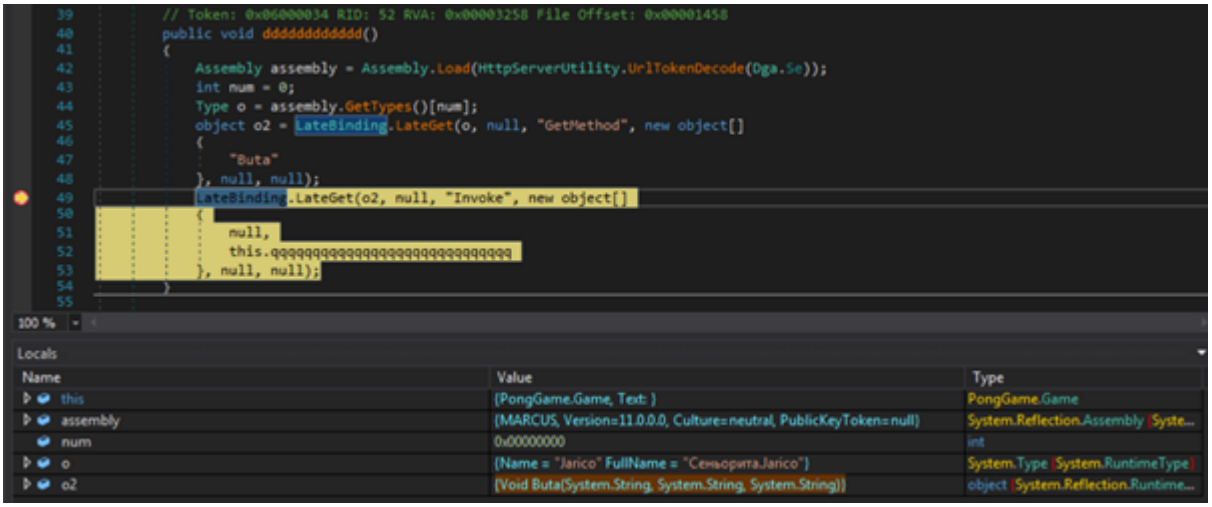
```
// C:\Users\Administrator\Downloads\swift copy.exe
// B7, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

// Entry point: PongGame.Program.Main
// Timestamp: <Unknown> (FF4FE818)

using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
```

Before stepping through the execution, we review the program resources, of which there are two that stand out. It is well known that AgentTesla makes heavy use of steganography, so it is safe to assume the single image (XDDVe) will at some point be passed through a decoding routine. However, there is no reference to it in the loader:



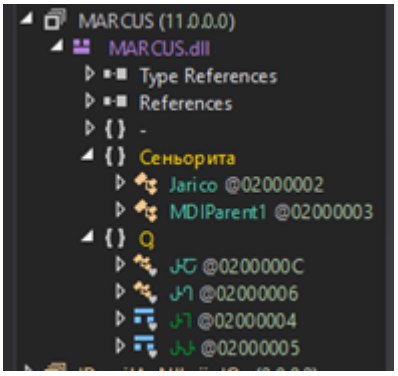
There is also a long string that is preceded with TVqQ which is base64 for “MZ”, the magic number for the MS-DOS EXE format:

A reference to this is seen in method “ddddddddddd”, where the value of this resource converted from a URL string token to a byte array using System.Web.HttpServerUtility.UrlTokenDecode, and the byte array then loaded as an assembly. A breakpoint is set prior to the assembly being invoked and execution is run through to this:



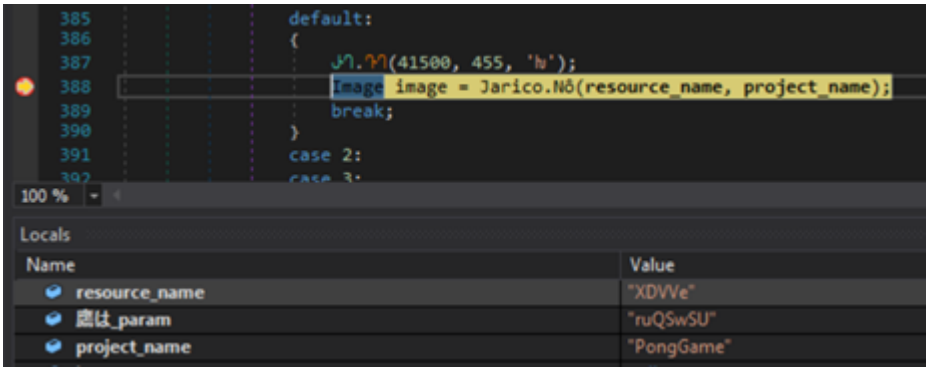
Second Stage

This unpacked assembly is MARCUS.dll and the method that will first be invoked is Jarico.Buta. The DLL is also obfuscated and has relatively few classes:

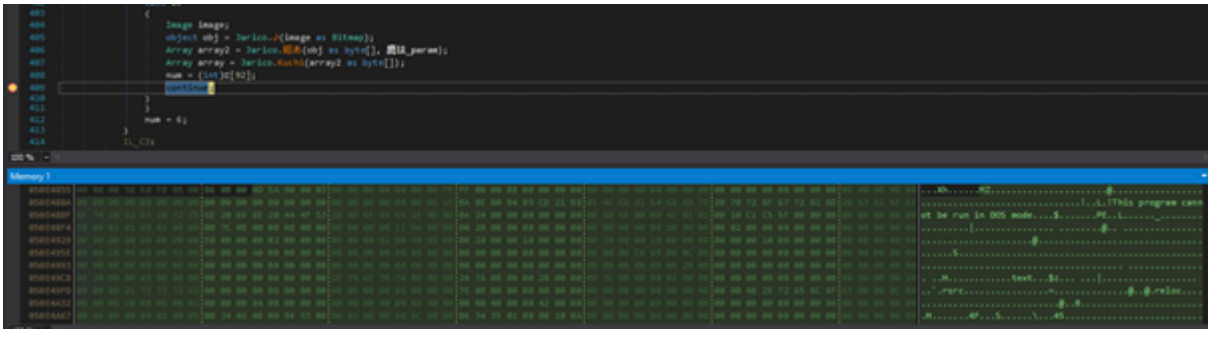


A breakpoint is set on Jarico.Buta and execution is continued through to here:

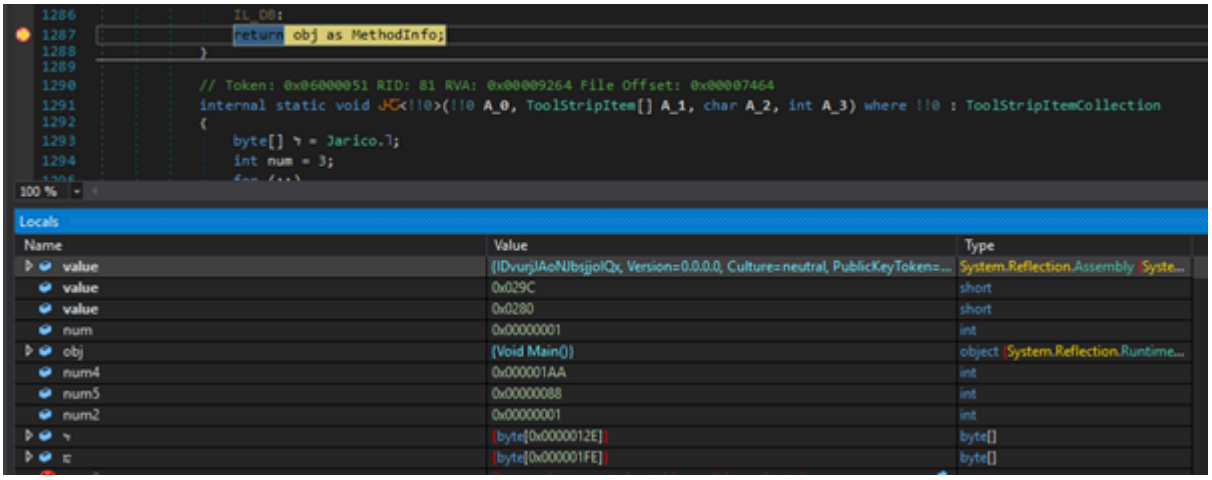
Shortly after this is an image object is returned by a method that takes a resource name and project name as parameters. Breaking at this point shows that the project and resource are the loader and image:



The image is run through several decoding methods which produces an additional executable:

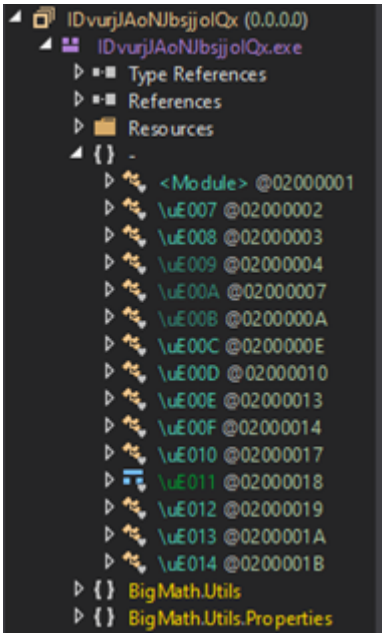


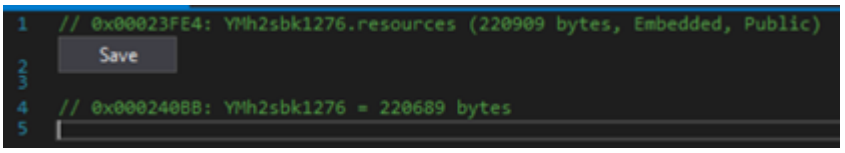
Stepping through a little further we see the executable is named “IDvurjJAoNJbsjjolQx” and the entry point is the Main method:



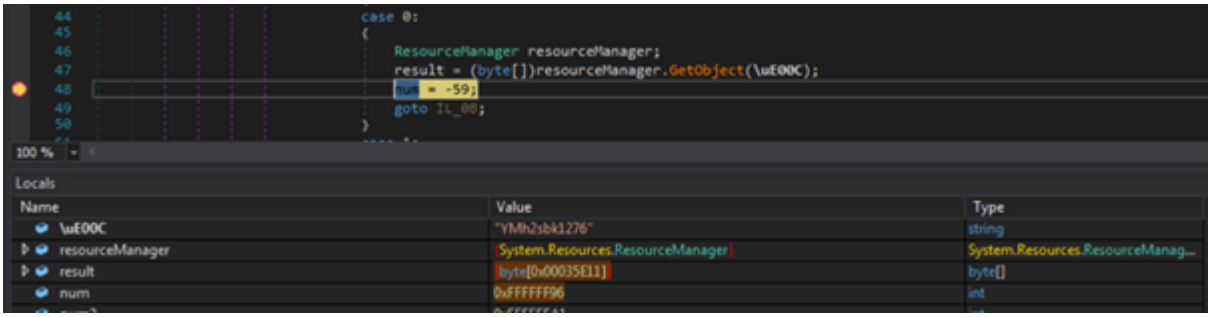
Third Stage

IDvurjJAoNJbsjjolQx is an obfuscated executable with a sizable resource named “YMh2sbk1276”:

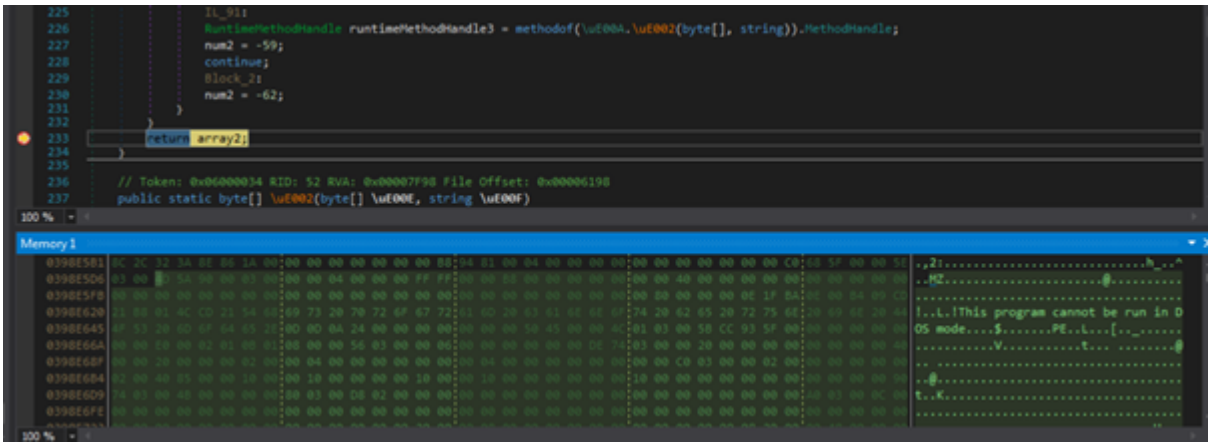




No direct reference to this is found, however a method is found where a resource is loaded into a byte array, so a breakpoint is set after the array has been formed:

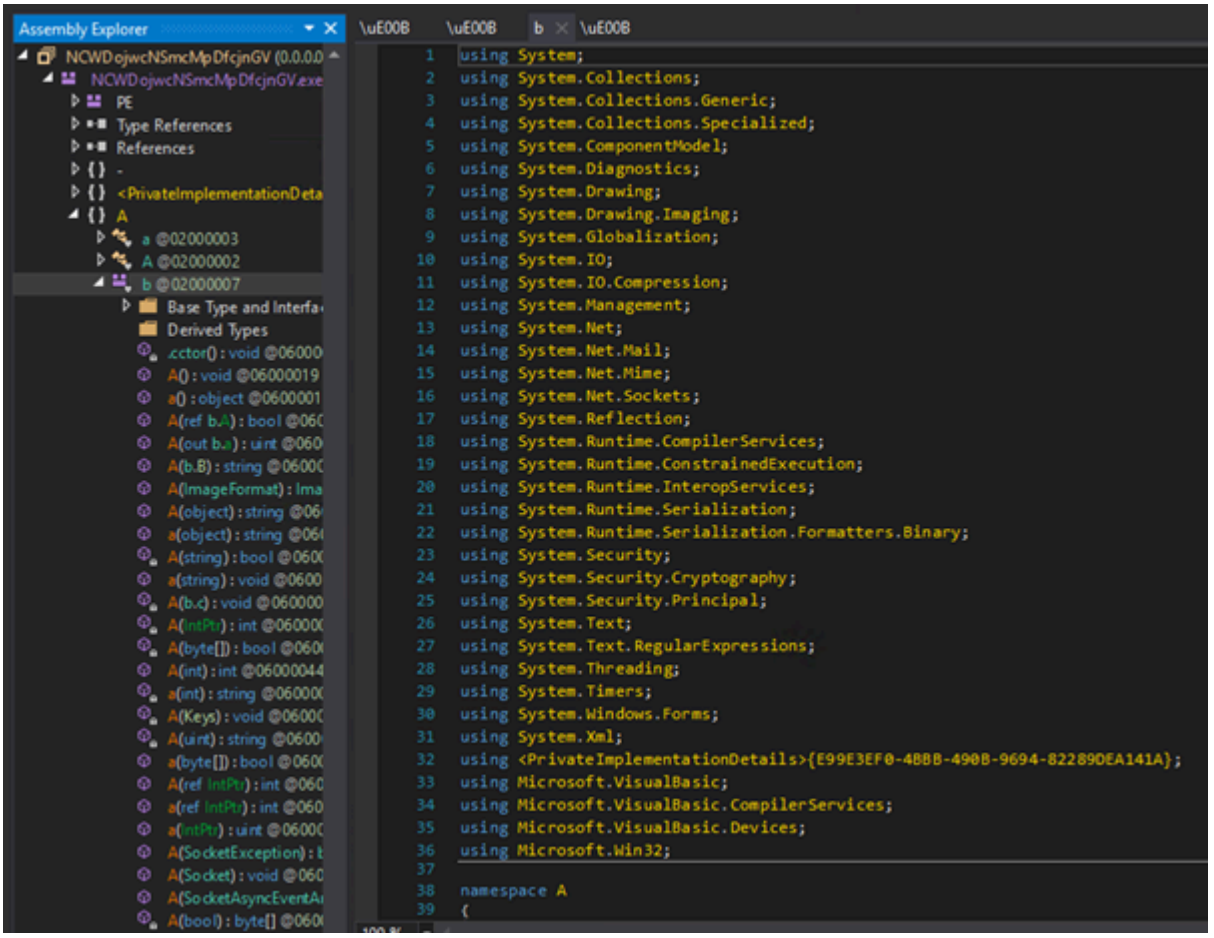


The value of \uE00C confirms that the loaded resource is what was expected. Contents of this resource are passed through several decoding routines to produce another executable:

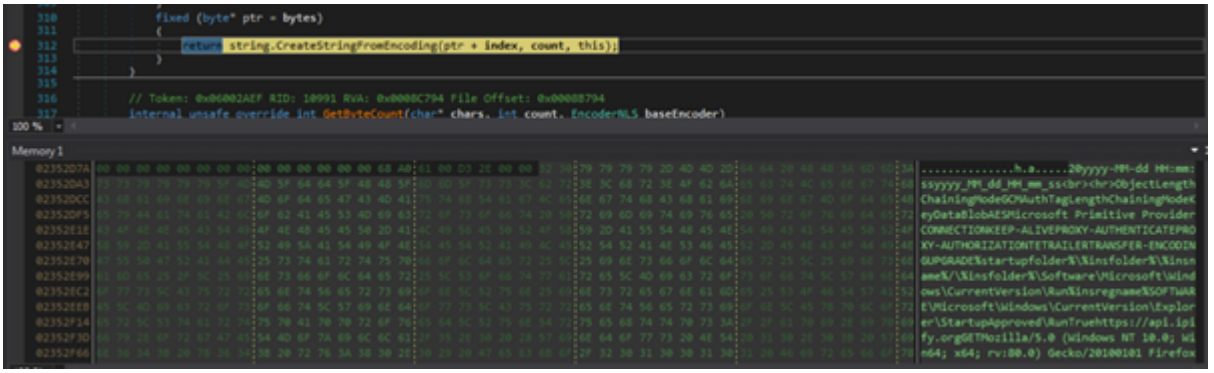


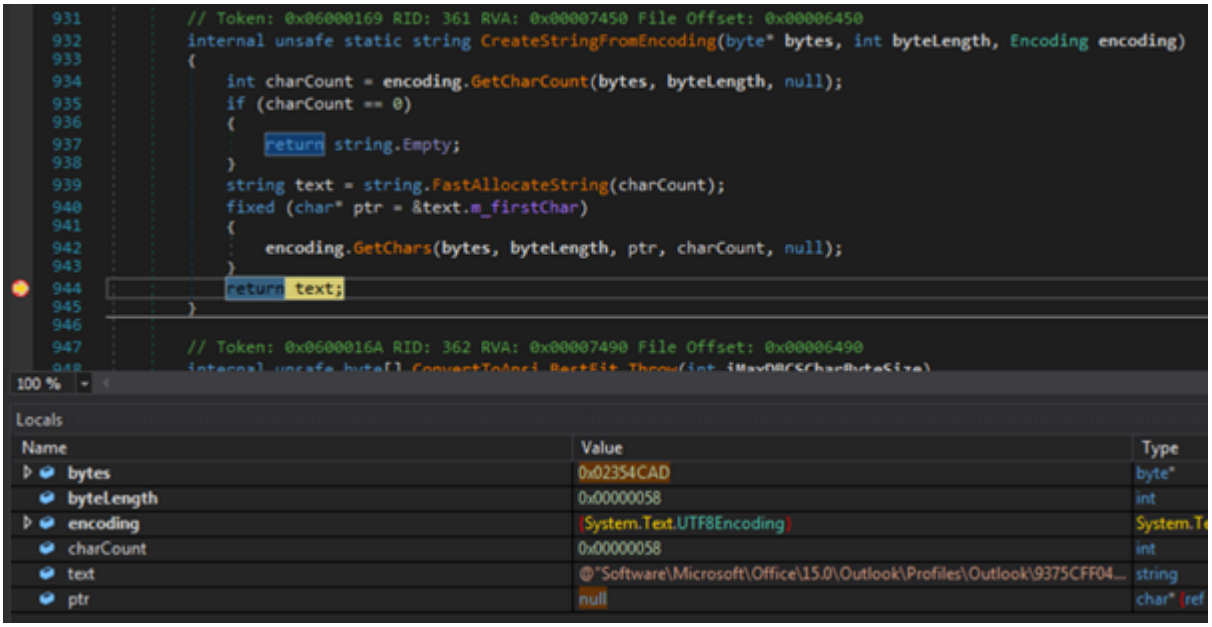
Unpacked Stealer

The memory section is dumped to disk and the resulting file – the AgentTesla stealer – is opened in a new dnSpy session. While still a little obfuscated, the source is relatively easy to read through:

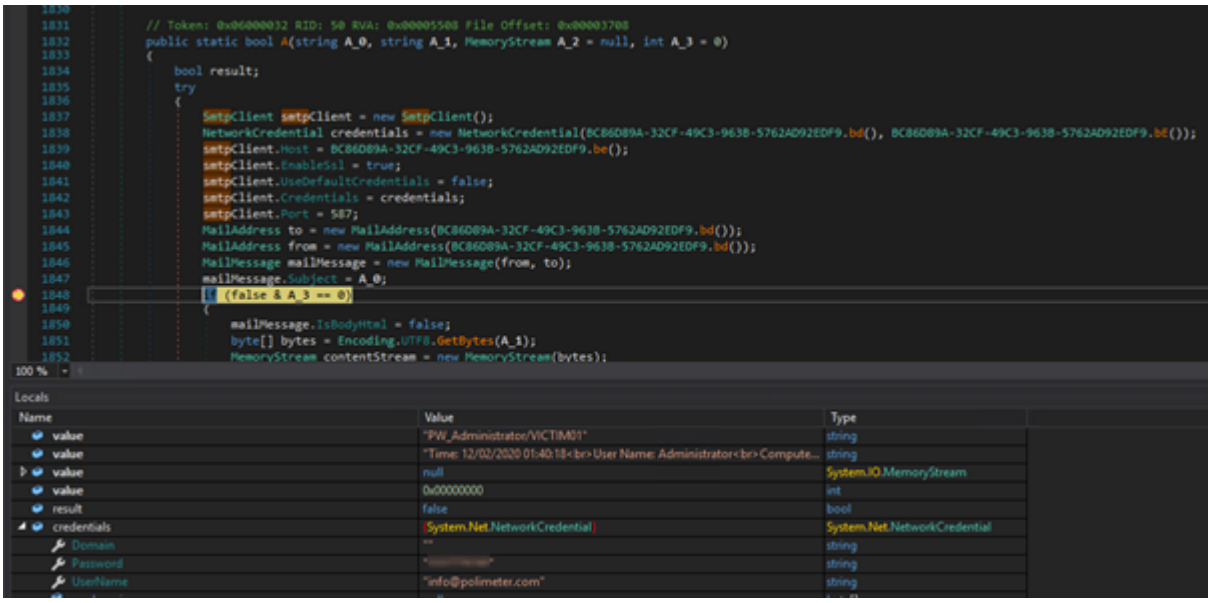


In this sample, configuration items are extracted from a specific position (i.e. offset and length) within a UTF8 byte array and converted to string format:





We can also set a breakpoint prior to HTTP POSTs or email messages being sent to obtain the respective config (i.e. HTTP request or SMTP credentials):



Imports are made for the kernel functions required by the keylogger:

```
// Token: 0x0600003C RID: 60
[DllImport("user32.dll", EntryPoint = "GetForegroundWindow")]
private static extern IntPtr G();

// Token: 0x0600003D RID: 61
[DllImport("user32.dll", EntryPoint = "GetWindowText")]
private static extern int A(IntPtr, StringBuilder, int);

// Token: 0x0600003E RID: 62
[DllImport("user32.dll", CharSet = CharSet.Auto, EntryPoint = "GetWindowTextLength", SetLastError = true)]
private static extern int A(IntPtr);

// Token: 0x0600003F RID: 63
[DllImport("user32.dll", EntryPoint = "GetKeyboardState")]
private static extern bool A(byte[]);

// Token: 0x06000040 RID: 64
[DllImport("user32.dll", EntryPoint = "MapVirtualKey")]
private static extern uint A(uint, uint);
```

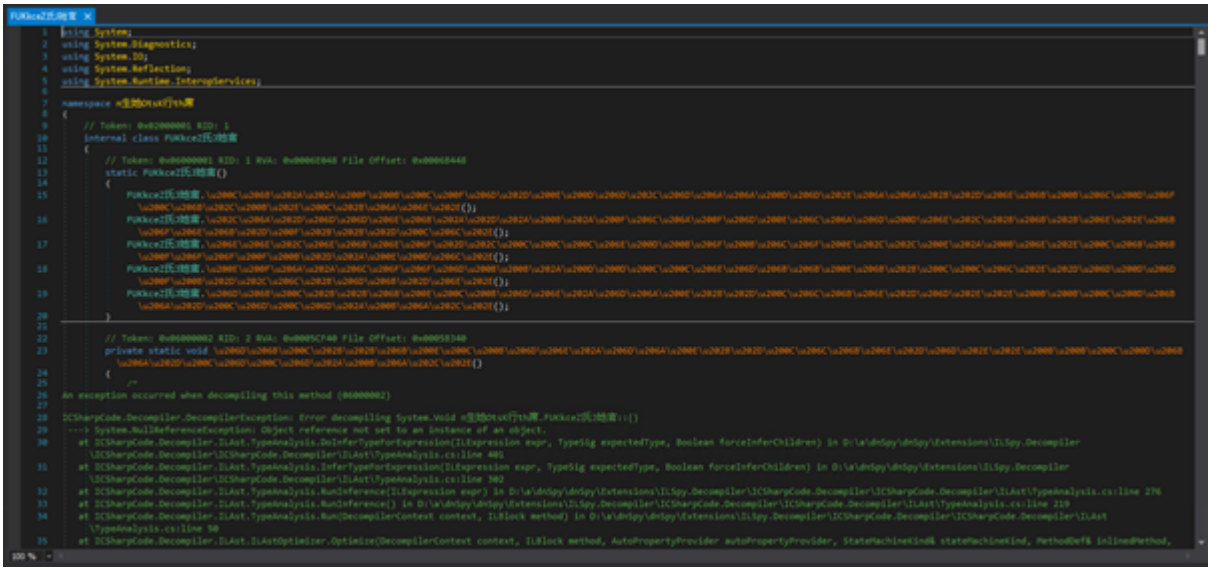
And below these is the method that implements the keylogger:

```
if (A_0 == Keys.Back)
{
    if (global::A.b.b == Conversions.ToBoolean(BC86D89A-32CF-49C3-963B-5762AD92EDF9.bS()))
    {
        global::A.b.A += BC86D89A-32CF-49C3-963B-5762AD92EDF9.bs();
    }
    else if (Operators.CompareString(global::A.b.A, BC86D89A-32CF-49C3-963B-5762AD92EDF9.A(), false) != 0 &&
      Operators.CompareString(global::A.b.A.Substring(global::A.b.A.Length - global::A.b.h.Length, global::A.b.h.Length),
      global::A.b.h, false) != 0)
    {
        string left = global::A.b.A.Substring(global::A.b.A.Length - 7);
        if (Operators.CompareString(left, BC86D89A-32CF-49C3-963B-5762AD92EDF9.bT(), false) != 0 & Operators.CompareString
          (global::A.b.A.Substring(global::A.b.A.Length - 4), global::A.b.e, false) != 0)
        {
            global::A.b.A = global::A.b.A.Substring(0, global::A.b.A.Length - 1);
        }
    }
}
else if (global::A.B.Computer.Keyboard.AltKeyDown & A_0 == Keys.Tab)
{
    global::A.b.A += BC86D89A-32CF-49C3-9638-5762AD92EDF9.bt();
}
else if (global::A.B.Computer.Keyboard.AltKeyDown & A_0 == Keys.F4)
{
    global::A.b.A += BC86D89A-32CF-49C3-9638-5762AD92EDF9.bU();
}
else if (A_0 == Keys.Tab)
{
    global::A.b.A += BC86D89A-32CF-49C3-9638-5762AD92EDF9.bu();
}
else if (A_0 == Keys.Escape)
{
    global::A.b.A += BC86D89A-32CF-49C3-9638-5762AD92EDF9.bV();
}
else if (A_0 == Keys.LWin | A_0 == Keys.RWin)
{
```

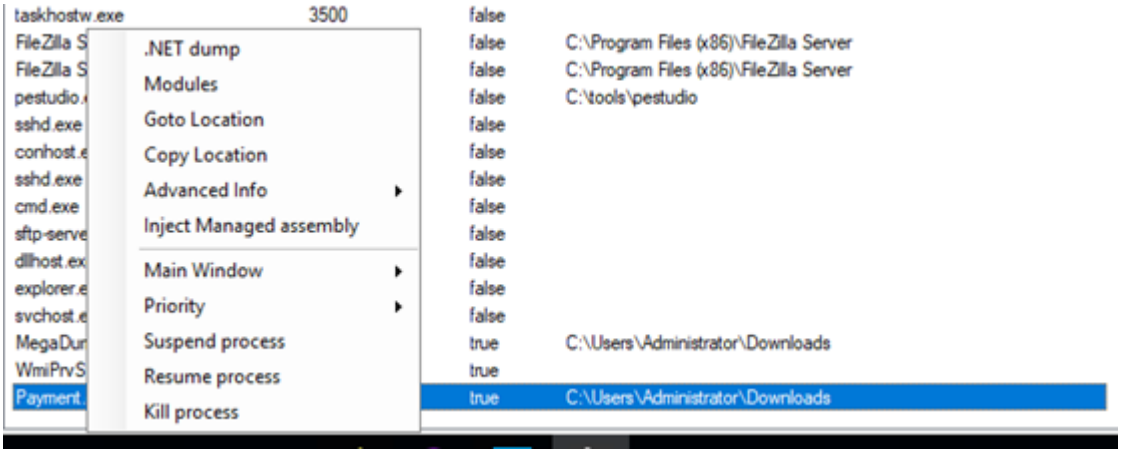
Another Loader

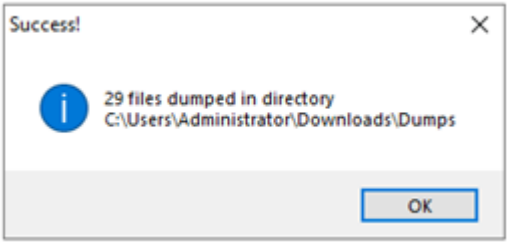
This second sample illustrates a couple of different aspects of .NET malware: anti-tamper measures and persistence.

Upon loading the sample into dnSpy and jumping to the module initialiser, we are presented with decompiler errors intentionally resulting from the anti-decompiler measures implemented by the obfuscator:



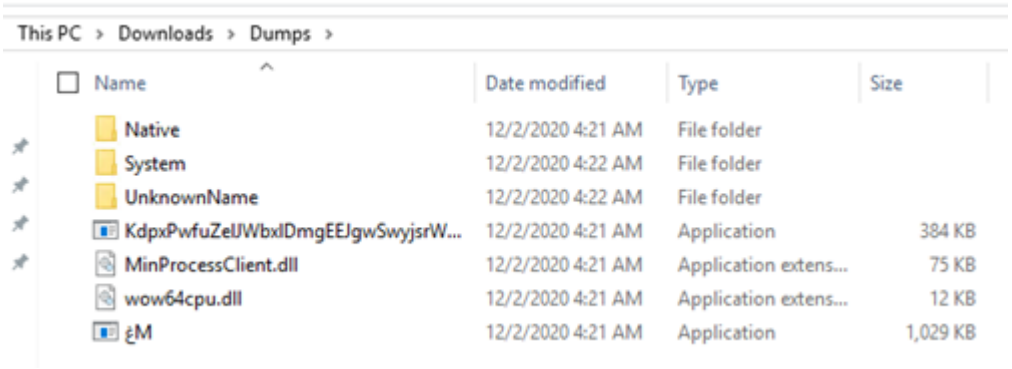
Attempting to run the sample also fails and the method at the entry point of the program appears to be empty. While it is possible to remove these protections with dnSpy by editing the IL instructions, a much faster method is simply running the payload and dumping the unpacked assemblies with a tool such as MegaDumper (https://github.com/CodeCracker-Tools/MegaDumper):





Hollows Hunter (https://github.com/hasherezade/hollows_hunter) is also a useful tool in similar situations.

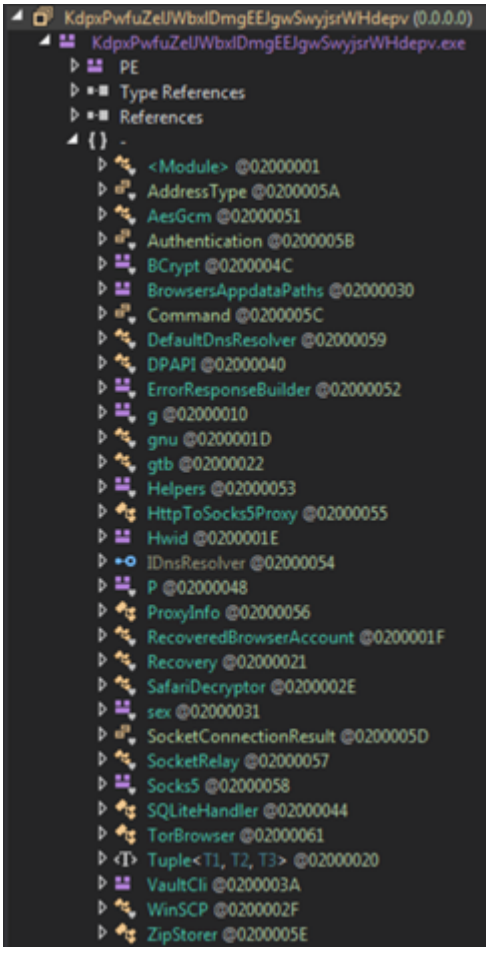
This produced two executables and a handful of DLLs:



The smaller executable is AgentTesla and the larger is a loader.

Modules

In this case, the level of obfuscation is much lower than the previous sample, so more insight into the capabilities can be gleamed thanks to cleaner class, method and attribute names (e.g. DPAPI, HttpToSocks5Proxy, SafariDecrypter, TorBrowser, VaultCli, etc):

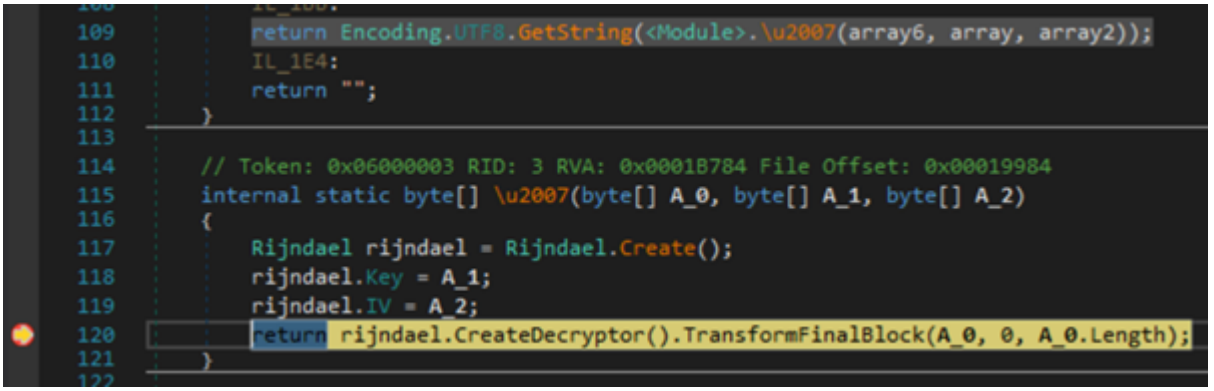


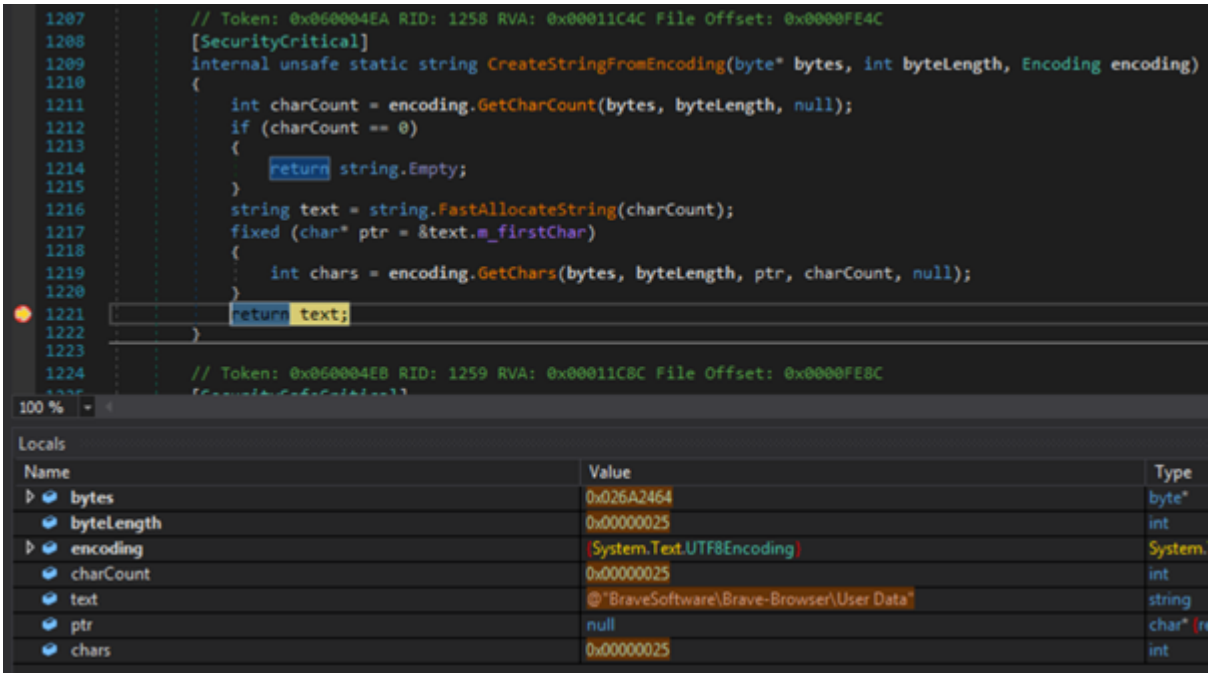
```
// Token: 0x06000254 RID: 596 RVA: 0x0004CD04 File Offset: 0x0004AF04
public TorBrowser()
{
    this.TorDirectory = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), "Tor");
}

// Token: 0x06000255 RID: 597 RVA: 0x0004CD30 File Offset: 0x0004AF30
public void StartTorPorxy()
{
    object obj = new IPEndPoint(IPAddress.Parse(<Module>.\u2008(182252)), 9051);
    this.ProxyServer = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    for (;;)
    {
        IL_28:
        uint num = 3318806002U;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 2600621499U)) % 7U)
            {
            case 0U:
                this.ProxyServer.Shutdown(SocketShutdown.Both);
                num = (num2 * 3488844385U ^ 2843327143U);
                continue;
            case 1U:
            {
                byte[] array;
                int count = this.ProxyServer.Receive(array);
                string @string = Encoding.ASCII.GetString(array, 0, count);
                if (!@string.Contains("250"))
                {
                    num = (num2 * 3926642812U ^ 577646878U);
```

While the previous sample used the simple method of storing the configuration as plaintext in a UTF8 byte array, this instead stores an encrypted configuration as a list of unsigned integer arrays:

```
// Token: 0x04000001 RID: 1
public static object[] \u202E = new object[]
{
    new uint[]
    {
        1007275925U,
        1662714935U,
        3526674679U,
        55266071U,
        1024465695U,
        2016776768U,
        349185261U,
        3297360032U,
        4036748837U,
        2195156222U,
        1840043526U,
        3858945328U,
        3153443422U,
        316485289U,
        192104990U,
        3671624579U
    },
    new uint[]
    {
```

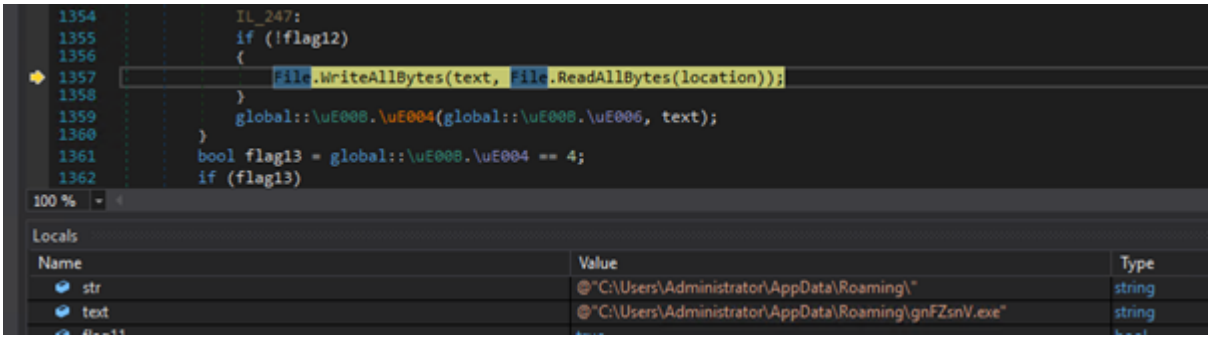
```
case 5U:
    list.Add(new global::Tuple<string, string, bool>(<Module>.\u2008(156256), Path.Combine(folderPath, <Module>.\u2008(156300)), true));
    num = (num2 * 4017678135U ^ 3136435389U);
    continue;
```



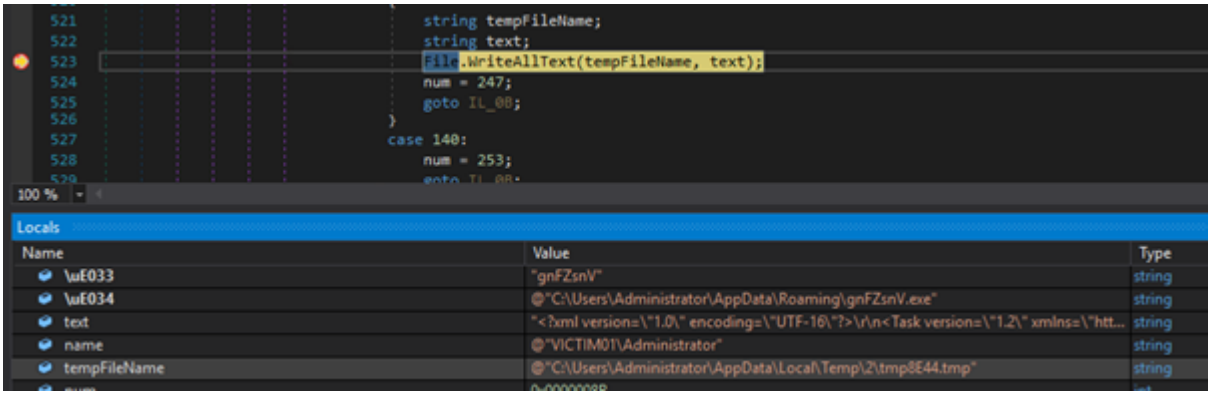


Persistence

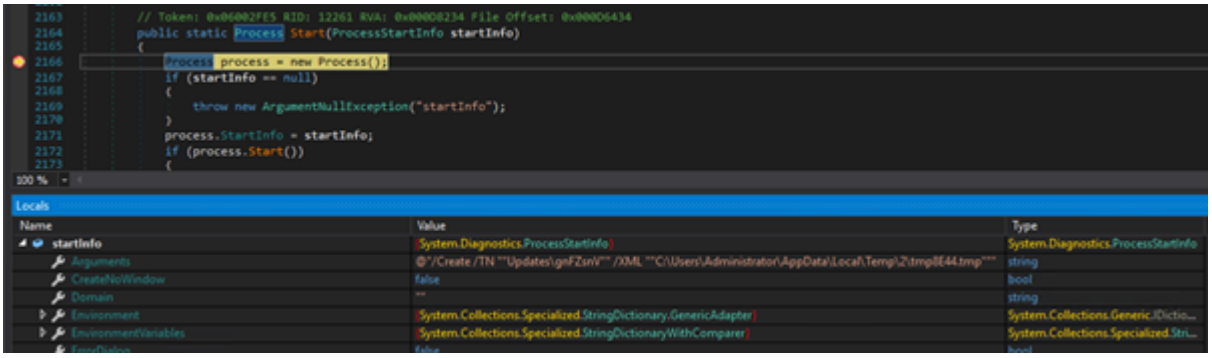
Among the functions of the loader is setting up persistence via scheduled task. The bytes of the current assembly are first written to a path under %APPDATA%:

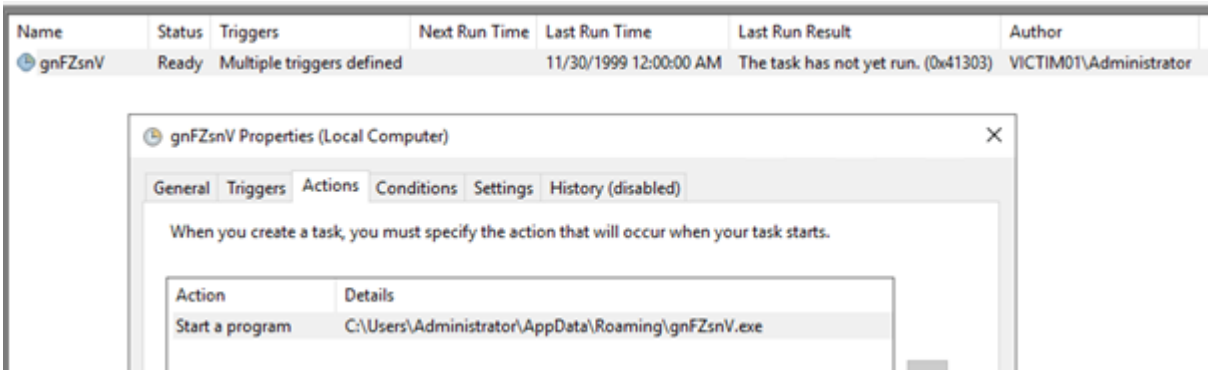


A scheduled task XML configuration is then formed, referencing the path of the dropped executable, and written out to a temporary text file:

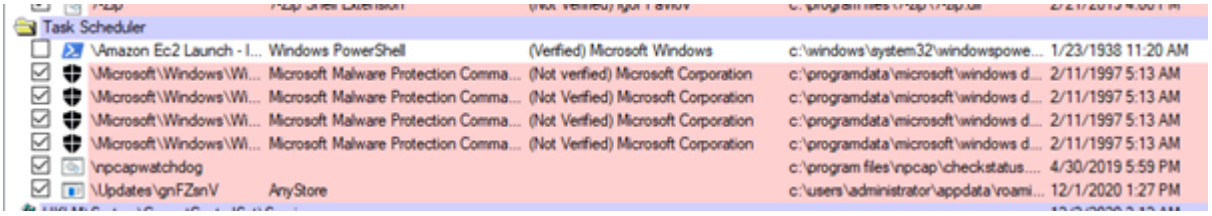


The configuration is passed to schtasks.exe which sets up the scheduled task according to the XML:





In effect, the scheduled task will run gnFZsnV.exe upon logon. Scheduled tasks aren't a particularly stealthy method of persistence, so should a suspected victim of AgentTesla be triaged, the scheduled task will be highlighted by Sysinternals AutoRuns (https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns):



Detection and Mitigation

- **Mail:** Given the predominant method of delivery is mail, robust mail filters, external sender warnings and end-user education form a significant part of defense against AgentTesla.
- **Network:** Depending on the capabilities of your firewall IPS, blocking traffic to known Tor nodes or traffic identified as Tor, combined with SMTP whitelisting, will help to prevent exfiltration. If your firewall supports TLS inspection, do it. FortiGate, ForcePoint and Palo Alto all have signatures for AgentTesla. Public IP lookups using the ipify service are also a potential indicator.
- **Endpoint:** Detection of AgentTesla is not difficult. Reputable EDR products, such as Defender ATP and SentinelOne, will have you sleeping easy. There are also a number of hardening steps that can be taken to prevent the impact of maldocs, including blocking the execution of macros and ASR rules to block Office child processes.

Samples

- swift copy.exe (SHA256: 2ba9db3110899e60daeecb086d4f53adc1cfab127820db3d230c383e74f7172c): https://tria.ge/201201-lbk71xggyx
- exe (SHA256: bd648199b17ff21db3d45cfd10eb3b70fdcbdf42c405061025de6cd1a59c212e): https://tria.ge/201201-3yr3d6cakn

Want More?

If you enjoyed this blog post and want to follow other interesting malware finds that I make, I regularly share them on Twitter: @phage_nz

If you'd like to find out more about how Inde can help detect this security threat, you can contact us here.

Chris Campbell

Chris was that notoriously disobedient kid who sat at the back of the class and always seemed bored, but somehow still managed to ace all of his exams. Obsessed with the finer details and mechanics of everything in both the physical and digital realms, Chris serves as the Technical Director within the Inde Security Team. His ventures into computer security began at an early age and haven't slowed down since. After a decade spent across security and operations, and evenings spent diving into the depths of malware and operating systems, he brings a wealth of knowledge to Inde along with a uniquely adversary focused approach to defence. Like many others at Inde, Chris likes to unwind by hitting the bike trails or pretending to be a BBQ pitmaster. He is also heavily involved in the leadership of security events, trust groups and research projects.
