# Agent Tesla .NET Loader Analysis

Kindred Security

https://twitter.com/kindredsec

*This writeup is a synopsis of a "real-time malware analysis" text feed on the Kind Red Security discord. Because of this, the analysis presented will be much less detailed and flushed out than a standard analysis. You can find the discord here to check out a real-time analysis when they occur: https://discord.gg/CCZCJCu*

In this analysis walkthrough, we're going to be looking at a loader written in .NET which loads an AgentTesla payload. To perform both static and dynamic analysis, we will be using the *dnSpy* tool, which is used for analyzing .NET applications.

---

Starting off, we can look at the symbols used for the main namespace, and see that this loader pretends to be an application called "lexCalculator":

```
// Entry point: lexCalculator.Program.Main
// Timestamp: <Unknown> (92111052)

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

Looking into the main function, it's clear that there's some heavy obfuscation:

```
[STAThread]
public static void Main(string[] args)
{
    int num = 251367141;
    bool flag = num > 251367163;
    if (!flag)
    {
        bool flag2 = num <= 251367196;
        if (flag2)
        {
            num++;
        }
    }
    bool flag3 = false;
    bool flag4 = !flag3;
    if (!flag4)
    {
        bool flag5 = false;
        if (flag5)
        {
        }
    }
```

This is common for .NET malware, and it's usually pretty easy to just ignore all this noise and look for actual function calls to see what the control flow is. For this sample specifically, at the very bottom of this function contains the only call made; everything

else is just pointless boolean and integer math. Therefore, this is where control flow must go:

```
int num20 = 251367149;
bool flag75 = num20 > 251367196;
if (!flag75)
{
    bool flag76 = num20 <= 251367107;
    if (flag76)
    {
        num20++;
    }
}
LexMath lexMath = new LexMath();
```

Looking at the constructor of this LexMath class, there is an immediate call to a method named *CDCDCDCD:*

```
// Token: 0x06000014 RID: 20 RVA: 0x00003552 File Offset: 0x00001752
public LexMath()
{
    this.CDCDCDCD();
}
```

This *CDCDCDCD* method certainly looks to be doing some strange code execution; we can tell due to some of the reflective assembly loading and invoking:

```
// Token: 0x06000017 RID: 23 RVA: 0x000035D8 File Offset: 0x000017D8
public string CDCDCDCD()
{
    byte[] rawAssembly = LexMath.AXAXAXAX();
    Assembly o = this.ac().Load(rawAssembly);
    object o2 = LateBinding.LateGet(o, null, this.Reverse("epy TteG"), new object[]
    {
        "BitBuc ket.Main Hash".Replace(" ", "")
    }, null, null);
    object o3 = LateBinding.LateGet(o2, null, this.Reverse("doht eMteG"), new object[]
    {
        "Star tHashing".Replace(" ", "")
    }, null, null);
    LateBinding.LateGet(o3, null, this.Reverse("eko vnI"), new object[]
    {
        null,
        new object[]
        {
            BinaryMath.Directory,
            "lexCalculator"
        }
    }, null, null);
    return "";
}
```

The actual assembly seems to come from this *AXAXAXAXAX* function, so let's take a look at it. If we can obtain this raw assembly in an independent form, we can move forward and analyze it as well, making it easier to understand what is being invoked.

The *AXAXAXAX* function simply returns one of the resources contained in the .NET assembly named **BitBucket4:**

```
public static byte[] AXAXAXAX()
{
    return Resources.BitBucket4;
}
```

For those of you unfamiliar, resources are extra pieces of data that can be packed into an executable, for example an embedded icon. .NET resources aren't technically PE resources (like the ones you can find in the .rsrc section), but they essentially function the exact same.

So, what we must do next is quite clear; extract this **BitBucket4** resource from the assembly. In order to do so, we're going to patch this executable to, instead of loading the assembly and invoking it, it will dump out the assembly onto disk. We can do that with the following adjustment:

```
public string CDCDCDCD()
{
    byte[] rawAssembly = LexMath.AXAXAXAX();
    File.WriteAllBytes("C:\\users\\jada\\desktop\\bitbucket4.rsc", rawAssembly);
    Environment.Exit(0);
    /*
    Assembly o = this.ac().Load(rawAssembly);
    object o2 = LateBinding.LateGet(o, null, this.Reverse("epy TteG"), new object[]
    {
        "BitBuc ket.Main Hash".Replace(" ", "")
    }, null, null);
    object o3 = LateBinding.LateGet(o2, null, this.Reverse("doht eMteG"), new object[]
    {
```

Upon executing the newly patched executable, we're introduced with our dump file on the Desktop:


Throwing this raw data into a hex editor, we can immediately identify the well known MZ header, indicating that this is an entire PE file:

```
MZ.........ÿÿ..
¸.......@.......
................
............€...
..°..´.Í!¸.LÍ!Th
is program canno
t be run in DOS
mode....$.......
PE..L...“°ì^....
....à..!..... ..
........¾>... ..
.@....@.. ......
................
.€............@…
................
................
p>..K....@..Ä...
```

So, to recap, the executable is loading an embedded PE file from its resource section, then invoking one of the functions contained in that PE file. Since we're loading the entire PE and calling a specific function, this is a good indicator that we're dealing with a DLL.

---

With the embedded DLL extracted and on disk, the next step is to obviously analyze it. Before doing so, let's take a look back at the calling executable, and see what function within this DLL is actually being invoked. We can see by just figuring out the light obfuscation, that it's calling a function named **StartHashing**, and passing two arguments to it:

```
object o3 = LateBinding.LateGet(o2, null, this.Reverse("doht eMteG"), new object[]
{
    "Star tHashing".Replace(" ", "")
}, null, null);
LateBinding.LateGet(o3, null, this.Reverse("eko vnI"), new object[]
{
    null,
    new object[]
    {
        BinaryMath.Directory,
        "lexCalculator"
    }
}, null, null);
return "";
```

Get this Method

Invoke Method with these arguments

Luckily for us, the embedded DLL is also written in .NET, so we can again use *dnSpy* to look at it. If we peek at the list of functions present in the DLL, we do in fact see the **StartHashing** function:

```
Qsn7lxu887o3AAEl5v() : object @06000012
RrgB784lKc67LN5Ncn(object, int, int) : Color @0600000B
StartHashing(string, string) : void @06000004
tsV04EX0yot4Xkj0lC(object, int, object) : void @0600000E
um4PrGZ5dpkTWihoCV() : bool @06000008
```

Looking at this function, there looks to be even heavier obfuscation than there was in the original executable:

```
object obj;
for (;;)
{
    IL_80:
    switch (num)
    {
    case 0:
    case 3:
        goto IL_AF;
    case 1:
        goto IL_3B;
    case 5:
        return;
    }
    IL_1E:
    MainHash.D7Wf5BQ42XHDvfxbEl(0);
    num = 5;
    if (!true)
    {
        goto IL_3B;
    }
    continue;
    IL_9D:
    goto IL_1E;
    IL_3B:
```

The strategy for analyzing this mirrors our strategy from the original executable; instead of getting caught up in the control flow, find the blocks of code which contain variance and notable function calls and variable types. Skimming through the code, I can immediately see another instance of some reflective assembly loading, though this one is much harder to determine what the source of said assembly is:

```
byte[] rawAssembly = MainHash.hp4yZRxX2Ih6crIPbA(MainHash.I3I(MainHash.dZ8IErby3r2E6dBxfb(I2I, I1I)));
Assembly assembly = Thread.GetDomain().Load(rawAssembly);
obj = MainHash.CmXZr7nP5hhlTmUbcC(assembly, null, MainHash.JlMankO2EvQRBgeKm6("[illegible]", 60813), null, null, null);
num = 1;
```

Since the complexity of how this is being loaded in is much greater, it will probably be easier to just debug, set a breakpoint at this spot, then dump out the assembly once hitting the breakpoint. This takes advantage of the fact that, at some point in execution, the raw assembly MUST be fully available in its unobfuscated/decoded form.

---

Since this is a DLL, however, there's a few caveats to how we can debug this function in *dnSpy*. We can easily set breakpoints within this DLL with no problem, and as soon as the executable which you're debugging invokes the DLL, those breakpoints will be hit. The problem is, the original executable which uses this DLL actually invokes the EMBEDDED version of this DLL, not the one we have on disk. This means that *dnSpy* will not catch those invocations, meaning our breakpoints won't be hit.

To remediate this issue, we can just patch the initial executable again, and instead of invoking the assembly embedded in the **BitBucket4** resource, we instead invoke the assembly which is sitting on disk. Remember, these two things are the same exact code; the difference is that using this method we can set and hit breakpoints within the DLL. So, a simple code adjustment allows us to load the DLL from disk instead of from the resource:

```
public string CDCDCDCD()
{
    //byte[] rawAssembly = LexMath.AXAXAXAX();
    byte[] rawAssembly = Assembly.LoadFrom("C:\\users\\jada\\desktop\\bitbucket4.rsc");
    Assembly o = this.ac().Load(rawAssembly);
```

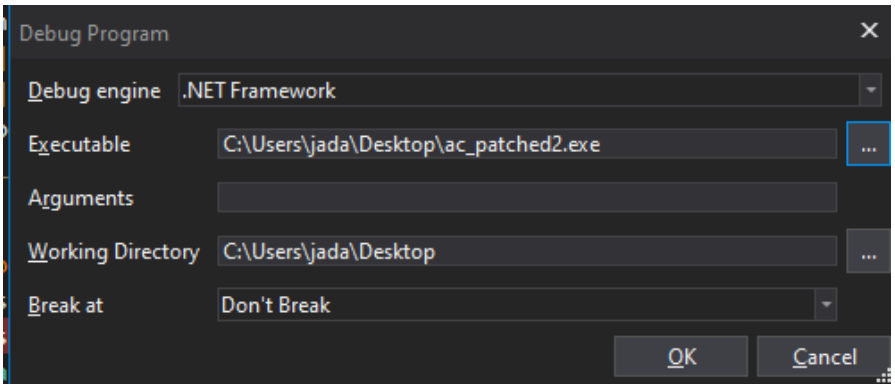
With that out of the way, let's now figure out where to set a breakpoint within the DLL to access the raw assembly. The best place would probably be the function call where the byte array is converted to an Assembly object:

```
55      byte[] rawAssembly = MainHash.hp4yZRxX2Ih6crIPbA(MainHash.I
56      Assembly assembly = Thread.GetDomain().Load(rawAssembly);
```

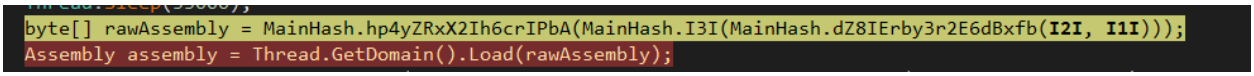
Since the raw assembly will be a parameter to this call, we'll be able to access it from the *dnSpy* interface.

---

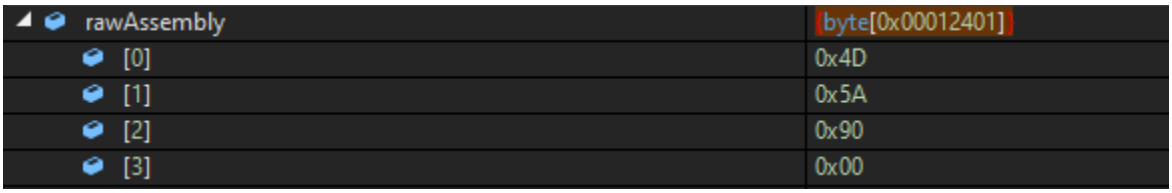
Now that we have our breakpoint set, let's hit it. In order to do so, we're going to use the *dnSpy* debugger and use the patched executable which loads the DLL from disk as the executable to debug:




Once executing, after a bit of time we hit our breakpoint:

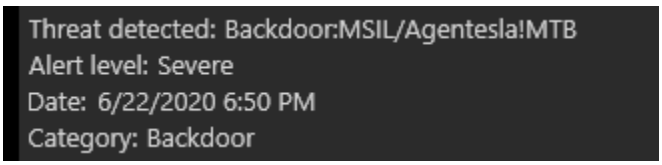



If we look at *dnSpy's* variable interface, we can see the *rawAssembly* variable, which is a byte array:




Notice the first few bytes of this array: 0x4D 0x5A 0x90. This is the *MZ* header which presides all PE files, meaning this assembly is also a full PE file. Let's use *dnSpy's* save functionality to dump this byte array to disk.

As soon as the PE file is written to disk, Microsoft notifies us that AgentTesla was detected:




So, in summary, this loader uses an embedded DLL to extract another embedded executable which serves as the core payload. In this instance, said payload is AgentTesla.