Challenge: AgentTesla Lab

Platform: CyberDefenders

Category: Malware Analysis

Difficulty: Medium

Tools Used: Detect It Easy (DiE), AutoIT Extractor, PE-sieve, Process Explorer, CFF Explorer, dnSpy, CyberChef, ProcMon

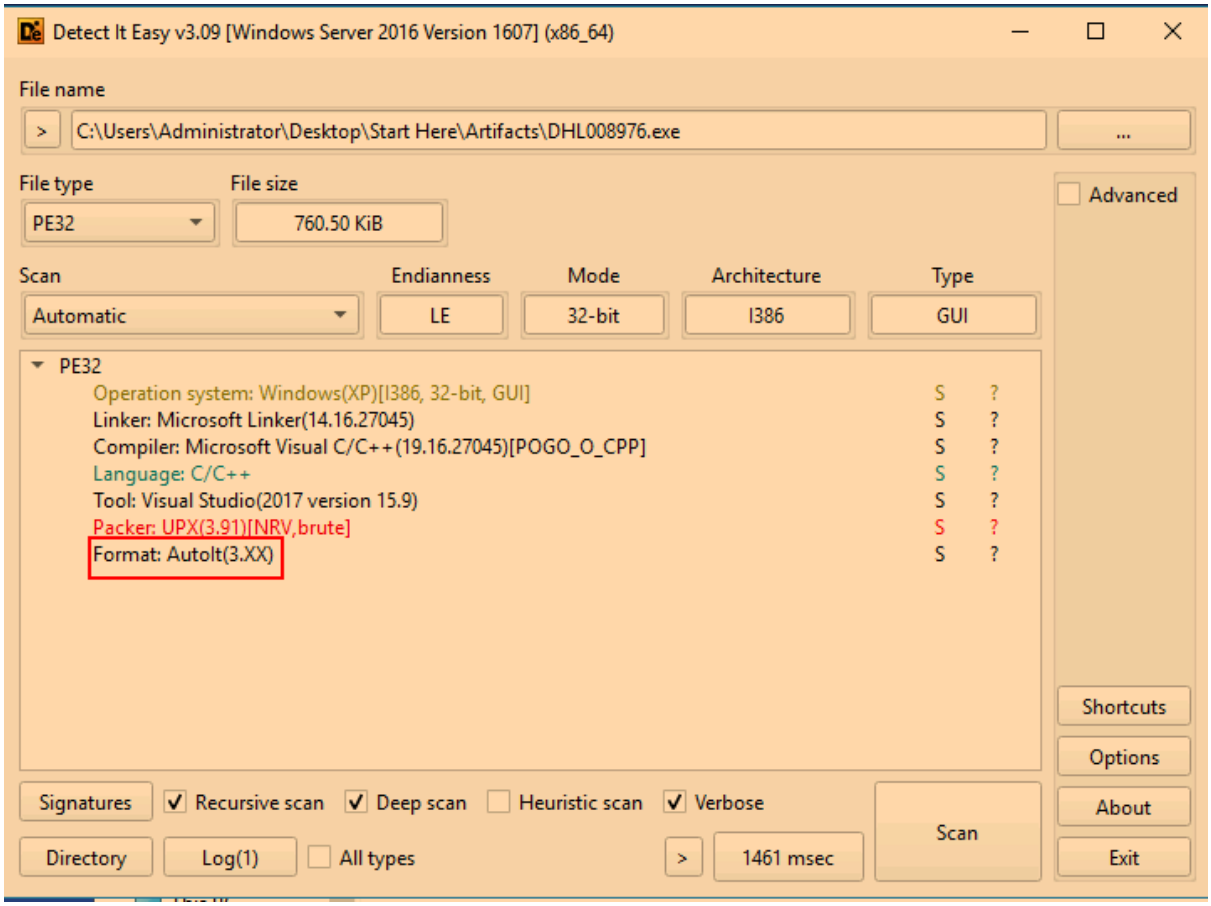
Summary: This lab involved investigating Agent Tesla, a Remote Access Trojan (RAT) and info-stealer malware written in .NET that has been active since 2014. Through the use of several tools, we unpack the malware, observe its behaviour at runtime, extract key IOCs, and examine functionality including its keylogger functions. It is not a deep dive into the sample, only covering a small subset of the malware's capabilities, nonetheless, it is still a great lab for practicing your malware analysis abilities.

Scenario: As a malware analyst at CyberResponse Inc., you are tasked with investigating a piece of malware that has been reported to steal sensitive information like passwords, keystrokes, and screenshots. Your goal is to dissect the malware sample, understand its components, and uncover its attack methods. This task is crucial for developing countermeasures to protect users and organizations.

Identifying the scripting engine or interpreter used in malware can provide insights into its functionality and potential behaviors. What is the name of the scripting engine embedded in this executable?

TLDR: Use Detect It Easy (DiE), focusing on the format field in the output.

Let's begin our analysis by using Detect It Easy (DiE) which is an extremely powerful file type identification tool that is popular among malware analysis. Once you drag the sample into DiE, we can find key information about the file, including the packer used, compiler, and much more. We are interested in the format field:



Here we can see that the scripting language used is AutoIT, which is a free BASIC-like scripting language for Windows.

Answer: AutoIt

Determining the hash of executable components is essential for verifying integrity and identifying malware. When the malware is executed, there are scripts running, one of them is a bin file. What is the MD5 hash of this file?

TLDR: Unpack the malware using the UPX utility. Once unpacked, use AutoIT Extractor to dump the resources contained within the file.

From the previous analysis using DiE, we know that this file is packed using UPX. Therefore, let's start by unpacking this binary using the UPX utility:

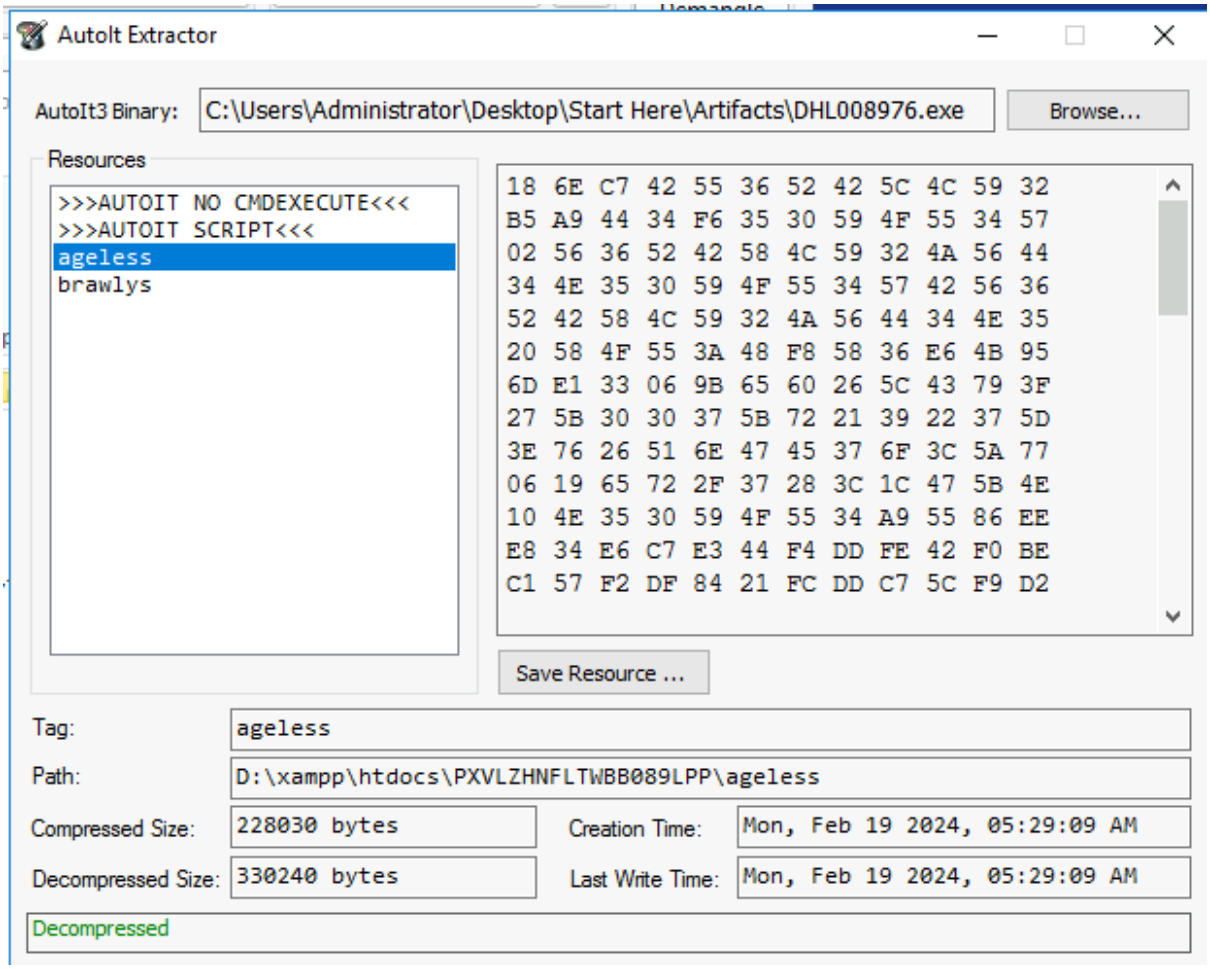
- `.\upx.exe -d "DHL008976.exe"`

```
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2024
UPX 4.2.4       Markus Oberhumer, Laszlo Molnar & John Reiser    May 9th 2024

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   1285632 <-    778752   60.57%    win32/pe     DHL008976.exe

Unpacked 1 file.
```

To continue analysing the unpacked malware, we can use a tool called AutoIT Extractor, which is used to view and extract resources in AutoIT compiled executables like this sample. Once the file has been extracted, we can see multiple resources:



The resource we are interested in is called “ageless”, highlight that file and click save resource. Once the resource is saved to disk, you can generate its MD5 hash by using the Get-FileHash cmdlet:

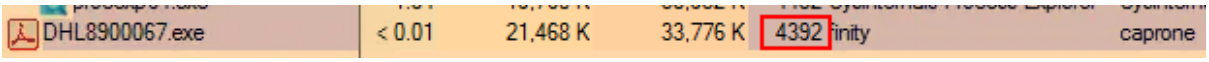
- `Get-FileHash -Algorithm MD5 -Path .\ageless.bin`

```
Algorithm       Hash
---------       ----
MD5             A657189456D164A28B0EB9F5A2654B26
```

Answer: A657189456D164A28B0EB9F5A2654B26

Recognizing key methods in the malware's code helps to pinpoint its main functionalities. What is the name of the method that has main logging functionalities, such as keylogging and screen logging?

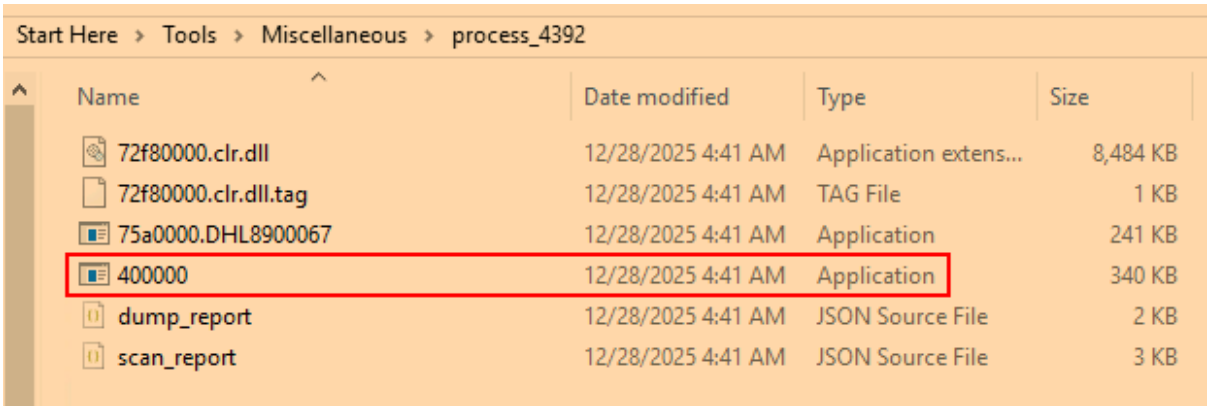
To determine the method for the malware's keylogging and screen logging functionalities, we can start by using a tool called PE-sieve. PE-sieve is a tool that enables you to dump an executable during runtime. This is extremely helpful for analysing obfuscated or packed malware as it eventually gets unpacked or deobfuscated at runtime. Once you have executed the sample, locate its PID using a tool like Process Explorer:



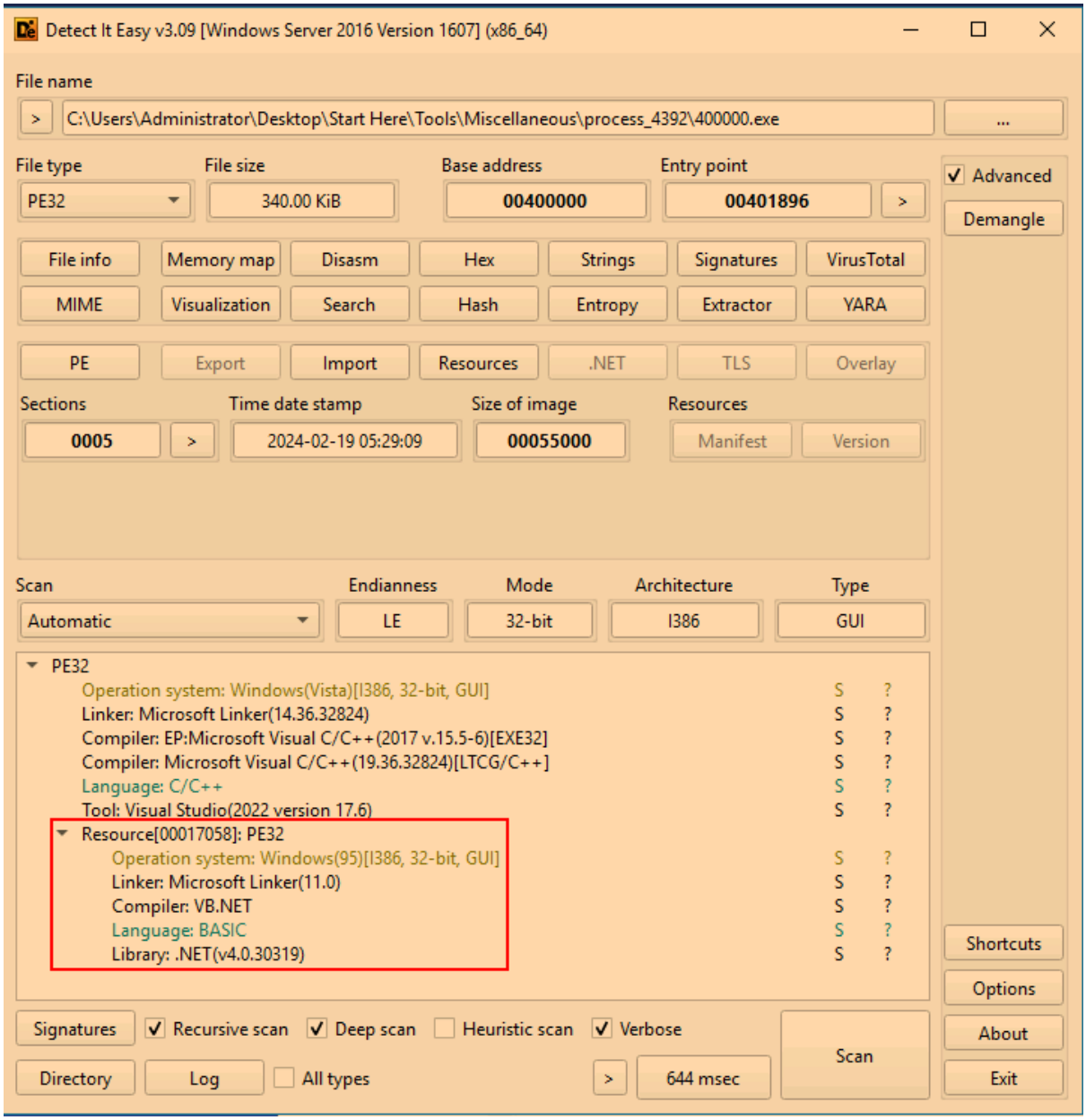
The PID given to the process in my case is 4392, let's now dump this process using PE-sieve:

- `.\pe-sieve.exe /pid <pid>`

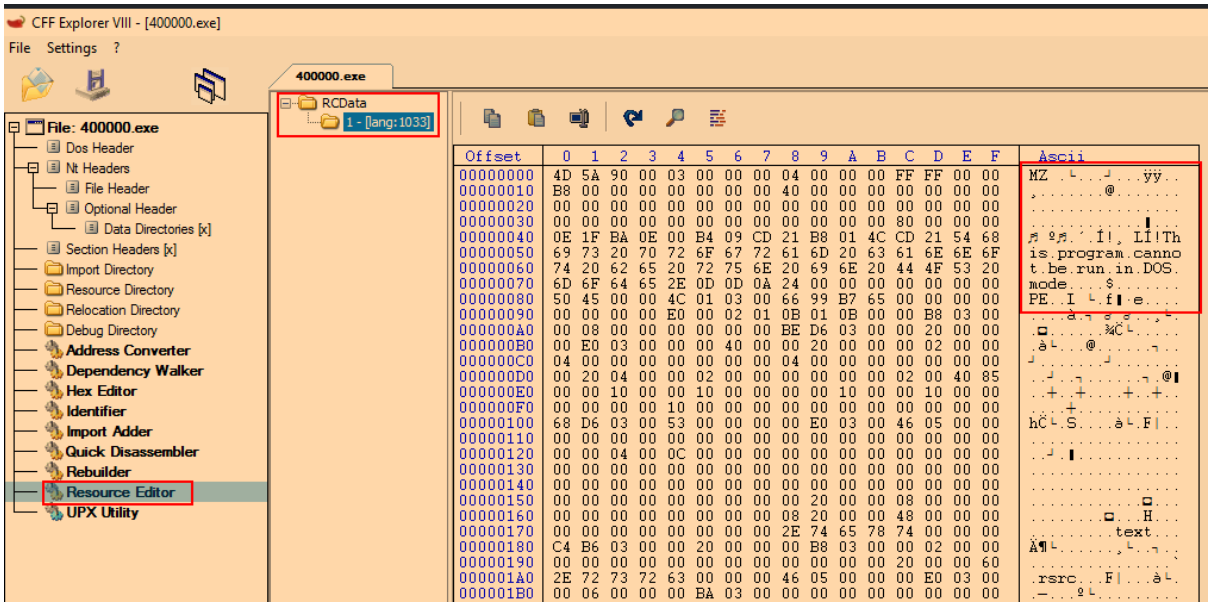
After PE-sieve has completed running, it will create a folder in the working directory with the format process_<PID>, within this folder, we can find the dumped memory:



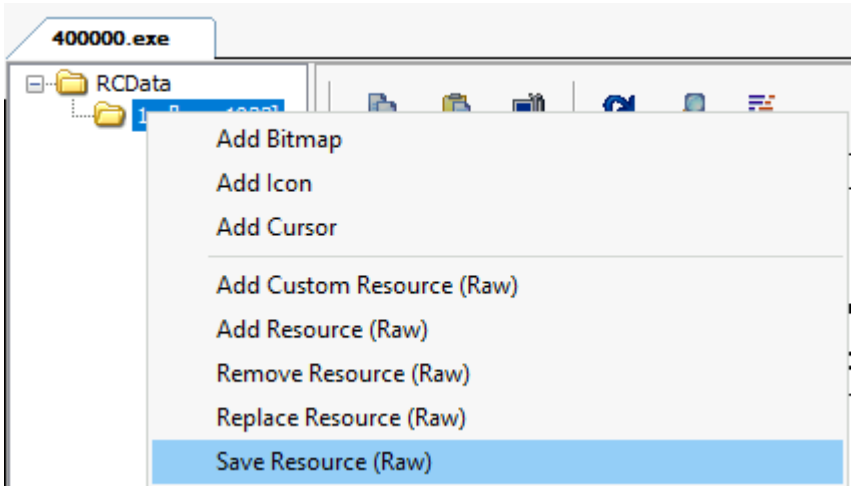
If you load this binary into DiE, we can see that there is an embedded .NET resource:



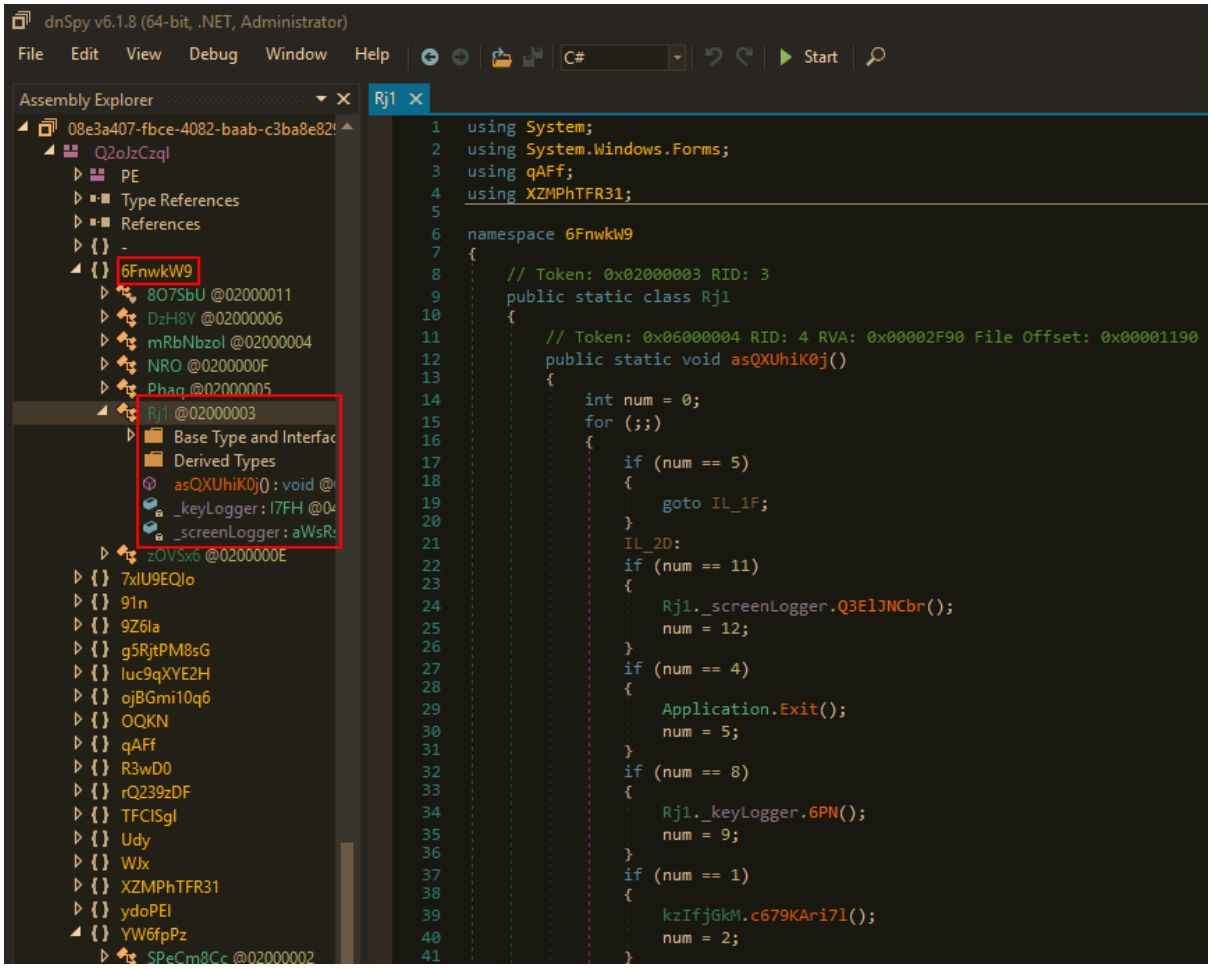
To dump this resource, I am going to use a tool called CFF Explorer and navigate to the Resource Editor section:



To save this resource, right-click it and select “Save Resource (Raw)”:



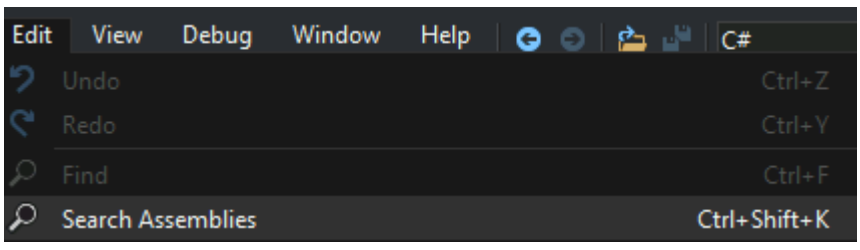
To analyse the .NET binary, we can use dnSpy, which is a debugger and .NET assembly editor. After exploring the methods, I came across the asQXUhiK0j method in the Rj1 class which is responsible for the keylogger and screenshot logger functionality:



Answer: asQXUhiK0j

Understanding the output format of keylogging functionalities assists in tracing and decoding captured data. What is the specific format (programming language) for the output of the malware keylogging functionality?

I started by using the Search Assemblies feature in dnSpy to search for the keyword “keyboard” hoping to come across mentions of keyboard within the code:



Within the output, I can see multiple mentions of keyboard, one that stands out is _keyboardHook:

If you navigate to this class (I7FH), we can see behaviour consistent with keylogging:

```
private void ZjZV4uBvHH(object WYSPMF7IoH, ElapsedEventArgs 1CKzh)
{
    int num = 0;
    bool flag;
    do
    {
        if (num == 1)
        {
            if (string.IsNullOrEmpty(this.KeylogText))
            {
                return;
            }
            num = 2;
        }
        if (num == 2)
        {
            flag = false;
            num = 3;
        }
        if (num == 0)
        {
            num = 1;
        }
    }
    while (num != 3);
    try
    {
        object @lock;
        Monitor.Enter(@lock = this._lock, ref flag);
        mRbNbzol.5glb84Yq(this.KeylogText);
        this.KeylogText = "";
    }
    finally
    {
        if (flag)
        {
            object @lock;
            Monitor.Exit(@lock);
        }
    }
}
```

After further examination of this class, you will come across the nMHN98Wpb method which is responsible for storing the captured keystrokes in HTML encoding:

```
try
{
    bool flag;
    object @lock;
    Monitor.Enter(@lock = this._lock, ref flag);
    string text;
    this.KeylogText = this.KeylogText + "<br><hr>Copied Text: <br>" + text + "<hr>";
}
finally
{
    bool flag;
    if (flag)
    {
        object @lock;
        Monitor.Exit(@lock);
    }
}
```

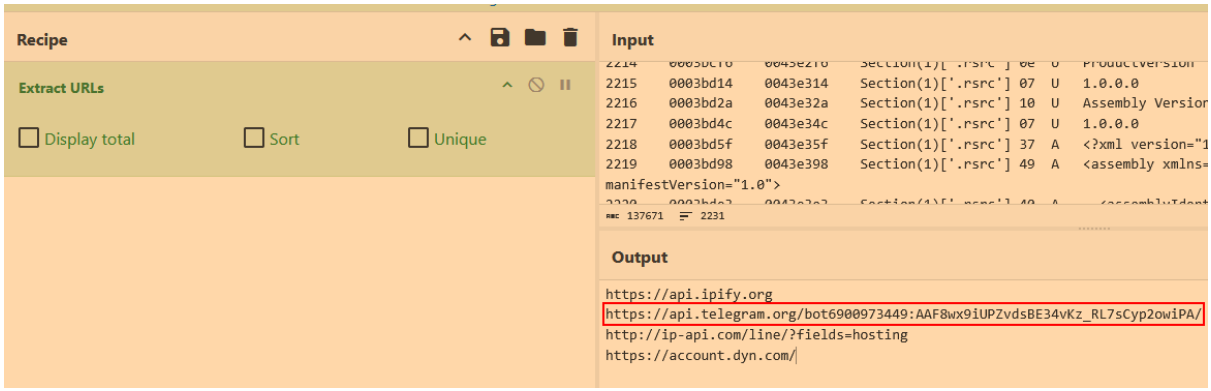
Standard HTML Syntax

Therefore, the output of the keylogging functionality is in HTML format.

Answer: HTML

Knowing the exfiltration methods used by malware is crucial for identifying data breaches and protecting sensitive information. What is the full URL the malware uses for exfiltration of data using Telegram?

To find the URL, I began by extracting strings from the file, you can do so by using the strings utility, FLOSS, or several other tools, in my case I chose to use the strings functionality within DiE. To extract URLs, I used the Extract URLs recipe within CyberChef, which identified 4 URLs:



All URLs are interesting, however, the telegram one stands out. Leveraging the Search Assemblies feature in dnSpy, we can see one class reference the TelegramApi:

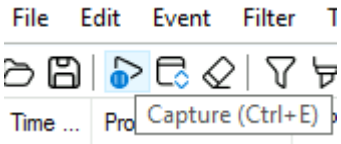

```
// Token: 0x04000019 RID: 25
public static string TelegramApi = "https://api.telegram.org/bot6900973449:AAF8wx9iUPZvdsBE34vKz_RL7sCyp2owiPA/";
```

We can also find other strings within this class that are consistent with configuration data.

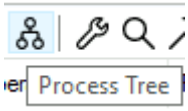
Answer: https://api.telegram.org/bot6900973449:AAF8wx9iUPZvdsBE34vKz_RL7sCyp2owiPA/

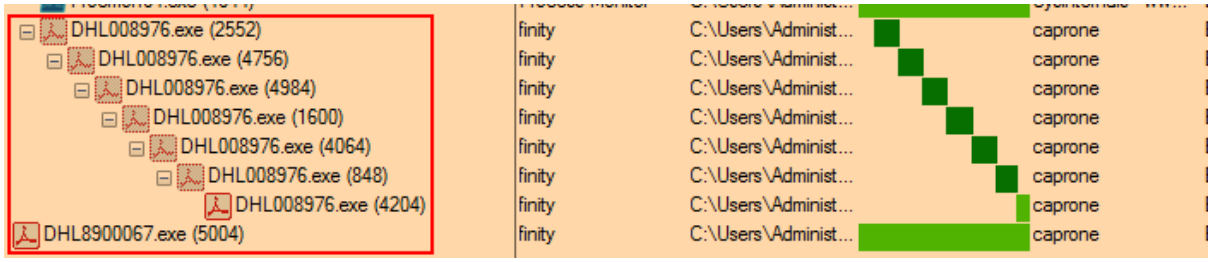
To better understand the persistence strategies of the malware, can you provide the name of the file that was dropped by the malware as part of its persistence mechanism?

To examine the behaviour of this malware during runtime, we can use a tool called Process Monitor (ProcMon). Start by launching ProcMon, pausing the capture and clearing the current events. Before executing the sample, make sure to click the capture button:

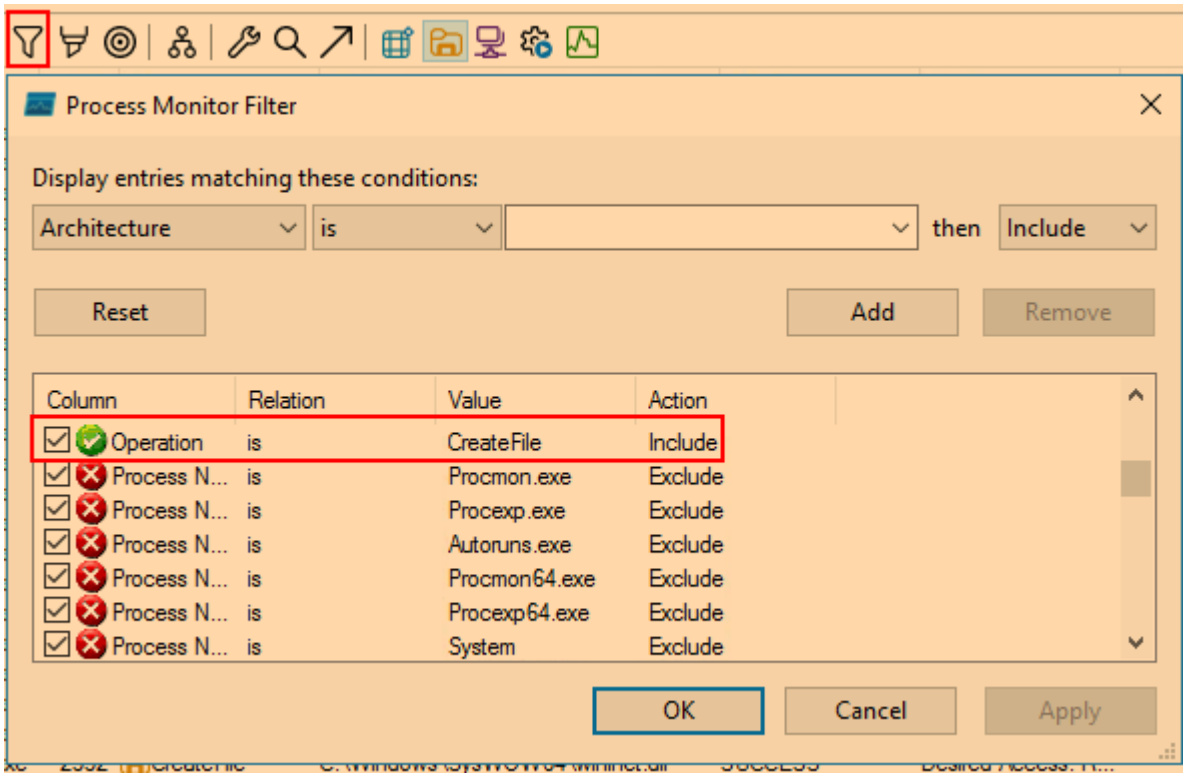


After a moment, we can pause the capture. If you navigate to the process tree, we can see that this malware spawned multiple child processes:

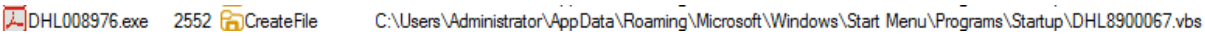




In this instance, we only want to see files created by the malware, therefore, we can create the following filter:



After scrolling through the events, you will eventually come across a file called "DHL8900067.vbs":



The Startup folder is a common persistence mechanism used by threat actors. Placing a file within the startup folder will cause that file to be executed when a user logs in.

Answer: DHL8900067.vbs

Knowing the legitimate services abused by malware can aid in recognizing suspicious network activities. Which legitimate service does the malware use to get the public IP address of the victim?

If you recall earlier, we came across the following string:

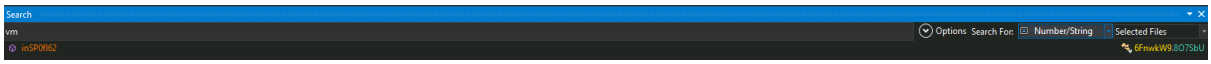
```
// Token: 0x04000009 RID: 9
public static string IpApi = "https://api.ipify.org";
```

ipify is a public IP address API which is used to retrieve the public IP address of the caller. Threat actors use services like this to determine the public IP address of compromised machines.

Answer: ipify

Understanding the anti-VM techniques used by malware is essential for bypassing detection in analysis environments. What is the function name used for the Anti-VM technique?

If you search for the string “vm”, there is only one result for the method inSP0fl62:



Upon double clicking this method, we can see a basic VM detection routine that uses WMI to query information about the host, and check whether it’s a common VM tool like VirtualBox for example:

```
try
{
    using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
    {
        foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
        {
            if ((managementBaseObject["Manufacturer"].ToString().ToLower() == "microsoft corporation" && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) || managementBaseObject
              ["Manufacturer"].ToString().ToLower().Contains("vmware") || managementBaseObject["Model"].ToString() == "VirtualBox")
            {
                return true;
            }
        }
    }
}
catch
{
    return true;
}
```

Answer: inSP0fl62