# Snake Keylogger

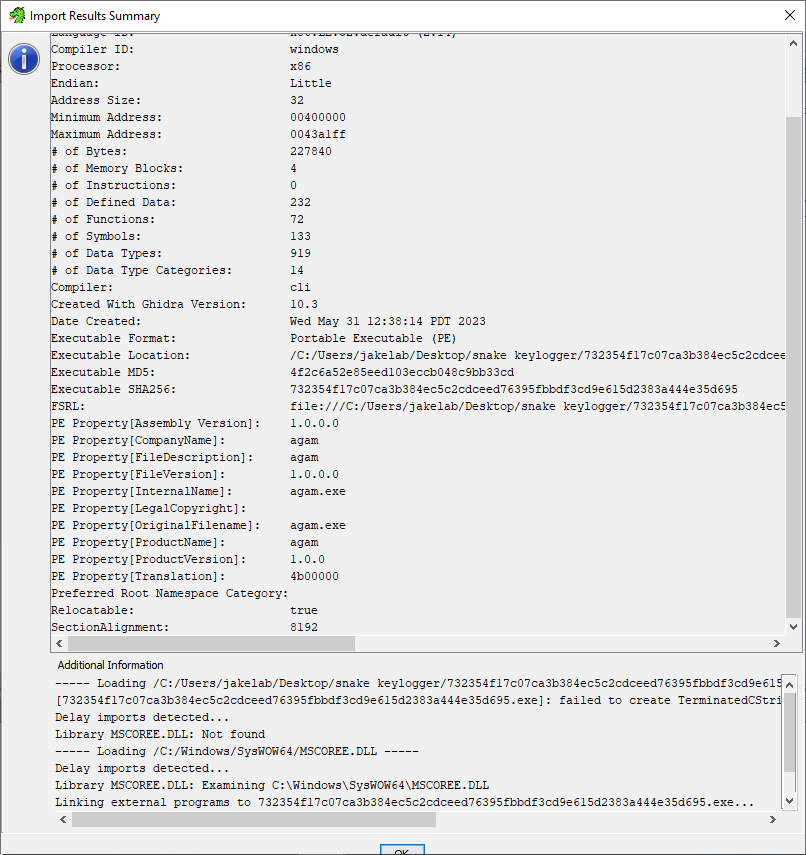
Jake Danson, Theron Hawley, Josh Danson

### Purpose

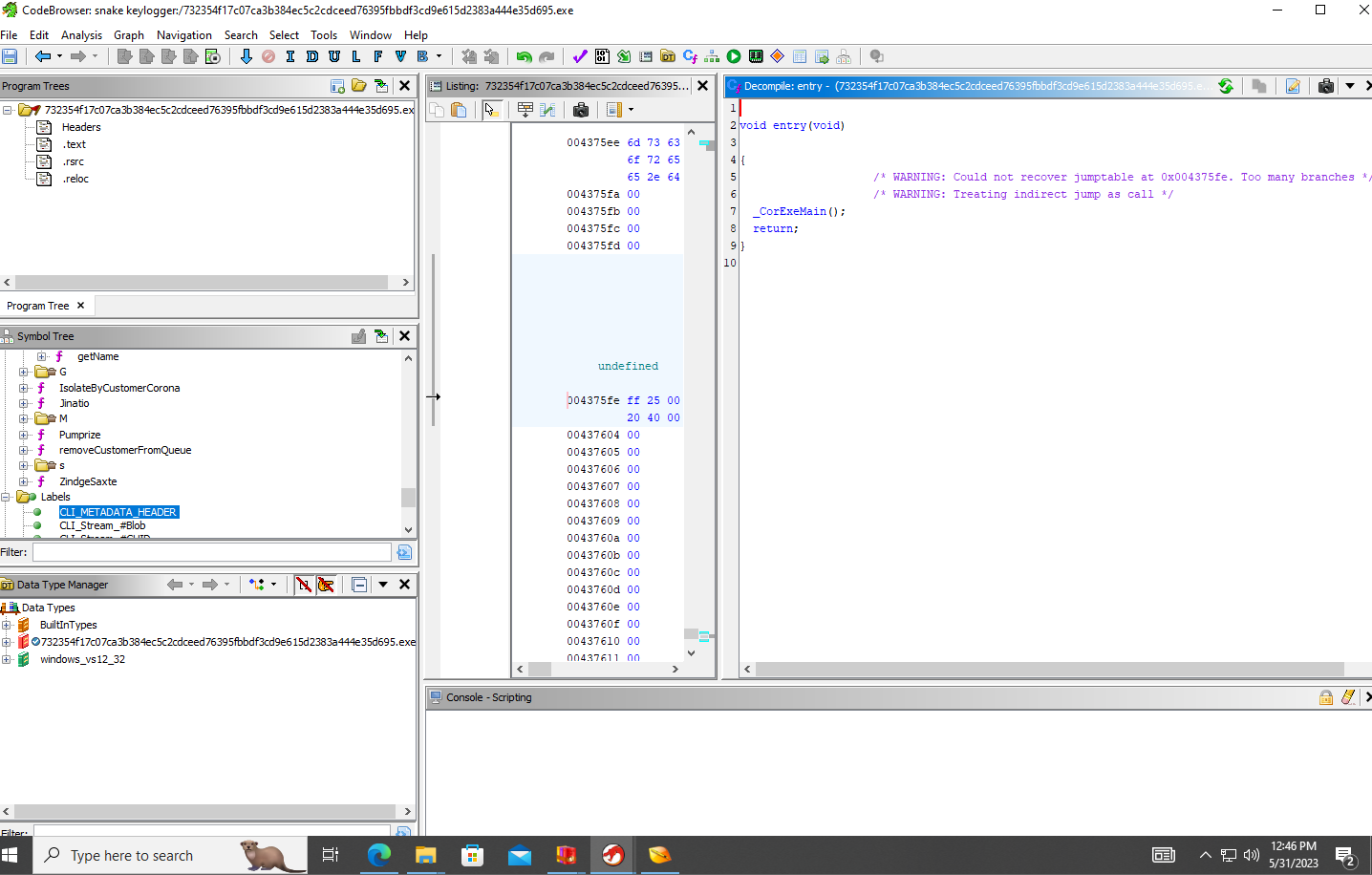
For this assignment, our goal is to investigate and reverse engineer the Snake Keylogger and create a report and presentation on our findings.

### Process

The first step taken was to research the Snake Keylogger. It can be found [here](https://x-junior.github.io/malware%20analysis/2022/06/24/Snakekeylogger.html#iocs). Next, we setup our isolated environment & download the keylogger.



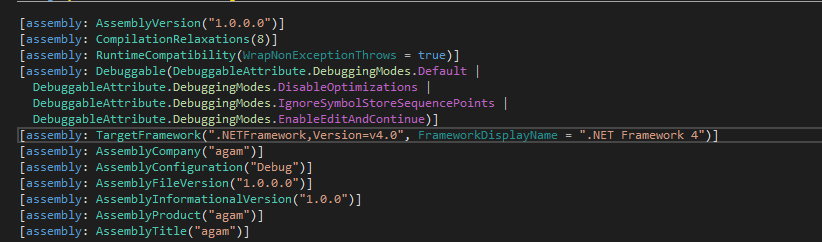
Next, we will begin the debugging process. We chose to start with Ghirdra. On first import to Ghidra, we see the properties of the file. Once we start looking at the function, for some reason the functions all point to \_CorExeMain(), so we need to figure this part out:



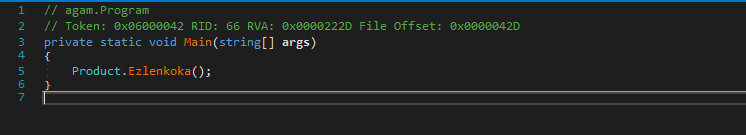
After some research, the best way to reverse engineer a .NET application is to use the dn spy tool. This can be found [here](https://github.com/0xd4d/dnSpy).

After some troubleshooting, we had to disable the antivirus on the machine. Simple & dumb, but good to note.

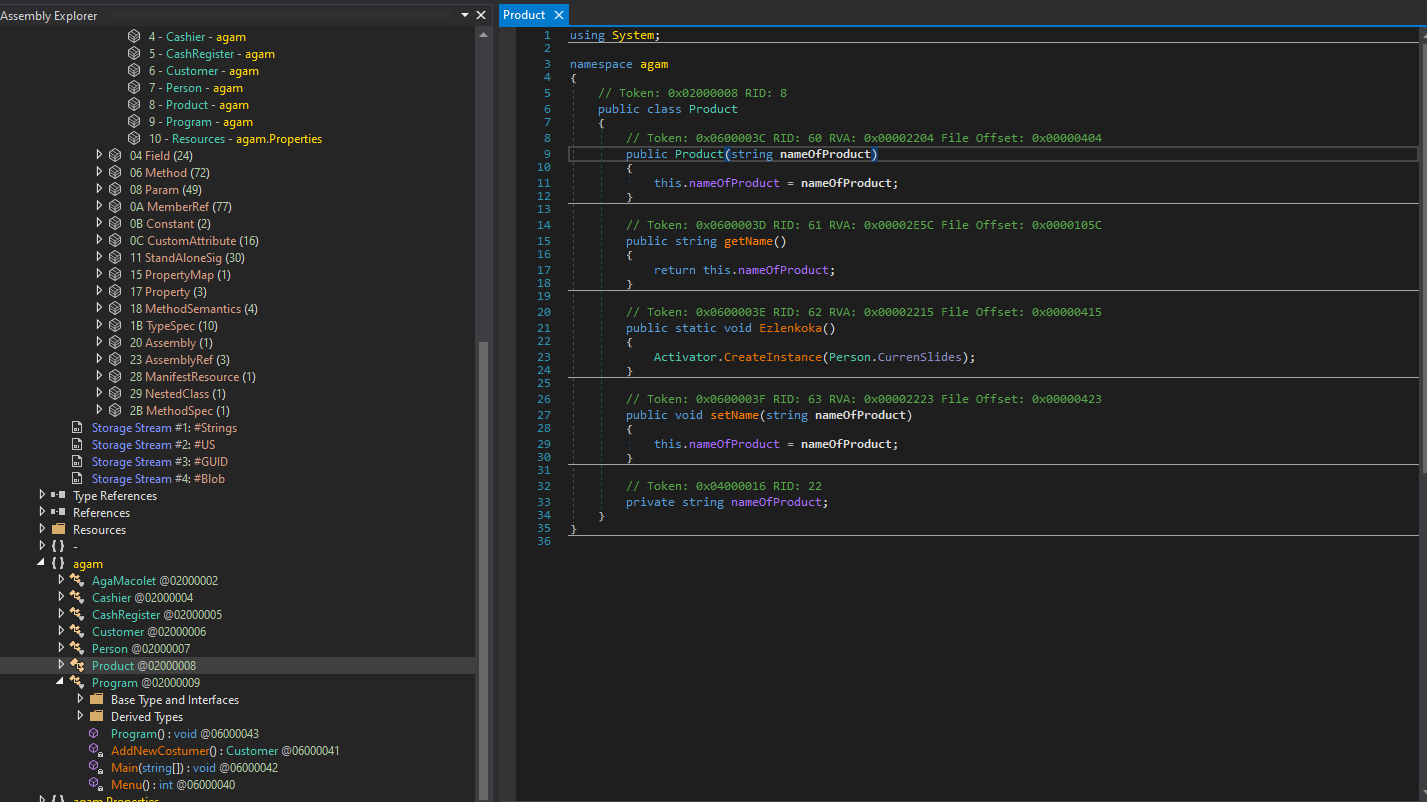
We are working with the .NET version 4 framework. .NET framework can only be run on Windows devices and therefore it is exclusively a Windows vulnerability:



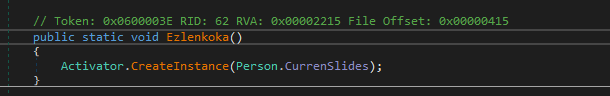
Next, we load the keylogger into DNSpy. This can be found here. We have now found the enterypoint:



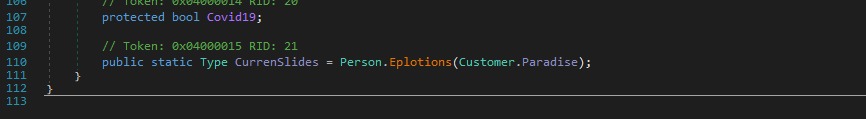
Clicking on Product class, we get:



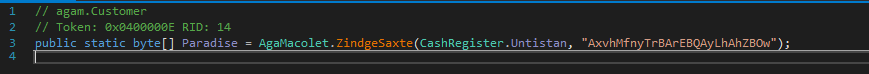
We can see it is creating an instance (note quite sure of what - yet). Since the Activator.CreateInstance method is a System class, lets take a look at the Person class:

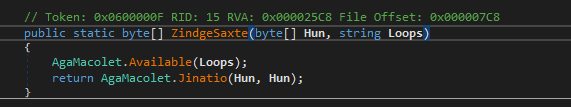


This is interesting. It looks like this malware is attempting to hide itself in a Covid related software. We find at the bottom we find a call to create a Customer class:



Let's look into the Customer class. We found something interesting at the bottom of the class. There is a function and it looks like it is taking a byte array and a string - notably called ‘Loops’.

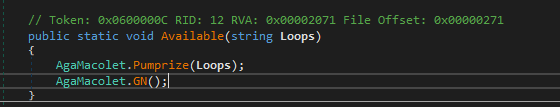




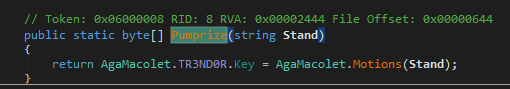
In the next sequence of events, we will be going down a rabbit hole of functions that are called. These functions are of note because they include hashing algorithms, decoding and keys. This is out of the ordinary for a class thats purpose is to establish a person & Covid related information.

We hope to go down this rabbit hole to figure out what exactly everything does, and piece everything together at the end.

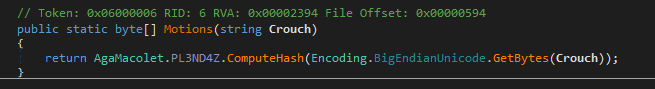
Taking a look of the Available function:



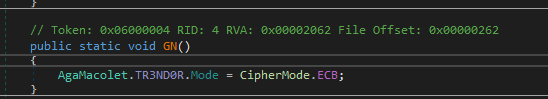
Taking a look inside Pumprize

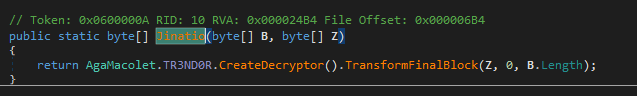
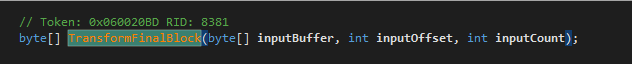


Taking a look inside Motions:



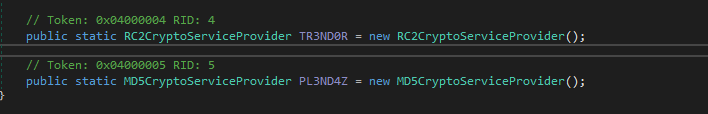
Taking a look inside GN. Setting the CHipherMode to ECB:



Inside Jinato:  
What TransformFinalblock Does:  


— END OF THIS TRAIL —

Taking a look inside of the ZindgeSaxte function which seems to be taking in an array(input buffer), array length & offset:



We found that these functions are creating the ECB mode & setting the encryptor. It is returning some type of decrypted bytes.

To further analyse and understand these functions, we created our own program replicating the functions above:

using System.Reflection;

using System.Security.Cryptography;

using System.Text;

namespace ConsoleApp1;

class Program

{

static RC2CryptoServiceProvider Tren;

static MD5CryptoServiceProvider plen;

static void Main()

{

var Tren = RC2CryptoServiceProvider.Create();

var plen = MD5CryptoServiceProvider.Create();

ConfigureEncryption(); // Configure the encryption settings

byte[] bytes = DecryptData(GetEncryptedData());

var something = Assembly.Load(bytes).GetExportedTypes()[0];

Console.WriteLine(BitConverter.ToString(bytes));

}

static void ConfigureEncryption()

{

// Set up the crypto service providers

// Derive key from a random value

var randomValue = Encoding.BigEndianUnicode.GetBytes("AxvhMfnyTrBArEBQAyLhAhZBOwz");

var key = plen.ComputeHash(randomValue);

Tren.Key = key;

Tren.Mode = CipherMode.ECB; // Set the encryption mode to ECB

}

// Doesn't work, we don't know the encrypted data portion

static byte[] DecryptData(byte[] encryptedData)

{

byte[] decryptedData = Tren.CreateDecryptor().TransformFinalBlock(encryptedData, 0, encryptedData.Length);

// Remove padding if needed

int paddingLength = decryptedData[decryptedData.Length - 1];

Array.Resize(ref decryptedData, decryptedData.Length - paddingLength);

return decryptedData;

}

static byte[] GetEncryptedData()

{

int length = 24; // Number of bytes in the array

byte[] byteArray = new byte[length];

Random random = new Random();

random.NextBytes(byteArray);

// Padding to ensure byte array length is a multiple of block size

int remainder = length % Tren.BlockSize;

if (remainder != 0)

{

int paddingLength = Tren.BlockSize - remainder;

Array.Resize(ref byteArray, length + paddingLength);

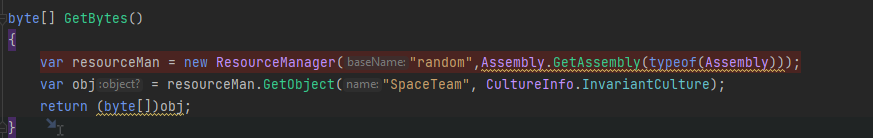
}

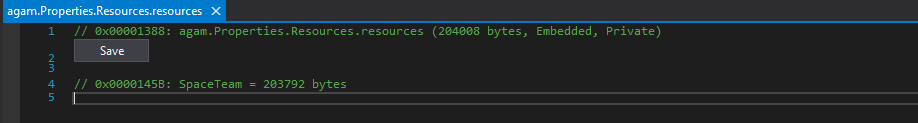
return byteArray;

}

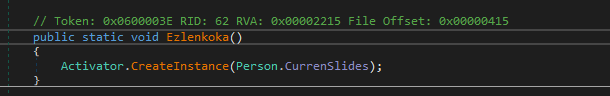
}

After writing, we ran into a problem.

Problem:

* I do not know what the byte array is because I do not have access to the Resources from the properties file. This file is unknown:
  + 

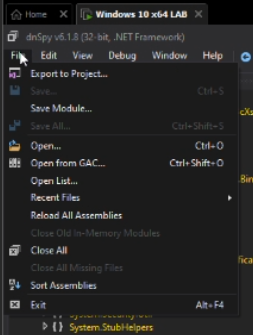
So to recap, the entry point to the program:



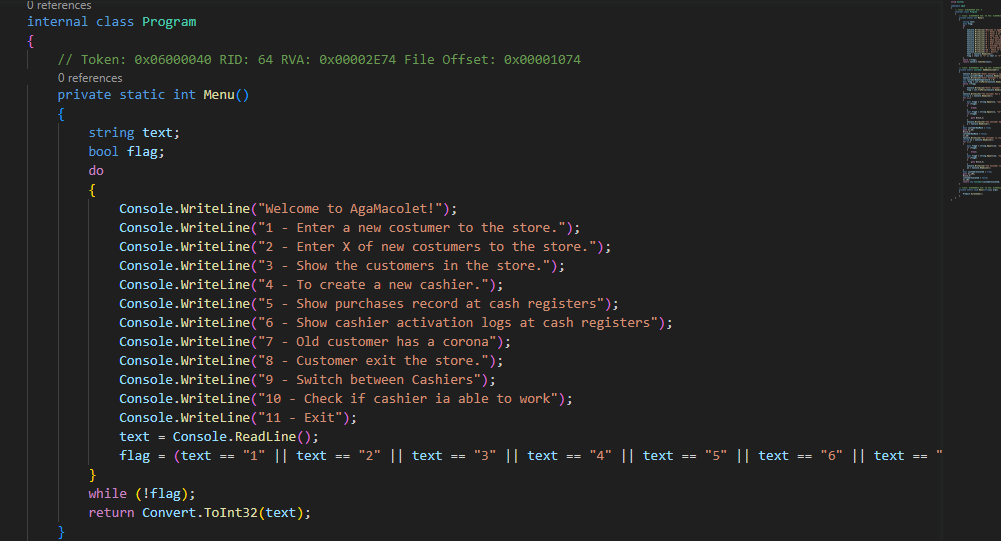
Now that we know what the internals are doing, we can make some assumptions based on the definition of the Activator.CreateInstance method:   
*“An assembly is a logical container that can be either a dynamic assembly generated at runtime or a static assembly stored as a file on disk. It is typically represented by an assembly file (with the file extension .dll or .exe) that contains the compiled code of one or more .NET types.”*

The assumption can be made that the file that we don’t have access to in the properties file is that of some malware. The Person.CurrenSlides is taking a secret that is stored in the program and decrypting the executable within the properties file. Once decrypted, the malicious executable is then called by the Activator.CreateInstance() method, which will run the executable on the victim’s machine.

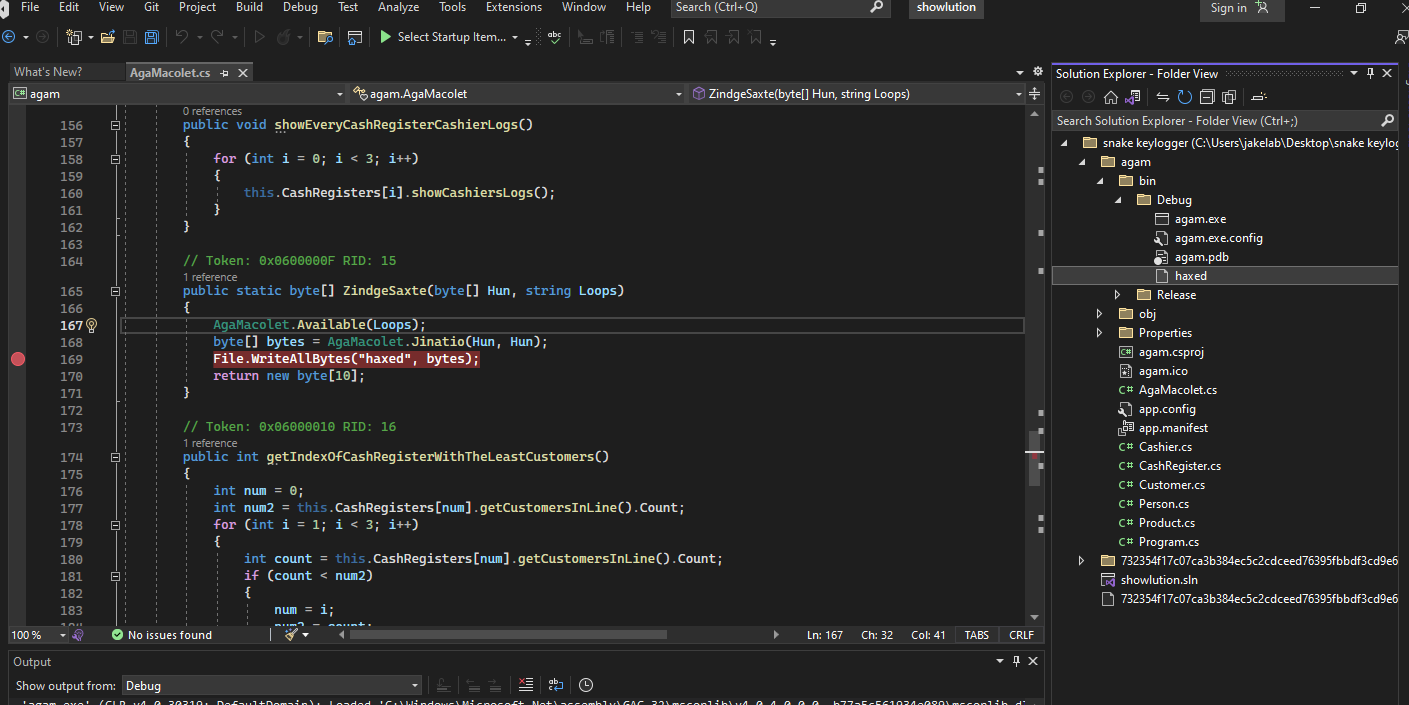
So we exported the solution so we can modify the files and try to rewrite the decrypted assembly file to a new file that we can further debug.



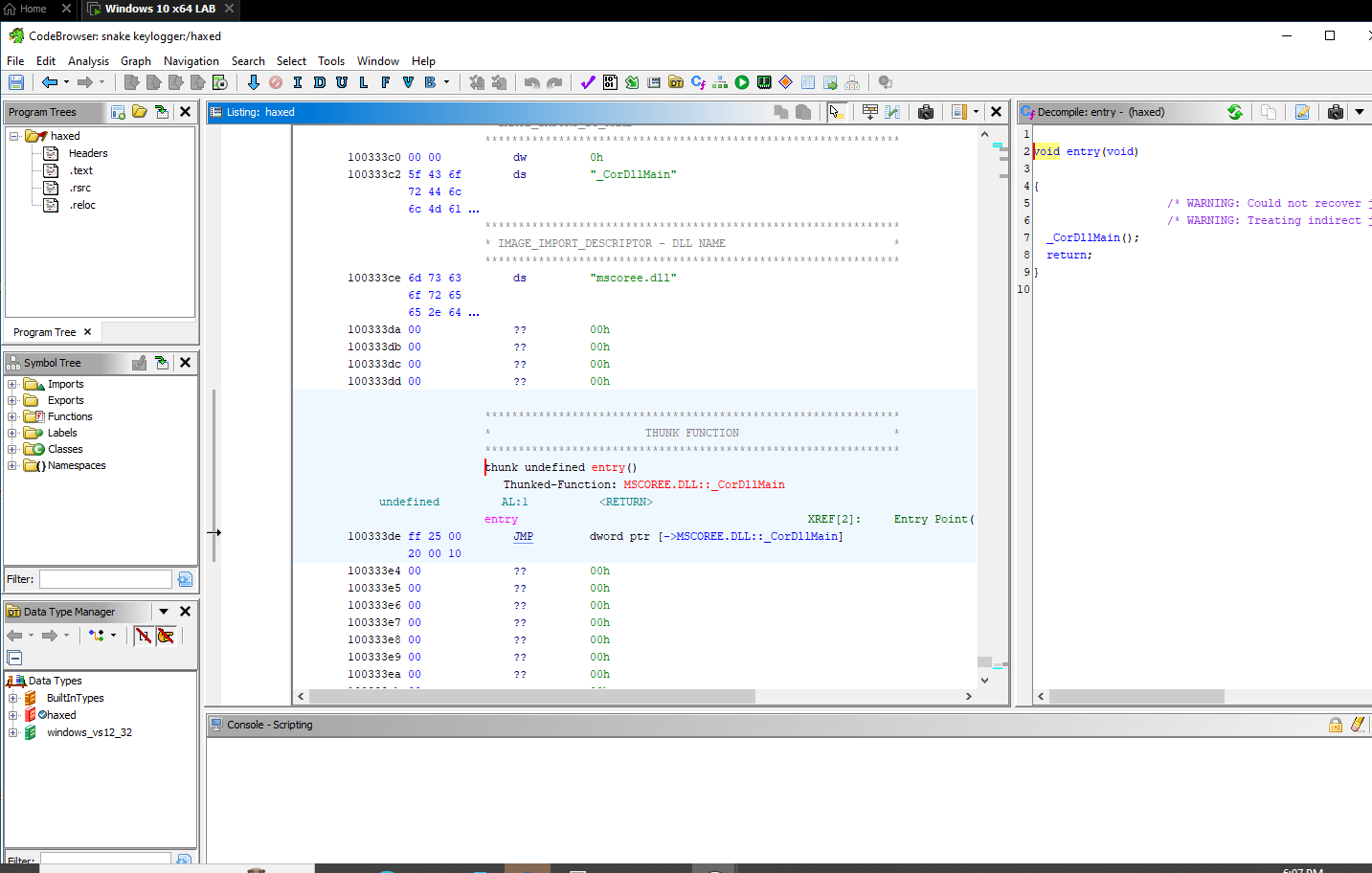
The program’s base interactable menu:



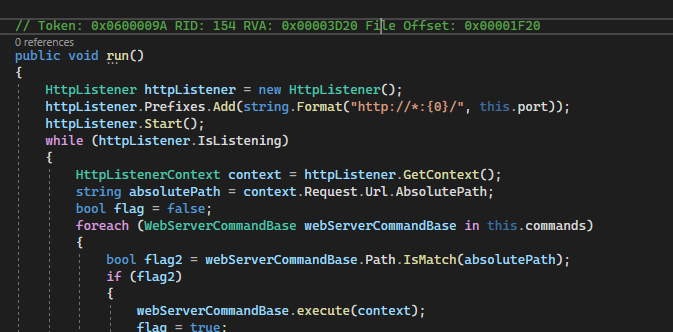
Next, we need to decrypt the payload:



After producing the decrypted file we opened it with Ghidra and POW a file.

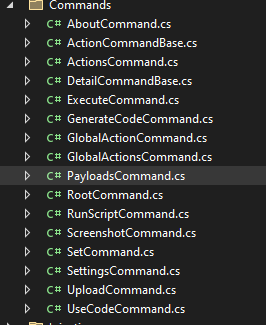


Here is the malware generating an httpListener that hosts html pages with javascript imbedded in the pages.



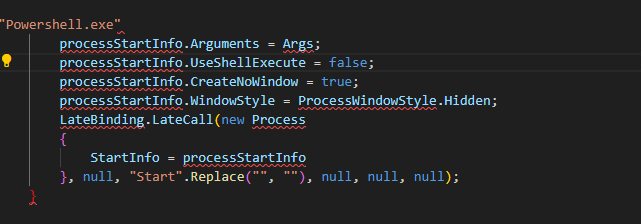
There are different routes on the web page that the hacker can go to, which will send them data that the malicious software was able to collect

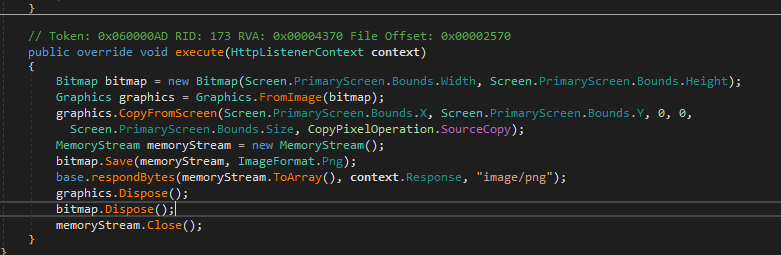
Commands and backdoors that the malware accepts:

* + 

Here is the execution of the malware copying itself to startup folder:

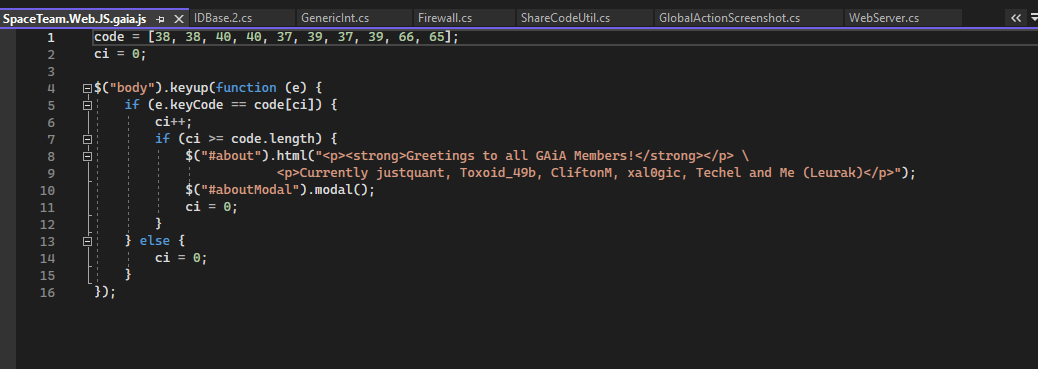




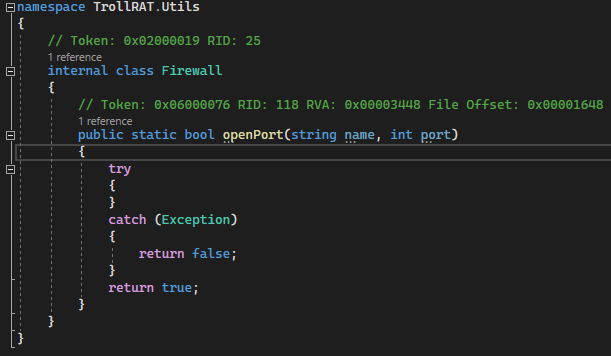


Other significant items:

Found the members & hacker group who made this:



Notable firewall openPort method that doesn’t seem to do anything? Maybe the http listener opens one up by default?:



### Learning Opportunities

Here some things we learned in the process of completing this assignment:

* Remember to disable antivirus when attempting to run a virus!
* Take an extra 5 minutes! - We reached a dead end and decided to go down one more rabbit hole that lead us to decrypting the malicious file.
* Control + enter allows you to find and replace newlines in VS code.

### Conclusion

It became apparent that the keylogger is being activated by the Activator.CreateInstance() method which is calling to run an executable or dll file. The activator is getting the information from the Person.CurrentSlides() function which then goes deep into the other method calls and eventually decrypts the dll file stores in the agam.Properties.Resrouces.resources file. This is where the keylogger is called and runs. It turns out the keylogger isn’t just a keylogger, but in fact a RAT as well. The rat runs a web service locally and creates a backdoor. Overall, this was a great learning experience and found a lot of interesting things that we did not expect.