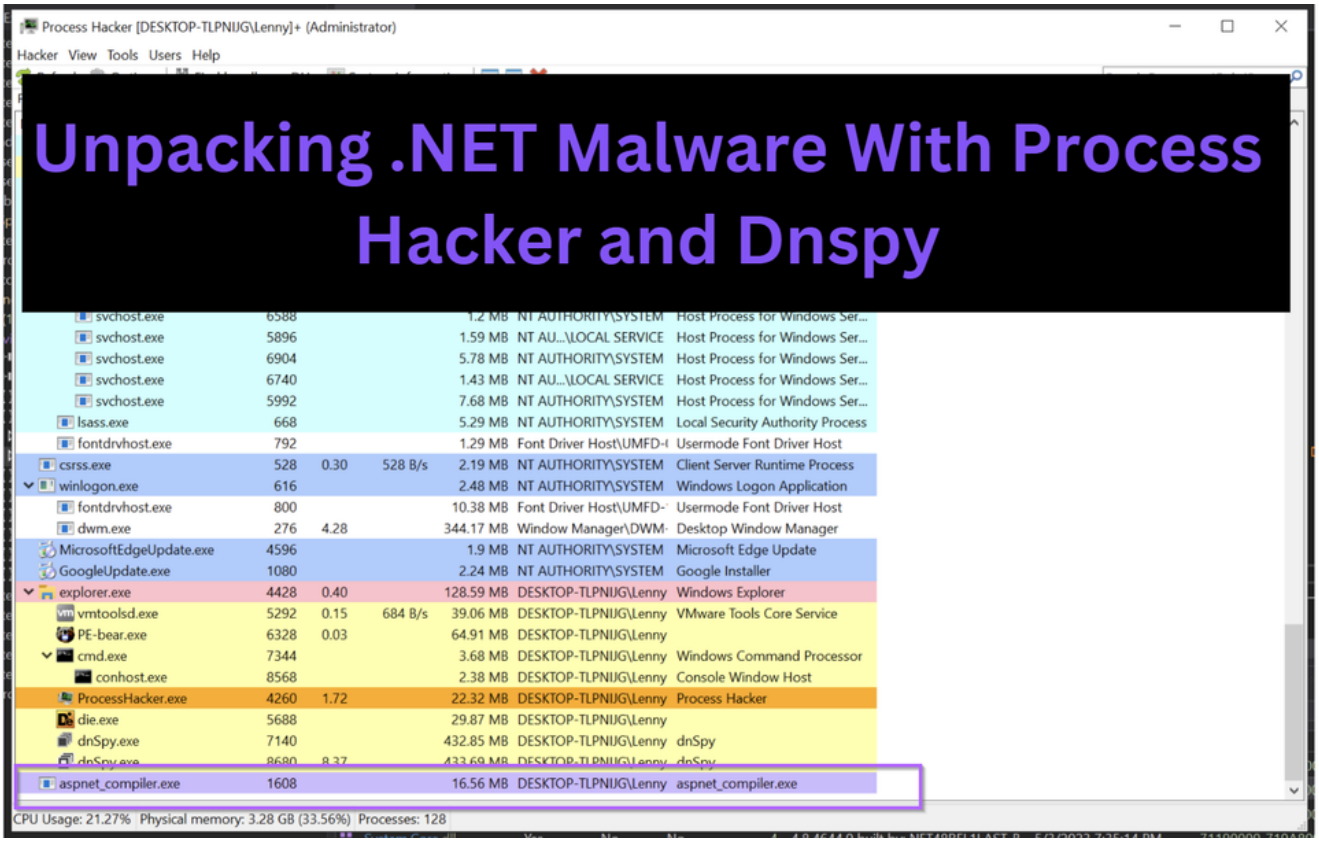
Unpacking .NET Malware With Process Hacker and Dnspy

embee-research.ghost.io/unpacking-net-malware-with-process-hacker/

Matthew October 30, 2023

Last updated on Oct 30, 2023



Unpacking malware can be a tedious task. Often involving intensive static analysis and in-depth knowledge of debugging.

In this post, I'll demonstrate an easy method that can be used to unpack files that ultimately load a .NET based malware.

This method primarily involves running the file and monitoring for process executions using Process Hacker. Upon execution, Process Hacker can be used to observe any .NET files loaded into memory. If a file is identified, it can then be obtained using Dnspy.

Link to the File

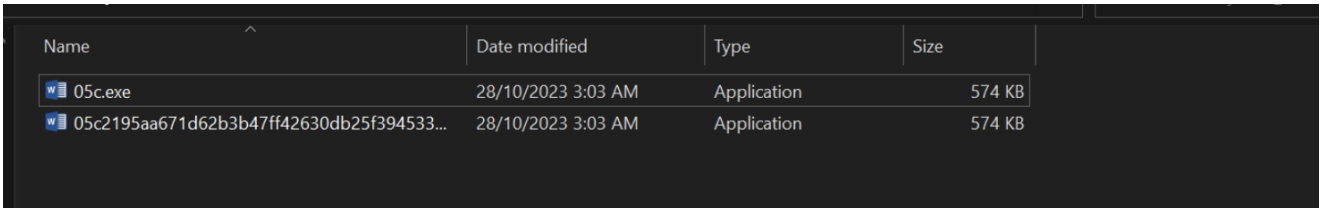
Sha256: `05c2195aa671d62b3b47ff42630db25f39453375de9cffa92fc4a67fa5b6493b`

Malware Bazaar

Analysis

I will begin analysis by saving the file into my virtual machine and unzipping it with the password `infected`.

After unzipping, I will also create a copy of the file with a shorter filename.

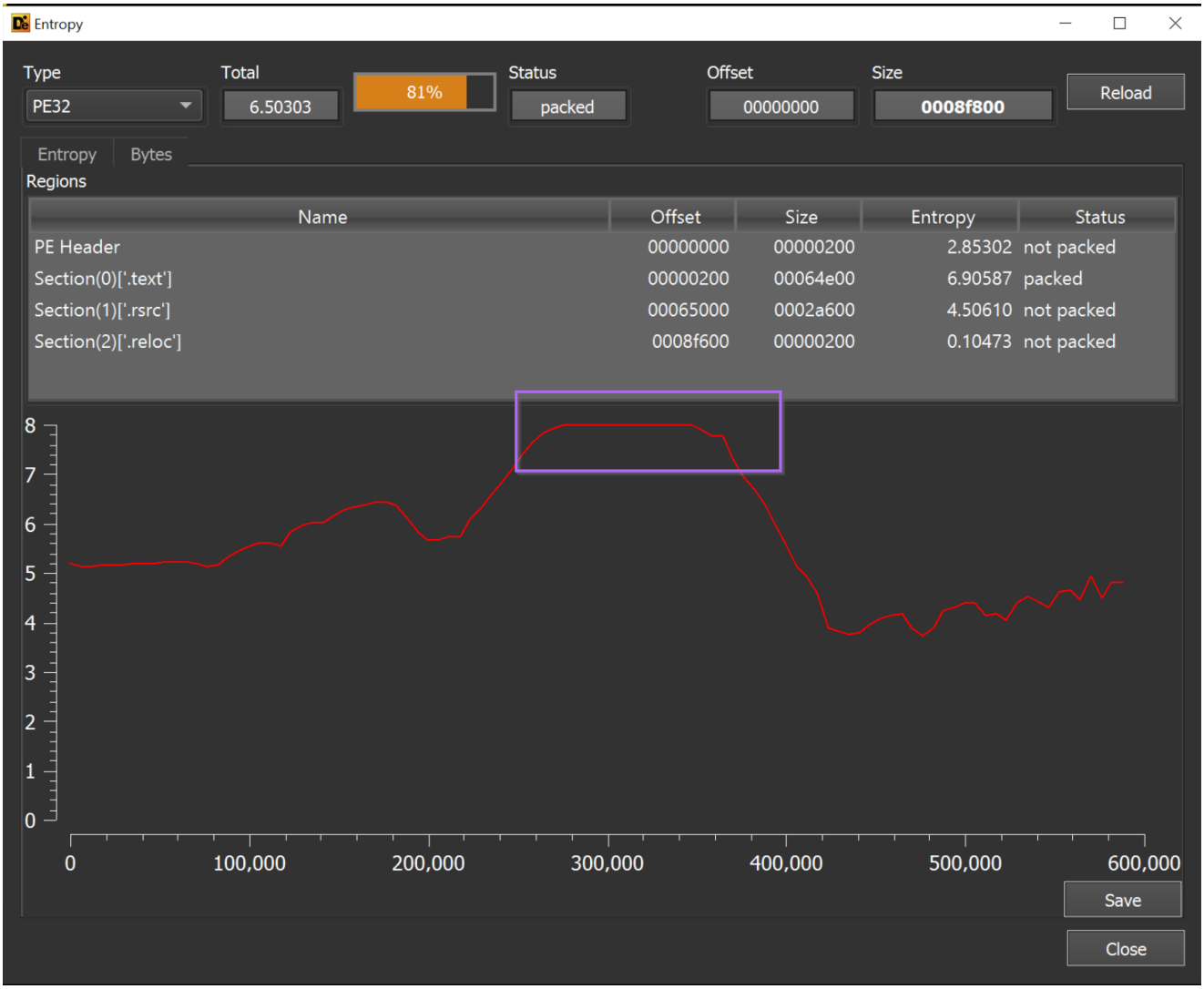


I will also perform a basic initial assessment using Detect-it-easy.

Initial Assessment with Detect-it-easy

My primary goal here is to review the entropy graph. Here I can determine if there are any high-entropy areas large enough to store a file.

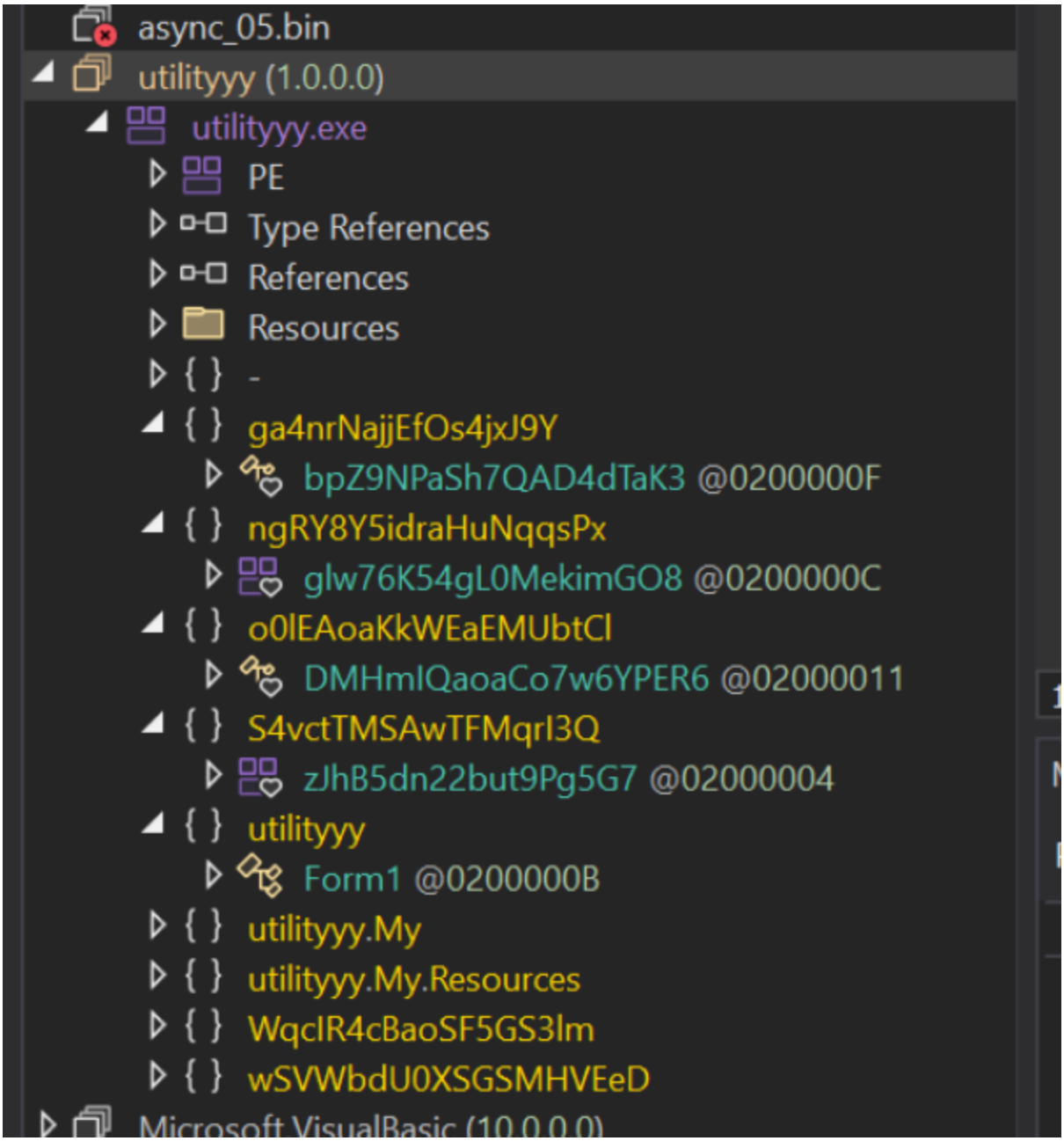
In this case, there is such an area (as seen in below screenshot). This area suggests that the file could be a loader (as it contains a possible encrypted payload).



Initial Assessment With DnSpy

Before attempting to unpack the file, I will also open it within DnSpy.

This is to make sure that the file is not already unpacked. In my initial assessment, I didn't see any functionality that suggested the file was already unpacked.



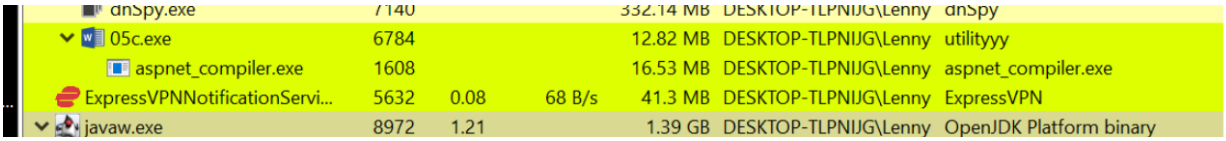
Observing Unpacked Content With Process Hacker

At this point, I want to run the file and attempt to let it unpack itself.

This can be achieved by running the file for a few seconds, and observing the process as well as any new processes that are spawned.

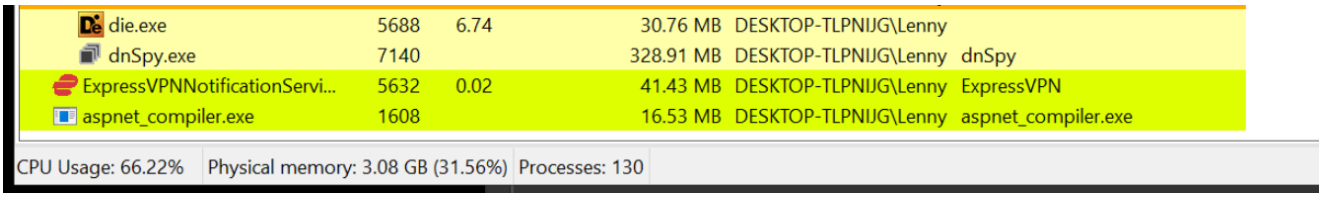
After a few seconds have passed, we can go ahead and view the process to see if any new .NET modules have been loaded.

Running the file for a few seconds, we can see that it spawns `aspnet_compiler.exe`. This is suspicious and something we can hone in on.



We can also observe that after the new process is spawned, the original process `05c.exe` exits a few seconds later.

This is an indicator that any suspicious or unpacked content is likely contained within `aspnet_compiler.exe`.



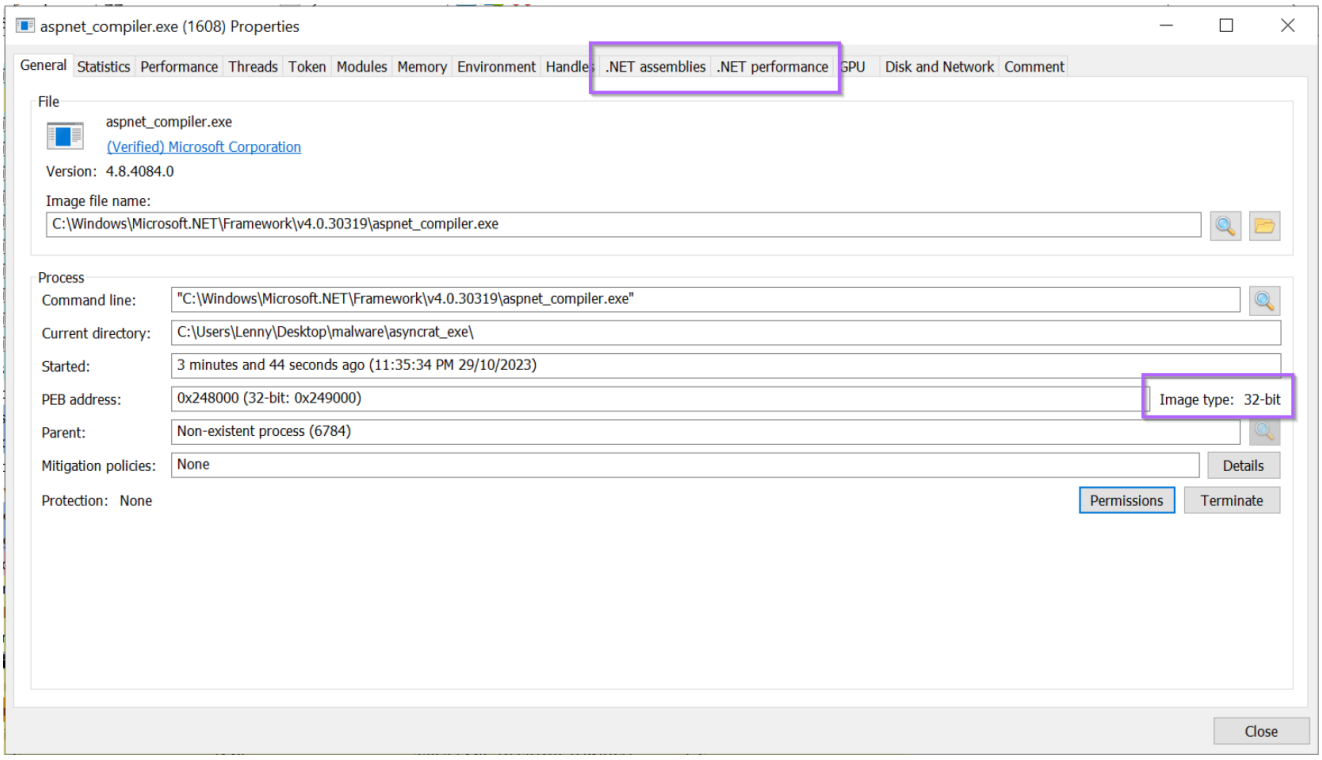
Identifying Unpacked .NET Files Using Process Hacker

With the suspicious `aspnet_compiler.exe` identified, we can go ahead and inspect it using Process Hacker.

We can do this by double clicking on the process name, or right-clicking and selecting "Properties".

This will open a window like the following. There are two main points here.

- `.NET Assemblies` tab - This shows us that some kind of .NET module is loaded into the process.
- `Image Type - 32bit` - The process is 32-bit, this tells us that any future debugging will require a 32-bit debugger (eg Dnspy x86)
- `(Verified) Microsoft Corporation` - This is likely a legitimate process that has been hijacked.

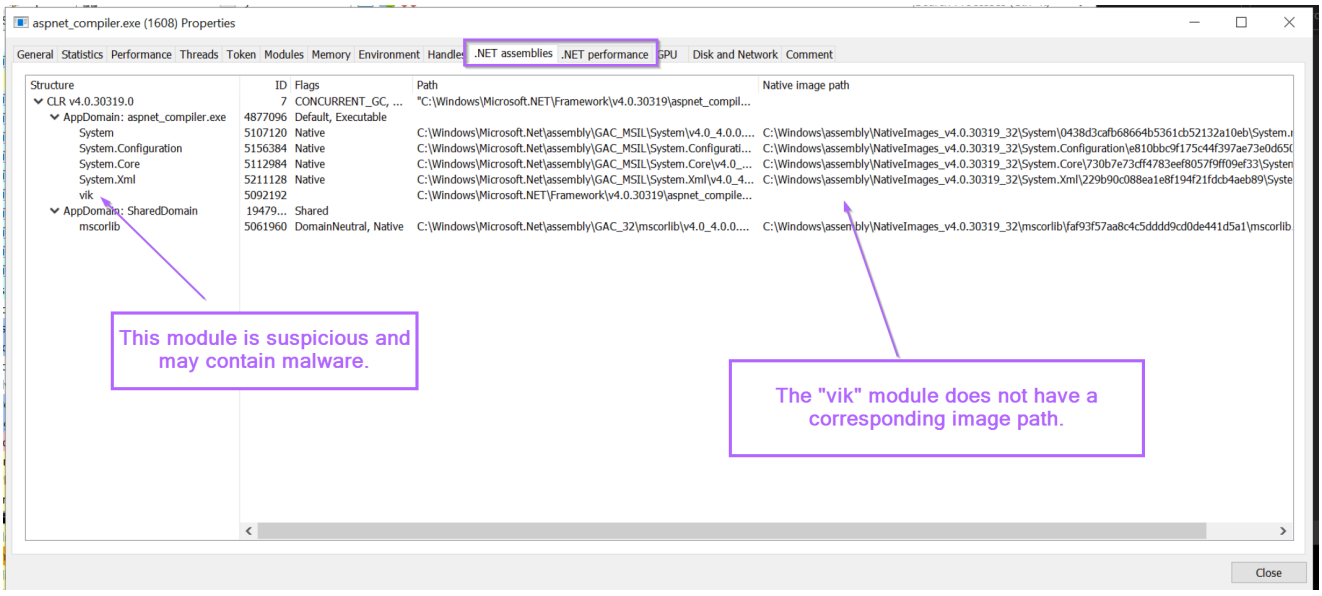


Inspecting Loaded .NET Modules With Process Hacker

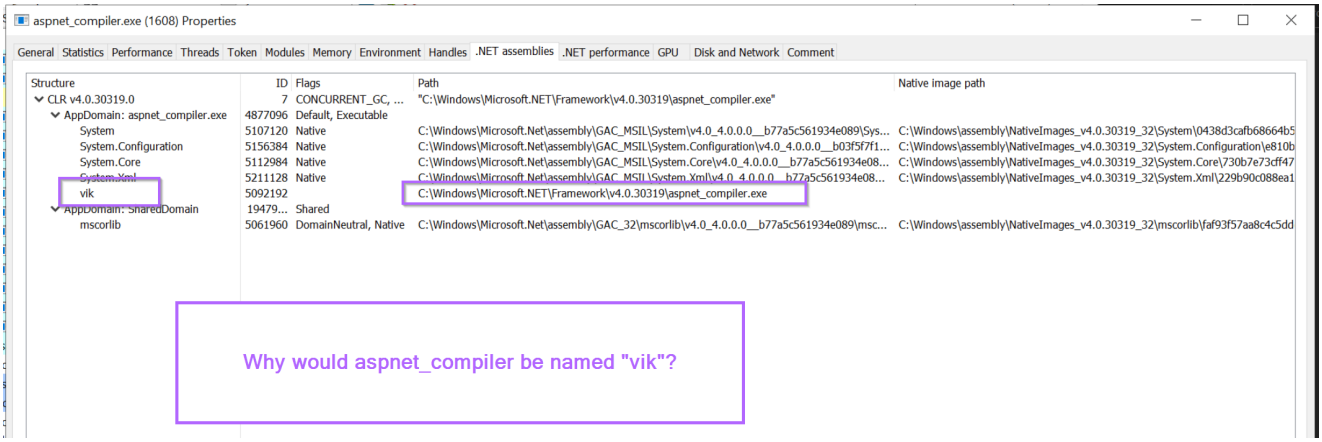
We can go ahead an inspect any loaded modules with the `.NET assemblies` tab.

This will list any loaded .NET modules within the current process. As well as information for each module. We want to look for loaded modules that look out of place, or different to the others.

In this case, there is a loaded module named `vik` that doesn't look right. It has a completely different style of name to the other modules, and doesn't have a corresponding native image path (like all the other modules)



If we look closer, we can also see that the "regular" path is that of `aspnet_compiler.exe`. This is suspicious, why would aspnet_compiler be named `vik`?



Verifying Suspicious .NET Modules Using DnSpy

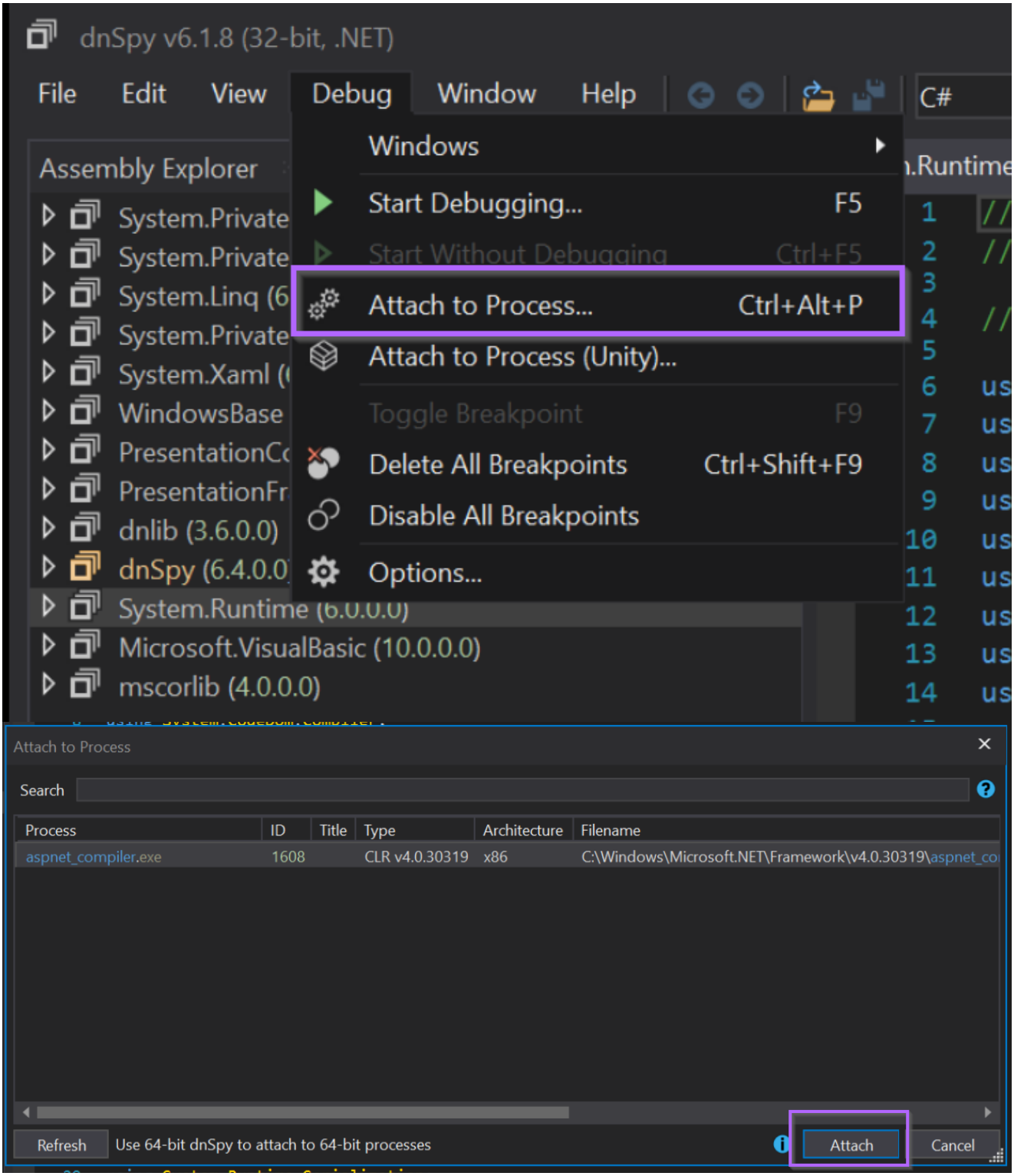
Now that we have identified a suspicious module, we can go ahead and obtain it using DnSpy.

To obtain the file, we can open up Dnspy (32-bit) and attach to the `aspnet_compiler.exe` process.

This will allow us to inspect the loaded modules and view their corresponding source code.

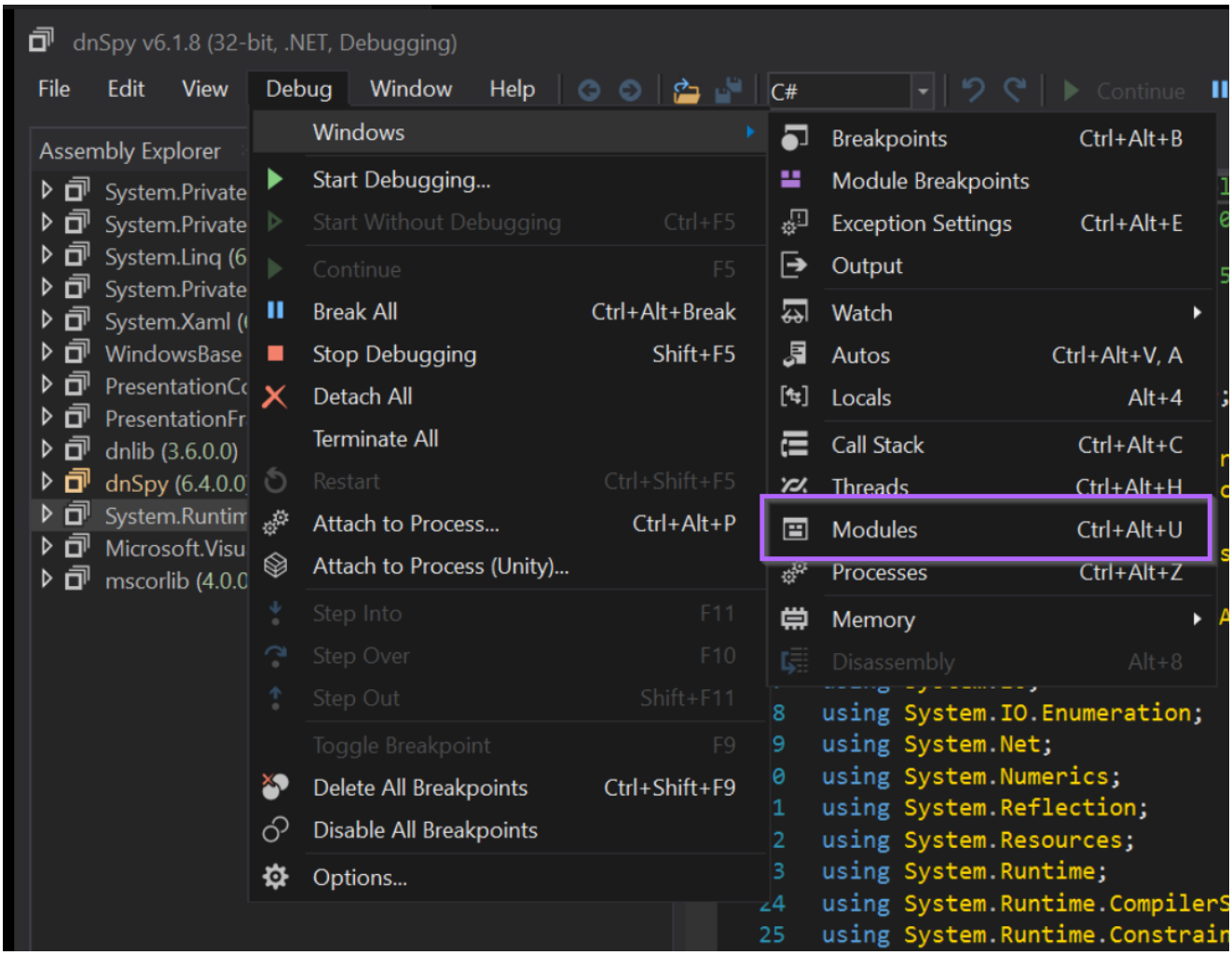
Attaching Dnspy To a .NET Process

We can attach to `aspnet_compiler.exe` using Debug -> Attach To Process -> Aspnet_compiler.exe



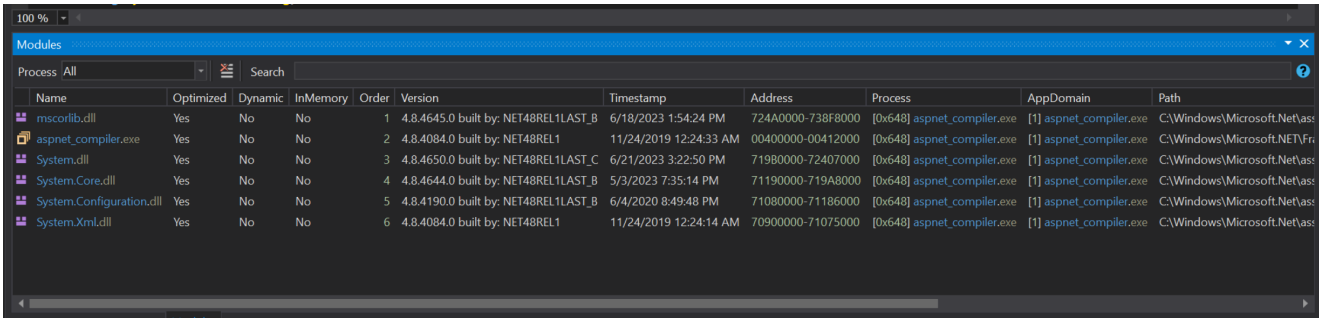
With the process attached, we now want to inspect any loaded modules.

We can do this by opening a "Modules" tab, using Debug -> Windows -> Modules.

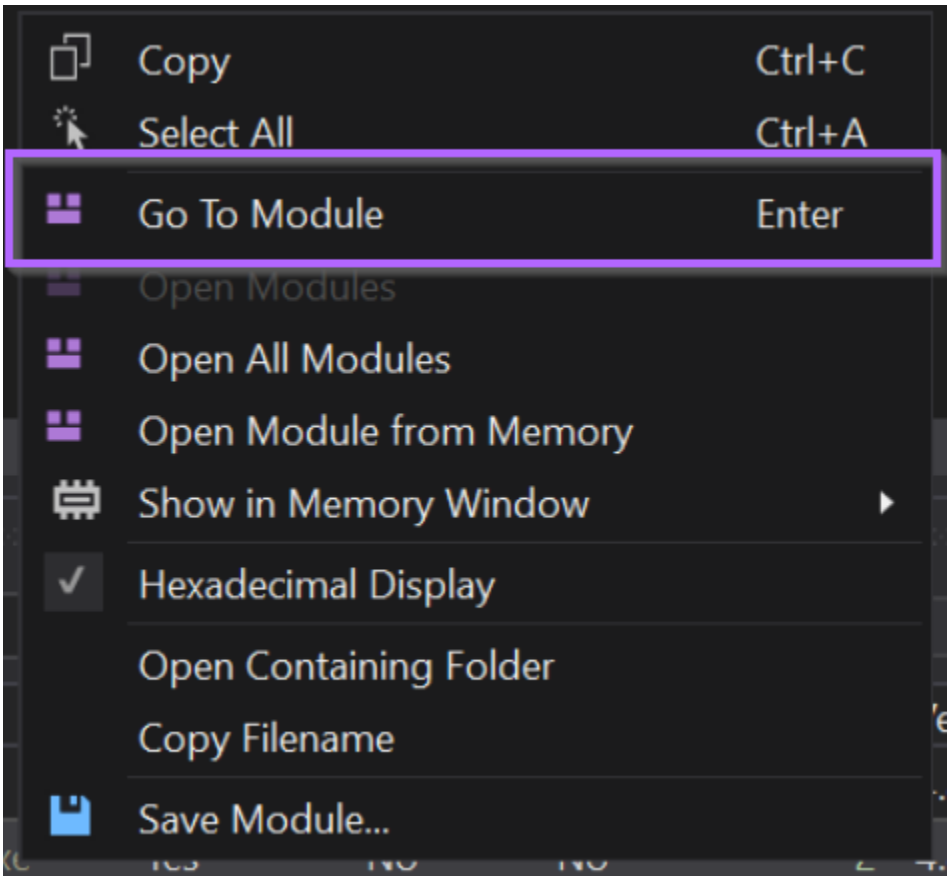


With the new Modules tab, we can list the same loaded modules that were observed with Process Hacker.

Interestingly, there is no `vik` module, but there is an `aspnet_compiler.exe` module that we know was associated with `vik`.

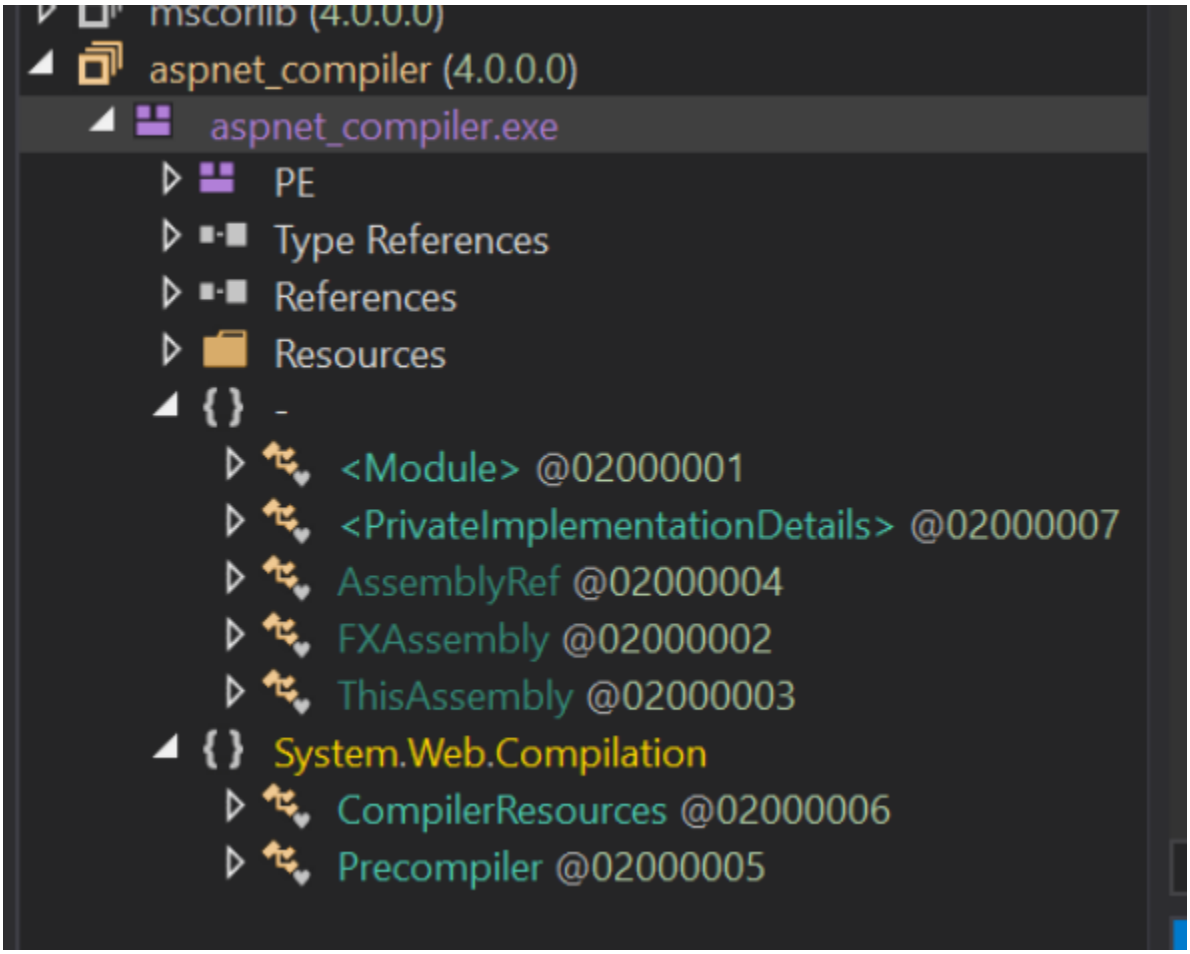


By clicking on `aspnet_c0mpiler.exe`, and selecting "Go To Module", we can view the module contents and corresponding decompiled code.

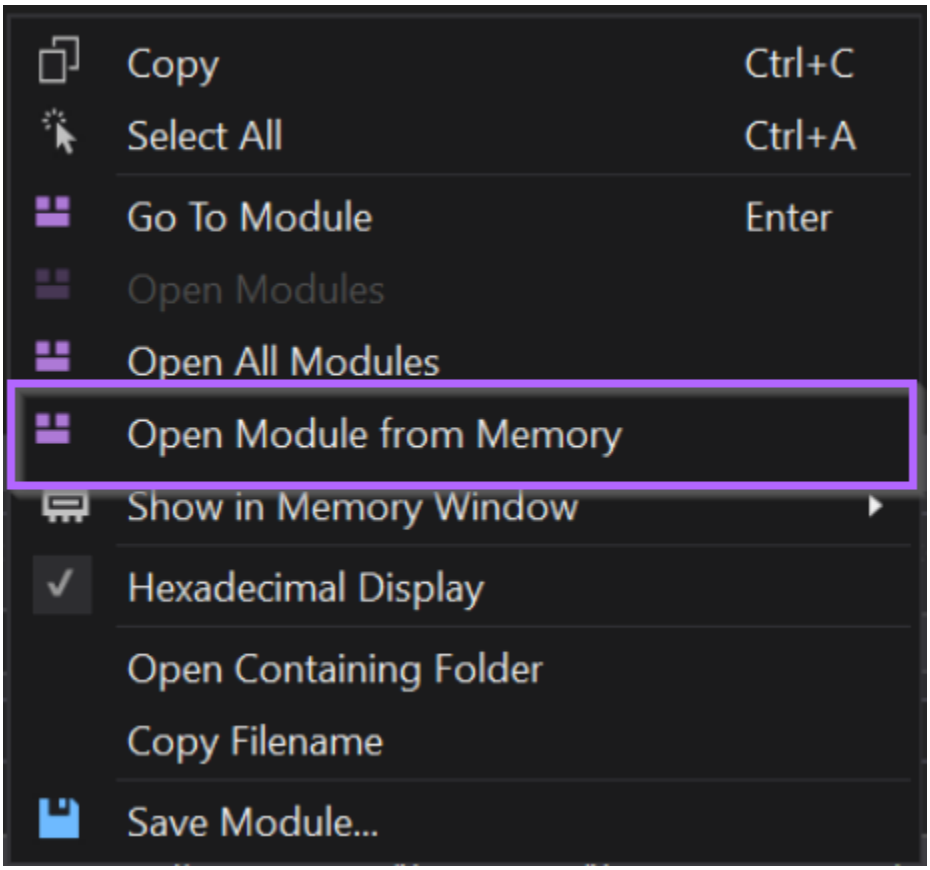


However, this will open the original `aspnet_compiler.exe` file from disk and not from within memory.

Hence, the "real" file will be loaded and we won't see anything suspicious.

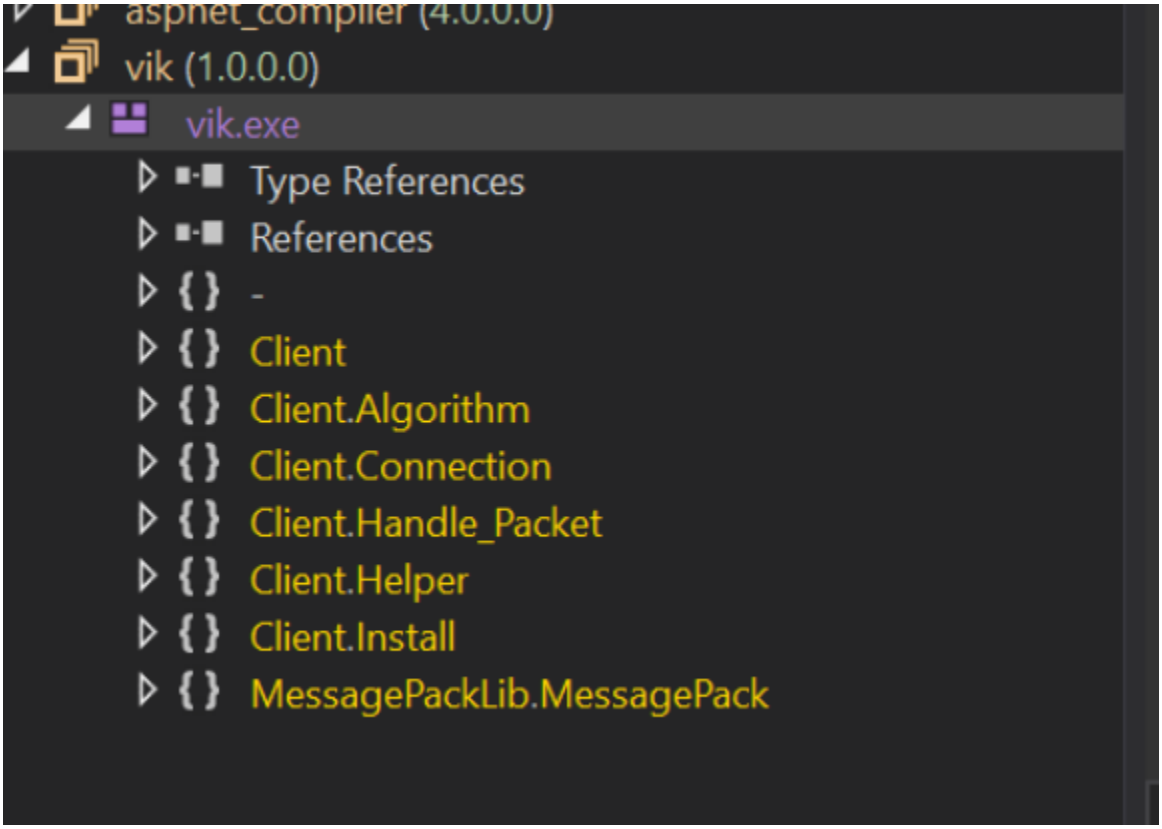


Instead, we can go back and re-open the file from memory.



With the file opened "from memory", we can obtain the real suspicious content. Which has likely been used to overwrite the original file in memory.

Here we can see the `vik` module loaded into DnSpy.



Jumping to the Entry Point of .NET Malware

To inspect the `vik` file more closely, we can righ-click on `vik` and select "Go To Entry Point".

This will take us to the beginning of the code. Which very closely resembles that of Asyncrat.

```
using System;
using System.Threading;
using Client.Connection;
using Client.Helper;
using Client.Install;

namespace Client
{
    // Token: 0x02000002 RID: 2
    public class Program
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002608 File Offset: 0x00002608
        public static void Main()
        {
            for (int i = 0; i < Convert.ToInt32(Settings.Delay); i++)
            {
                Thread.Sleep(1000);
            }
            if (!Settings.InitializeSettings())
            {
                Environment.Exit(0);
            }
            try
            {
                if (!MutexControl.CreateMutex())
                {
                    Environment.Exit(0);
                }
                if (Convert.ToBoolean(Settings.Anti))
                {
                    Anti_Analysis.RunAntiAnalysis();
                }
```

Clicking on the `Settings.InitializeSettings()` method, we can see where the configuration values are decrypted and loaded into the file.

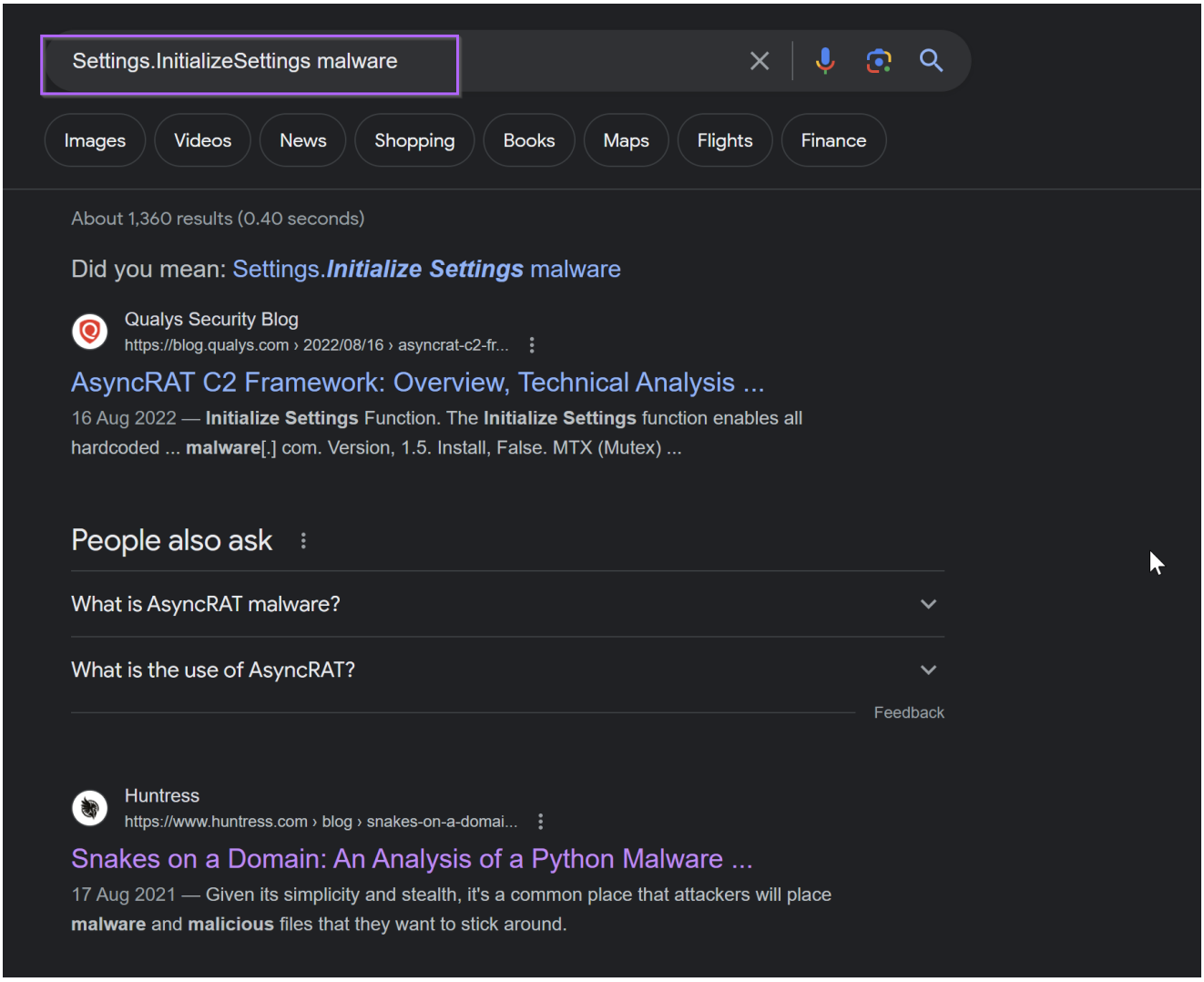
```
public static class Settings
{
    // Token: 0x06000003 RID: 3 RVA: 0x000026F8 File Offset: 0x000026F8
    public static bool InitializeSettings()
    {
        bool result;
        try
        {
            Settings.Key = Encoding.UTF8.GetString(Convert.FromBase64String(Settings.Key));
            Settings.aes256 = new Aes256(Settings.Key);
            Settings.Ports = Settings.aes256.Decrypt(Settings.Ports);
            Settings.Hosts = Settings.aes256.Decrypt(Settings.Hosts);
            Settings.Version = Settings.aes256.Decrypt(Settings.Version);
            Settings.Install = Settings.aes256.Decrypt(Settings.Install);
            Settings.MTX = Settings.aes256.Decrypt(Settings.MTX);
            Settings.Pastebin = Settings.aes256.Decrypt(Settings.Pastebin);
            Settings.Anti = Settings.aes256.Decrypt(Settings.Anti);
            Settings.BDOS = Settings.aes256.Decrypt(Settings.BDOS);
            Settings.Group = Settings.aes256.Decrypt(Settings.Group);
            Settings.Hwid = HwidGen.HWID();
            Settings.Serversignature = Settings.aes256.Decrypt(Settings.Serversignature);
            Settings.ServerCertificate = new X509Certificate2(Convert.FromBase64String(Settings.aes256.Decrypt(Settings.Certificate)));
            result = Settings.VerifyHash();
        }
        catch
        {
            result = false;
        }
        return result;
    }
```

Identifying the Malware With Google

If you haven't seen Asyncrat before, you could instead take some of the values in the "unpacked" sample and google them.

If the malware is known and there are existing reports, you will likely encounter reports that will suggest which family the malware belongs to.

You may have to experiment with which values to google, some return better results than others. Below we can see Asyncrat comes up straight away when googling `Settings.InitializeSettings Malware`

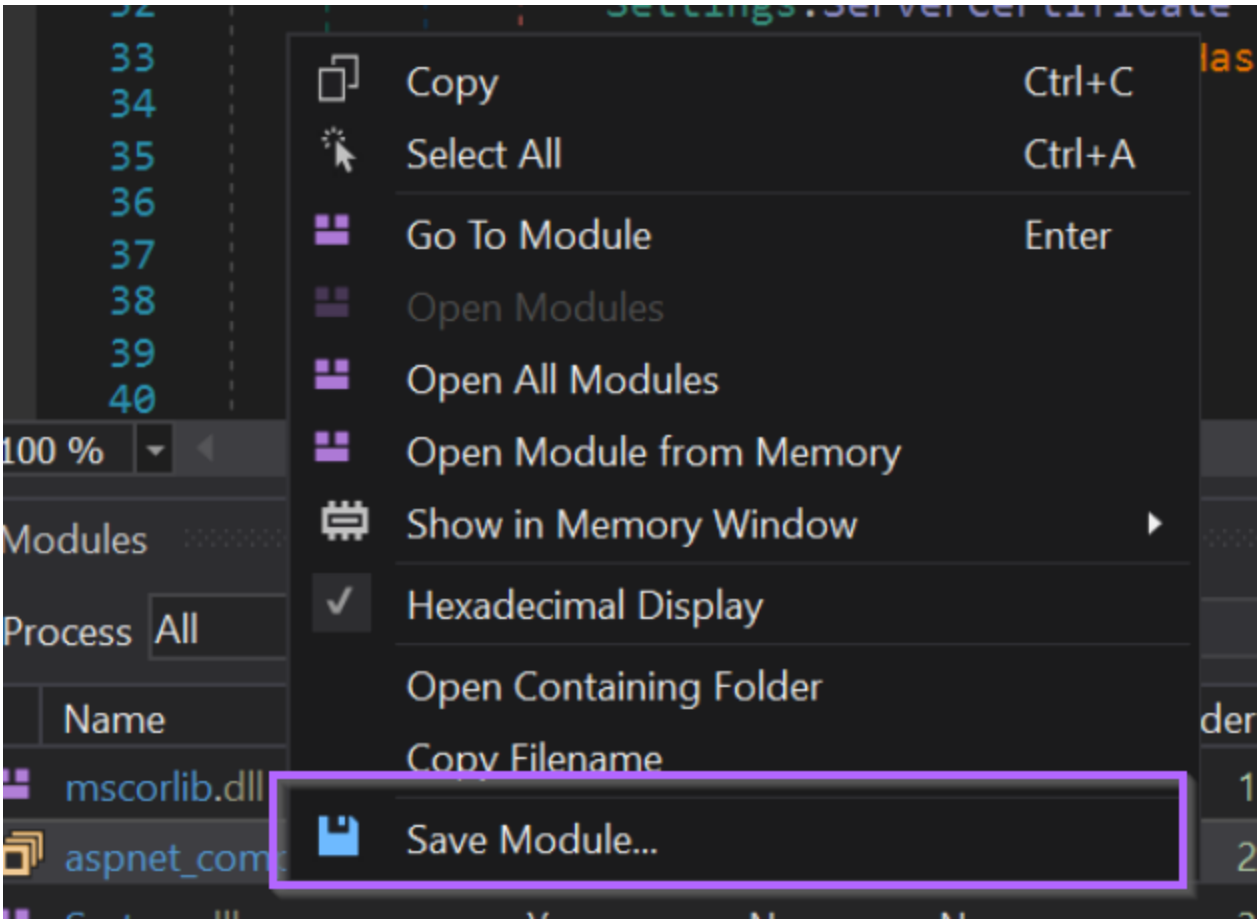


Verifying With a Sandbox

With an unpacked module now obtained, you can use DnSpy to save the file for additional analysis.

From here, you can submit the unpacked file to a sandbox or scan it against a set of Yara rules. This is useful if the strings/functions within the file are obfuscated or you aren't able to obtain a good result from google.

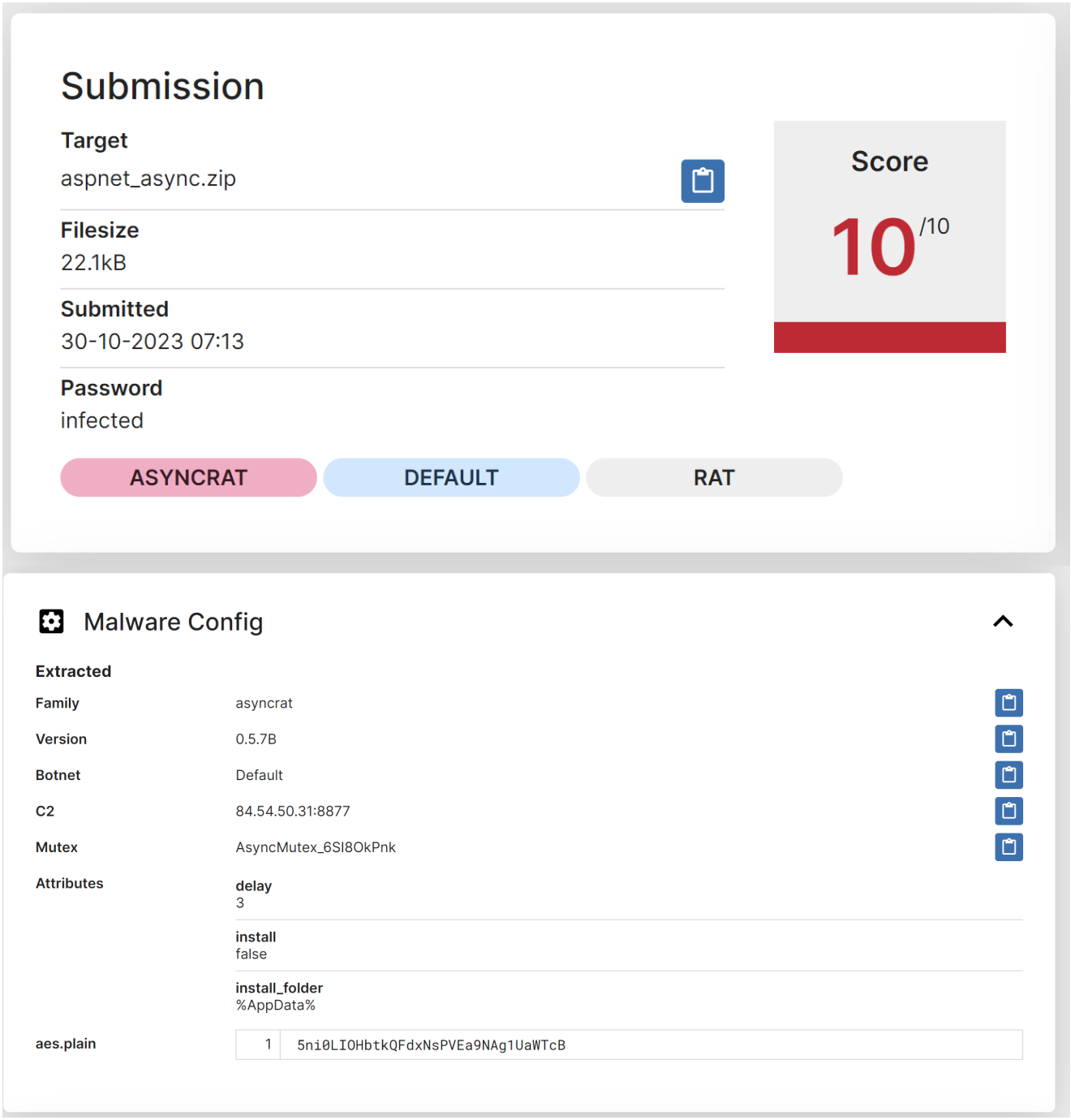
> This will save the file from memory, so you don't have to worry about saving the "wrong" file



Submitting the File To Hatching Triage

After saving, you can submit the file to an online sandbox like Hatching Triage.

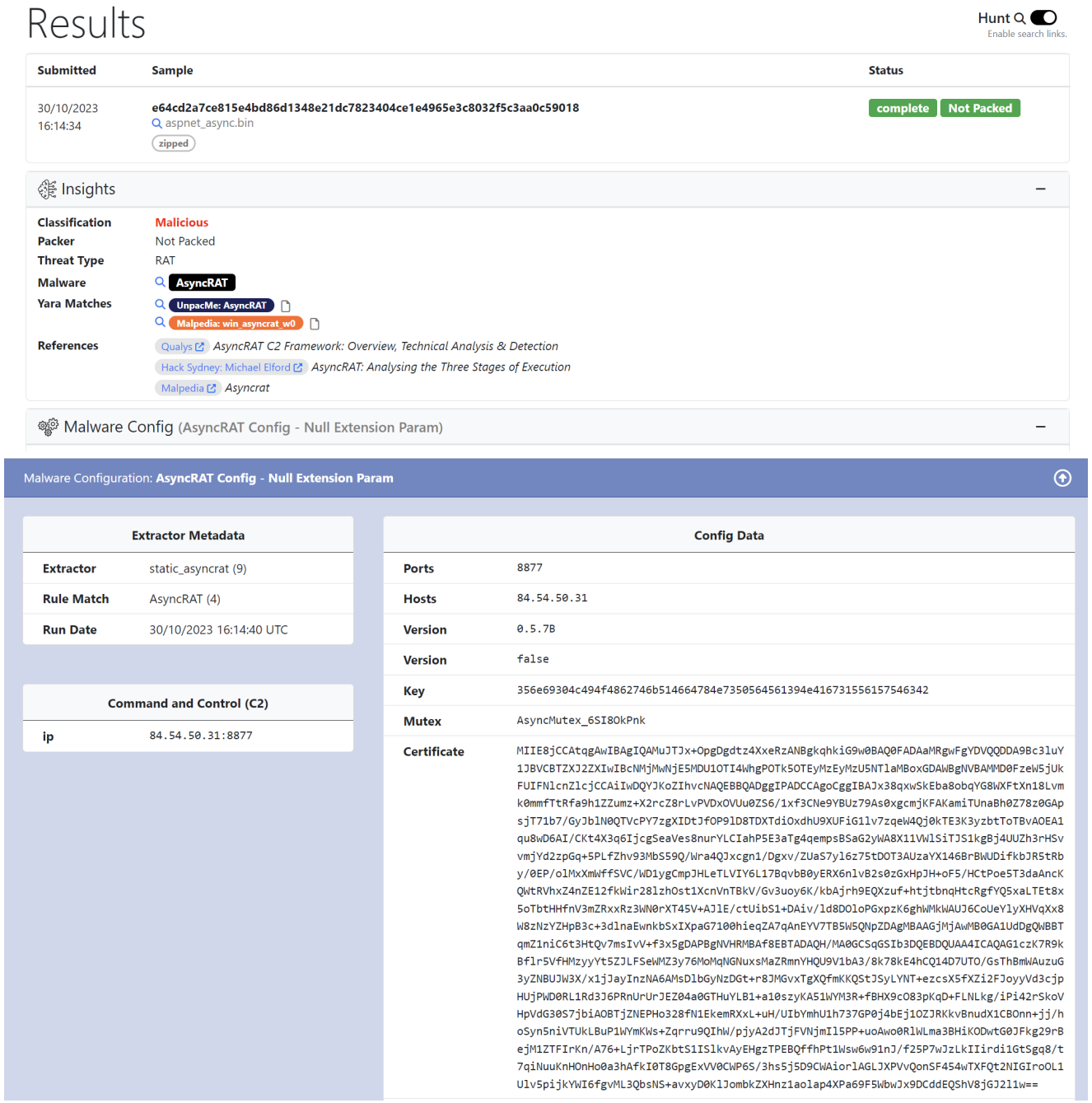
Hatching Triage is correctly able to identify the file as Asyncrat and extract the associated configuration values.



Submitting the File to Unpacme

Another option which is effective and significantly cheaper for researchers, is Unpacme.

Unpacme is correctly able to identify the file as Asyncrat and extract all configuration values.



Conclusion

In this post, we performed some basic analysis of an Asyncrat loader, and utilised Process Hacker to identify an unpacked payload. We then used Dnspy to obtain the unpacked malware and identify it as Asyncrat.