UNIVERSITY OF AMSTERDAM

Detecting Fileless Malicious Behaviour of .NET C2 Agents using ETW

Alexander Bode abode@os3.nl
Niels Warnars nwarnars@os3.nl

February 9, 2020

***Abstract*—Attackers are interested in bypassing malware detection. One of the techniques they rely on is by ensuring that malicious code only reside in memory and not on disk. This can amongst others things be achieved using .NET's *Assembly.Load* function, which can load .NET executables into memory without touching the disk. Multiple command-and-control frameworks implement the technique for its usefulness.**

Our research looked into whether Event Tracing for Windows (ETW) could be used to detect this type of executable loading that uses the .NET software framework. An analysis and comparison of ETW logs generated in our experiments showed that it is possible to identify assembly loading from memory using ETW. The proposed detection method relied on the *ModuleLoad* event provided by the .NET runtime ETW provider and a specific set of expected values. In the performed test cases, the detection signature could correctly identify in-memory .NET executable loading operations. The aim of this study is to provide defenders with valuable insight into detecting malicious fileless behaviour of .NET C2 agents.

I. INTRODUCTION

One of an attacker's goal may be to stay undetected during a compromise. To do so, the adversary may make use of tools and functionality provided by the operating system and script-based malware relying on, e.g., PowerShell [1]. Increased awareness and detection capabilities of defenders result in attackers shifting away from PowerShell to, amongst other things, malicious .NET applications [2] [3]. One of the advantages .NET offers is that code can be dynamically loaded and executed in memory, thus allowing malicious code to be executed from memory by a lightweight agent instead of it residing in an executable on disk [4]. Recent attention to Event Tracing for Windows (ETW) has led to the idea of using event tracing to detect malicious behaviour of applications. ETW includes mechanisms to log and trace events issued by user-mode applications and kernel-mode drivers. Relatively little research has addressed the question of how ETW could be used for the detection of malicious software. However, challenges remain due to the complexity and volume of generated data. This study aims to address the challenges and to provide a generic solution for detecting malicious .NET software that is loaded into memory by popular command and control (C2) frameworks.

RESEARCH QUESTIONS

The focus of our research is the detection of .NET agents used by popular C2 frameworks, also known as command and control frameworks, using Event Tracing for Windows. The main research question is therefore defined as:

How can ETW be leveraged to detect fileless malicious behaviour of .NET agents used by popular C2 frameworks?

This results in the following sub-questions:

- What language-specific features can be used by .NET C2 agents for fileless attacks?
- Which event types are relevant for detecting malicious .NET behaviour?

A. Structure

The remainder of this paper has the following structure; In section II we look at highlights of research performed by others that relates to ours. In section III we give a high-level overview of ETW and the C2 frameworks that are used during the experiments. In section IV we define our approach to determine how ETW can be used to detect malicious .NET C2 agents. In section V we present the findings of our evaluation. In section VI we give interpretations, discuss the implications and highlight the limitations of the study. Using the results of the experiments, we will draw conclusions and present those in section VII. The last section, VIII, contains suggestions for future work.

II. RELATED WORK

Earlier work on ETW arises primarily from industry research and is presented in blog posts.

A. Detecting .NET code injection

F-Secure has for example investigated methods for detecting .NET code injection using ETW, but only showed which related events exist. Their Python-based proof of concept relied for its detection on, among other things, the names of high-risk methods and namespaces, while using the inefficient PyWinTrace library. They also state that their PoC is only intended as a research and exploration tool and is

inefficient at processing data [4] [5]. More generic attempts of detecting malicious behaviour have also been made. Lelonek et al. of CyberPoint, e.g., showcased an ETW-based PoC for detecting ransomware by monitoring read and write operations performed by suspicious applications [6].

B. Bypassing ETW

First impressions of ETW made us hypothesise that Event Tracing for Windows can give an in-depth picture of the behaviour of .NET agents. ETW bypasses exist though; Tsukerman demonstrated that an ETW sensor for detecting Asynchronous Procedure Calls can be circumvented [7]. Palentir documented that ETW logging can be subverted if malware or a malicious user has administrator or SYSTEM-level privileges [8]. Bypasses are also used in the wild. Kaspersky documented that the Slingshot APT group, amongst others, avoids detection by renaming ETW log files [9].

C. C2 Frameworks

The C2 Matrix Team actively investigate popular C2 frameworks and share details that are useful for adversary emulation plans [10]. This includes information, such as programming languages used for the C2 servers and agents, the type of user interface, API presence and the support of common capabilities for each C2 framework.

III. Design

In this section we will give an overview of ETW and the C2 frameworks that will be investigated. We will take a look at functionalities of ETW that can be leveraged for detecting malicious software. For each C2 framework, relevant characteristics of the C2 framework will be briefly mentioned, followed by the specification of hardware and software used during the experiments.

A. Event Tracing for Windows

Event Tracing for Windows enables logging kernel level or user-mode application data. It can, amongst other things, be used for debugging an application or detecting performance issues. Event Tracing for Windows has been included in the Windows kernel since Windows 2000 [11]. Kernel events that ETW can log include executed system calls, Windows Registry access, memory allocation and more [12] [13]. The type of log traces produced by individual user-mode ETW providers differs per provider. The trace data can be consumed in real-time or by writing it to a file first. The output of the sessions can be highly detailed and large in volume. The verbosity of the trace data gives potential for leveraging the logs to detect malicious behaviour of software.

ETW consists of three type of components, including:

1) Controllers
2) Consumers
3) Providers

Controllers are utilities that are used to start or stop logging sessions and enable or disable providers [14]. These can be software, such as applications, libraries or services, either custom or developed by Microsoft. ETW uses Win32 API's to control the sessions and associate sessions with providers [15].

Consumers are tools that are used to view and parse trace data that are generated from trace sessions, either from stored log files or in real-time [14]. These can be software, such as applications or libraries, either custom or developed by Microsoft. The consumer must support the format of the event data to be able to consume the events.

Providers are applications that contain instrumentation [14], i.e. the conceptual components that provide the events [15]. Event tracing sources are separated under providers, e.g., Microsoft-Windows-VolumeControl for volume control and Microsoft-Windows-USB-UCX for USB devices. Events can further be filtered per category using keywords, which are 64-bit bitmask values used to group similar events. The events do not have to come from a single source, such as a single dynamic-link library or executable [15].

B. C2 Frameworks

All tested C2 frameworks can generate agents written in C# that use the .NET software framework. The .NET framework includes a large class library, supports multiple languages including C# and is developed by Microsoft [16].

More importantly, the studied C2 frameworks can load .NET assemblies from memory. A .NET assembly is a .NET executable or library that can exist standalone or can be used by other assemblies [17]. Functionality to load assemblies into memory is provided by methods from the Assembly class of the "System.Reflection" namespace [18]. The investigated C2 frameworks load additional assemblies onto infected machines by pushing an assembly from the C2 server to the agent that subsequently executes the assembly from memory. Various proof of concept implementations that can load assemblies into memory are included in appendix section IX-A.

We investigated the following C2 frameworks:

- Covenant v0.4
- PoshC2 v5.2
- Faction C2 v20.10.2019
- SilentTrinity v0.46

Covenant is a C2 framework that focuses on the usage of .NET on Windows systems. The server is written in C# and can generate .NET Core 2.1, .NET 3.5 and .NET 4.0 agents. Covenant includes a set of .NET assemblies which can be executed by issuing a new task from within the web interface [19].

PoshC2 is written in Python and can generate agents written in PowerShell, C# or Python. It allows users to extend its functionality by following a modular format. PoshC2 payloads are frequently updated to bypass anti-virus products [20].

Faction is another C2 framework that focuses on the usage of .NET on Windows systems. The server is written using the .NET framework and can generate .NET 3.5 and .NET 4.5

agents. However, Faction supports custom agents of any programming language. The framework supports using language-specific modules for interacting with its agents [21].

SilentTrinity is a C2 framework that is written in Python and can generate C# agents that support using the open-source IronPython implementation, which allows loading .NET assemblies into memory [22]. The agent requires that .NET 4.5 or a newer version is installed on the system so that the ZipArchive library is included and the IronPython DLLs can be compiled against .NET 4.0 [23].

IV. Methodology

To determine how ETW can be used to detect malicious .NET agents, a test environment was setup containing:

- *Kali Linux host running a C2 server for controlling the agents*
- *Windows 10 host running benign or malicious software and tooling for controlling and logging event traces*

We used the following hardware and operating system software to conduct our experiments on:

- Intel Core i7 Quad-Core CPU @ 3.60GHz
- 16GB RAM
- Kali GNU/Linux Rolling 2019.2 VM
- Microsoft Windows 10 Pro Build 18362 VM

The steps taken for gathering and analysing data can be outlined as follows:

1) Determine relevant ETW providers and events
2) Generate event trace logs of the execution of:
   - Malicious .NET C2 agents
   - Assembly loading PoCs
   - Benign .NET software
3) Compare the generated logs side-by-side
4) Create detection method and automate detection
5) Identify limitations of detection methods

The first step to our approach was to determine the relevant ETW providers, ETW keywords and event names. Previous research by F-Secure served as a starting point. F-Secure documented two providers that are relevant for logging event traces of .NET software [4]:

1) *Microsoft-Windows-DotNETRuntime*
2) *Microsoft-Windows-DotNETRuntimeRundown*

F-Secure's Python-based proof of concept consumed .NET-Runtime Loader events [24]. Nevertheless, we wanted to independently determine the relevant .NET-Runtime events. Therefore, for one of the assembly loading proof of concepts, a full log trace was generated without filters. Irrelevant and verbose event types were subsequently manually removed.

The next step included generating the event tracing logs of the different test cases. SilkETW v0.8 was used to control and consume the ETW traces during the experiments [25]. SilkETW is a C# wrapper for ETW that abstracts complexities of ETW. It provides an interface to start or stop trace sessions and outputs logged data to a serialized JSON file [26].

The first test case consisted of tracing the execution of four agents that were generated by the C2 servers of the different frameworks. Depending on the framework, agents built for the following .NET versions were tested: v3.5, v4.0, and v4.5. During the experiments, the agents were instructed from the C2 server to execute at least one command. This would ensure that a second stage .NET payload containing additional functionality or a .NET assembly containing code to execute the issued command was loaded from memory and executed by the agent.

The second test case was intended to investigate the impact of different assembly loading methods and .NET versions on the generated events. The .NET versions are taken into consideration because the C2 frameworks can generate agents for different .NET versions. For the study, traces were logged of the execution of custom .NET executables that load a .NET assembly from memory or disk in one of three ways:

- Loading an assembly from memory using *Assembly.Load*
- Loading an assembly from disk using *Assembly.LoadFile*
- Creating an assembly dynamically and subsequently loading the payload into the dynamic assembly using *Module.LoadModule*, based on an implementation from Graeber, 2018 [27].

Proof-of-concept code of these implementations is included in appendix section IX-A.

The third test case was designed to compare the event trace logs of malicious software with the trace logs of benign software. In this experiment, ETW logs were generated of three benign .NET applications. These programs included:

- Paint.NET 4.2.8
- KeePass 2.44
- Visual Studio 2019.16.4.2

The logs were manually inspected and compared side by side to see if there are any anomalies. This was done primarily to detect the behaviour of the .NET agents that invoke assemblies into memory using the methods of the .NET reflection namespace [18]. Based on the anomalies, a method was devised to detect in-memory loaded assemblies.

Finally, the detection method was automated using a custom script written in Python, after which we looked at ways to bypass the constructed detection method. The script works on logs generated by SilkETW and makes it possible to more easily test for detection.

V. Results

The results section covers the outcomes of the performed experiments to understand assembly loading by .NET applications and describes how a detection method was developed for in-memory assembly loading.

A. ETW Providers and Filters

F-Secure documented that two providers generate events for .NET applications [4]. Initial testing showed that the *Microsoft-Windows-DotNETRuntime* was the runtime that should be used. To determine the relevant events, a full ETW log trace without filters was created of the *Assembly.Load* proof of concept.

The log trace contained 99937 generated events, divided over 26 event types. This number was reduced by manually removing irrelevant event types and included, e.g., method loading, unloading and garbage collection events.

In the end, we determined that only *Loader* events had to be logged, which contain information related to the loading of application domains, assemblies and modules [28]. This matches with the events logged by F-Secure's detection proof of concept [24].

Our tests show that assembly loading in a .NET 4.x application results in the logging of up to three events, namely:

- *AssemblyLoad*
- *ModuleLoad*
- *DomainModuleLoad*

For .NET 3.5 applications, only *ModuleLoad* events are logged.

B. Event analysis

The test runs revealed that two types of .NET-runtime events could be useful for detecting in-memory assembly loading, namely *AssemblyLoad* and *ModuleLoad* events.

1) AssemblyLoad event: The first observation we made, is that for many legitimate .NET modules a Public Key Token value is set. The assemblies loaded from memory by the C2 agents were unsigned and did not have a Public Key Token value set, as is shown in table II. The PublicKeyToken value is only set when a .NET assembly is strong named [29]. Strong naming is optional and therefore, the in-memory loaded assemblies together with our assembly loading PoCs and also some benign software applications are not strong-named and have a Public Key Token value of null.

The observation that the assemblies loaded from memory have no Public Key Token value, matches with the AssemblyLoad events generated by our assembly loading PoCs. A list of AssemblyLoad events generated by our assembly loading PoCs can be found in appendix IX-B.

Legitimate Module	Assembly name	PublicKeyToken
mscorlib.dll	"mscorlib"	b77a5c561934e089

TABLE I: *AssemblyLoad* event values when a legitimate module, in this case mscorlib.dll, is loaded

2) ModuleLoad event: A comparison between ModuleLoad events generated by legitimate modules (table III) and assemblies loaded from memory (table IV and V), shows that loading assemblies from memory results in ModuleLoad events containing a ModuleILPath value with merely the assembly name. The ModuleILPath field contains the path of the Microsoft intermediate language image of a module [30] [28]. Assemblies that are loaded directly into memory lack a full path value. Instead, only the name of the assembly is given as the ModuleILPath value. Therefore, a signature for this behaviour can filter on ETW events of the .NET runtime that have a ModuleILPath value where the backslashes are missing.

C2 Agent	Assembly name	PublicKeyToken
Covenant	"jhyfwkp2.hwm"	null
PoshC2 Sharp	"Core"	null
Faction	"stdlib"	null
SilentTrinity	"Stage"	null

TABLE II: Overview of *AssemblyLoad* event values when a second stage assembly (or equivalent) is loaded by an agent

Additionally, none of the ModuleLoad events resulting from assemblies loaded by the C2 agents contained a ModuleNativePath, as is shown in appendix IX-C. The ModuleNativePath and the "Native" module flag are set if a native image is present, meaning that an assembly has been pre-compiled into machine code. Using pre-compiled native images increases performance because the .NET runtime does not have to compile the module every time it is loaded [31].

The observation that in-memory loaded assemblies result in a ModuleILPath value containing only the assembly name and not a full path, matches with the ModuleLoad events generated by our assembly loading proof-of-concept implementations. The same holds for the absence of the ModuleNativePath value, which if present is the path of the native image [28]. A list of ModuleLoad events generated by our assembly loading PoCs can be found in appendix IX-B.

Legitimate Module	ModuleILPath	ModuleFlags
mscorlib.dll	"C:\\[...]\\mscorlib.dll"	"DomainNeutral| Manifest"

TABLE III: *ModuleLoad* event values when a legitimate module, in this case mscorlib.dll, is loaded

C2 Agent	ModuleILPath	ModuleFlags
Covenant	"jhyfwkp2.hwm"	"Manifest"
PoshC2 Sharp	"Core"	"Manifest"
Faction	"stdlib"	"Manifest"
SilentTrinity	"Stage"	"Dynamic"

TABLE IV: Overview of *ModuleLoad* event values when a second stage assembly (or equivalent) is loaded by a .NET 4.x agent

C2 Agent	ModuleILPath	ModuleFlags
Covenant	""	"0"
Faction	""	"0"

TABLE V: Overview of *ModuleLoad* event values when a second stage assembly (or equivalent) is loaded by a .NET 3.5 agent

C. Detection Method

Based on the ModuleLoad-events observed in the test cases, we can see that in-memory assembly loading results in certain values being logged. Based on the observed values, the following detection signature can be derived that can detect in-memory assembly loading performed by the C2 agents:

Field	Value
ModuleILPath	No absolute path (i.e. exclude slashes)
ModuleNativePath	Empty string
ModuleFlags (if present)	”0”, ”Dynamic” or ”Manifest”

TABLE VI: Signature to detect in-memory assembly loading by the C2 agents based on the *ModuleLoad* event

The signature has been implemented in a Python script that can consume a JSON log file generated by SikETW and is available in appendix IX-D.

D. Reducing False Positives

For completeness, the detection signature was tested against multiple benign .NET applications. It is important to know whether the developed detection method triggers on ModuleLoad events generated by legitimate software, therefore the signature was tested on ETW logs generated while opening the following .NET applications:

- Paint.NET 4.2.8
- KeePass 2.44
- Visual Studio 2019.16.4.2

In this limited testing, the signature did not flag on any events.

VI. Discussion

The overall purpose of this study is to determine whether ETW can be used to detect fileless malicious behaviour of C2 agents that make use of the .NET software framework.

A. Interpretations

An analysis of the investigated C2 frameworks shows that multiple frameworks dynamically load additional components into the .NET agent process. ETW logs generated while tracing the behaviour of the agents show that this technique leaves specific traces.

B. Implications

Based on a comparison of in-memory assembly loading and benign module loads, the characteristics of assemblies loaded from memory appear to be distinct enough to be useful in helping to identify malicious .NET behaviour. The proposed detection method can detect assembly loading performed by the studied agents and our proof of concepts.

C. Limitations

Despite the promising indicators, our research has some limitations:

- First of all, benign software may also use in-memory assembly loading. It is unknown how often this occurs, only limited false-positive testing was performed to study whether legitimate applications load assemblies into memory the same way malicious software does. Nevertheless, observing dynamically loaded modules is still an indicator of potentially malicious behaviour.
- Secondly, attackers with knowledge of implemented detection methods can bypass or complicate detection. The proposed ModuleLoad-signature identifies an assembly loaded from memory by checking whether the ModuleILPath field contains slashes or backslashes. Only the assembly name is logged for an assembly loaded from memory, not a full path. The assembly’s name is hardcoded in the .NET executable and can be replaced with a name that resembles a fake path. This way, whenever the assembly is loaded in memory a fake path is logged in the ModuleILPath field, thus creating the appearance of an absolute file path in logs. This method can bypass the proposed detection method. However, this bypass technique was not observed in any of the tested C2 frameworks.

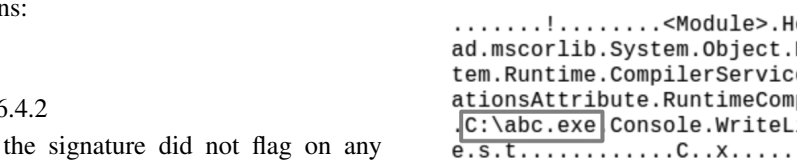

Fig. 1: .NET executable with patched assembly name

VII. Conclusion

Due to increased awareness and detection capabilities of defenders, attackers are shifting towards the use of .NET for its evasive capabilities. Our research aimed to identify methods to detect malicious fileless behaviour of .NET agents used by popular C2 frameworks using ETW. Based on a qualitative and experimental analysis, it can be concluded that multiple .NET agents acquire additional functionality by loading assemblies, that were sent by the C2 server, from memory. Loading assemblies into memory using .NET was detectable in the performed test runs. The test runs included a selection of C2 agents, custom software that loaded .NET assemblies into memory and a selection of benign software. Besides a detection method that can detect in-memory assembly loading, our research also identified the limitations of the proposed detection technique. Despite new efforts to detect malicious fileless malicious behaviour of .NET C2 agents, more research is necessary to implement detection with ETW into production environments.

VIII. Future work

Our research has shown that in-memory assembly loading is used by multiple .NET C2 frameworks and that the agents

exhibit specific properties that can be logged with ETW. Nevertheless, more work can be done on .NET malware detection.

- First, false-positive testing was only performed on a small number of benign .NET applications. More testing is required to determine false positive rates in a production environment.
- Secondly, the scope was limited to identifying unique characteristics that may be useful for detection. Investigating how detection using ETW can be practically and efficiently implemented on endpoints or in a security information and event management system was out of scope and is left as future work.
- Finally, our research focused on detecting a specific type of malicious behaviour. Future research can possibly result in other use cases of ETW for malware detection.

To summarize, more research into additional use cases of ETW and practical .NET malware detection using ETW is favourable, our research has provided some building blocks on which future work can build on.

REFERENCES

[1] Demystifying fileless threats. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/white-papers/restricted/wp-demystifying-fileless-threats.pdf

[2] N. Mittal. Amsi: How windows 10 plans to stop script-based attacks and how well it does it. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Mittal-AMSI-How-Windows-10-Plans-To-Stop-Script-Based-Attacks-And-How-Well-It-Does-It.pdf

[3] C. de Plaa. Red team tactics: Active directory recon using adsi and reflective dlls. [Online]. Available: https://outflank.nl/blog/2019/10/20/red-team-tactics-active-directory-recon-using-adsi-and-reflective-dlls/

[4] Detecting malicious use of .net – part 1. [Online]. Available: https://blog.f-secure.com/detecting-malicious-use-of-net-part-1/

[5] Detecting malicious use of .net – part 2. [Online]. Available: https://blog.f-secure.com/detecting-malicious-use-of-net-part-2/

[6] N. R. B. Lelonek. Make etw great again. - exploring some of the many uses of event tracing for windows (etw). [Online]. Available: https://ruxcon.org.au/assets/2016/slides/ETW_16_RUXCON_NJR_no_notes.pdf

[7] P. Tsukerman. Bypassing the microsoft-windows-threat-intelligence kernel apc injection sensor. [Online]. Available: https://medium.com/@philiptsukerman/bypassing-the-microsoft-windows-threat-intelligence-kernel-apc-injection-sensor-92266433e0b0

[8] Tampering with windows event tracing: Background, offense, and defense. [Online]. Available: https://medium.com/palantir/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63

[9] The slingshot apt. [Online]. Available: https://s3-eu-west-1.amazonaws.com/khub-media/wp-content/uploads/sites/43/2018/03/09133534/The-Slingshot-APT_report_ENG_final.pdf

[10] A. M. J. Orchilles, B. Bort. C2 matrix. [Online]. Available: https://www.thec2matrix.com/matrix

[11] Event tracing portal. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal

[12] Etw kernel logger events. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/evntrace/ns-evntrace-event_trace_properties

[13] perform etw kernel event tracer source code. [Online]. Available: https://github.com/microsoft/perfview/blob/master/src/TraceEvent/Parsers/KernelTraceEventParser.cs

[14] Event tracing. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing

[15] T. Soulami, *Inside Windows Debugging (Developer Reference)*. Microsoft Press, 2012. [Online]. Available: https://www.amazon.com/Inside-Windows-Debugging-Developer-Reference/dp/0735662789?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0735662789

[16] What is .net framework? [Online]. Available: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework

[17] Assemblies in .net. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/assembly/

[18] Reflection (c#). [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection

[19] Covenant .net c2 framework. [Online]. Available: https://github.com/cobbr/Covenant

[20] Poshc2. [Online]. Available: https://github.com/Nettitude/PoshC2

[21] Factionc2. [Online]. Available: https://www.factionc2.com/

[22] Ironpython documentation. [Online]. Available: https://ironpython.net/documentation/dotnet/

[23] Silenttrinity. [Online]. Available: https://github.com/byt3bl33d3r/SILENTTRINITY

[24] .net etw poc. [Online]. Available: https://gist.github.com/countercept/7765ba05ad00255bcf6a4a26d7647f6e

[25] Silketw. [Online]. Available: https://github.com/fireeye/SilkETW

[26] Silketw: Because free telemetry is ... free! [Online]. Available: https://www.fireeye.com/blog/threat-research/2019/03/silketw-because-free-telemetry-is-free.html

[27] M. Graeber. Assembly.loadmodule powershell implementation. [Online]. Available: https://gist.github.com/mattifestation/8958b4c18d8bca9e221b29252cfee26b

[28] Loader etw events. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/performance/loader-etw-events

[29] Assembly names. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/assembly/names

[30] What is "managed code"? [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/managed-code

[31] Ngen.exe (native image generator). [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/tools/ngen-exe-native-image-generator

IX. Appendix

A. Assembly.Load PoCs

1) Assembly.Load:

```
var payload = Convert.FromBase64String("[PAYLOAD REMOVED]");

// Load assembly
var asm = Assembly.Load(payload);

// Execute payload
asm.EntryPoint.Invoke(0, new object[] { new string[] { } });
```

2) Assembly.LoadFile:

```
var path = "[PATH REMOVED]";

// Load assembly
var asm = Assembly.LoadFile(path);

// Execute payload
asm.EntryPoint.Invoke(0, new object[] { new string[] { } });
```

3) Assembly.LoadModule:

```
// Derived from:
↪  https://gist.github.com/mattifestation/8958b4c18d8bca9e221b29252cfee26b

var payload = Convert.FromBase64String("[PAYLOAD REMOVED]");

// Define assembly name
var an = new AssemblyName("asm");

// Create builder for dynamic assembly
var ab = AssemblyBuilder.DefineDynamicAssembly(an, AssemblyBuilderAccess.Run);

// Define dummy module necessary for assembly creation
var mb = ab.DefineDynamicModule("dummy");

// Define payload module, the payload module is loaded at a later stage
ab.DefineDynamicModule("payload");

// Define dummy class necessary for assembly creation
var tb = mb.DefineType("dummy");

// Create type
var type = tb.CreateType();

// Load payload module into assembly
var module = type.Assembly.LoadModule("payload", payload);

// Execute payload
module.GetTypes()[0].GetMethods()[0]Invoke(0, new object[] { new string[] { } });
```

B. Event comparison of assembly loading PoCs

1) .NET 4.0: AssemblyLoad event:

Action triggering AssemblyLoad event	Assembly name	AssemblyFlags	PublicKeyToken
Loading of mscorlib library	”mscorlib”	”DomainNeutral|Native”	b77a5c561934e089
Loading of main executable file	”[name]”	”0”	null
Assembly.Load	”[name]”	”0”	null
Assembly.LoadFile	”[name]”	”0”	null
Assembly.LoadModule	”asm”	”Dynamic”	null

Note: *AssemblyLoad* events are not generated for .NET 3.5 applications.

2) .NET 4.0: ModuleLoad event:

Action triggering ModuleLoad event	ModuleILPath	ModuleNativePath	ModuleFlags
Loading of mscorlib library	”[path]\\mscorlib.dll”	”[path]\\mscorlib.ni.dll”	”DomainNeutral|Native |Manifest|0x10”
Loading of main executable file	”[path]\\[name].exe”	””	”Manifest”
Assembly.Load	”[name]”	””	”Manifest”
Assembly.LoadFile	”[path]\\[name].exe”	””	”Manifest”
Assembly.LoadModule (1)	”asm”	””	”Dynamic|Manifest”
Assembly.LoadModule (2)	”payload”	””	”Dynamic”

3) .NET 3.5: ModuleLoad event:

Action triggering ModuleLoad event	ModuleILPath	ModuleNativePath	ModuleFlags
Loading of mscorlib library	”[path]\\mscorlib.dll”	”[path]\\mscorlib.ni.dll”	”3” / ”DomainNeutral|Native”
Loading of main executable file	”[path]\\[name].exe”	””	”0”
Assembly.Load	””	””	”0”
Assembly.LoadFile	”[path]\\[name].exe”	””	”0”
Assembly.LoadModule (1)	”asm”	””	”4” / ”Dynamic”
Assembly.LoadModule (2)	”dummy”	””	”4” / ”Dynamic”
Assembly.LoadModule (3)	”payload”	””	”4” / ”Dynamic”

C. Event comparison of .NET agents

1) AssemblyLoad event:

Agent	.NET Version	Assembly name	AssemblyFlags	Public KeyToken
Covenant	4.0	”jhyfwkp2.hwm”	”0”	null
PoshC2 Sharp	4.0	”Core”	”0”	null
Faction	4.5.2	”stdlib”	”0”	null
SilentTrinity	4.0	”Stage”	”Dynamic”	null

Note: The *AssemblyLoad* events listed above were generated when a second stage assembly (or equivalent) was loaded into the agent process. *AssemblyLoad* events are not generated for .NET 3.5 applications.

2) ModuleLoad event:

Agent	.NET Version	ModuleILPath	Module NativePath	ModuleFlags
Covenant	3.5	""	""	"0"
Covenant	4.0	"jhyfwkp2.hwm"	""	"Manifest"
PoshC2 Sharp	4.0	"Core"	""	"Manifest"
Faction	3.5	""	""	"0"
Faction	4.5.2	"stdlib"	""	"Manifest"
SilentTrinity	4.0	"Stage"	""	"Dynamic|Manifest"
SilentTrinity	4.0	"Stage.exe"	""	"Dynamic"

Note: The *ModuleLoad* events listed above were generated when a second stage assembly (or equivalent) was loaded into the agent process.

D. moduleload.py - ModuleLoad detection script

```
#!/usr/bin/env python3
#
# Script to detect .NET assemblies loaded from memory.
# The expected input file is a JSON log file created by SilkETW.
import json
import sys


def check_moduleload(j):
    if (
        "EventName" not in j or
        "XmlEventData" not in j or
        "ModuleILPath" not in j["XmlEventData"] or
        "ModuleNativePath" not in j["XmlEventData"]
    ): return False

    event_name = j["EventName"]
    xml_event_data = j["XmlEventData"]
    module_il_path = xml_event_data["ModuleILPath"]
    module_native_path = xml_event_data["ModuleNativePath"]

    if (
        not event_name.endswith("/ModuleLoad") or
        "\\" in module_il_path or
        "/" in module_il_path or
        module_native_path != ""
    ): return False

    if "ModuleFlags" in xml_event_data:
        module_flags = xml_event_data["ModuleFlags"]
        if module_flags not in ["0", "Dynamic", "Manifest"]: return False

    return True

def main():
    if len(sys.argv) != 2:
        sys.exit("Usage: moduleload.py <SilkETW JSON file>")

    lines = open(sys.argv[1]).readlines()

    for line in lines:
        try:
            j = json.loads(line)
            if check_moduleload(j):
                print(j)
        except ValueError:
            pass

if __name__ == '__main__':
    main()
```