BOHOPS

A blog about cybersecurity research, education, and news

WRITTEN BY BOHOPS

MARCH 16, 2021

INVESTIGATING .NET CLR USAGE LOG TAMPERING TECHNIQUES FOR EDR EVASION

QUICK LINKS

- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence (Part 2)
- Abusing .NET Core CLR Diagnostic Features (+ CVE-2023-33127)
- Abusing the COM Registry Structure (Part 2): Hijacking & Loading Techniques
- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence
- Abusing the COM Registry Structure: CLSID, LocalServer32, & InprocServer32
- Analyzing and Detecting a VMTools Persistence Technique
- Investigating .NET CLR Usage Log Tampering Techniques For EDR Evasion (Part 2)
- WS-Management COM: Another Approach for WinRM Lateral Movement
- Executing Commands and Bypassing AppLocker with PowerShell Diagnostic Scripts

INTRODUCTION

In recent years, there have been numerous published techniques for evading endpoint security solutions and sources such as A/V, EDR and logging facilities. The methods deployed to achieve the desired result usually differ in sophistication and implementation, however, effectiveness is usually the end goal (of course, with thoughtful consideration of potential tradeoffs). Defenders can leverage the native facilities of the operating system and support frameworks to build quality detections. One way to detect potentially interesting .NET behavior is by monitoring the Common Language Runtime (CLR) Usage Logs (“UsageLogs”) for .NET execution events.

In this quick post, we will identify how defenders are (likely) leveraging .NET Usage Logs for detection and forensic response, investigate ways to circumvent detection log monitoring, and discuss potential monitoring opportunities for catching Usage Log tampering behavior.

USING .NET CLR USAGE LOGS TO DETECT SUSPICIOUS ACTIVITY

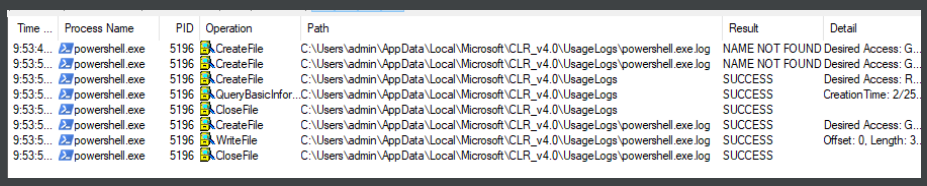
- Vshadow: Abusing the Volume Shadow Service for Evasion, Persistence, and Active Directory Database Extraction

When .NET applications are executed or when assemblies are *injected* into another process memory space (by the Red Team), the .NET Runtime is loaded to facilitate execution of the assembly code and to handle various and sundry .NET management tasks. One task, as initiated by the CLR (crl.dll), is to create a *Usage Log* file named after the executing process once the assembly is finished executing for the first time in the (user) session context. This log file contains .NET assembly module data, and its purpose serves an information file for .NET native image autogeneration (auto-NGEN).

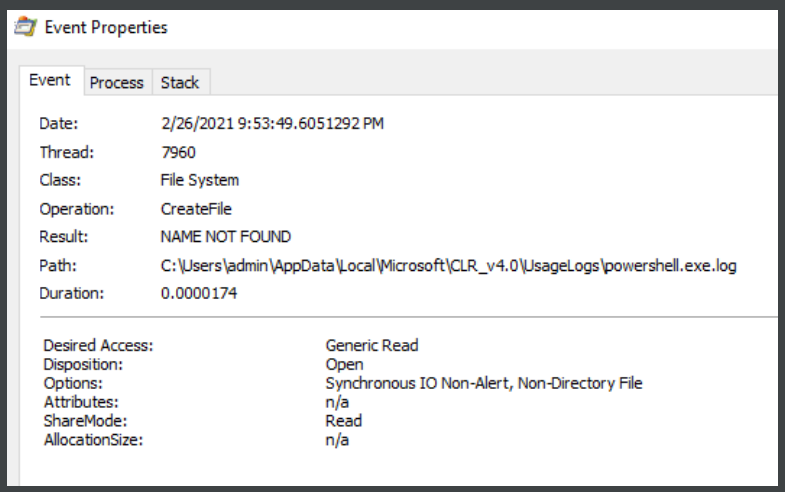
Prior to process exit, the CLR typically writes to one of these file paths (although there could be others):

- <SystemDrive>:\Users\<user>\AppData\Local\Microsoft\CLR_<version>_(arch)\UsageLogs
- <SystemDrive>:\Windows\<System32|SysWOW6$=4>\config\systemprofile\AppData\Local\Microsoft\CLR_<version>_(arch

As an example, we can see that the *powershell.exe.log* Usage Log is created for the first time just prior to 'gracefully' terminating the powershell.exe process:



Time ...	Process Name	PID	Operation	Path	Result	Detail
9:53:4...	powershell.exe	5196	CreateFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs\powershell.exe.log	NAME NOT FOUND	Desired Access: G...
9:53:5...	powershell.exe	5196	CreateFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs\powershell.exe.log	NAME NOT FOUND	Desired Access: G...
9:53:5...	powershell.exe	5196	CreateFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs	SUCCESS	Desired Access: R...
9:53:5...	powershell.exe	5196	QueryBasicInfor...	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs	SUCCESS	CreationTime: 2/25...
9:53:5...	powershell.exe	5196	CloseFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs	SUCCESS	
9:53:5...	powershell.exe	5196	CreateFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs\powershell.exe.log	SUCCESS	Desired Access: G...
9:53:5...	powershell.exe	5196	WriteFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs\powershell.exe.log	SUCCESS	Offset: 0, Length: 3...
9:53:5...	powershell.exe	5196	CloseFile	C:\Users\admin\AppData\Local\Microsoft\CLR_v4.0\UsageLogs\powershell.exe.log	SUCCESS	



From a DFIR and threat hunting perspective, analyzing the Usage Logs is very opportunistic for investigatory purposes as outlined in this excellent blog post by the MENASEC Applied Research Team. From an endpoint monitoring standpoint, Endpoint Detection & Response Solutions ('EDRs') are likely monitoring Usage Log file creation events to identify suspicious or unlikely processes that have loaded the .NET CLR. As an example, Olaf Hartong (@olafhartong) maintains the incredible Sysmon-Modular project and has graciously provided a rule config that monitors *Usage Log* activity for .NET 2.0 activity and risky LOLBINs.

Red Teamers can certainly expect that many commercial vendors are monitoring Usage Logs in a similar fashion (e.g. to catch Cobalt Strike's *execute-assembly*).

Before diving into the evasive techniques, let's briefly discuss *Configuration Knobs* in .NET...

A QUICK PRIMER ON .NET CLR CONFIGURATION KNOBS

While maintaining a wealth of valuable documentation for .NET Framework and subsequently releasing open-source .NET Core, Microsoft has provided valuable (explicit and implicit) insight into the inner workings of the functional components of the .NET ecosystem. .NET, in general, is a very powerful and capable development platform and runtime framework for building and running .NET managed applications. A powerful feature of .NET (on Windows in particular), is the ability to adjust the configuration and behavior of the .NET Common Language Runtime (CLR) for development and/or debugging purposes. This is achievable through .NET CLR Configuration Knobs controlled by environment variables, registry settings, and/or configuration files/property settings as retrieved by the *CLRConfig*.

Abusing configuration knobs is not a new concept. Other researchers have explored various techniques for leveraging knob settings to execute arbitrary code and/or evade defensive controls. A few recent examples include Adam Chester's (@_xpn_) use of the *ETWEnabled* CLR configuration knob to disable Event Tracing for Windows (ETW) and Paul Laîné's (@am0nsec) use of the *GCName* CLR configuration knob to specify a custom Garbage Collector (DLL) for loading arbitrary code and bypassing application control solutions. And of course, Casey Smith (@subTee) for exploring all things .NET including COR_PROFILER unmanaged code loading for defense evasion/UAC bypass and the Ghost Loader *AppDomainManager* injection technique (as further described by @netbiosX).

ADJUSTING .NET CONFIGURATION KNOB REGISTRY SETTINGS TO EVADE CLR USAGE LOG FILE CREATION

Interestingly, .NET Usage Log output location can be controlled by setting the *NGenAssemblyUsageLog* CLR configuration knob in the Registry or by configuring an environment variable (as described in the next section). By simplify specifying an arbitrary value (e.g. fake output location or junk data) for the expected value, a Usage Log file for the .NET execution context will not be created. The *NGenAssemblyUsageLog* CLR configuration knob string value can be set at the following Registry keys:

- HKCU\SOFTWARE\Microsoft\.NETFramework
- HKLM\SOFTWARE\Microsoft\.NETFramework

Configuring the value within the HKCU hive will apply for the active user context and influence logs output that would otherwise log to: <SystemDrive>:\Users\<user>\AppData\Local\Microsoft\CLR_<version>_(arch)\UsageLogs directory and/or Microsoft Office Hub paths. Configuring the value within the HKLM hive will apply for the system context and influce logs output that would otherwise log to: <SystemDrive>:\Windows\<System32|SysWOW64>\config\systemprofile\AppData\Local\Microsoft\CLR_<version>_(arch)\Usa directory paths. Let's walk through a simple example to demonstrate expected and tampered behavior...

The following source code is compiled as a 64-bit NET application called 'test.exe':

```
using System.Windows.Forms;

namespace test
{
    -references
    class Program
    {
        -references
        static void Main(string[] args)
        {
            MessageBox.Show("Hello World!", "");
        }
    }
}
```

Before executing the application, take note that the *UsageLogs* directory is empty on this test machine. The directory may be well populated on production or test machines.

Once executed, a simple message box appears:

Upon inspecting the *UsageLogs* directory, a file named *test.exe.log* is created that contains assembly module information:

Next, let's remove the test.exe.log file from the *UsageLogs* directory to demonstrate tampering behavior:

Before re-executing the .NET application, let's validate the existence of the .NETFramework registry (sub)key in *HKCU* with the following command:

```
reg query "HKCU\SOFTWARE\Microsoft\.NETFramework"
```

In this case, the Registry key exists and does not contain additional values or subkeys. (Note: If the *.NETFramework* key does not exist, it can be created). Next, add the *NGenAssemblyUsageLog* configuration knob string value to the .NETFramework key and verify the change:

```
reg.exe add "HKCU\SOFTWARE\Microsoft\.NETFramework" /f /t REG_SZ /v
"NGenAssemblyUsageLog" /d "NothingToSeeHere"
```

The program is executed again:

And as expected, the text.exe.log file does not appear after viewing the contents of the target *UsageLogs* directory:

So you may be asking – what does the CLR do when you supply the *NGenAssemblyUsageLog* name an arbitrary value? Well, it really just inserts the arbitrary string into a 'properly' constructed path. For instance, if we set the path data to 'eeeee' and execute a .NET application, the CLR inserts the string value into the constructed path::

Since the path is not found, the Usage Log does not write to disk. As shown in the following screenshot, the partial *UsageLogs* path suffix is hardcoded and pulled from *clr.dll*:

ADJUSTING .NET CONFIGURATION KNOB ENVIRONMENT VARIABLES TO EVADE CLR USAGE LOG FILE CREATION

CLR configuration knobs are also configured by setting environment variables with the the *COMPlus_* prefix. In the following example, the *COMPlus_NGenAssemblyUsageLog* is set to an arbitrary value (e.g. 'zzzz') in the Command Prompt. When PowerShell (a .NET application) is invoked, the *COMPlus_NGenAssemblyUsageLog* environment variable is inherited from the parent cmd.exe process:

After exiting PowerShell, we note that the Usage Log file (powershell.exe.log) is never created in the *UsageLogs* directory:

When Adam Chester (@_xpn_) blogged about the *ETWEnabled* .NET CLR knob configuration discovery for disabling ETW processing, he published a spoofing proof-of-concept to inject the *COMPlus_ETWEnabled* environment variable when launching a child process. After modifying a few variables in the program, the same spoofing technique can be used to disable the Usage Log output as shown in this code snippet:

After compiling and executing the program, PowerShell.exe is launched with the *COMPlus_NGenAssemblyUsageLog* environment variable set to an arbitrary value of "zz":

And as expected, the Usage Log is never created after exiting the PowerShell session:

Note: the modified environment variable spoofing POC can be found here.

DISRUPTING THE CLR USAGE LOG OUTPUT OPERATION VIA FORCEFUL PROCESS TERMINATION

.NET Configuration Knobs provide an elegant way to influence log flow. However, there are methods for disrupting the Usage Log creation process without having to make configuration changes. These methods pose greater risk for disrupting process and program workflow.

Usage Logs are generated when a process exits 'gracefully'. This occurs when an assembly completes the execution process, such as when using an implicit or explicit *return* statement or when using the Environment.Exit() method in (C#) managed code:

However, if the process is forced to terminate, the Usage Log process is disrupted and never written to disk. As an example, the Process.Kill() method can be used to achieve the desired result (at the risk of losing data *or a shell*):

DISRUPTING THE CLR USAGE LOG OUTPUT OPERATION VIA MODULE UNLOADING

In another other interesting albeit risky testing scenario, tampering with loaded modules (DLLs) can be used to disrupt CLR Usage Log creation by destabilizing the process and causing it to prematurely exit. To accomplish this, we leverage .NET delegate function pointers and the powerful DInvoke library authored by The Wover (@TheRealWover) and b33f (@FuzzySec). For the test case, a delegate function pointer is declared for the *FreeLibrary()* Win32 API function which is called to unload modules from the running .NET managed process. Removing a single module or a lesser combinations of modules could potentially achieve the same effect, however, we will unload several .NET modules to increase the chances of making the process unstable to force termination and disrupt Usage Log creation (Note: We are picking on .NET modules here but other DLLs could be unloaded as well)

To successfully unload a module, we must first get a pointer to the library address of the *FreeLibrary()* function with DInvoke's *GetLibraryAddress()*. Then, we convert the function

pointer to a callable delegate for the *FreeLibrary()* API method with the *GetDelegateForFunctionPointer()* method from .NET 'Interop' services. Next, we get a handle on each of the the loaded modules (DLLs) by searching for each module's base address reference in the Process Execution Block (PEB) of our .NET process with DInvoke's *GetPebLdrModuleEntry()* method. Lastly, we call the *FreeLibrary* delegate function with the handle to each module to unload it from memory. The POC code in this test case is appears as follows:

After compiling and executing the code, the Usage Log file creation process is disrupted (as expected):

For more information about DInvoke, check out this fantastic blog post by The Wover (@TheRealWover) and b33f (@FuzzySec). The POC code for unloading DLL modules can be found here.

DEFENSIVE CONSIDERATIONS

Continue to monitor Usage Logs files & directories. Implement analytics/signatures/detections for Usage Log creation and modification. Despite the *questionable* offensive techniques demonstrated here, such detections are still quite valuable. Offensive operators will not always account for Usage Log tampering while executing their .NET tools.

Look for log instances of (irregular) unmanaged binaries and script hosts that would not typically load the CLR to create a Usage Log. Leverage Olaf Hartong's (@olafhartong) Sysmon-Modular rule config and/or this Elastic Security rule query as a baseline for getting started with a rule set. Additionally, Samir (@SBousseaden) provides an excellent detection tip for monitoring WinRM Lateral Movement using .NET tools.

Furthermore, audit and monitor for attempts to remove Usage Log files as offensive operators may remove the Usage Log files from disk to cover their tracks. Note: This is tradecraft is mentioned in the MENASEC blog post.

Monitor for suspicious .NET runtime loads. Identifying suspicious .NET CLR runtime loads may be an interesting compensation detection mechanism if Usage Log evasions are deployed. Unmanaged processes that load the CLR (e.g. MS Office). could be an indicator of compromise.

Monitor for CLR configuration knob additions or modifications. Roberto Rodriguez (@Cyb3rWardog) authored a fantastic write-up for detecting the [COMPLUS_]ETWEnabled configuration knob adjustment behavior that includes SACL audit recommendations, Sysmon configuration settings, Sigma rules, and a Yara rule. The same methodologies can be applied to detect [COMPLUS_]NGenAssemblyUsageLog configuration knob modifications. A summary of (replicated) recommendations include the following:

- Hunt for the addition of the *NGenAssemblyUsageLog* string in the *HKCU\Software\Microsoft\.NETFramework* and *HKLM\Software\Microsoft\.NETFramework* Registry keys. As Roberto points out, Event ID 4657 is generated when the audit object access policy is enabled and the target key is audited for key write/set value events:

- Hunt for the prepending of *COMPlus_* in permanent user/system environment variables (see Roberto's notes), and temporary environment variables where applicable – e.g. process command line , transcription logs, etc.

Monitor for process module tampering. Monitoring for 'suspicious' process termination events may not be practical in most organizations. However, unloading DLLs from a running process could be an interesting detection opportunity. As described by spotheplanet (@spotheplanet) in this post, module unloads can be traced with the ETW *Microsoft-Windows-Kernel-Process* provider.

FUTURE RESEARCH & CONCLUSION

If you discover other Usage Log evasion techniques or have improved ideas for detecting them, please feel free to reach out on Twitter and I will link to your resource page. I am currently investigating an "in-depth" technique that may circumvent Usage Log creation, but my current approach hasn't quite worked out just yet :).

Of note, MSRC was notified of this issue prior to the release of this post. Microsoft does not consider Usage Log evasion a security boundary issue.

And as always, thanks for taking the time to read my posts!

~ Bohops

2 THOUGHTS ON "INVESTIGATING .NET CLR USAGE LOG TAMPERING TECHNIQUES FOR EDR EVASION"

Pingback: Investigating .NET CLR Usage Log Tampering Techniques For EDR Evasion (Part 2) – bohops

Pingback: Abusing and Detecting LOLBIN Usage of .NET Development Mode Features – | bohops |

Comments are closed.

PREVIOUS POST

NEXT POST