Understanding ETW Patching

jsecurity101.medium.com/understanding-etw-patching-9f5af87f9d7b

Jonathan Johnson April 12, 2024

Jonathan Johnson

Introduction

As of late, I have gotten a lot of questions around Event Tracing for Windows (ETW) patching, specifically the following questions:

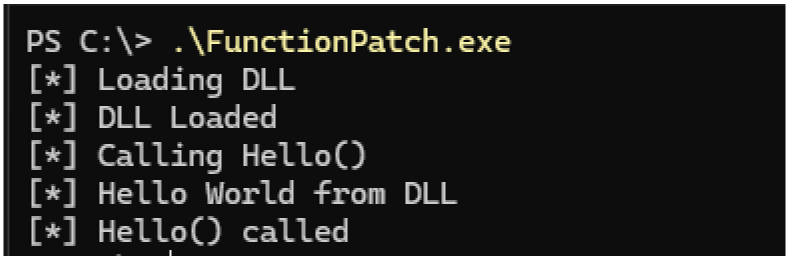
- Which ETW providers can be patched (kernel-mode vs. user-mode)?
- What does it mean to actually patch out an ETW Provider?
- How can you detect ETW patching?

These are all valuable questions, so I decided to write up on these questions and answer any misconceptions or misunderstandings people have about ETW patching.

Function Patching

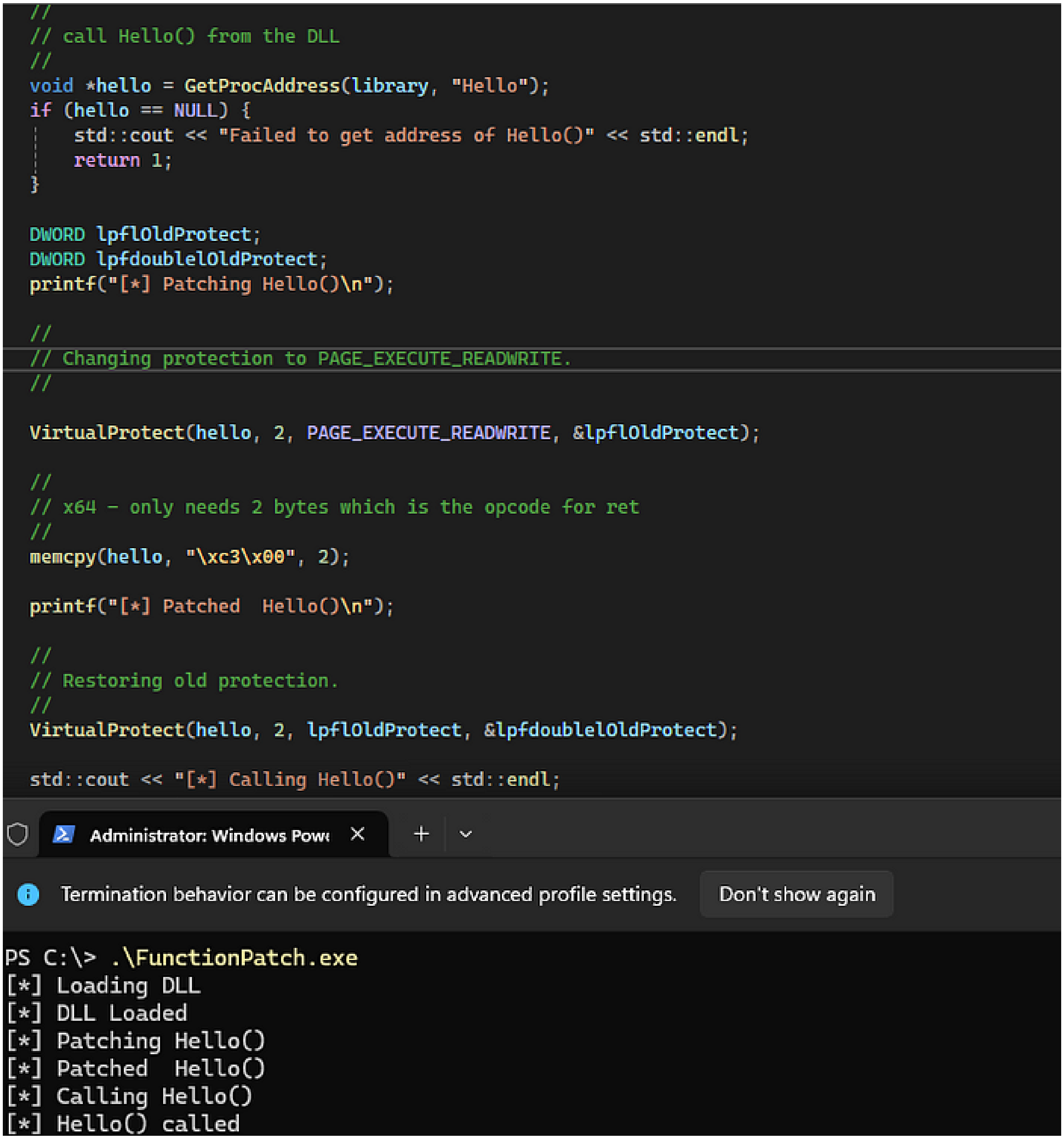
At a high level, function patching refers to changing the code flow of a given function by either making it fail, providing fake data, or having the function immediately return. This might benefit someone who wants a function to execute differently than usual or not execute anything at all. From a local process perspective, to patch this out, a function, the PE that holds the function (EXE or DLL) that holds the desired function, needs to be loaded, and a function pointer needs to be obtained. Once that has been received, someone can change the protection of the memory region where the function is located and then disrupt normal code flow.

Let’s take an example — say we have a DLL that has an exported function (Hello()) that prints “Hello World from DLL.”



```
PS C:\> .\FunctionPatch.exe
[*] Loading DLL
[*] DLL Loaded
[*] Calling Hello()
[*] Hello World from DLL
[*] Hello() called
```

You can see above that the DLL loads and executes Hello() fine. Let’s say we want to patch this function so that any time this function is called it immediately returns instead of actually executing completely.



```
//
// call Hello() from the DLL
//
void *hello = GetProcAddress(library, "Hello");
if (hello == NULL) {
    std::cout << "Failed to get address of Hello()" << std::endl;
    return 1;
}

DWORD lpflOldProtect;
DWORD lpfdoubleOldProtect;
printf("[*] Patching Hello()\n");

//
// Changing protection to PAGE_EXECUTE_READWRITE.
//

VirtualProtect(hello, 2, PAGE_EXECUTE_READWRITE, &lpflOldProtect);

//
// x64 - only needs 2 bytes which is the opcode for ret
//
memcpy(hello, "\xc3\x00", 2);

printf("[*] Patched  Hello()\n");

//
// Restoring old protection.
//
VirtualProtect(hello, 2, lpflOldProtect, &lpfdoubleOldProtect);

std::cout << "[*] Calling Hello()" << std::endl;
```

Administrator: Windows Powe
Termination behavior can be configured in advanced profile settings.
Don't show again

```
PS C:\> .\FunctionPatch.exe
[*] Loading DLL
[*] DLL Loaded
[*] Patching Hello()
[*] Patched  Hello()
[*] Calling Hello()
[*] Hello() called
```

As seen above, a function pointer is obtained to Hello() via GetProcAddress and then patched by copying the bytes that represent the ret opcode for x64 applications. After doing so, “Hello World from DLL” isn’t printed as expected.

ETW Patching

ETW patching is a function patching technique used by offensive operators (threat actors, red teams, etc.) to prevent telemetry from being produced for their action(s). ETW Providers are responsible for emitting events for specific actions (LDAP, .NET, AMSI, etc). Patching is simply changing the code flow of a given function by either making it fail, providing fake data, or having the function immediately return. This includes patching out ETW-specific write functions, like EtwEventWrite or NtTraceEvent, or it can be internal functions that will eventually invoke an ETW-specific write function. There are multiple ways to patch out logging-related functions, but before we dive into that, we have to understand a couple of foundational things that allow patching to be successful.

Prerequisites

In order to patch out an ETW logging function, one must be running code in the process that writes events to the targeted ETW provider. In practice, patching could involve forcing the function to return immediately, provide false data so that the function fails, etc. This holds true for user-mode and kernel-mode based providers (we will dive into this in the **Types of Patching** section below).

Let’s take a practical example of LDAP activity. The Microsoft-Windows-LDAP-Client ETW provider is stored within wldap32.dll. Whenever LDAP activity is executed, the wldap32.dll is loaded within the process. Now, suppose you want to execute an LDAP search without logging it. Before executing the LDAP activity, you need to locate the function’s memory address you want to patch (EtwEventWrite, LDAPSearchLoggingClientTraceEventNoReg, or NtTraceEvent) and provide an alternative instruction. If this was a kernel-mode ETW provider, you would need to be running code in the kernel to perform this.

This means you *can* patch providers within your process, a remote process, or in the kernel. However, each has its own challenges. It is much easier to patch out a function in your current process than it is in a remote process because you would need to leverage a function like WriteProcessMemory to properly patch out the function, which is most likely going to get an operator caught due to the large corpus of Process Injection detections out there. You can also patch out providers in the kernel, but you need to find a way to get code execution in the kernel. Which typically consists of finding a vulnerable signed driver that Microsoft’s driver block list doesn’t block. The kernel-patching will be the most challenging and, honestly, the most unrealistic scenario.

Types of Patching

GetProcAddress

GetProcAddress is a well-known Win32 API that allows someone to obtain a function pointer to an exported function within a DLL. For this to work properly, the callee must obtain a module handle to the DLL that holds the exported function. This usually results in someone calling LoadLibrary on the desired DLL. After the function has been returned it is common to change the protection of memory so that they can change the next instructions of the function. This is typically done via VirtualProtect. Afterwards, depending on the function you are trying to patch, after the pointer has been returned and the protection of that memory has been changed one can disrupt normal code flow a number of ways — having the call after be a return instruction for example. One limitation of GetProcAddress is that whatever function your patching needs to be in a DLLs exports table, so this wouldn’t work if you want to patch an internal function.

Manual Function Pointer

GetProcAddress is the most commonly used API for function patching I have seen, but someone can actually do the exact same thing without it. If someone finds a function, they want to patch all they have to do is find where the function’s virtual address offset within a DLL. Once that is found and a DLL is loaded within a process they can get where in memory that DLL is loaded and add that virtual address offset to it and then they have a pointer to that function. This looks something like this:

```
   DWORD offset = ; LPVOID ldapClient = <LPVOID>(<DWORD_PTR>(hModule) +
offset);std::cout <<  << ldapClient << std::endl;
```

They would then follow the same process of changing the protection of that region of memory and manipulating code flow. This works for both internal and external functions. A limitation to keep in mind with this is that it is possible the virtual address offset can change on versions of DLLs, however in my testing I was able to get my code to work fine on Windows 10 & 11 machines.

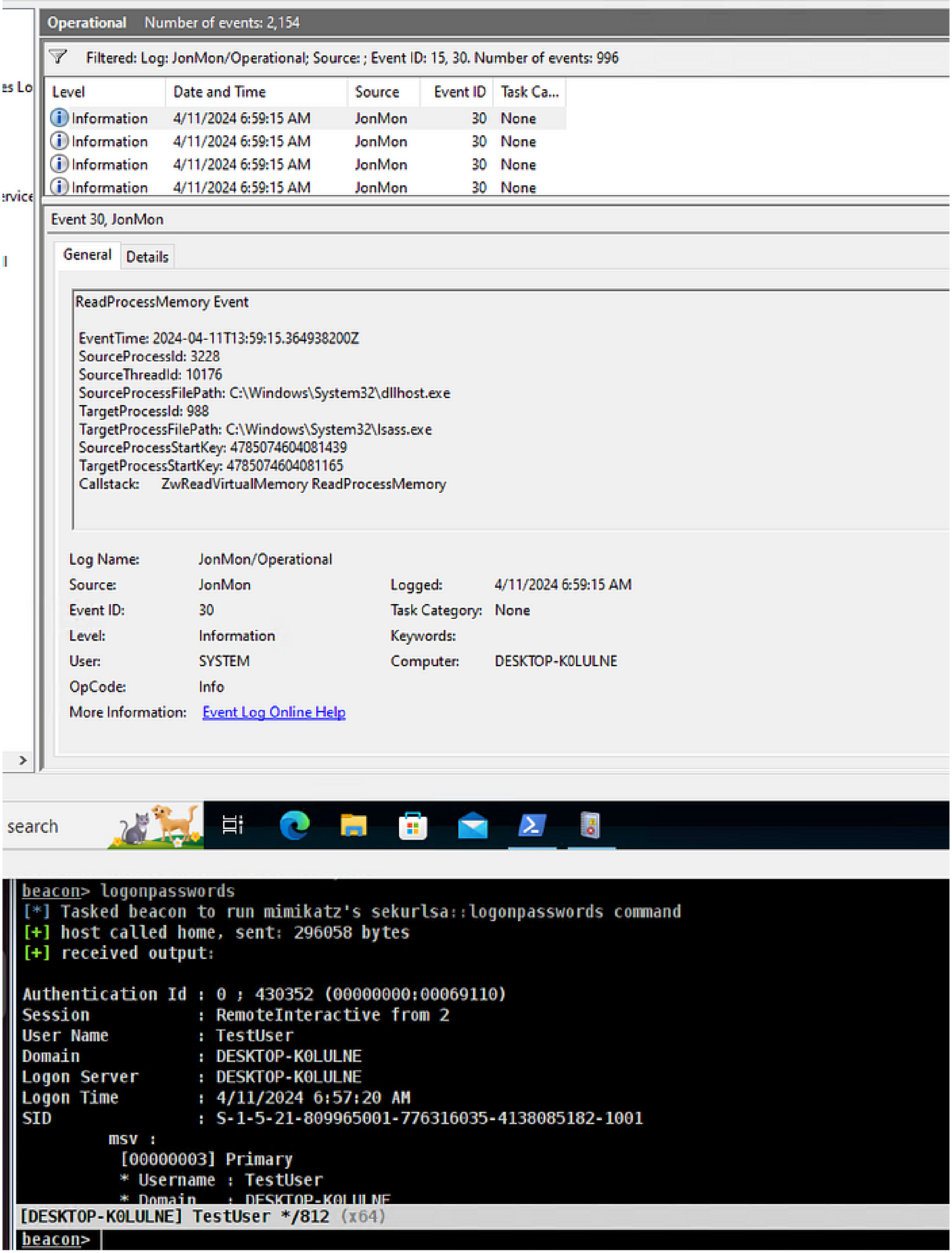
Kernel

All the examples I have shown above have been user-mode examples, however let’s dive into kernel-mode providers. I have seen comments about people patching out the Threat-Intelligence ETW provider. Unless someone has a code execution in the kernel by way of a driver or some other means, this isn’t possible. Just like with the user-mode providers someone needs to be working with the memory where the functions are called. I want to explore this a bit, the Threat-Intelligence provider collects information for ReadProcessMemory (useful for dumping LSASS telemetry). The functions used to complete ReadProcessMemory transition into the kernel and the function used to log this is EtwTiLogReadWriteVm (please see my project TelemetrySource for more). There is nothing of value for someone to patch within user-mode.

This goes for other kernel-mode ETW providers too. Just because someone patches out EtwEventWrite or NtTraceEvent doesn’t mean that providers will stop being written to. Let’s take a look at a practical example.

Example 1: Beacon LogonPasswords

This example has a process running as “beacon.exe” that calls mimikatz’s logonprocess function to dump LSASS. This does not patch ETW at all and we can see we get ReadProcessMemory events from the ETW Threat Intelligence provider. The process shown below is dllhost.exe, but that is a CobaltStrike-ism with how it executes commands via sacrificial processes. This is derived from the beacon.exe process.



Filtered: Log: JonMon/Operational; Source: ; Event ID: 15, 30. Number of events: 996

Level	Date and Time	Source	Event ID	Task Ca...
Information	4/11/2024 6:59:15 AM	JonMon	30	None
Information	4/11/2024 6:59:15 AM	JonMon	30	None
Information	4/11/2024 6:59:15 AM	JonMon	30	None
Information	4/11/2024 6:59:15 AM	JonMon	30	None

Event 30, JonMon

```
ReadProcessMemory Event

EventTime: 2024-04-11T13:59:15.364938200Z
SourceProcessId: 3228
SourceThreadId: 10176
SourceProcessFilePath: C:\Windows\System32\dllhost.exe
TargetProcessId: 988
TargetProcessFilePath: C:\Windows\System32\lsass.exe
SourceProcessStartKey: 4785074604081439
TargetProcessStartKey: 4785074604081165
Callstack:     ZwReadVirtualMemory ReadProcessMemory
```

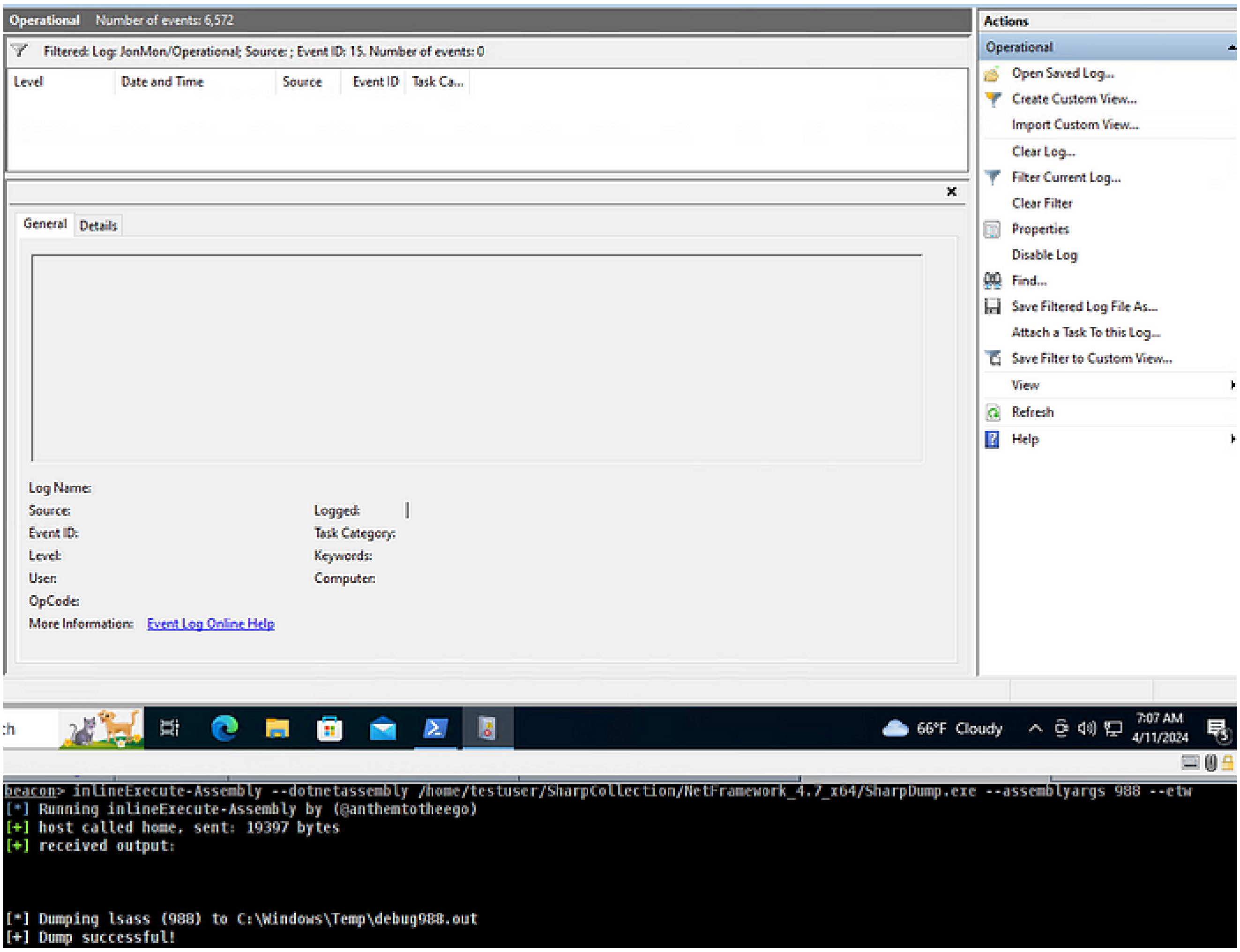
Log Name:	JonMon/Operational		
Source:	JonMon	Logged:	4/11/2024 6:59:15 AM
Event ID:	30	Task Category:	None
Level:	Information	Keywords:	
User:	SYSTEM	Computer:	DESKTOP-K0LULNE
OpCode:	Info		
More Information:	Event Log Online Help		

```
beacon> logonpasswords
[*] Tasked beacon to run mimikatz's sekurlsa::logonpasswords command
[+] host called home, sent: 296058 bytes
[+] received output:

Authentication Id : 0 ; 430352 (00000000:00069110)
Session           : RemoteInteractive from 2
User Name         : TestUser
Domain            : DESKTOP-K0LULNE
Logon Server      : DESKTOP-K0LULNE
Logon Time        : 4/11/2024 6:57:20 AM
SID               : S-1-5-21-809965001-776316035-4138085182-1001
        msv :
         [00000003] Primary
         * Username : TestUser
         * Domain   : DESKTOP-K0LULNE
[DESKTOP-K0LULNE] TestUser */812 (x64)
beacon>
```

Example 2: inlineExecute-Assembly + SharpDump

This example uses inlineExecute-Assembly to call SharpDump to dump LSASS and uses the " — etw" flag to patch EtwEventWrite. This should show events from the DotNet ETW provider, but it doesn't due to the " — etw" flag (see Picture 1).

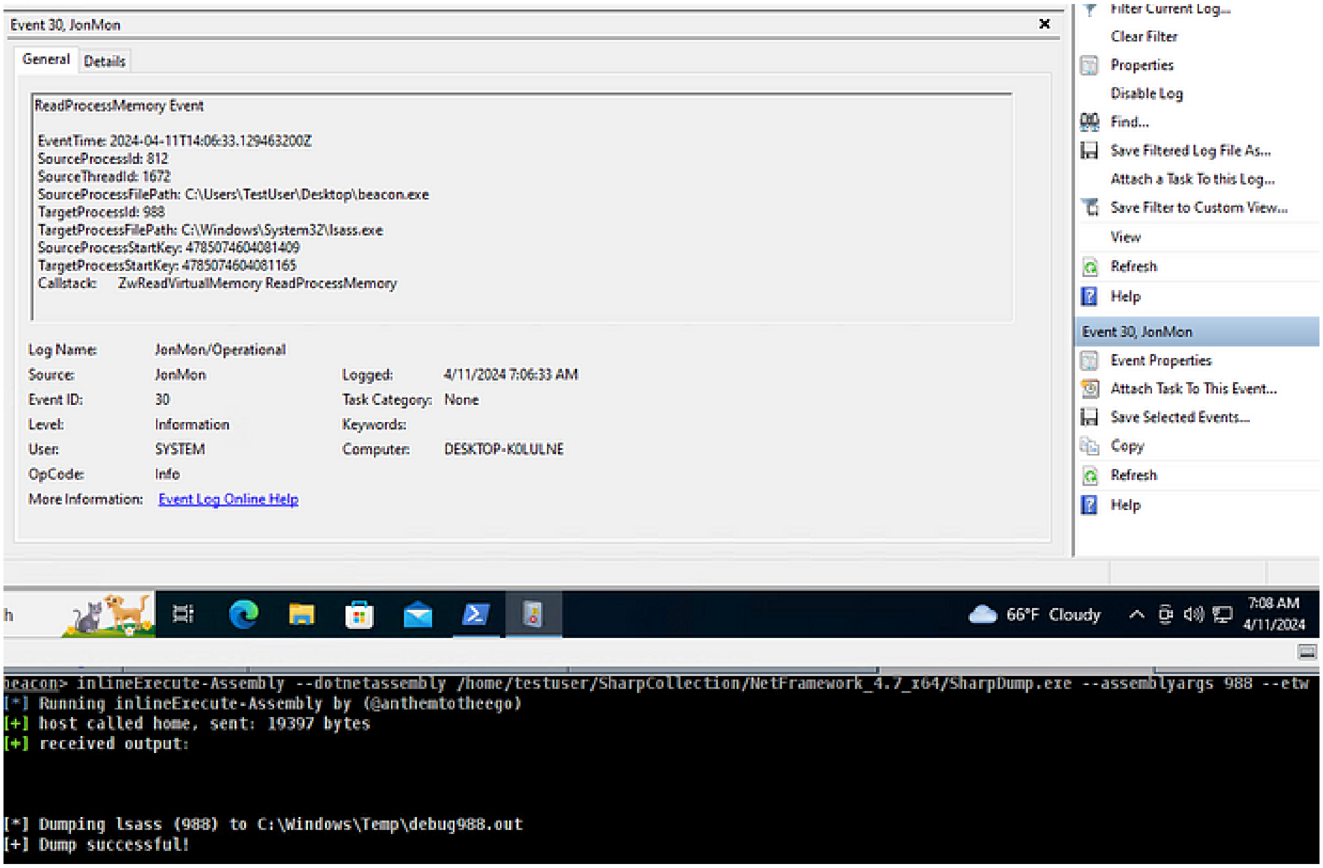

Operational Number of events: 6,572

Filtered: Log: JonMon/Operational; Source: ; Event ID: 15. Number of events: 0

```
beacon> inlineExecute-Assembly --dotnetassembly /home/testuser/SharpCollection/NetFramework_4.7_x64/SharpDump.exe --assemblyargs 988 --etw
[*] Running inlineExecute-Assembly by (@anthemtotheego)
[+] host called home, sent: 19397 bytes
[+] received output:


[*] Dumping lsass (988) to C:\Windows\Temp\debug988.out
[+] Dump successful!
```

The 2nd picture below shows that regardless of ETW patching via EtwEventWrite the Threat-Intelligence provider will still write a ReadProcessMemory event, because the Threat-Intelligence provider is in the kernel vs. user-mode.



```
ReadProcessMemory Event

EventTime: 2024-04-11T14:06:33.1294632002
SourceProcessId: 812
SourceThreadId: 1672
SourceProcessFilePath: C:\Users\TestUser\Desktop\beacon.exe
TargetProcessId: 988
TargetProcessFilePath: C:\Windows\System32\lsass.exe
SourceProcessStartKey: 4785074604081409
TargetProcessStartKey: 4785074604081165
Callstack:    ZwReadVirtualMemory ReadProcessMemory
```

Log Name: JonMon/Operational
Source: JonMon
Logged: 4/11/2024 7:06:33 AM
Event ID: 30
Task Category: None
Level: Information
Keywords:
User: SYSTEM
Computer: DESKTOP-K0LULNE
OpCode: Info
More Information: Event Log Online Help

```
beacon> inlineExecute-Assembly --dotnetassembly /home/testuser/SharpCollection/NetFramework_4.7_x64/SharpDump.exe --assemblyargs 988 --etw
[*] Running inlineExecute-Assembly by (@anthemtotheego)
[+] host called home, sent: 19397 bytes
[+] received output:


[*] Dumping lsass (988) to C:\Windows\Temp\debug988.out
[+] Dump successful!
```

Which functions to patch?

I want to take a moment to talk about the offensive benefit of patching out certain functions vs. others. Historically I have seen tools and malware patch functions like AmsiScanBuffer, EtwEventWrite or NtTraceEvent because these are all functions that are exported out of a DLL, so there is less complexity in patching these out. However, as of late it isn't uncommon for anti-virus (AV) products to pick up on these behaviors within a file through their scans. So, what is the alternative?

1. Manual patching common functions (, , , etc).
2. Manual patching internal logging functions that eventually call EtwEventWrite, like how I did above with LDAPSearchLoggingClientTraceEventNoReg.

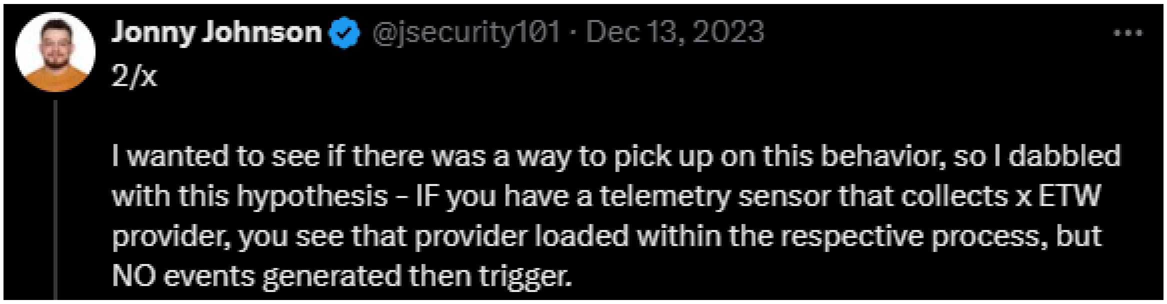
I also want to take a moment and touch on remote patching. Again, this is possible but someone would have to leverage WriteProcessMemory to the remote process which is more likely to get caught versus local patching. You might wonder — why is this the case if the activity originates from your process, why would you have to patch a function in a remote process? This comes down to code flow. Let's take a look at an example.

Say you want to create a scheduled task via Register-ScheduledTask in PowerShell. The code flow stays in your process (really your thread) and goes through a WMI provider, a COM server, which then invokes a RPC method. When this RPC method is invoked the code flow transitions from your process to wherever the RPC server is located, which in this case is schedsvc.dll which is loaded into a svchost.exe process. It's not until after that transition that the logging functions Auditor::AuditJobOperation / AuthziLogAuditEvent are executed. You can manipulate those functions so that they don't log properly, but you'd have to get code execution into the svchost.exe process that has the schedsvc.dll loaded. The key here is — understanding when code transitions out of your current processes context to another which is common with some WMI methods, a lot of COM methods, and almost every RPC method. More often than not the logging functions are going to be invoked at the end of the call stack which is why understanding these transitions is important.

Note: If you want to learn how to analyze this transition between PowerShell, WMI, COM and RPC check out my post: WMI Internals: Reversing a WMI Provider.

Defenders

It is imperative for defenders to understand OS internals as well, if not more than offensive engineers. In my opinion defenders should understand patching, even though they aren't running operations and performing patching all the time. That being said, there isn't a great way to detect patching. I dabbled with an idea a while back in this tweet:



Jonny Johnson @jsecurity101 · Dec 13, 2023
2/x

I wanted to see if there was a way to pick up on this behavior, so I dabbled with this hypothesis - IF you have a telemetry sensor that collects x ETW provider, you see that provider loaded within the respective process, but NO events generated then trigger.

I have seen good success with this, but it isn't a silver bullet and requires some development and fine tuning. **I think it is important to keep trying to innovate new ideas like this to pick up on common offensive tradecraft.** If anyone would like to discuss more strategies on this, please reach out to me.

Also, keep in mind when creating detections which providers are likely to get patched vs. not. I always suggest that when people dive into telemetry that can be used for a detection to try to find something where the ETW provider isn't writing events from the source process so that you don't have to worry as much about patching. An example — although the DotNet ETW provider is a good telemetry source for seeing malicious .NET assemblies being loaded, I personally wouldn't have high confidence in using it as a primary data source for a detection. I would still use it, but I would try to look for another telemetry source to help me pick up on the activity I am interested in. That way if it fires — great, that is an easy win. If it doesn't, I am still covered detection wise.

Conclusion

I wanted to write a blog discussing how ETW patching is really just function patching and explain the basics behind that. It is imperative for defenders and offensive engineers to understand the foundations behind offensive capabilities. ETW patching is something that is used quite a lot (I've seen it in more red team tools than in-the-wild malware). Due to the large number of tools out there, I didn't see a reason to release any proof-of-concepts. I will link valuable resources below. Please let me know if people would be interested in a write-up on how to find the virtual address offset of functions for manual patching. I hope this was helpful for some!

Resources