Busting Red Team Trends With Style

Lessons Learned From Building an ETW Based Sysmon Replacement From Scratch

Philipp Schmied, Sebastian Feldmann | 13.06.24

Who Are We?

- Philipp Schmied
- Sebastian Feldmann
- Dominik Phillips (Can’t be with us today ☹)

@CaptnBanana 🍌

- Members of the CSIRT of Deutsche Bahn AG
- Both former Red Teamers
- Now Detection Engineering and Incident Response
  - Building detection rules, evaluating telemetry sources, staring at logs …

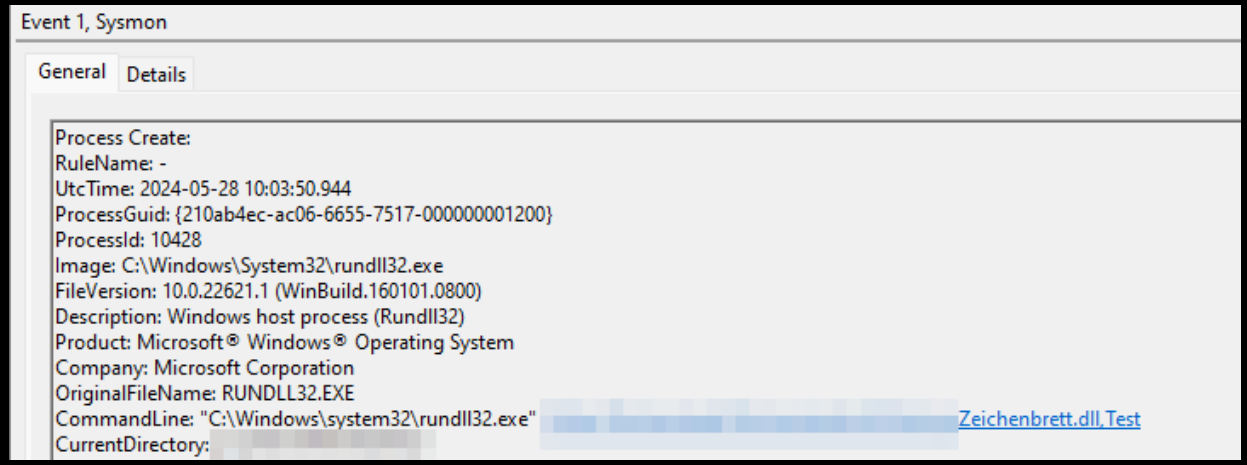
@thefLinkk

But Why?!

- Reliable and context-rich telemetry is crucial for defenders
- Key-Role for detection rules, hunting and incident response
- Different sensors have pros and cons
- Defenders have to trust third party sensors and cannot customize the sensor
  - Cannot add new events
  - Cannot enrich existing events
  - Have to trust the sensor to reliably forward events
- This makes defenders live difficult …

But Why?!

- Example: Sysmon EventID 3
- Rundll connects to the Internet
  - Difficult to determine why
- Need to corelate to EventID 1
  - Requires joining events
- No access to source code of Sysmon
  - We can not customize the event!



MDE

- MDE has more rich data for us
- Allows defenders to build complex rules
  - Reduces FPs

```
1   DeviceNetworkEvents
2   | where InitiatingProcessFolderPath =~ @"c:\windows\system32\rundll32.exe"
3   | where InitiatingProcessCommandLine contains @"c:\windows\temp\"
```

- There is only one problem however …

Event Sampling

- It is up to MDE if an event is forwarded or not
- Potentially not forwarded if:
  - Event is deemed not crucial
  - Too many events
  - … ? Fully opaque
- ImageLoad events e.g. are sometimes forwarded. Sometimes not.
- Often, only the first NetworkConnection event is forwarded. Subsequent (sometimes) not
- WriteProcessMemory Events probably depends on the TargetProcess
- **Unpredictable telemetry makes writing detection rules difficult!**

CallStacks

- MDE fully lacks CallStacks
- Sysmon has one for EventID 10 (ProcessAccess)
- CallStacks are crucial to write fine granular detection rules
  - Enables check for private memory regions (= injected tools)
  - Allows (In-)DirectSycall detection
  - So much more

Goal

- We want to have the best of both worlds
- A customized security sensor under our control
  - Allowing us to fine-tune events to our needs
- To be used alongside MDE
  - Still used for alerting, IR, Isolation…
- Implemented in user-land
  - For stability reasons
- Sysmon Compatibility
  - Generate Sysmon-like events for all relevant Event IDs
  - Because we don’t want to edit every Splunk rule

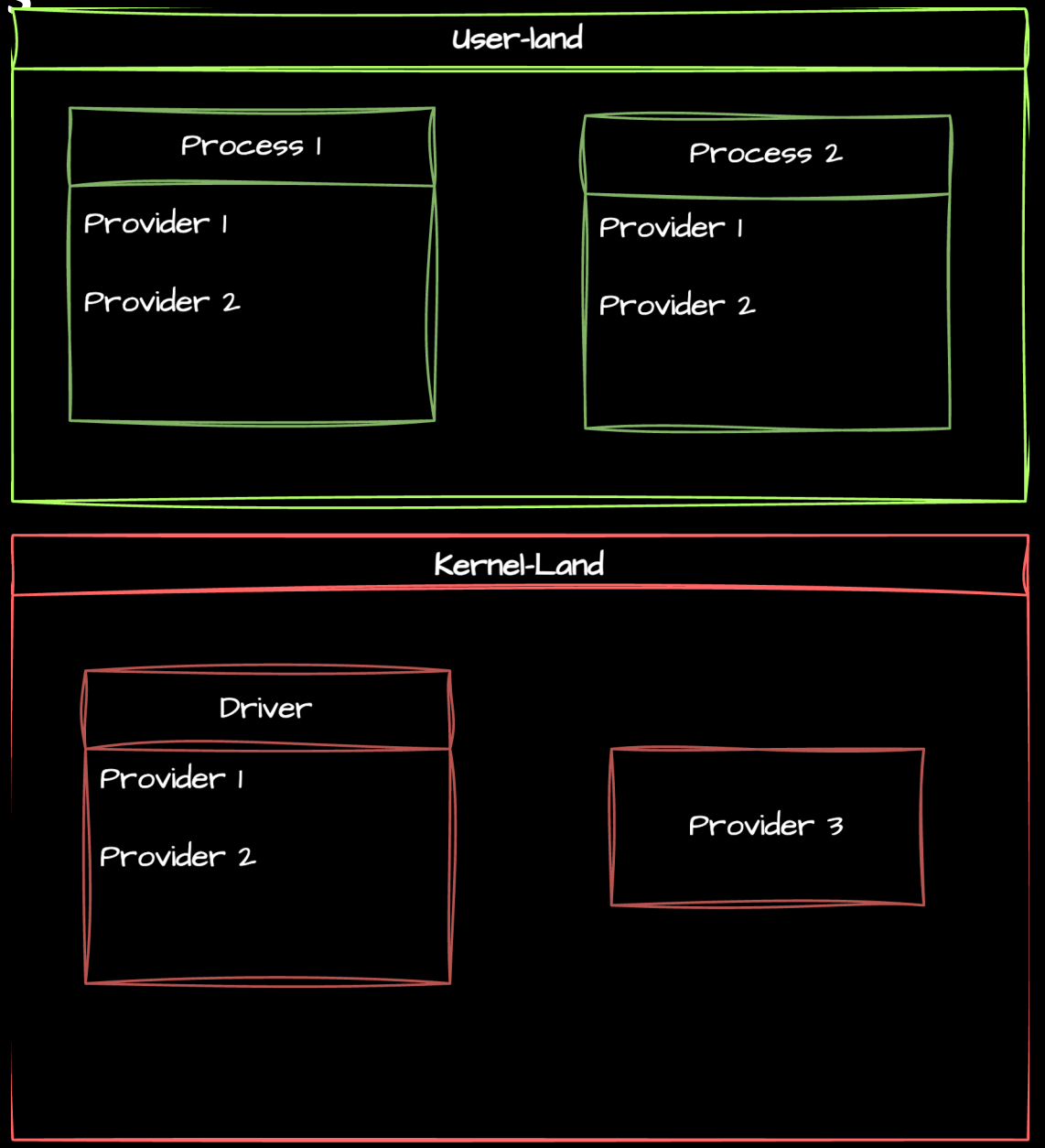
Event Tracing for Windows

Event Tracing for Windows (ETW) Basics

- High-Speed communication mechanism
  - Originally: Logging mechanism for troubleshooting and diagnostics
  - Allows real-time consumption of events
  - **Developers instrument their programs to emit events**
  - Event IDs → Event type
    - E.g. process start, image load, DNS request
- Events are pre-defined by developers
  - **We have no control here, except some configuration**
- Apart from actual payload, ETW events contain meta data, such as
  - PID
  - Timestamp

ETW Basics: Kernel and User Providers

- Difference: Origin of events
- Kernel providers
  - Events generated by kernel-land component
  - E.g. Driver …
- User providers
  - Event source is a user-land component
  - E.g. in DLL / Exe …



First Steps

- Initial Approach:
  - Use known ETW providers to implement Sysmon events
    - Microsoft-Windows-Kernel-Process
    - NT System Trace
    - Microsoft-Windows-TCPIP
    - …

- **That didn’t work out** 🫠

Encountered Issues: Overview

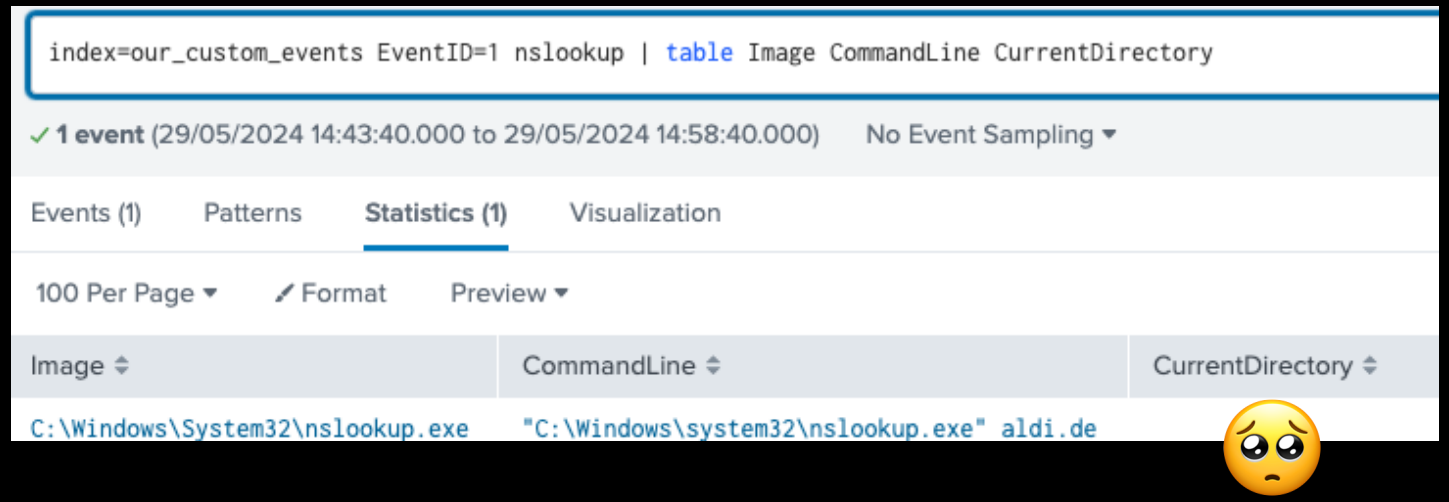
- Performance: High CPU Load due to image hashing
- Missing telemetry: Named pipes and timestomping
- Inconsistent telemetry: Especially for registry operations
- Short-lived processes: Some values have to be determined via process access
- And more 😭

Naïve Approach: Performance Issues

- Sysmon Event ID 7: ImageLoad
  - Contains file hashes
  - One of the most common events
- We have to hash every loaded image
  - → We need optimization
- Sysmon seems to use caching
  - We will do that too
  - Needs mechanism to invalidate cache entries
    - Race condition – Timing issue
      - Sysmon *seems* to have that too
    - Typical issue with ETW: We get notified **after** something happened
      - **We are working with “old” information**

Naïve Approach: Short-Lived Processes

- Sysmon Event ID 1: ProcessCreate
  - Contains CWD (Current Working Directory), LUID (Logon UID)
  - Known ETW providers do not supply these values
- Have to be determined dynamically by accessing the new process
  - Race condition (short-lived processes) → not guaranteed to work
- **Again: Timing-Issue**

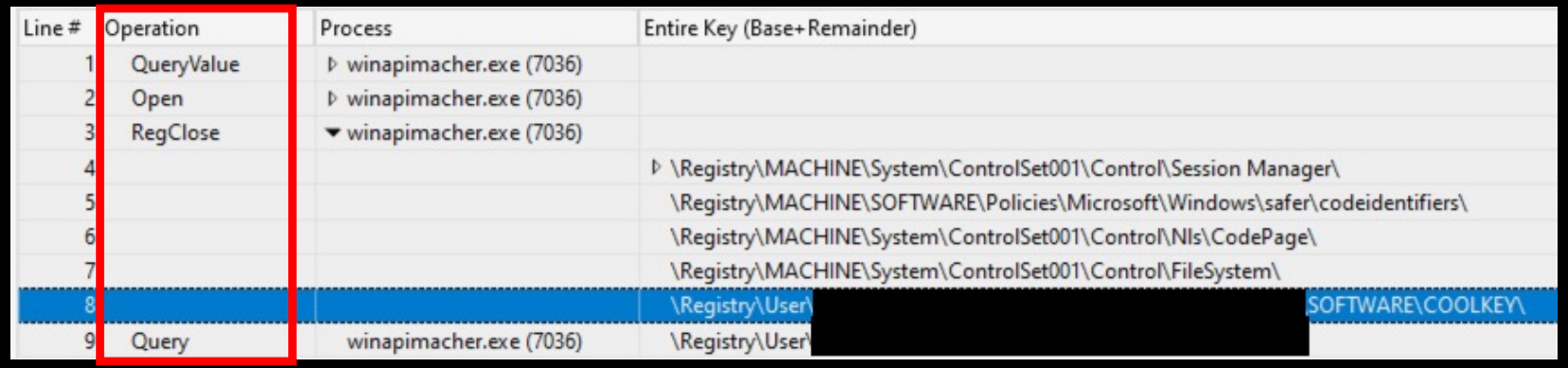


Naïve Approach: Missing Telemetry

- Sometimes, there is no known ETW provider available
  - EventID 2: Timestomping
    - No dedicated event available
      - Therefore no previous timestamp value
  - Named Pipe Events (Create, Connect)
    - No dedicated events, too
    - ObjectManager trace works *in theory*
      - However: High CPU load in practice
      - Due to too many events for this trace
      - No way to do efficient filtering: NamedPipe objects are handled like file objects in the kernel
- More on these events later

Naïve Approach: Registry Events

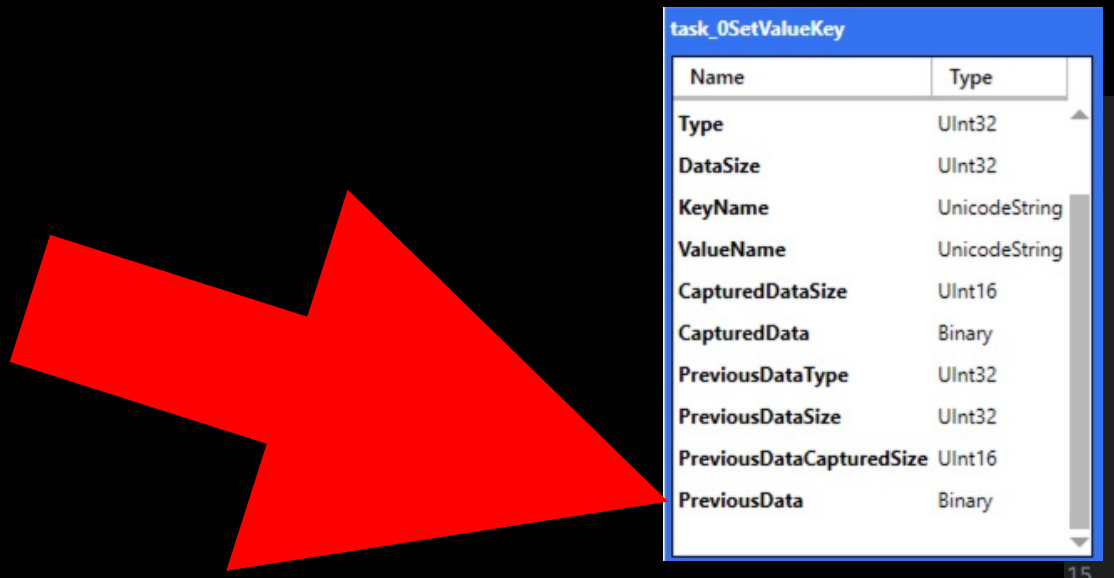
- For threat hunting: Telemetry is quite bad
  - Need to use both Microsoft-Windows-Kernel-Registry and NT System Trace
    - Latter used for Open and Create operations
  - Registry paths in telemetry are sometimes relative, sometimes absolute
    - Also: Sometimes need to be correlated by handle value with Open() events
- Telemetry for registry key rename with RegRenameKeyA()
  - **There are no dedicated rename events**



Line #	Operation	Process	Entire Key (Base+Remainder)
1	QueryValue	▷ winapimacher.exe (7036)	
2	Open	▷ winapimacher.exe (7036)	
3	RegClose	▼ winapimacher.exe (7036)	
4			▷ \Registry\MACHINE\System\ControlSet001\Control\Session Manager\
5			\Registry\MACHINE\SOFTWARE\Policies\Microsoft\Windows\safer\codeidentifiers\
6			\Registry\MACHINE\System\ControlSet001\Control\Nls\CodePage\
7			\Registry\MACHINE\System\ControlSet001\Control\FileSystem\
8			\Registry\User\ [redacted] SOFTWARE\COOLKEY\
9	Query	winapimacher.exe (7036)	\Registry\User\ [redacted]

Naïve Approach: Registry Events

- Manifest: Previous registry values are available
- Reality: Empty 🤯
- Undocumented Flag: Enable previous values



task_0SetValueKey

Name	Type
Type	UInt32
DataSize	UInt32
KeyName	UnicodeString
ValueName	UnicodeString
CapturedDataSize	UInt16
CapturedData	Binary
PreviousDataType	UInt32
PreviousDataSize	UInt32
PreviousDataCapturedSize	UInt16
PreviousData	Binary

```
ULONG64 dwVal = 0x02;
EVENT_FILTER_DESCRIPTOR EnableFilterDesc = { 0 };
EnableFilterDesc.Ptr = (ULONGLONG)&dwVal;
EnableFilterDesc.Size = 4;

[...]

status = EnableTraceEx(
    &ProviderGuid,
    0,
    SessionHandle,
    1,
    TRACE_LEVEL_VERBOSE,
    0xFFFFF,
    0,
    0,
    &EnableFilterDesc
);
```

```
<Data Name="KeyObject">0xffffb48e34c55440</Data>
<Data Name="Status">0</Data>
<Data Name="Type">1</Data>
<Data Name="DataSize">48</Data>
<Data Name="KeyName" />
<Data Name="ValueName">Path64</Data>
<Data Name="CapturedDataSize">0</Data>
<Data Name="CapturedData" />
<Data Name="PreviousDataType">1</Data>
<Data Name="PreviousDataSize">46</Data>
<Data Name="PreviousDataCapturedSize">46</Data>
<Data
    Name="PreviousData">43003A005C00500072006F006700720061006D00
</EventData>
</Event>
```

Thanks to ZeroBuffer (https://www.anquanke.com/post/id/171298)

Encountered Issues: Summary

- We tried really hard, but:
- Telemetry of known providers is not enough for threat hunting purposes
- We managed to implement workarounds for most issues
- We came up with theoretical solutions for *some* missing telemetry
  - However, nothing we would use in production due to performance issues

- Until this point: We are not satisfied
  - Many detection gaps
  - Poor performance
  - In general: Too many limitations

Is ETW still our way to go?

- Do we want to proceed with using ETW?
- Many products, including MDE, rely on kernel drivers
  - Kernel callbacks provide rich and reliable telemetry
- It seems we need kernel-level information too
  - We need a kernel driver
  - But we do not want to rely on Sysmon driver for performance reasons
  - We don’t feel like coding our own driver to run it on production systems (it’s hard)

- **It would be ideal to find an existing kernel driver that emits events using ETW**
  - We also hope for telemetry that’s relevant for threat hunting purposes
- We decided to do more research on such telemetry sources

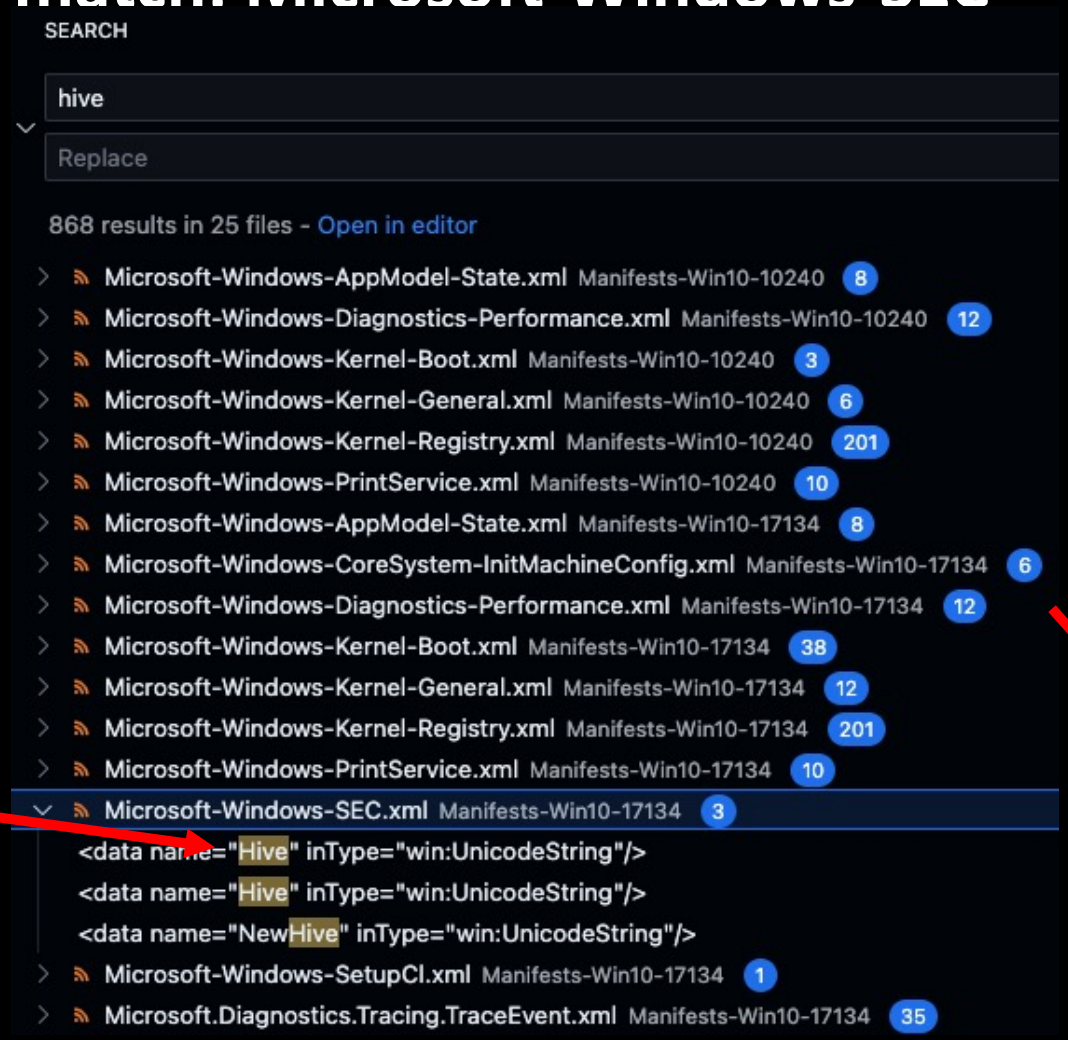
ETW Research

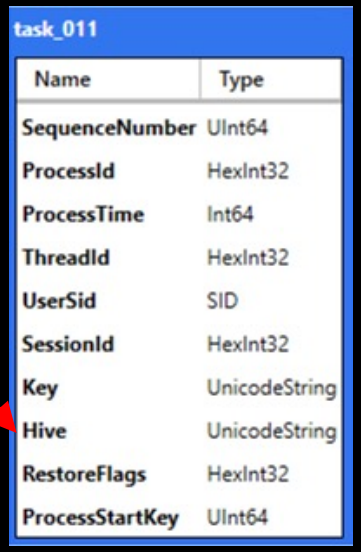
Searching for More Telemetry

- There exist various provider types
- First idea: Look at manifest-based providers (XML)
- We exported all ETW manifests with logman
- Grep for attributes related to registry events
  - E.g. “*hive*”

Searching for More Telemetry

- **Promising match: Microsoft-Windows-SEC**





The SEC-Provider

- Implemented in mssecflt.sys driver
  - Therefore has access to kernel telemetry
  - Not loaded by default
  - Related to Sense-Service
    - Running on all systems **onboarded in MDE**
      - We are using MDE, so this is fine for us
- **This provides raw Sense telemetry**
- This may solve our problems

Accessing the SEC Provider

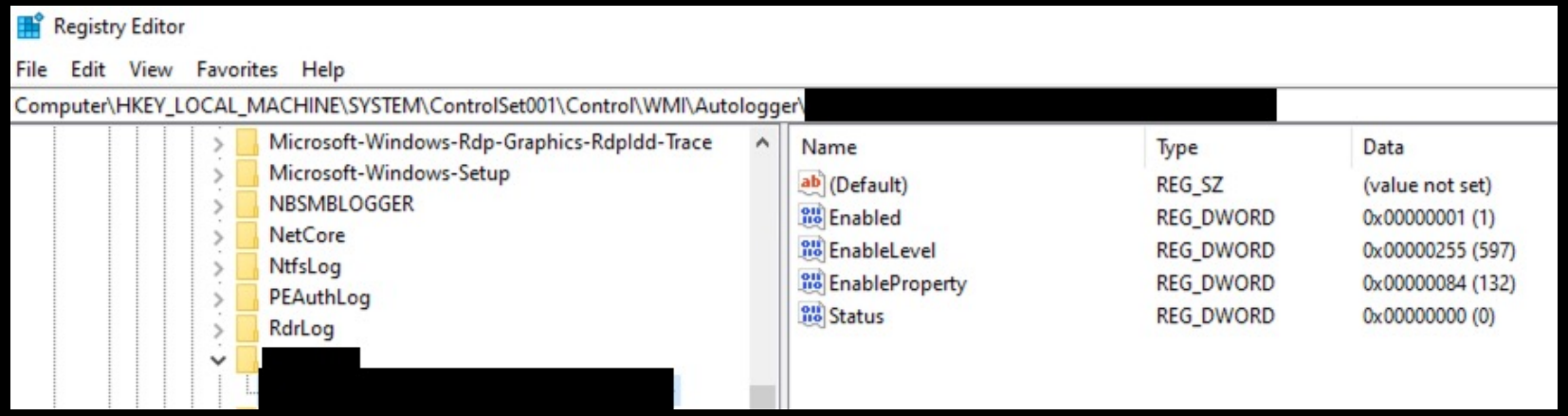
- Accessing Microsoft-Windows-SEC gives “Access Denied” 🤯
  - Even as SYSTEM
  - Driver checks permissions: MS-signed binary required

```
static void Main(string[] args)
{
    var trace = new UserTrace("trace1337");
    var secProvider = new Provider("Microsoft-Windows-SEC");

    trace.Enable(secProvider);
    trace.Start();
}
```

```
C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>C:\Users\                                                    etwtester.exe

Unhandled Exception: System.UnauthorizedAccessException: Need to be admin
   at Microsoft.O365.Security.ETW.UserTrace.Start() in D:\a\_work\1\s\Microsoft.O365.Security.Native.ETW\UserTrace.hpp:l
ine 211
   at etwtester.Program.Main(String[] args) in C:\Users\
```

Accessing the SEC Provider

- Instead of accessing the provider ourselves, we let Windows do it for us
  - We create an autologger session to access the provider
    - Allows system tracing starting from boot process
- **Autologger is started by the kernel: Passes the permission check**
- We can access the autologger using our own tooling
- But: We need a system reboot for that
  - Autologgers are only started upon Windows boot



SEC Provider: Telemetry

- Similar to Sysmon driver telemetry
  - But events are forwarded using ETW instead of IOCTL
  - Better performance in our tests, especially on multi-user systems
- Named Pipe events: Creation and connections
- Process Start w/ hashes and LUID
- Timestomping information
- CallTraces
- No sampling
- **Much potential for custom detections**
- **SEC provider clearly has focus on threat hunting** 

Forking KrabsETW: BlueKrabs

- Changes to ETW library are required
- We created our own fork: BlueKrabs
  - Open/Close existing traces instead of creating new ones (SEC provider)
  - Improved kernel-level filtering
    - Based on ID, PID, flags, payload
- It’s public and you can use it
  - https://github.com/threathunters-io/bluekrabsetw

SEC Provider: Telemetry Examples

ProcessCreate 👇

task_0

Name	Type
ProcessId	HexInt32
ProcessTime	Int64
ThreadId	HexInt32
UserSid	SID
SessionId	HexInt32
CreatorProcessId	HexInt32
CreatorProcessTime	Int64
CreatorProcessName	UnicodeString
ProcessName	UnicodeString
CommandLine	UnicodeString
ImageSHA256	Binary

CreateRemoteThread 👇

task_018

Name	Type
TargetProcessId	HexInt32
TargetProcessTime	Int64
TargetProcessName	UnicodeStrin
TargetThreadId	HexInt32
TargetThreadStartAddress	Pointer
StartAddressVadQueryResult	UInt32
StartAddressVadAllocationBase	Pointer
StartAddressVadAllocationProtect	UInt32
StartAddressVadRegionType	UInt32
StartAddressVadRegionSize	Pointer
StartAddressVadProtect	UInt32

OpenNamedPipe 👇

task_017

Name	Type
SequenceNumber	UInt64
ProcessId	HexInt32
ProcessTime	Int64
ThreadId	HexInt32
UserSid	SID
SessionId	HexInt32
PipeName	UnicodeString
RemoteClientsAccess	UInt32
NamedPipeEnd	UInt32
DesiredAccess	HexInt32
FileOperation	UInt32

Remaining Issues

- Registry Events
  - Manifest of SEC describes interesting registry events
    - Has rename events
  - However, does not generate such events
  - SEC is configured using a bit field in kernel driver
    - Currently, registry events are disabled 😐
    - Hopefully they will be enabled in the future
  - Currently, we're using an alternative
    - Microsoft-Antimalware-Engine
    - Event ID 105: Registry events
  - Registry key renames may still be an issue, though

WEASEL

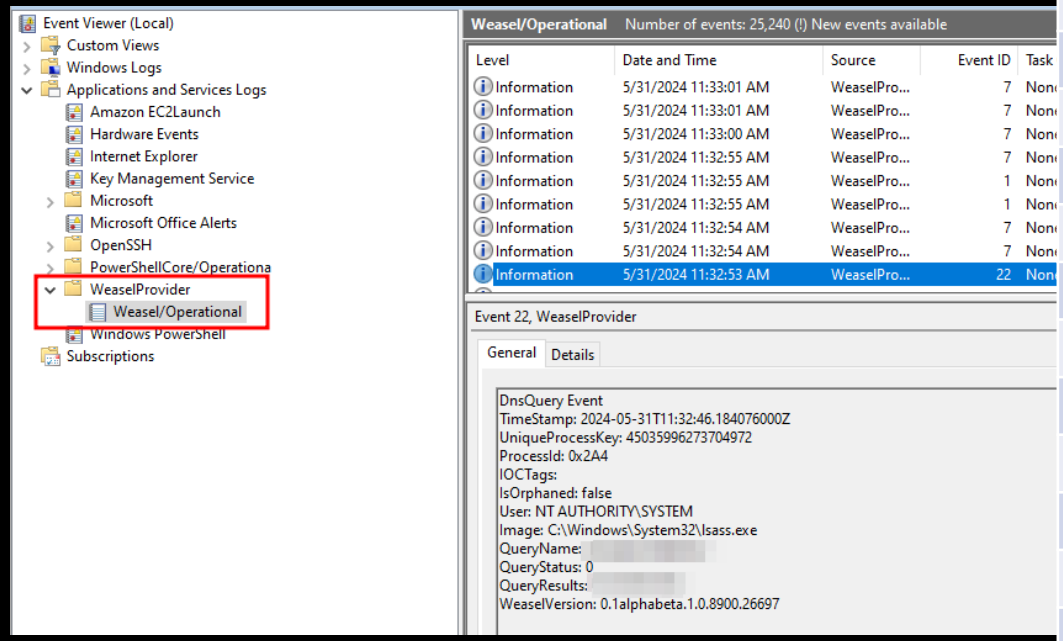
Windows Event And Security Logging

Our New Approach

- Uses SEC provider as base-line of telemetry
  - All other events are enriched with information we get via SEC
- Additional providers are still required
  - Microsoft-Windows-DNS-Client
  - Microsoft-Windows-Kernel-File
  - Microsoft-Windows-TCPIP
  - Microsoft-Windows-WMI-Activity
  - Microsoft-Antimalware-Engine

Weasel

- 22 highly enriched and customizable events
  - Includes all Sysmon-Events
  - Except some registry fields
- Additional Events
- Runs as background service
  - Writes into Windows Event Log
  - Can be forwarded to Splunk
- Event filters configurable via config (JSON)
- More events to be implemented
  - Microsoft-Windows-Crypto-DPAPI



ID	Event
1	ProcessCreation
2	TimeStomp
3	NetworkConnection
5	ProcessStop
6	DriverLoad
7	ImageLoad
8	CreateRemoteThread
9	RawDiskAccess
10	ProcessAccess
11	FileCreate
12	RegCreateDelete
13	RegSet
15	FileStream
17	NamedPipeCreated
18	NamedPipeConnected
21	WMIConsumerBinding
22	DNSQuery
26	FileDelete
108	AMSI
109	RPCClientCall
110	RPCServerCall
400	DotnetAssemblyLoaded

Events

Event 7, WeaselProvider

General Details

ImageLoad Event
TimeStamp: 2024-05-28T13:11:29.951960400Z
UniqueProcessKey: 45035996274053495
ProcessId: 0x1F58
IOCTags: IOCModuleProxying
IsOrphaned: false
User:
Image \WorkItemLoadLibrary.exe
ImageLoaded: C:\Windows\System32\wininet.dll
ImageLoadedOriginalName: wininet.dll
FileVersion: 11.00.20348.1 (WinBuild.160101.0800)
Product: Internet Explorer
Company: Microsoft Corporation
Description: Internet Extensions for Win32
Hashes: SHA1=F0C98FA9E452E450E4AEDC5F526F69BEBC44ACAE,MD5=8745BAEB996212003E4049712BB3638A,SHA256=8417F61976D666821DC9AD4E437A31757F8668F62
SignatureStatus: Valid
Signed: true
Signature: Microsoft Windows
SignatureValid: true
ImageBase: 0x7FF918720000
Motw: false
Calltrace: ntdll.dll+a06d4|ntdll.dll+9ea30|ntdll.dll+33a56|ntdll.dll+3356c|ntdll.dll+3c3e0|ntdll.dll+3c114|ntdll.dll+3b751|ntdll.dll+26df8|ntdll.dll+34f78|ntdll.dll+34946|kerne
|kernel32.dll+14de0|ntdll.dll+7ed9b|
WeaselVersion: 0.1alphabeta.1.0.8900.26697

General Details

Event RPC
UtcTime 2024-01-05 12:29:42.151
ProcessName C:\Windows\System32\mmc.exe
ProcessId 12352
ProcessGuid 9560a0e9-0768-4e92-a050-c21ef02a148e
InterfaceUuid f6beaff7-1e19-4fbb-9f8f-b89e2018337c
ProcNum 9
Protocol 3
Networkaddress NULL
Options NULL
AuthenticationLevel 6
AuthenticationService 20
ImpersionationLevel 3
Calltrace 0x7FFF5E32F954 -> ntdll.dll+f954
0x7FFF5A755258 -> rpcrt4.dll+75258
0x7FFF5A891628 -> rpcrt4.dll+1b1628
0x7FFF5A892FA4 -> rpcrt4.dll+1b2fa4
0x7FFF4EDD4260 -> wevtapi.dll+14260
0x7FFF4EDD5830 -> wevtapi.dll+15830
0x7FFF4EDE4DC4 -> wevtapi.dll+24dc4
0x7FFE92F2BBA4 -> miguicontrols.ni.dll+64bba4
0x7FFE92DE2E30 -> miguicontrols.ni.dll+502e30
0x7FFE92DEA030 -> miguicontrols.ni.dll+50a030
0x7FFE92EE22DC -> miguicontrols.ni.dll+6022dc
0x7FFE92EA6478 -> miguicontrols.ni.dll+5c6478
0x7FFE92EAD0BC -> miguicontrols.ni.dll+5cd0bc
0x7FFF1456D0D0 -> system.windows.forms.ni.dll+d5d0d0
0x7FFF14573514 -> system.windows.forms.ni.dll+d63514
0x7FFF145748B4 -> system.windows.forms.ni.dll+d648b4
0x7FFE92E19A7C -> miguicontrols.ni.dll+539a7c
0x7FFF13ADD4E8 -> system.windows.forms.ni.dll+2cd4e8
0x7FFF142F95FC -> system.windows.forms.ni.dll+ae95fc
0x7FFF176C185C -> clr.dll+185c
0x7FFF5AB62204 -> user32.dll+12204
0x7FFF5AB61AEC -> user32.dll+11aec
0x7FFF3E00A7A0 -> comctl32.dll+a7a0
0x7FFF3E00A540 -> comctl32.dll+a540
0x7FFF3E05D534 -> comctl32.dll+5d534
0x7FFF3E00A7A0 -> comctl32.dll+a7a0
0x7FFF3E00A650 -> comctl32.dll+a650
0x7FFF5AB62204 -> user32.dll+12204

```
index=our_custom_events EventID=3 | head 5 | table Computer Image DestinationIp DestinationHostname CommandLine
```

Last 15 minutes

✓ 5 events (29/05/2024 07:51:28.000 to 29/05/2024 08:06:28.000) No Event Sampling Job Verbose

Events (5) Patterns Statistics (5) Visualization

100 Per Page Format Preview

Computer	Image	DestinationIp	DestinationHostname	CommandLine
CS 1E [redacted]	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.24040.4-0\MsMpEng.exe	20.223.15.161	europe.cp.wd.microsoft.com	"C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.24040.4-0\MsMpEng.exe"

Event Customization

- We are now able to customize our events
  - Easier SPLs
    - Able to include every information we have about the process
    - Parent and grandparent process information in each event
  - More information
    - Add context fields to existing events, e.g. IsManaged for .NET processes
    - Built-In fine-grained Detections (IOCTags)

Config

```
"ImageLoad":
[
    { "IsInclude": true, "Operation":"And", "Comment":"LSASS Suspicious Set of DLLs loaded", "Filters":[
        { "Operation":"Equals", "Field":"Image", "Value":"C:\\Windows\\System32\\lsass.exe" },
        { "Operation":"ContainsAny", "Field":"ImageLoaded", "Value":
    ]},
```

```
{ "IsInclude": true, "Operation":"And", "Comment":"Dumping credentials from services", "Filters":[
    { "Operation":"Equals", "Field":"TargetImage", "Value":"C:\\Windows\\system32\\lsass.exe" },
    { "Operation":"BeginsWith", "Field":"GrantedAccessStr", "Value":  },
    { "Operation":"EndsWith", "Field":"GrantedAccessStr", "Value":  }
]},
```

```
  "RPC":
  [
      { "IsInclude": true, "Operation":"Contains", "Field":"InterfaceUuid", "Value":
      { "IsInclude": true, "Operation":"Contains", "Field":"InterfaceUuid", "Value":
      { "IsInclude": true, "Operation":"Contains", "Field":"InterfaceUuid", "Value":
      { "IsInclude": true, "Operation":"Contains", "Field":"InterfaceUuid", "Value":
      { "IsInclude": true, "Operation":"Contains", "Field":"InterfaceUuid", "Value":
  ],
  "AMSI": [],
  "DotnetAssemblyLoaded":
  [
      { "IsInclude": false, "Operation":"Contains", "Field":"ModuleILPath", "Value":"\\"}
  ]
```

```
"Detections": {
    "ProcessCreate": [],
    "Timestomp": [],
    "NetworkConnect": [],
    "ProcessTerminate": [],
    "DriverLoad": [],
    "ImageLoad": [ "ModuleProxying" ],
    "CreateRemoteThread": [ "ModuleProxying", "DirectSyscall", "IndirectSyscall" ],
    "RawAccessRead": [],
    "ProcessAccess": [ "ModuleProxying", "DirectSyscall", "IndirectSyscall" ],
    "FileCreate": [],
    "RegistryEvent": [],
    "FileCreateStreamHash": [],
    "PipeEvent": [],
    "DNSQuery": [],
    "FileDelete": [],
    "RPC": [],
    "ProcessTampering": []
}
```

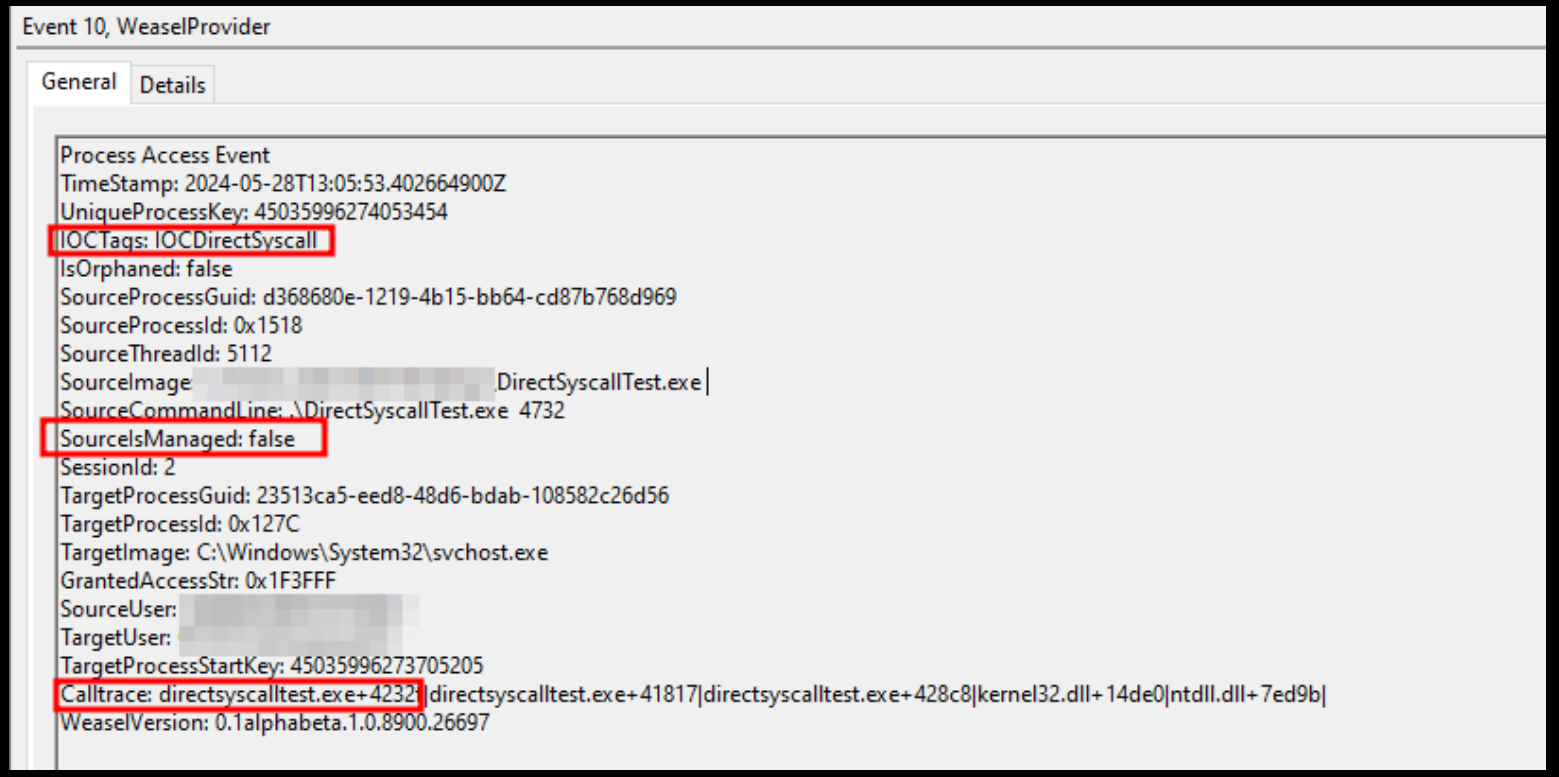
Built-In Detections

Custom Detections

- Many offensive techniques can barely be detected using splunk rules alone
  - They are too dependend on the endpoint (e.g. offsets in ntdll)
  - IOC footprint too small, need to sum up smaller IOCs
  - Requires corelation of a lot of events on splunk (= huge load)
- As we have a custom sensor we can detect and corelate on the endpoint itself
  - CallStack analysis done on the endpoint
- Example: ProcessAccess. Happens quite a lot even with process_all_access
  - Can only store a limited amount of events
  - **Some evasion techniques introduce unnecessary IOCs**

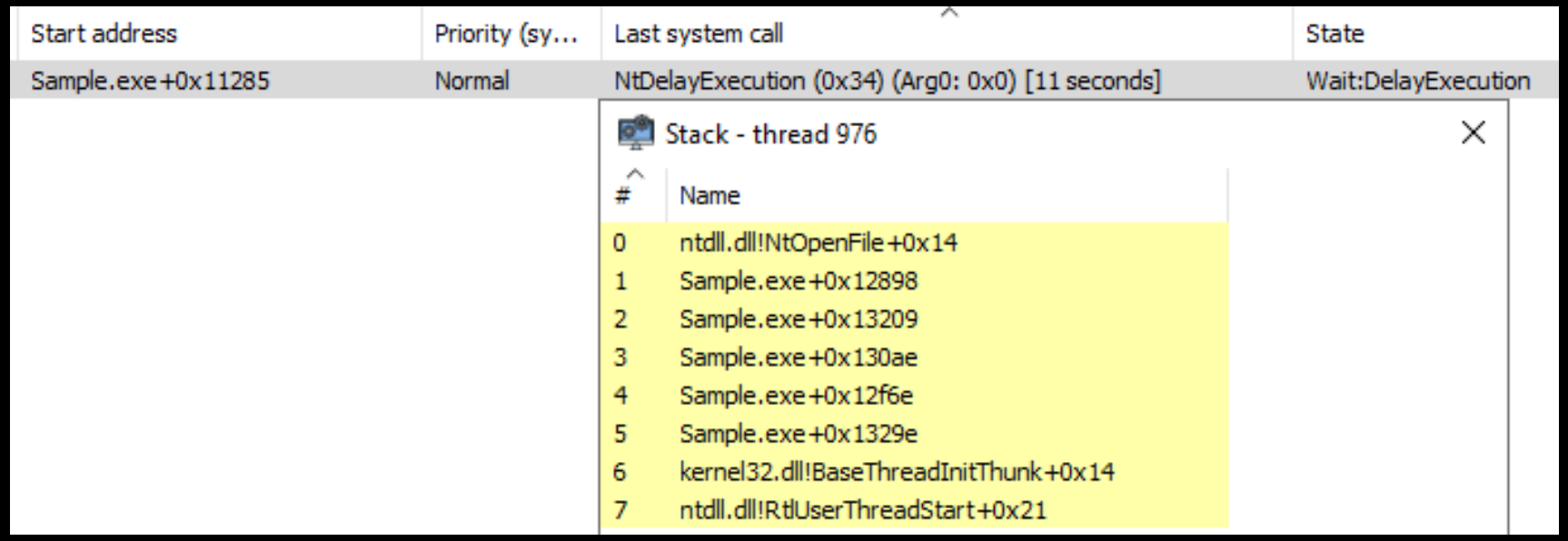
Direct Syscall Detection

- Very simple
- Last module in calltrace != ntdll
- IOCTag: IOCDirectSyscall



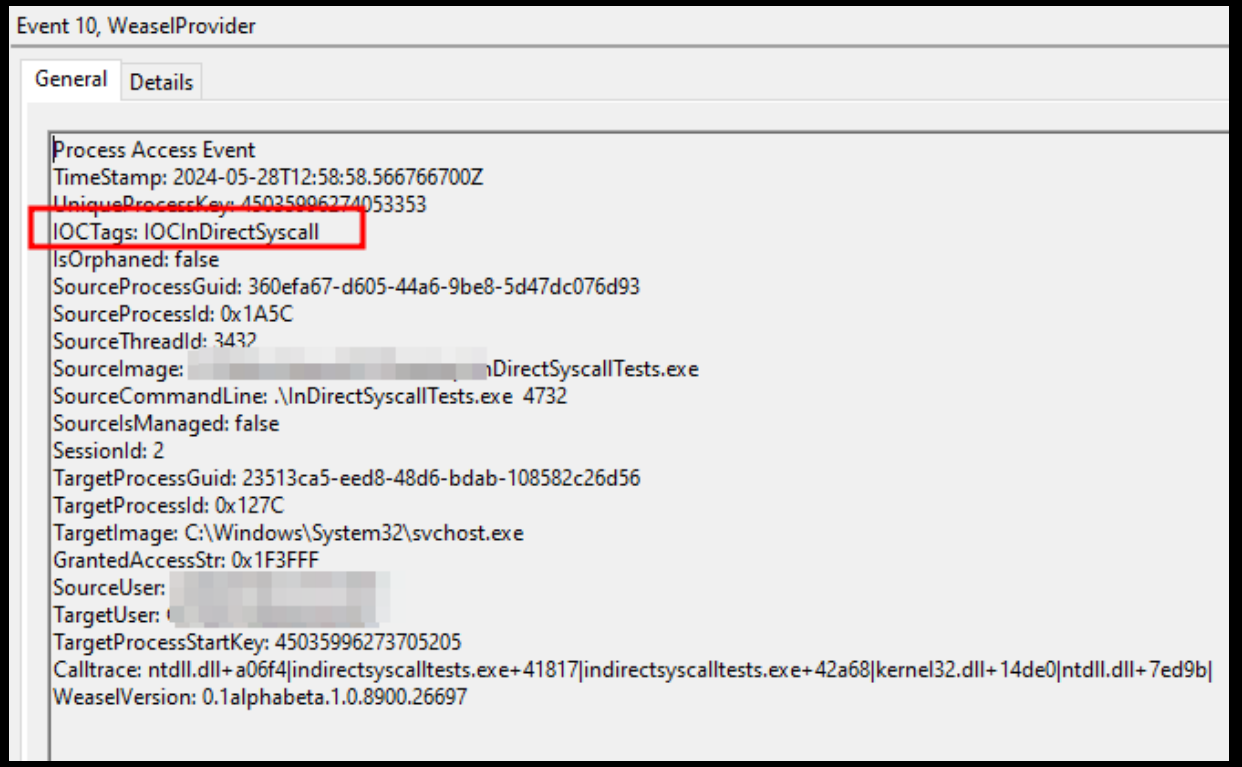
InDirect Syscall Detection

- Improvement for direct syscalls
- InDirect Syscalls abuse that not every syscall stub in ntdll is hooked
  - All implementations use a clean syscall stub but with a non-expected syscall number
- IOC: Syscall number does not match the syscall stub



InDirect Syscall Detection

- We build a static list of syscall stub offsets in ntdll
  - Stubs whose related syscalls are expected in CallStack
  - NtOpenProcess, NtDuplicateObject, NtAlpcOpenSenderProcess …
  - NtCreateThread …
- Check if expected stub appears in CallStack
- IOCTag: IOCInDirectSyscall
- Please don’t do this syscall stuff anymore
  - Especially if the EDR does not hook anything
- Bypass is obvious tho (:D)

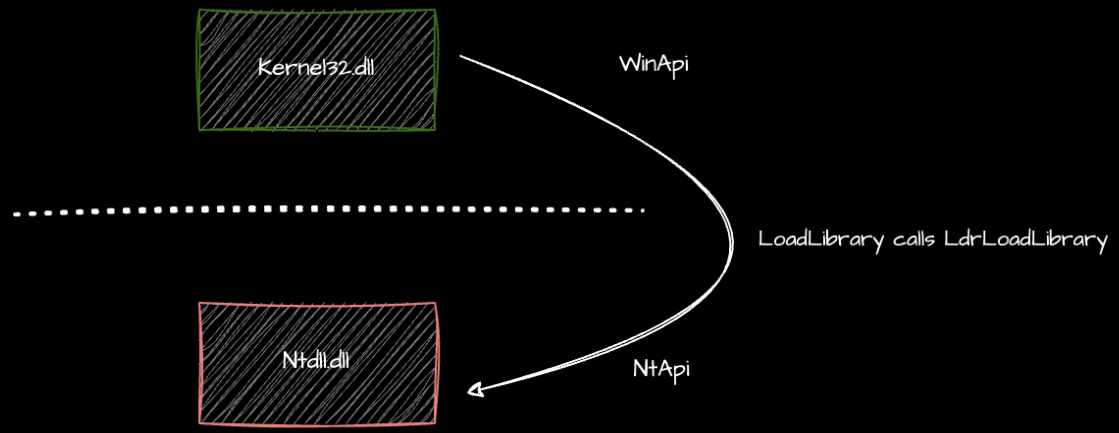


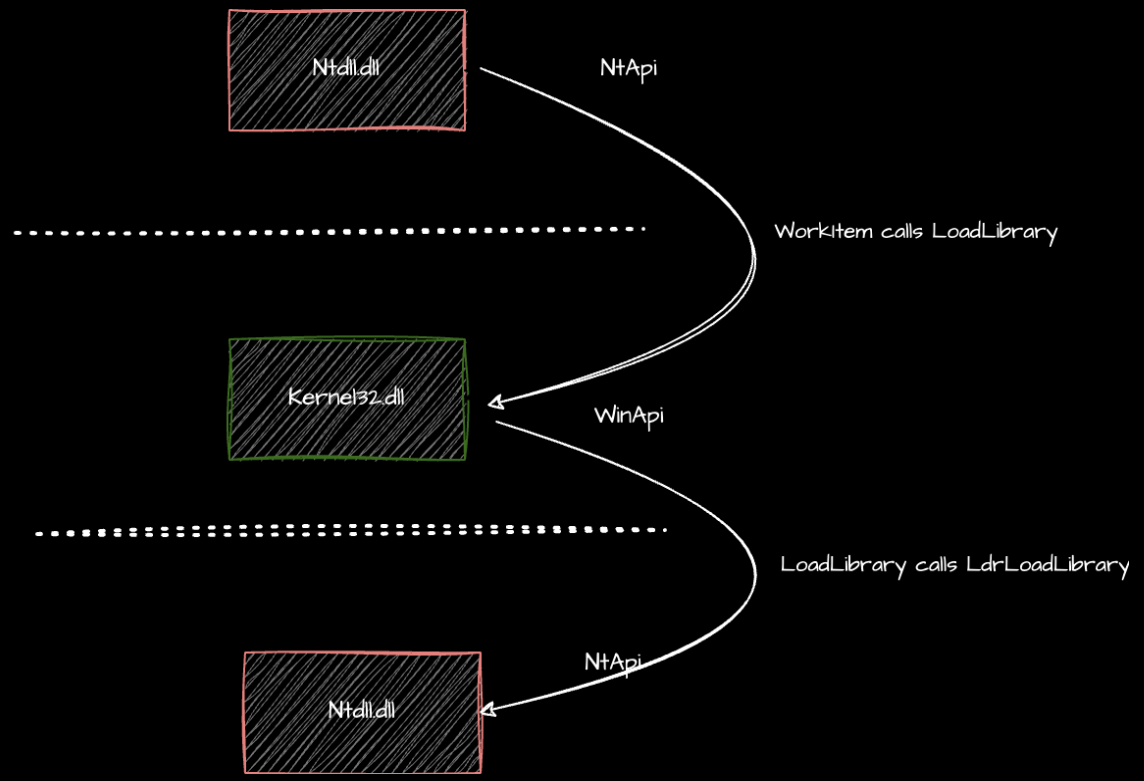
Suspicious ImageLoad Events

- Some DLLs are often loaded by C2 beacons: wininet, netapi, dpapi …
- A private page in a CallStack to kernel32!LoadLibrary might be an IOC
- Bypass: Proxy the call to kernel32!LoadLibrary through Ntdll
  - Using Workerthreads (ThreadPool)
  - A separate Workerthread will pick up the item and load the Dll into the process
  - Produces a clean CallStack without suspicious pages
    - But is it really clean!?

```
HMODULE MyLoadLibrary ( PCSTR mName ) {

        HANDLE hThread = NULL;
        RtlQueueWorkItem _RtlQueueWorkItem = ( RtlQueueWorkItem ) GetProcAddress ( GetModuleHandleA ( "ntdll.dll" ), "RtlQueueWorkItem" );
        _RtlQueueWorkItem ( LoadLibraryA, ( PVOID ) mName, WT_EXECUTEDEFAULT );

        Sleep ( 1000 ); // Dirty :-)

        return GetModuleHandleA ( mName );

}
```

Suspicious ImageLoad Events





Suspicious ImageLoad Events

```
1   ntdll.dll+a06d4
2   ntdll.dll+9ea30
3   ntdll.dll+33a56
4   ntdll.dll+3356c
5   ntdll.dll+3c3e0
6   ntdll.dll+3b8f8
7   ntdll.dll+3b677
8   ntdll.dll+26df8
9   ntdll.dll+34f78
10  ntdll.dll+34946
11  kernelbase.dll+43fb2
12  uxtheme.dll+206c9
13  uxtheme.dll+1760e
14  uxtheme.dll+1dd26
15  uxtheme.dll+1dbf8
16  uxtheme.dll+144d1
17  user32.dll+101af
18  user32.dll+f5ac
19  user32.dll+f30f
20  uxtheme.dll+21ba4
21  conhost.exe+6619
22  conhost.exe+3f37
23  conhost.exe+3439
24  conhost.exe+3258
25  conhost.exe+30f5
26  kernel32.dll+14de0
27  ntdll.dll+7ed9b
```

Call into LoadLibrary

Thread Initialization

```
ntdll.dll+a06d4
ntdll.dll+9ea30
ntdll.dll+33a56
ntdll.dll+3356c
ntdll.dll+3c3e0
ntdll.dll+3c114
ntdll.dll+3b751
ntdll.dll+26df8
ntdll.dll+34f78
ntdll.dll+34946
kernelbase.dll+43fb2
kernelbase.dll+4c9f1
kernelbase.dll+8cb8f
ntdll.dll+5522
ntdll.dll+bb26
kernel32.dll+14de0
ntdll.dll+7ed9b
```

Again NTAPI

LoadLibrary

NTAPI Calling WinApi!?

Thread Initialization

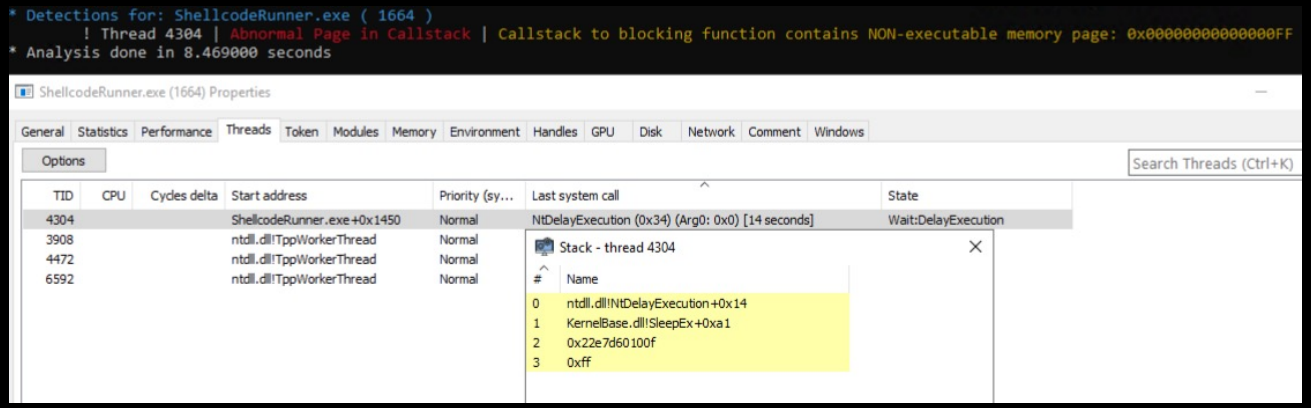
Detection ModuleProxying

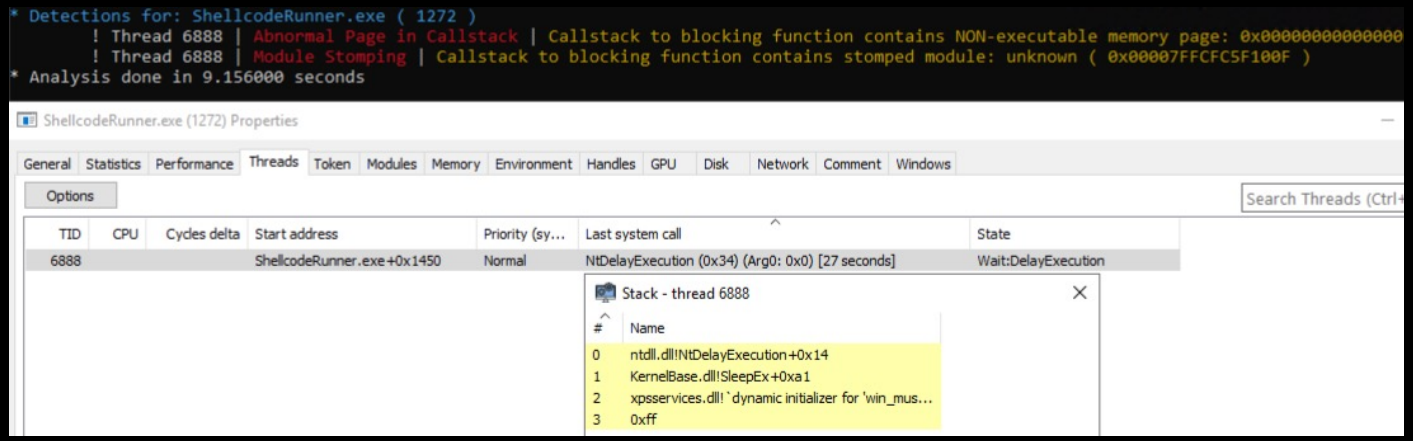
Process Rating

- Observe process behaviour
  - Talks to the Internet, loads certain dlls …
  - Threshold hit? Report process
- Additionally processes are periodically scanned for IOCs
  - Abnormal memory allocations or thread states
- HTTP based beacons wait between their callbacks
  - Idea: Enumerate all idling threads and check their CallStack to the blocking function
  - A finding of this scan is added to the process rating

Beacon Detection

- Thread is idling
- CallStack contains private r(w)x
- Severity: Medium



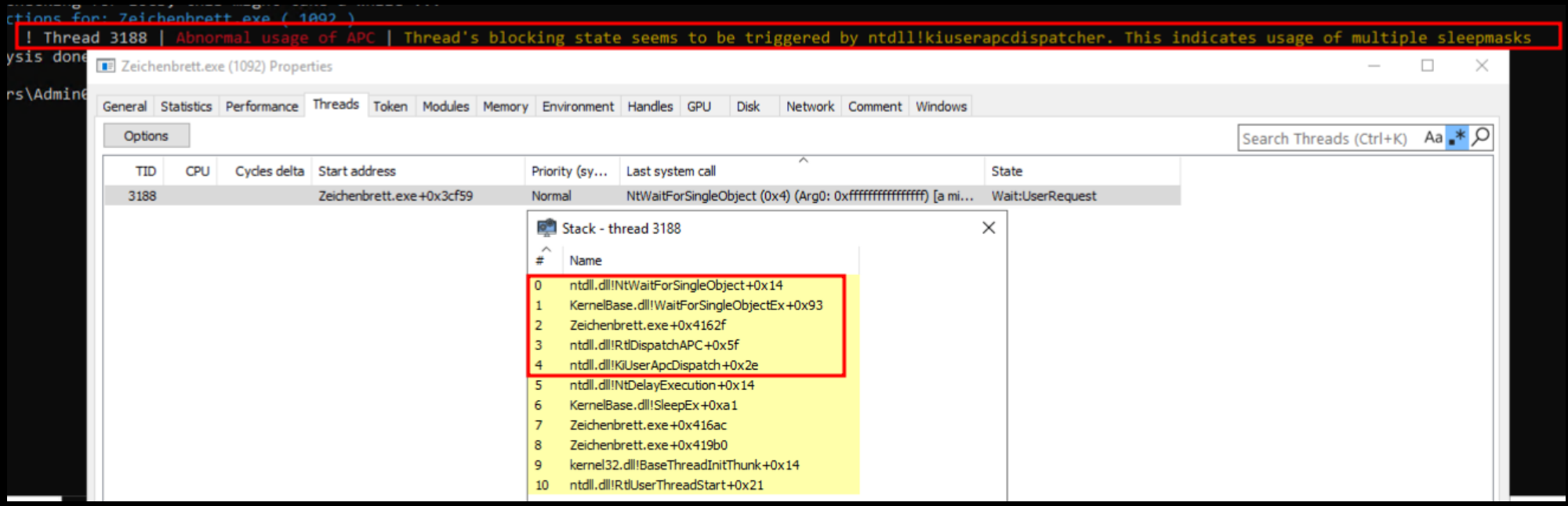


- Thread is idling
- CallStack contains stomped module
- Severity: Low

MEMORY_WORKING_SET_EX_INFORMATION. u1.VirtualAttributes.SharedOriginal == 0

Sleepmask Detection (APC)

- Some SleepMask implementation trigger a sequence of APCs
  - One of which calls a blocking function
- Ntdll!KiUserAPCDispatcher in callstack to blocking function
  - Severity: High

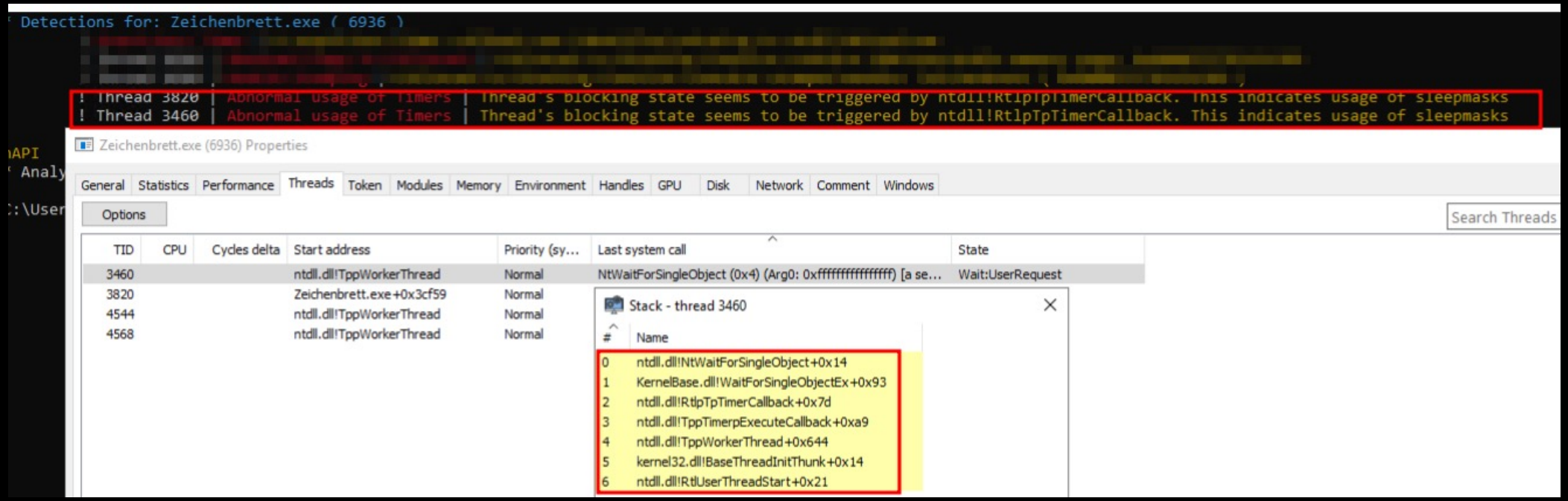


Sleepmask Detection (Timer)

- Some SleepMask implementation trigger a sequence of Timers
  - One of which calls a blocking function
  - A routine called by a waitable timer should return somewhat quickly

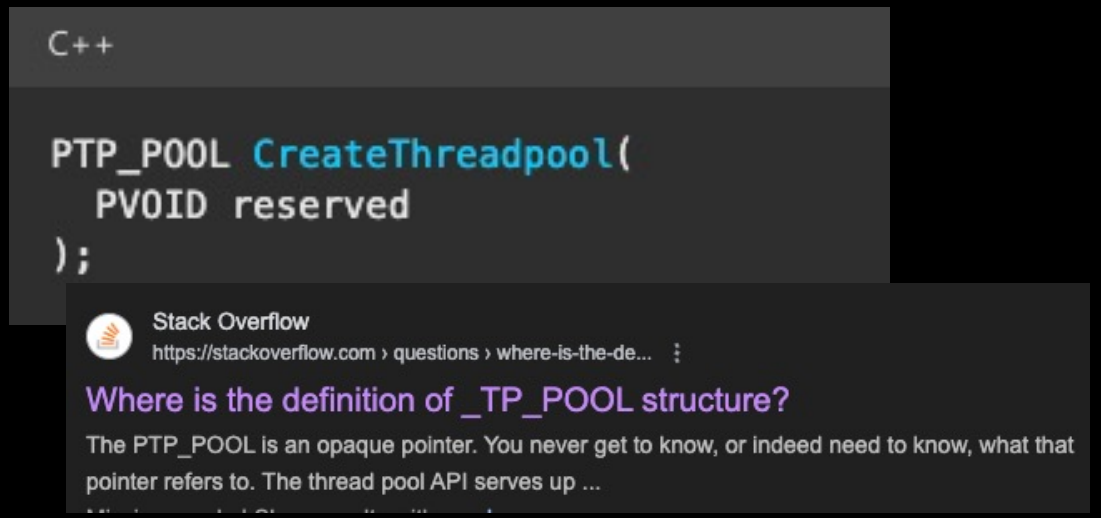
WT_EXECUTEINTIMERTHREAD 0x00000020	The callback function is invoked by the timer thread itself. This flag should be used only for short tasks or it could affect other timer operations.

- RtlpTpTimerCallback in CallStack to WaitForSingleObject
  - Severity: High



Sleepmask Detection (Timer)

- The idea to check for RtlpTpTimerCallback on the callstack is not ideal
- Does not find pending timers
  - Big dection gap
  - CobaltStrikes CallStackMasker bypasses this (https://github.com/Cobalt-Strike/CallStackMasker)
- Need to enumerate all pending timers and their callbacks
- Difficult task
- Timer internals are barely documented
  - Built on top of threadpools
  - Threadpool-internals also are not officially documented



thanks, google

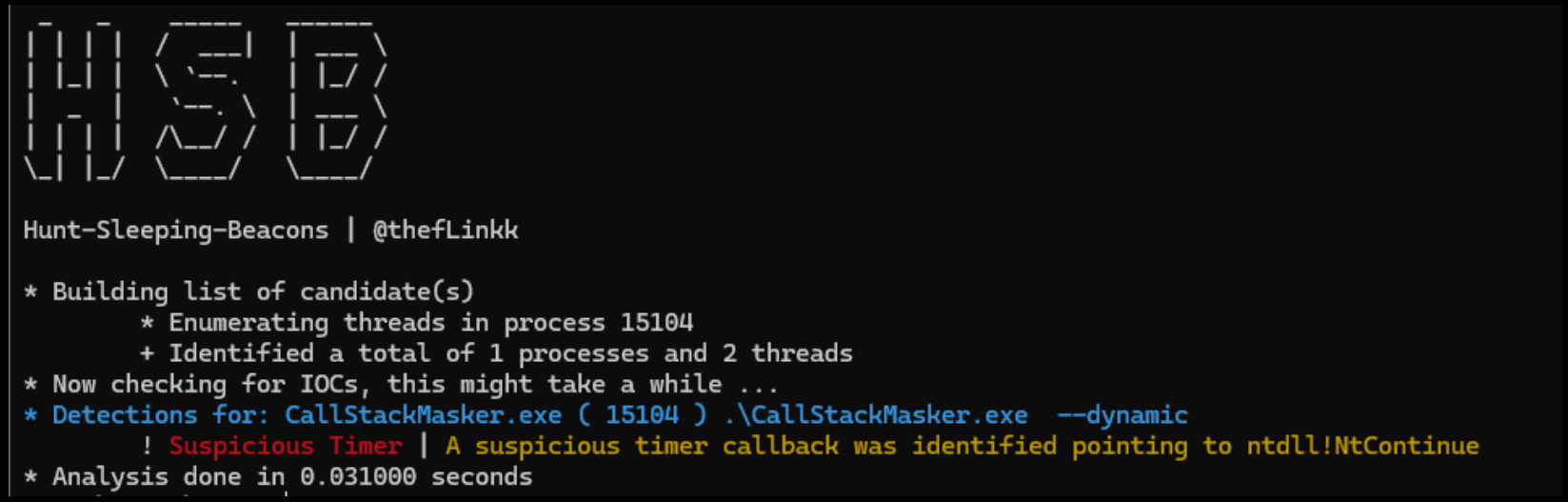
Sleepmask Detection (Timer)

- Structs undocumented but released by SafeBreach-Labs: ☺ ☺
  https://github.com/SafeBreach-Labs/PoolParty
- Enumerate all WorkerFactories and query: `NtQueryInformationWorkerFactory`
  - `WORKER_FACTORY_BASIC_INFORMATION -> FULL_TP_POOL-> TimerQueue`
  - Timer.Work.CleanupGroupMember.Context
  - `Contains FinalizationCallback` ¯_(ツ)_/¯
- Suspicious Callbacks:
  - NtContinue
  - RtlCaptureContext
  - RtlCopyMemory
  - …

```
bSuccess = EnumTools::GetHandlesOfTypeInProcess(pProcess, L"TpWorkerFactory", WORKER_FACTORY_ALL_ACCESS, workerFactories);
if (bSuccess == FALSE)
    goto Cleanup;

for (HANDLE hWorkerFactory : workerFactories) {

    if (NtQueryInformationWorkerFactory(hWorkerFactory, WorkerFactoryBasicInformation, &wfbi, sizeof(WORKER_FACTORY_BASIC_INFORMATION), NULL) == STATUS_SUCCESS) {

        bSuccess = ReadProcessMemory(pProcess->hProcess, wfbi.StartParameter, &full_tp_pool, sizeof(FULL_TP_POOL), &len);
        if (bSuccess == FALSE)
            continue;

        if (full_tp_pool.TimerQueue.RelativeQueue.WindowStart.Root)
            p_tp_timer = CONTAINING_RECORD(full_tp_pool.TimerQueue.RelativeQueue.WindowStart.Root, FULL_TP_TIMER, WindowStartLinks);
        else if (full_tp_pool.TimerQueue.AbsoluteQueue.WindowStart.Root)
            p_tp_timer = CONTAINING_RECORD(full_tp_pool.TimerQueue.AbsoluteQueue.WindowStart.Root, FULL_TP_TIMER, WindowStartLinks);
        else
            continue;


        bSuccess = ReadProcessMemory(pProcess->hProcess, p_tp_timer, &tp_timer, sizeof(FULL_TP_TIMER), &len);
        if (bSuccess == FALSE)
            continue;

        PLIST_ENTRY pHead = tp_timer.WindowStartLinks.Children.Flink;
        PLIST_ENTRY pFwd = tp_timer.WindowStartLinks.Children.Flink;
        LIST_ENTRY entry = { 0 };

        do {

            bSuccess = ReadProcessMemory(pProcess->hProcess, tp_timer.Work.CleanupGroupMember.Context, &ctx, sizeof(TPP_CLEANUP_GROUP_MEMBER), &len);
            if (bSuccess == FALSE)
                break;

            for (SUSPICIOUS_CALLBACK suspiciousCallback : this->SuspiciousCallbacks) {
                if (suspiciousCallback.addr == ctx.FinalizationCallback) {

                    wsprintfA(message, "A suspicious timer callback was identified pointing to %s", suspiciousCallback.name.c_str());
```

Callstack Spoofing Detection

- Now able to enumerate pending timers and their callbacks
- Currently finds most implementations of timer-based Sleepmasks
  - Depending on which function the callback executes.
- Severity: Critical



Callstack Spoofing Detection

- CallStacks are very valuable for defenders
- No surprise: attackers are trying to spoof stacks
  - https://github.com/klezVirus/SilentMoonwalk
- Most of them make use of a ROP-Gadget jmp [non-volatile register]

```
BYTE patternJmpDerefRbx [ 2 ] = { 0xFF, 0x23 };
BYTE patternJmpDerefRbp [ 3 ] = { 0xFF, 0x65, 0x00 };
BYTE patternJmpDerefRdi [ 2 ] = { 0xFF, 0x27 };
BYTE patternJmpDerefRsi [ 2 ] = { 0xFF, 0x26 };
BYTE patternJmpDerefR12 [ 4 ] = { 0x41, 0xff, 0x24, 0x24 };
BYTE patternJmpDerefR13 [ 4 ] = { 0x41, 0xff, 0x65, 0x00 };
BYTE patternJmpDerefR14 [ 3 ] = { 0x41, 0xff, 0x26 };
BYTE patternJmpDerefR15 [ 3 ] = { 0x41, 0xff, 0x27 };

for ( int i = 0; i < pCandidate->calltrace->size ( ); i++ ) {

    bSuccess = ReadProcessMemory ( hProcess, ( PVOID ) pCandidate->calltrace->at ( i ), instructions, sizeof ( instructions ), &nRead );
    if ( bSuccess == FALSE )
        goto Cleanup;

    if ( memcmp ( instructions, patternJmpDerefRbx, sizeof ( patternJmpDerefRbx ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefRbp, sizeof ( patternJmpDerefRbp ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefRdi, sizeof ( patternJmpDerefRdi ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefRsi, sizeof ( patternJmpDerefRsi ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefR12, sizeof ( patternJmpDerefR12 ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefR13, sizeof ( patternJmpDerefR13 ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefR14, sizeof ( patternJmpDerefR14 ) ) == 0 )
        bSuspicious = TRUE;
    else if ( memcmp ( instructions, patternJmpDerefR15, sizeof ( patternJmpDerefR15 ) ) == 0 )
        bSuspicious = TRUE;
```

```
 _   _    _____   ______
| | | |  /  ___|  | ___ \
| |_| |  \ `--.   | |_/ /
|  _  |   `--. \  | ___ \
| | | |  /\__/ /  | |_/ /
\_| |_/  \____/   \____/

Hunt-Sleeping-Beacons | @thefLinkk

* Building list of candidate(s)
        * Enumerating threads in process 11024
        + Identified a total of 1 processes and 6 threads
* Now checking for IOCs, this might take a while ...
        * Including stackspoofing detections
* Detections for: SilentMoonwalk.exe ( 11024 )
        ! Thread 14192 | Return Address Spoofing | Thread continues to JMP gadget after delay. Gadget in: KERNELBASE!CreateFileW
* Analysis done in 0.047000 seconds
```

Putting It All Together

```
 _   _   _____   _____
| | | | /  ___| |  _  \
| |_| | \ `--.  | |_/ /
|  _  |  `--. \ |  __ \
| | | | /\__/ / | |_/ /
\_| |_/ \____/  \____/

Hunt-Sleeping-Beacons | @thefLinkk

* Building list of candidate(s)
        * Enumerating processes and threads ( ignoring Dotnet and 32Bit processes ). This might take a while ...
        + Identified a total of 23 processes and 199 threads
* Now checking for IOCs, this might take a while ...
* Detections for: Ekko.exe ( 4696 )
        ! Suspicious Timer | A suspicious timer callback was identified pointing to ntdll!NtContinue
        ! Thread 5164 | Abnormal Page in Callstack | Callstack to blocking function contains NON-executable memory page: 0x00007FF6A71B1CBD
        ! Thread 5164 | Module Stomping | Callstack to blocking function contains stomped module: Ekko ( 0x00007FF6A71B1CBD )
        ! Thread 5164 | Abnormal usage of Timers | Thread's blocking state seems to be triggered by ntdll!RtlpTpTimerCallback. This indicates usage of sleepmasks
        ! Thread 2228 | Abnormal usage of Timers | Thread's blocking state seems to be triggered by ntdll!RtlpTpTimerCallback. This indicates usage of sleepmasks
        ! Thread 2228 | Abnormal intermodular call | ntdll!RtlAddRefActivationContext called KERNELBASE!WaitForSingleObjectEx, this indicates module proxying. NtAPI Should not call WinAPI
* Analysis done in 7.469000 seconds
```

Standalone Scanner:
https://github.com/thefLink/Hunt-Sleeping-Beacons

Summary

Summary and Future Work

- Relying on third party sensors is challenging for defenders
  - No customization, no flexibility
- Well-Known ETW-Providers not always suitable for Threat-Hunting
- We found a way to use raw Sense-Telemetry
  - SEC-Provider (kernel-based), requires MDE onboarded device
- Custom sensor: Weasel based on SEC-Provider
  - Events customizable, Fine-Grained detection mechanisms
  - Built-In scanner to detect C2 agents
- Future work:
  - Protecting our ETW-session(s) and sensor itself
  - Implement more events

Thänk you for travelling with WEASEL