

# ExaTrack

Sysmon Internals

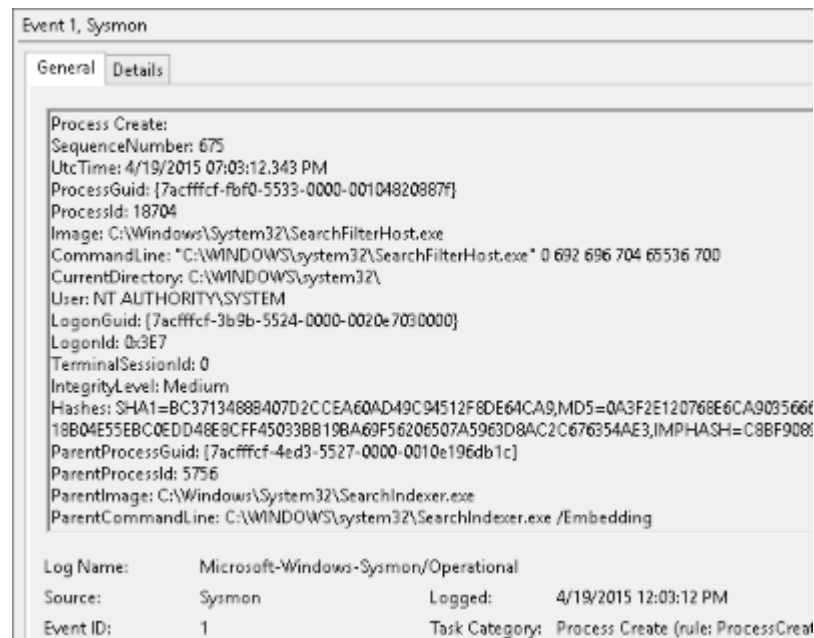


# Objectives of the talk

- What is Sysmon?
- How Sysmon works?
- What Sysmon see?
- Suspicious activities identifications!

## What is Sysmon ?

- Sysmon is a monitoring tool of « SysInternals » suite
- It grab a lot of operations onto the system and log them into « Event logs » of Windows
- A set of XML rules can be edited to have a more specific logging



# What intercept Sysmon ?

- Events traced by Sysmon :
  - Process Create
  - File creation time
  - Network connections
  - Sysmon service state change (cannot be filtered)
  - Process terminated
  - Driver Loaded
  - Image loaded
  - CreateRemoteThread
  - RawAccessRead
  - Process accessed
  - File created
  - Registry object added, deleted, value set, object renamed
  - File stream created
  - Sysmon configuration change (cannot be filtered)
  - Named pipe created, connected
  - WMI Events
  - DNS query

## XML filters

- Sysmon have a logical tree to take decision of logging or not
- If event == ProcessCreate and ("timeout.exe" in Image) and ("100" in CommandLine))

```
<EventFiltering>
  <RuleGroup name="group 1" groupRelation="and">
    <ProcessCreate onmatch="include">
      <Image condition="contains">timeout.exe</Image>
      <CommandLine condition="contains">100</CommandLine>
    </ProcessCreate>
  </RuleGroup>
  <RuleGroup groupRelation="or">
    <ProcessTerminate onmatch="include">
      <Image condition="contains">timeout.exe</Image>
      <Image condition="contains">ping.exe</Image>
    </ProcessTerminate>
  </RuleGroup>
  <ImageLoad onmatch="include"/>
</EventFiltering>
```

# Sysmon installation

- 2 files are dropped on the disk :
  - C:\Windows\Sysmon.exe
  - C:\Windows\SysmonDrv.sys
- 1 service is installed, it run « Sysmon.exe » when the system have booted (late loading)
- Microsoft have produce some documentations to deploy Sysmon by GPO
  - Objective : Each computer have his Sysmon running ♥

## Some Windows kernel bases

- Executables run into User land
- Drivers run into Kernel land
- To discuss with drivers to use NtDeviceIoControl
- Some actions are easier in kernel land and some other actions are easier (or impossible) in userland



Sysmon.exe

SysmonDrv.sys

## Some Windows kernel bases

- Windows kernel able to notify some action to other drivers
- This is a « Callback » and it's really usefull to do complicated action without modifying the Windows kernel
- For example, when you run an executable an AV can analyze it and block this execution, those actions are done with this process

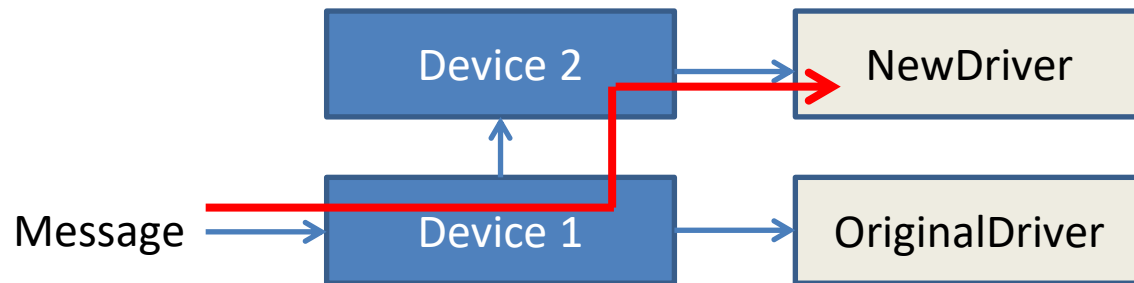


## Some Windows kernel bases

- When a driver is loaded a table is affected to the module (1 table per driver)
- This table is « empty » and can be partialy or totally overwrite to handle some actions
- The table name is « IRP Table »
- For exemple to handle a « read » on our driver object we need to have set the « IRP\_MJ\_READ » entry

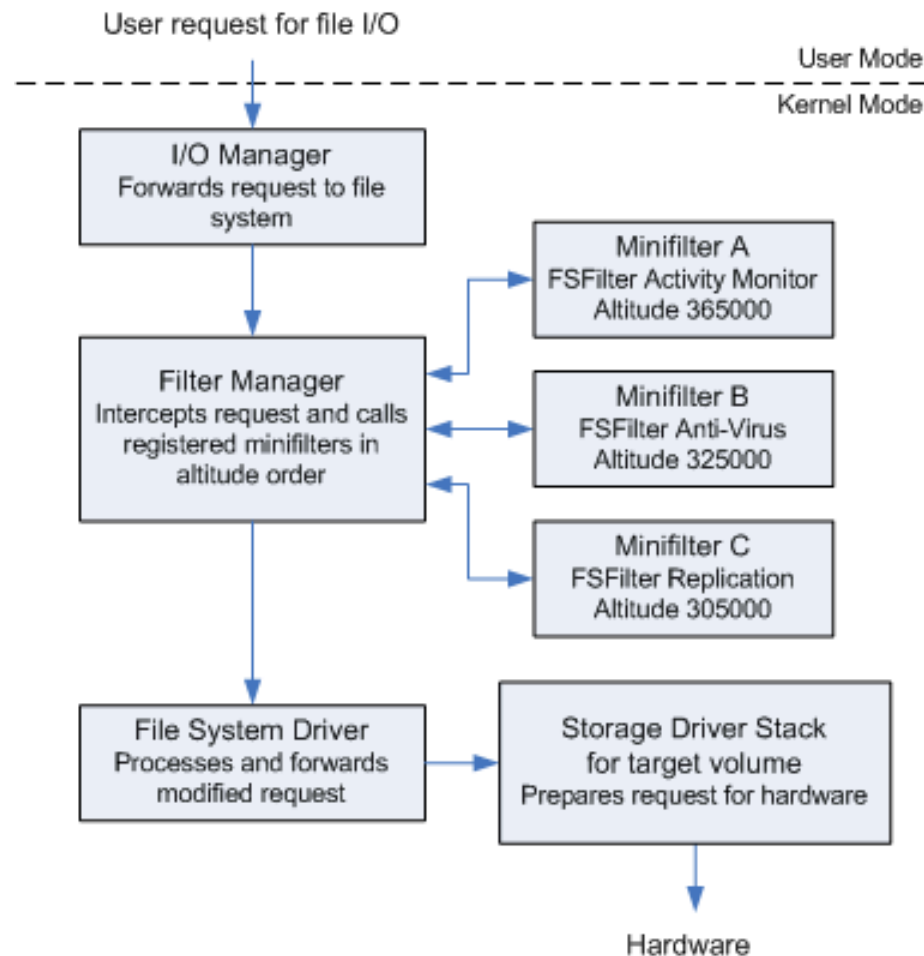
## Some Windows kernel bases

- To communicate with a driver we need to open a handle on a device
- Almost devices of drivers are in \Device
- When we send a message to a device it can be followed to an upper driver (this is the device stack)



## Some Windows kernel bases

- FltMgr : Windows filters manager (or MiniFilters)

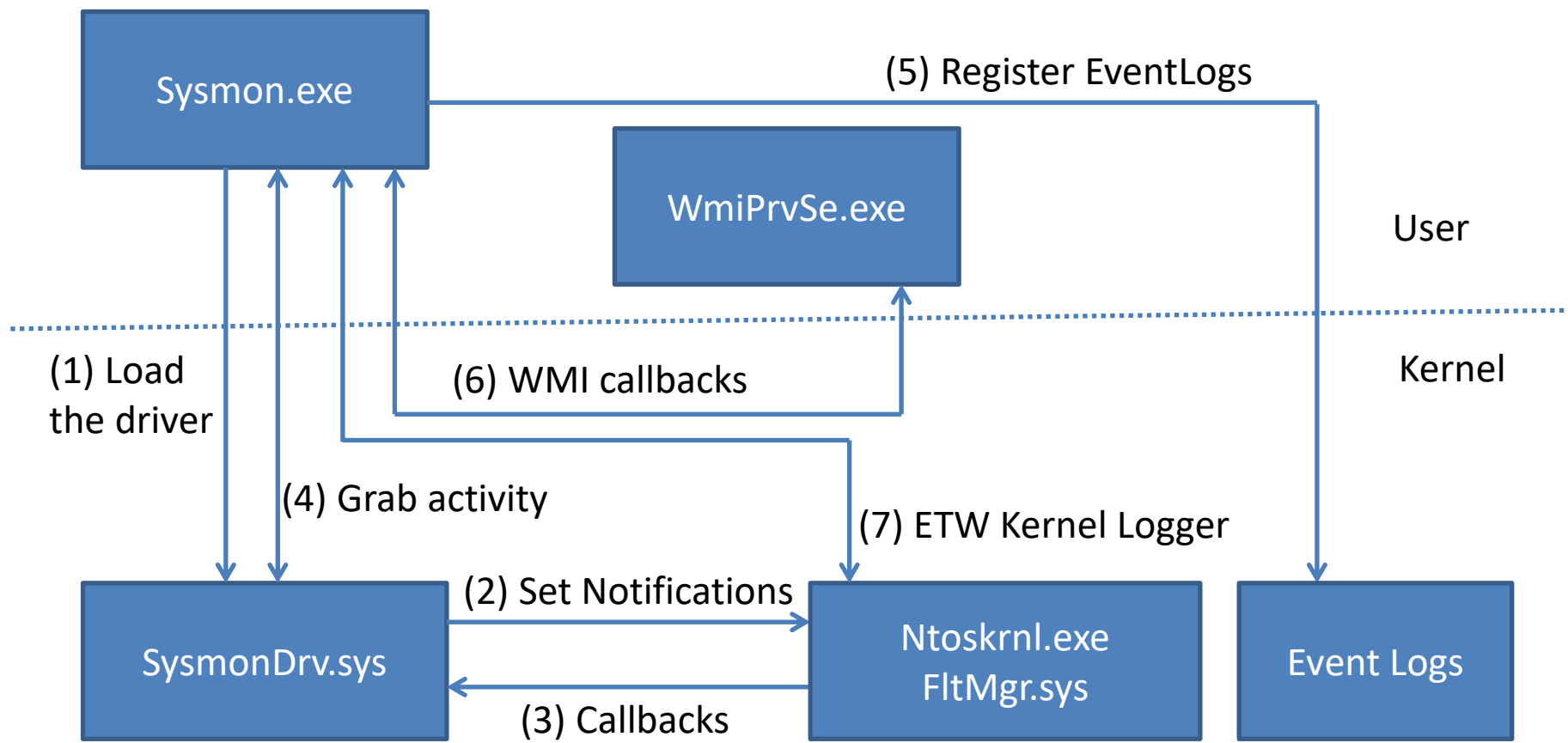


## Some Windows kernel bases

- FltMgr filter can be applied on resources accesses
- There are 2 operations :
  - PRE-OPERATION
  - POST-OPERATION

## Sysmon Global Architecture

- Sysmon.exe continually request the driver to get state of events generated.



# Open Sysmon handle

- Basically I try this :

```
>>> open(r"\\.\SysmonDrv", 'rb')
```

```
IOError: [Errno 13] Permission denied: '\\\\.\SysmonDrv'
```

- ACL Problem ?

```
->Dacl      : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl      : ->Ace[0]: ->AceFlags: 0x0
->Dacl      : ->Ace[0]: ->AceSize: 0x14
->Dacl      : ->Ace[0]: ->Mask : 0x001201bf
->Dacl      : ->Ace[0]: ->SID: S-1-1-0 (Well Known Group:
localhost\Every Body)
```

## Open Sysmon handle

```
wchar_t local_228 [264];
ulonglong local_18;

local_18 = DAT_1400f4560 ^ (ulonglong)&stack0xffffffffffffd88;
vswprintf(local_228,0x104,L"\\\\\\.\\%s",PTR_u_SysmonDrv_1400f5c48);
hDevice =
CreateFileW(local_228,0xc0000000,0,(LPSECURITY_ATTRIBUTES)0x0,3,0x40000080,(HANDLE)0x0);
pvVar1 = DAT_1400fd630;
if (hDevice != (HANDLE)0xffffffffffffffff) {
    local_238 = 800;
    BVar2 = DeviceIoControl(hDevice,0x83400000,&local_238,4,(LPVOID)0x0,0,local_234,
        (LPOVERLAPPED)0x0);

    pvVar1 = hDevice;
    if (BVar2 == 0) {
        CloseHandle(hDevice);
        pvVar1 = DAT_1400fd630;
    }
}
```

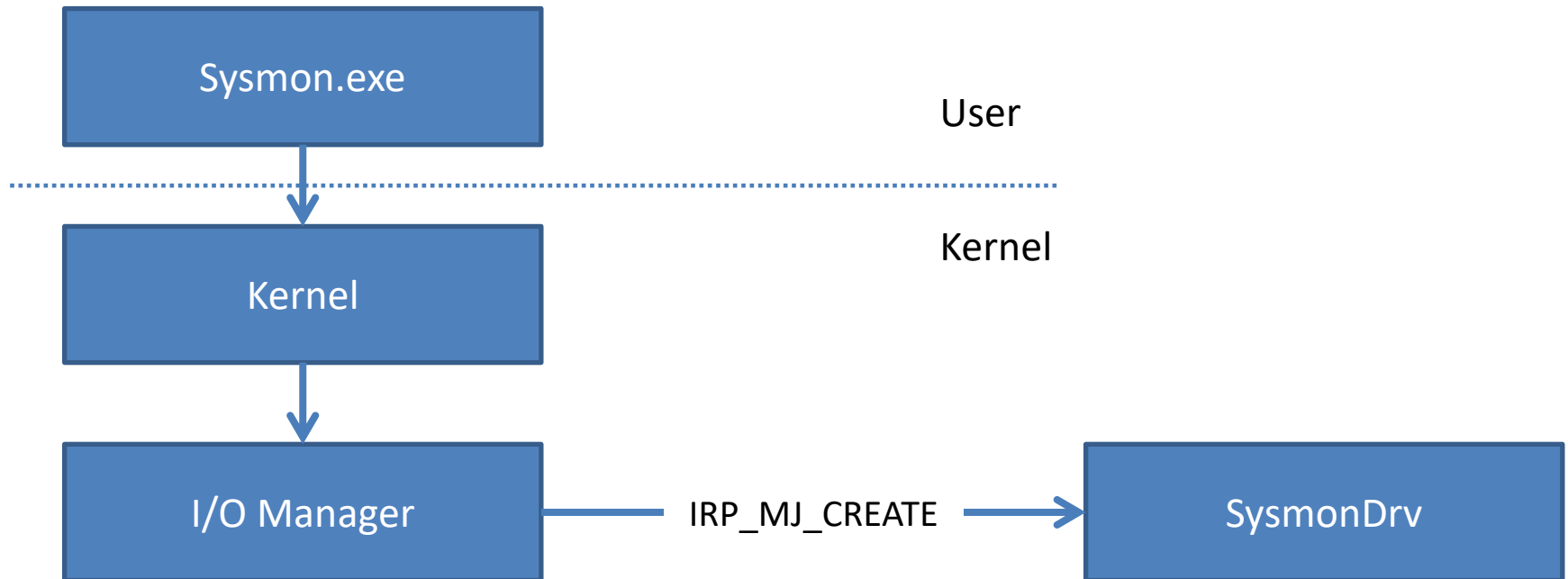
- Try with same flags :

```
>>> windows.winproxy.CreateFileA(r"\\.\SysmonDrv",
0xC0000000, 0, None, 3, 0x40000080, 0)
```

**CreateFileA: [Error 5] Access Deny.**

## Open Sysmon handle

- When you open a handle on a file, Windows kernel send an IRP request « IRP\_MJ\_CREATE » to the driver linked to the device





# Open Sysmon handle

- The dispatch table is :

Dispatch routines:

<b>[00] IRP_MJ_CREATE</b>	<b>ffffff80bc0468d40</b>	<b>SysmonDrv+0x8d40</b>
[01] IRP_MJ_CREATE_NAMED_PIPE	ffffff803da528ed0	
nt!IopInvalidDeviceRequest		
<b>[02] IRP_MJ_CLOSE</b>	<b>ffffff80bc0468d40</b>	<b>SysmonDrv+0x8d40</b>
[03] IRP_MJ_READ	ffffff803da528ed0	
nt!IopInvalidDeviceRequest		
[...]		
[0d] IRP_MJ_FILE_SYSTEM_CONTROL	ffffff803da528ed0	
nt!IopInvalidDeviceRequest		
<b>[0e] IRP_MJ_DEVICE_CONTROL</b>	<b>ffffff80bc0468d40</b>	<b>SysmonDrv+0x8d40</b>
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL	ffffff803da528ed0	
nt!IopInvalidDeviceRequest		

- Function at SysmonDrv+0x8d40 receive ALL requests for CREATE/CLOSE/DEVICE\_CONTROL

## Open Sysmon handle

```
pcVar1 = *(char **)(lParm2 + 0xb8);
cVar3 = *pcVar1;
if (cVar3 == 0) {
    uStack104 = 0x14;
    uStack64 = 1;
    uStack60 = 1;
    uStack48 = 0;
    uStack56 = 0x14;
    SeCaptureSubjectContext (auStack96);
    bVar2 = ExGetPreviousMode();
    cVar3 = SePrivilegeCheck (&uStack64, auStack96, (ulonglong)bVar2);
    iVar7 = 0;
    if (cVar3 == 0) {
        iVar7 = -0x3ffffffde;
    }
    SeReleaseSubjectContext (auStack96);
}
```

→ ID of the function in the IRP\_MJ (0 -> CREATE)

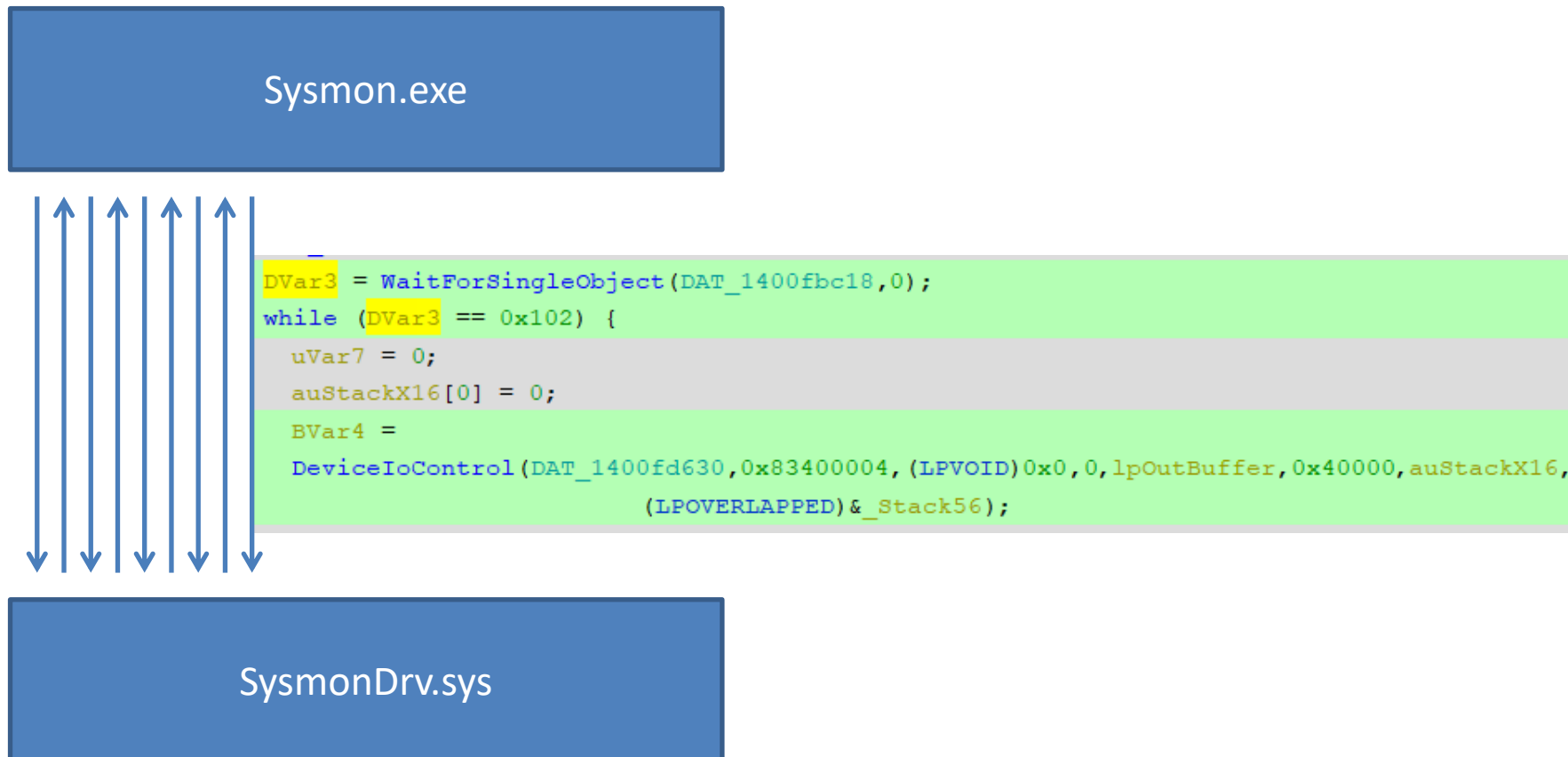
→ ID of « SeDebugPrivilege »

- Try with the privilege :

```
>>> windows.utils.enable_privilege("SeDebugPrivilege", True)
>>> windows.winproxy.CreateFileA(r"\\.\SysmonDrv", 0xC0000000, 0,
None, 3, 0x40000080, 0)
```

# Sysmon events pull

- Sysmon.exe continually request the driver to get state of events generated.



# Sysmon.exe

```
DVar3 = WaitForSingleObject(DAT_1400fbc18,0);
while (DVar3 == 0x102) {
    uVar7 = 0;
    auStackX16[0] = 0;
    BVar4 =
        DeviceIoControl(DAT_1400fd630,0x83400004,(LPVOID)0x0,0,lpOutBuffer,0x40000,auStackX16,
                        (LPOVERLAPPED)& Stack56);
}
```

# SysmonDrv.sys

## Sysmon events pull

- Screen of a message exchanged with the kernel

01 00 00 00 5c 04 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	☺... \♦.....
00 00 00 00 00 00 00 00	24 09 00 00 28 03 00 00	..... \$... (♥..	
00 00 00 00 00 00 00 00	3A FE F9 73 0B 2C D5 01	..... :..s♂, ..☺	
E7 03 00 00 00 00 00 00	00 00 00 00 23 65 83 EC	.♥..... #e..	
00 00 00 00 01 00 00 00	0C 00 00 00 0C 00 00 00	....☺... ♀... ♀...	
3E 00 00 00 80 00 00 00	14 00 00 00 2A 00 00 00	>..... ¶... *...	
01 01 00 00 00 00 00 05	12 00 00 00 01 01 00 00	☺☺..... ♣↑... ☺☺...	
00 00 00 10 00 40 00 00	43 00 3A 00 5C 00 57 00	...▶. @... C.:. \.W.	
69 00 6E 00 64 00 6F 00	77 00 73 00 5C 00 53 00	i.n.d.o.w.s.\.S.	
79 00 73 00 74 00 65 00	6D 00 33 00 32 00 5C 00	y.s.t.e.m.3.2.\.	
73 00 76 00 63 00 68 00	6F 00 73 00 74 00 2E 00	s.v.c.h.o.s.t...	
65 00 78 00 65 00 43 00	3A 00 5C 00 57 00 49 00	e.x.e.C.:. \.W.I.	
4E 00 44 00 4F 00 57 00	53 00 5C 00 73 00 79 00	N.D.O.W.S.\.s.y.	
73 00 74 00 65 00 6D 00	33 00 32 00 5C 00 73 00	s.t.e.m.3.2.\.s.	
76 00 63 00 68 00 6F 00	73 00 74 00 2E 00 65 00	v.c.h.o.s.t...e.	
78 00 65 00 20 00 2D 00	6B 00 20 00 6E 00 65 00	x.e. .-k. .n.e.	
74 00 73 00 76 00 63 00	73 00 20 00 2D 00 70 00	t.s.v.c.s. .-p.	
20 00 2D 00 73 00 20 00	58 00 62 00 6C 00 41 00	.-.s. .X.b.l.A.	
75 00 74 00 68 00 4D 00	61 00 6E 00 61 00 67 00	u.t.h.M.a.n.a.g.	
65 00 72 00 00 00 C0 2E	C8 13 B2 E6 CB A9 2E 1C	e.r.....!!.....L	
72 37 68 50 73 7D F2 04	D4 C5 43 00 3A 00 5C 00	r7hPs}.♦...C.:. \.	
57 00 49 00 4E 00 44 00	4F 00 57 00 53 00 5C 00	W.I.N.D.O.W.S.\.	
73 00 79 00 73 00 74 00	65 00 6D 00 33 00 32 00	s.y.s.t.e.m.3.2.	
5C 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	\.....	

## Driver/Module loading

- To register a function who will handle all images loading there a a simple function :  
« PsSetLoadImageNotifyRoutine »


### PsSetLoadImageNotifyRoutine function

04/30/2018 • 2 minutes to read

The **PsSetLoadImageNotifyRoutine** routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).

#### Syntax

C++

 Copy

```
NTSTATUS PsSetLoadImageNotifyRoutine(  
    PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine  
);
```

# CreateProcess/Thread

- Kernel export 2 functions linked to 2 lists of callbacks
  - PsSetCreateProcessNotifyRoutine
  - PsSetCreateThreadNotifyRoutine
- Really simple and old lists

## Registry interceptions

- Registry callback registration is  
« CmRegisterCallback », usage is similar to  
« PsSetLoadImageNotifyRoutine »

### CmRegisterCallback function


04/30/2018 • 2 minutes to read

The **CmRegisterCallback** routine is **obsolete** for Windows Vista and later operating system versions. Use [CmRegisterCallbackEx](#) instead.

The **CmRegisterCallback** routine registers a [RegistryCallback](#) routine.

### Syntax

C++

 Copy

```
NTSTATUS CmRegisterCallback(  
    PEX_CALLBACK_FUNCTION Function,  
    PVOID Context,  
    PLARGE_INTEGER Cookie  
);
```

## Sysmon NamedPipe

- First Method: For kernel previous than version 6
- Sysmon create a new device named « SysmonPipeFilter » and attach it to « NamedPipe »

```
RtlInitUnicodeString((PUNICODE_STRING)&l_str_namedpipe,L"\\Device\\NamedPipe");
NVar2 = IoGetDeviceObjectPointer
    ((PUNICODE_STRING)&l_str_namedpipe,0x1f01ff,local_res18,&objNamedPipe);
if (NVar2 == 0) {
    ObfDereferenceObject(local_res18[0]);
    RtlInitUnicodeString((PUNICODE_STRING)&l_str_sysmonpipe,L"\\Device\\SysmonPipeFilter");
    NVar2 = IoCreateDevice((PDRIVER_OBJECT)lParm1,8,(PUNICODE_STRING)&l_str_sysmonpipe,
        objNamedPipe->DeviceType,0,0,(PDEVICE_OBJECT *)&objSysmonPipe);
    if (NVar2 == 0) {
        lParm1->FastIoDispatch = (_FAST_IO_DISPATCH *)&DAT_18001b090;
        PDEVICE_OBJECT_18001c3f8 =
            (PDEVICE_OBJECT)
                IoAttachDeviceToDeviceStack((PDEVICE_OBJECT)objSysmonPipe,objNamedPipe);
```



## Sysmon and FltMgr

- Second Method
- Sysmon use FltMgr to monitor actions on FileSystem
- With this we don't need to directly attach to the device stack to grab all messages (such 2.0)

```
Filter List: fffffdb04562e30c0 "Frame 0"  
  FLT_FILTER: fffffdb045d6e9ba0 "wcnfs" "409900"  
    FLT_INSTANCE: fffffdb0455821010 "wcnfs Instance" "409900"  
  FLT_FILTER: fffffdb04569bbc10 "SysmonDrv" "385201"  
    FLT_INSTANCE: fffffdb04551fca30 "Sysmon Instance" "385201"  
    FLT_INSTANCE: fffffdb045927ba40 "Sysmon Instance" "385201"  
    FLT_INSTANCE: fffffdb04592734a0 "Sysmon Instance" "385201"  
    FLT_INSTANCE: fffffdb045ac14cb0 "Sysmon Instance" "385201"  
    FLT_INSTANCE: fffffdb045effa2c0 "Sysmon Instance" "385201"  
  FLT_FILTER: fffffdb04562e30c0 "Frame 0" "385201"
```

## Sysmon and FltMgr

- Monitor :
  - \Device\HarddiskVolume\*
  - \Device\NamedPipe

```
FLT_INSTANCE: fffffdb04551fca30 "Sysmon Instance" "385201"  
CallbackNodes : (fffffdb04551fcad0)
```

```
CREATE (0)  
CALLBACK_NODE: fffffdb04551fcc60 Inst:(fffffdb04551fca30,"SysmonDrv","\Device\HarddiskVolume4") "Sysmon Instance" "385201"  
CREATE_NAMED_PIPE (1)  
CALLBACK_NODE: fffffdb04551fcd20 Inst:(fffffdb04551fca30,"SysmonDrv","\Device\HarddiskVolume4") "Sysmon Instance" "385201"  
CLOSE (2)  
CALLBACK_NODE: fffffdb04551fccf0 Inst:(fffffdb04551fca30,"SysmonDrv","\Device\HarddiskVolume4") "Sysmon Instance" "385201"  
SET_INFORMATION (6)  
CALLBACK_NODE: fffffdb04551fccc0 Inst:(fffffdb04551fca30,"SysmonDrv","\Device\HarddiskVolume4") "Sysmon Instance" "385201"  
CLEANUP (18)  
CALLBACK_NODE: fffffdb04551fcc90 Inst:(fffffdb04551fca30,"SysmonDrv","\Device\HarddiskVolume4") "Sysmon Instance" "385201"
```

```
FLT_INSTANCE: fffffdb045927ba40 "Sysmon Instance" "385201"  
CallbackNodes : (fffffdb045927bae0)
```

```
CREATE (0)  
CALLBACK_NODE: fffffdb045927bc70 Inst:(fffffdb045927ba40,"SysmonDrv","\Device\NamedPipe") "Sysmon Instance" "385201"  
CREATE_NAMED_PIPE (1)  
CALLBACK_NODE: fffffdb045927bd30 Inst:(fffffdb045927ba40,"SysmonDrv","\Device\NamedPipe") "Sysmon Instance" "385201"  
CLOSE (2)  
CALLBACK_NODE: fffffdb045927bd00 Inst:(fffffdb045927ba40,"SysmonDrv","\Device\NamedPipe") "Sysmon Instance" "385201"  
SET_INFORMATION (6)  
CALLBACK_NODE: fffffdb045927bcd0 Inst:(fffffdb045927ba40,"SysmonDrv","\Device\NamedPipe") "Sysmon Instance" "385201"  
CLEANUP (18)  
CALLBACK_NODE: fffffdb045927bca0 Inst:(fffffdb045927ba40,"SysmonDrv","\Device\NamedPipe") "Sysmon Instance" "385201"
```

# Process Access

- This kind of action have a stack of filters like FltMgr, to handle:
  - Threads
  - Processes
  - Desktops
- Like FltMgr there is a system of PRE/POST operations, only POST operations are used for logging

## ObRegisterCallbacks function

04/30/2018 • 2 minutes to read

The **ObRegisterCallbacks** routine registers a list of callback routines for thread, process, and desktop handle operations.

## WMI filters

WMI is really NOT funny...



## WMI filters

- A callback is called each 5 seconds
- This callback filter 3 WMI classes :
  - \_\_EventConsumer
  - \_\_EventFilter
  - \_\_FilterToConsumerBinding
- All those classes are used to set WMI persistence, so Sysmon just check persistence queries

## WMI filters

```
l_root_subscription = SysAllocString(L"ROOT\\Subscription");  
[...]  
wmi_str_filter =  
    ConvertStringToBSTR(  
        "SELECT * FROM __InstanceOperationEvent WITHIN 5WHERE  
        TargetInstance ISA \'__EventConsumer\' OR TargetInstance ISA  
        \'__EventFilter\' OR TargetInstance ISA  
        \'__FilterToConsumerBinding\'"  
    );  
*_Memory = wmi_str_filter;  
_Memory_00 = (longlong *)FUN_140049808(0x18);  
if (_Memory_00 != (longlong *)0x0) {  
    _Memory_00[1] = 0;  
    *(undefined4 *)(_Memory_00 + 2) = 1;  
    str_sql = ConvertStringToBSTR("WQL");  
    *_Memory_00 = str_sql;  
    iVar4 = (*(code *) (*IWbemServices)->ExecNotificationQueryAsync)  
        (IWbemServices, str_sql, wmi_str_filter, 0x80, 0, sink_class);
```

## WMI filters

- When the callback is triggered, the function « Indicate » of the sink object is called
- Sysmon check is the action is interesting to log on those 3 actions :
  - "Deletion"
  - "Creation"
  - "Modification"

## Network tracing

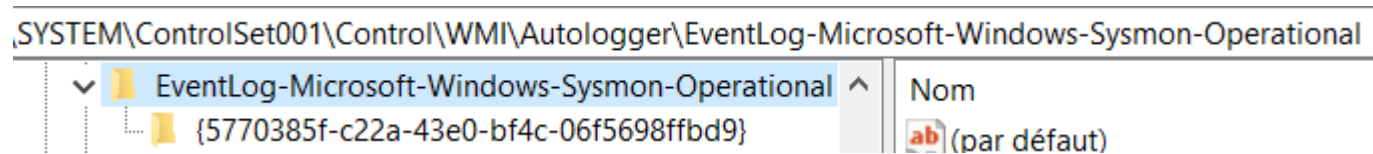
- To trace network traffic Sysmon use ETW callbacks "NT Kernel Logger" on `EVENT_TRACE_FLAG_NETWORK_TCPIP`
- Instance is named "SYSMON TRACE"
- Each event is described in a buffer. And Sysmon explore WMI objects inside "root\wmi" that described howto parse the buffer based on the event type
- More infos on ETW on :  
[https://exatrack.com/public/etw\\_for\\_the\\_lazy\\_rever\\_ser\\_beerump\\_2019.pdf](https://exatrack.com/public/etw_for_the_lazy_rever_ser_beerump_2019.pdf) (FR)



## Events registration

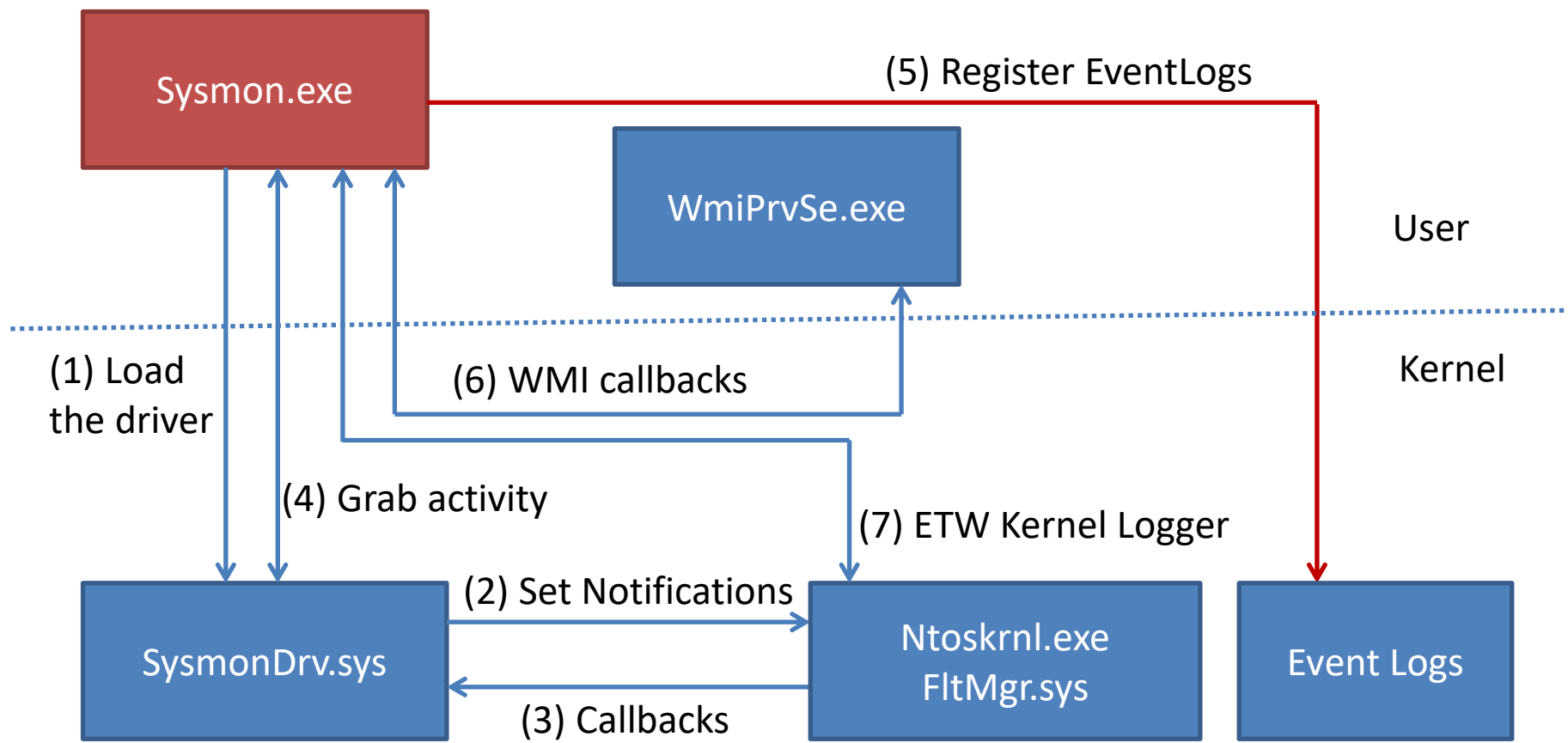
- Registration of handle to write eventlogs.

```
GUID_1400c0170 XREF[
GUID 5770385f-c22a-43e0-bf4c-06f5698ffbd9 ,
(*_f_EventRegister) (GUID_1400c0170, 0, 0, &REGHANDLE_1400fbc90);
```



## Design fail...

- All logs are register by « Sysmon.exe » service, so if (for an unknowns reason) it crash... no logs :(



## Sysmon intercept everything ?

- A lot of actions are handled by Sysmon
- But Windows have a lot of more cases to break logic of a « normal » execution
- For exemple Sysmon don't monitor win32k operations
- A full memory malware can hide itself to Sysmon if it tricks a lot
- But most of attackers can't burn all those tricks for a campaign, so they use standard actions and can left traces on Events

## Sysmon intercept everything ?

- A big problem is the Sysmon configuration
  - A lot of companies use standards shared XML
  - So they miss majority of system actions
  - When an attacker do tricky actions we can see side effects in OS activity and can reveal an attacker
    - For exemple a conhost.exe run by a critical executable -> maybe an injection
    - Anormal file writed by a critical process
    - DLL loaded by a critical process (often DLL loading is disable)
    - ...
- We recomand to log all major system actions (minimum all CreateProcess)!

# Attackers actions examples

- A lot of actions are done by a lot of attackers
- Exemple of actions:
  - PSEXEC execution
  - SVCHOST located in a temporary directory
  - Write of a RUN registry key and the file pointed
  - Powershell with a payload full of shit
  - Word executing a VBS
  - Scheduled Task with a file in %TEMP%
  - Creation of executable in « C:\Programdata »

# Detecting attacker

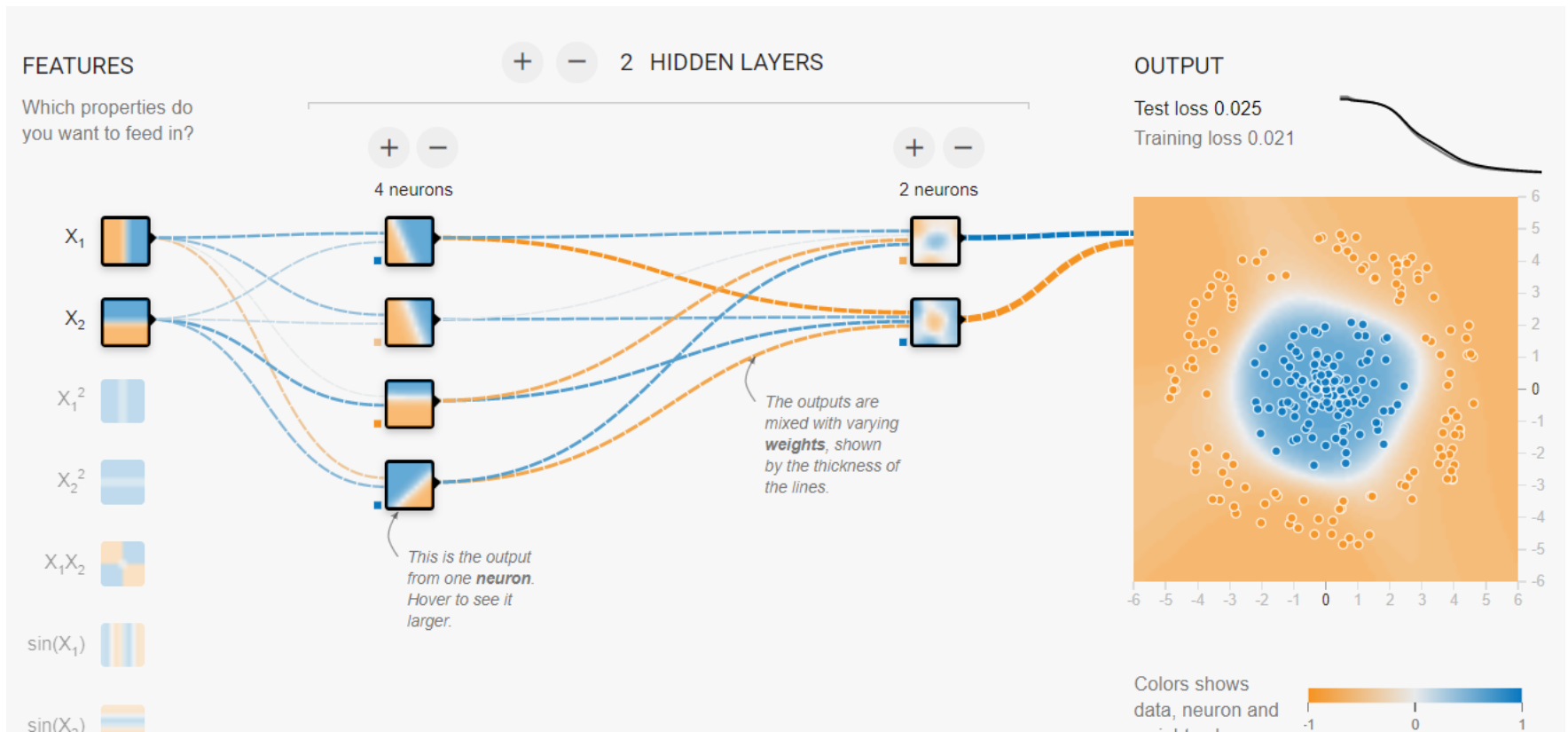
- Some rules based on previous artefacts
- Identify new executables hashes/paths on your SI
- Anormal connections of a user on multiple computers
- Have a list of normal runned services on your park
- Identify actions of user at suspicious hours (when they sleep for example)

# Sysmon – Deep Learning

- We can write a lot of rules, on a lot of cases
- But often when we see an event we can say « Hummm, it smell really bad! »
- This is the reflexion process we want to reproduce on our logs
- Honestly we have tried ML because everybody told us that it would perfectly apply to our usecase. We gave it a try without really expecting anything from it.

## Sysmon – Deep Learning

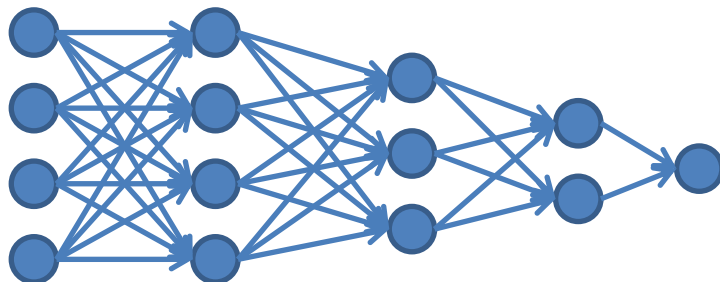
- Basics of Neural Networks / Deep Learning





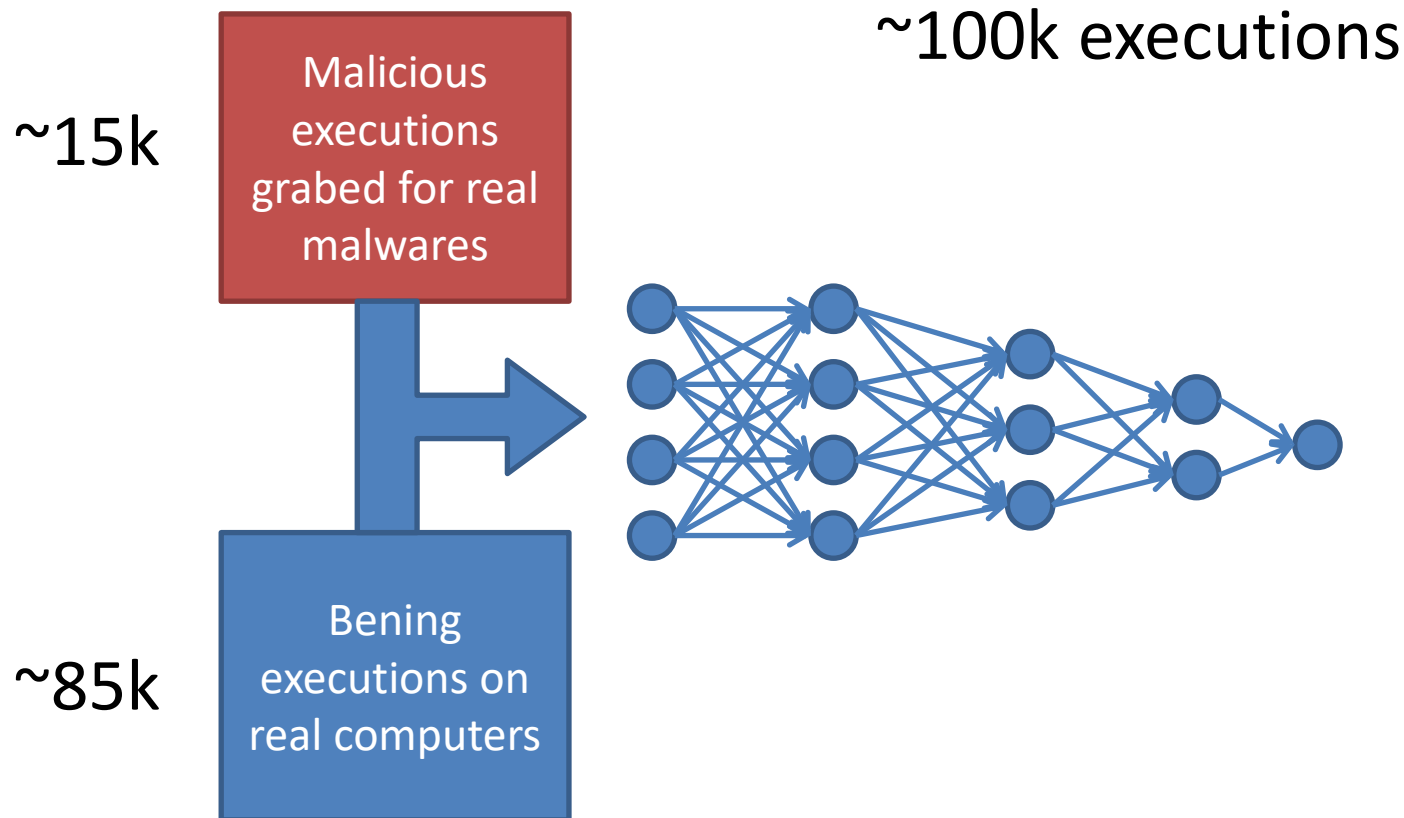
## Sysmon – Deep Learning

- Our goal is to identify a suspicious CreateProcess
- To do this we exact:
  - Current process name
  - Current command line
  - Parent process
  - Parent command line
- We have 1 exit node to say if it suspicious or not



## Sysmon – Deep Learning

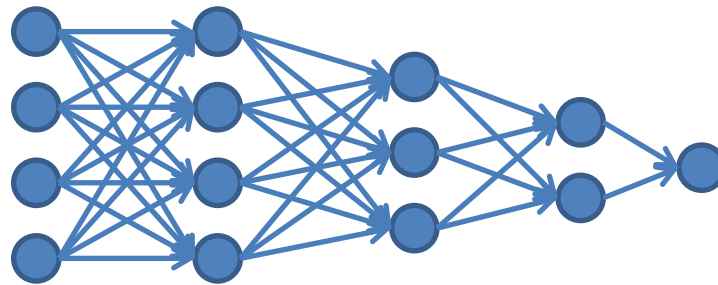
- Set of training



## Sysmon – Deep Learning

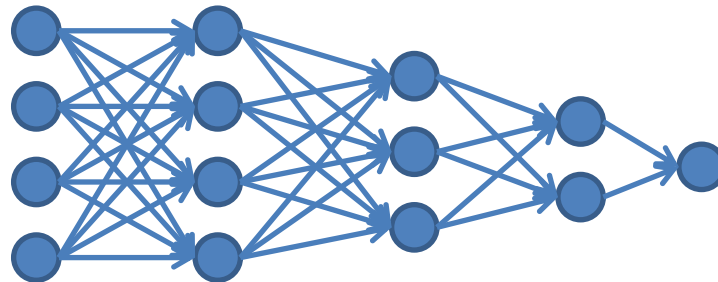
- Set of testing

New Malicious  
executions  
grabbed for real  
malwares



>95% of detection

Checks with  
real executions  
from other  
compaines



0,1% to 3% of false  
positives

# Sysmon – Deep Learning

- Funy detection
  - 'PPath':  
'C:\\Windows\\SysWOW64\\FlashPlayerInstaller.exe'
  - 'PCmdLine': 'FlashPlayerInstaller.exe -install -iv 11'
  - 'Path': 'C:\\Windows\\SysWOW64\\cmd.exe'
  - 'CmdLine': '"C:\\Windows\\ System32 \\cmd.exe" /c del "FlashPlayerInstaller.exe" >> NUL '

## Conclusion

We know how Sysmon grab his logs.

What it can to detect and what it can't.

How to parse logs to identify suspicious activities.

Now: **Just install Sysmon!**





Thank you for your attention.

Any questions ?

Stéfan Le Berre (@Heurs)  
stefan.le-berre [at] exatrack.com

<https://exatrack.com>

Nice work on sysmon internals too :

<https://ackroute.com/post/2017/08/08/sysmon-enumeration-overview/>

**Click here !**