

Hunting in the Sysmon Call Trace

lares.com/blog/hunting-in-the-sysmon-call-trace

January 29, 2021

Intro

The Sysmon ProcessAccess event has been used in threat hunting and detection efforts in order to alert on techniques such as process injection and credential access.

According to the Sysinternals website, the Sysmon ProcessAccess event reports when a process opens another process, an operation that's often followed by information queries or reading and writing the address space of the target process.

Johnny Johnson's research into which APIs map to Sysmon events shows us that the Sysmon ProcessAccess event gets its information from the `NtOpenProcess`` and ``NtAlpcOpenSenderProcess` Windows APIs.

We can also look at the Sysmon Community Guide for an explanation of the various fields contained in the Sysmon ProcessAccess event as well as a sample configuration file containing rules that correspond to this event.

A particular field of interest for this blog post is the CallTrace field, which, according to the Sysmon Community guide, includes `the DLL and the relative virtual address of the functions in the call stack right before the open process call.`

Very interesting indeed and leads to the question of whether we can extract any detection value out of this field.

With this in mind, the purpose of this blog post is to look at the above question and take a deeper dive into the Sysmon CallTrace field in order to demonstrate how it can be used for both writing Sysmon rules as well as hunting for particular TTPs.

Before diving in, it is important to note that hunting within the Call Trace field is not a brand new concept, Andrey Skablonsky presented on the topic for Zero Nights 2019. This presentation served as the inspiration for this blog post.

Methodology

A sample Sysmon Process Access event can be found here:

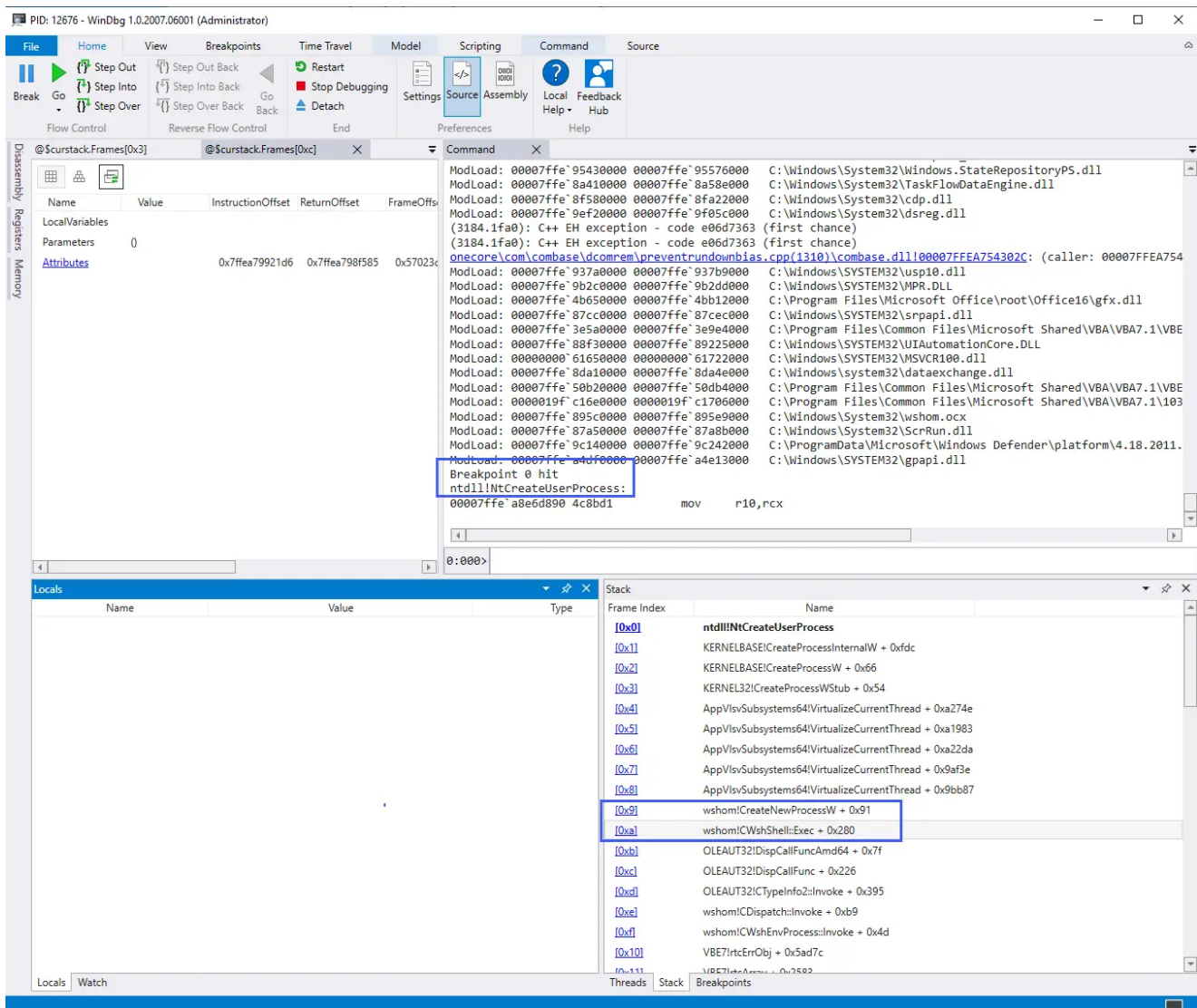
<https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90010> – taking a look at the CallTrace field, we see values like

C:\Windows\SYSTEM32\ntdll.dll+a5594` and `C:\Windows\system32\KERNELBASE.dll+1e865 — these are the relative virtual addresses of the functions contained within the call stack at the time that the ProcessAccess event occurred.

In order to extract value from these events and make them readable to humans, we need a way to look up these addresses and translate them to their corresponding functions.

There are probably numerous ways to accomplish this task, however, for the purposes of this post, WinDbg and Process Monitor will be used.

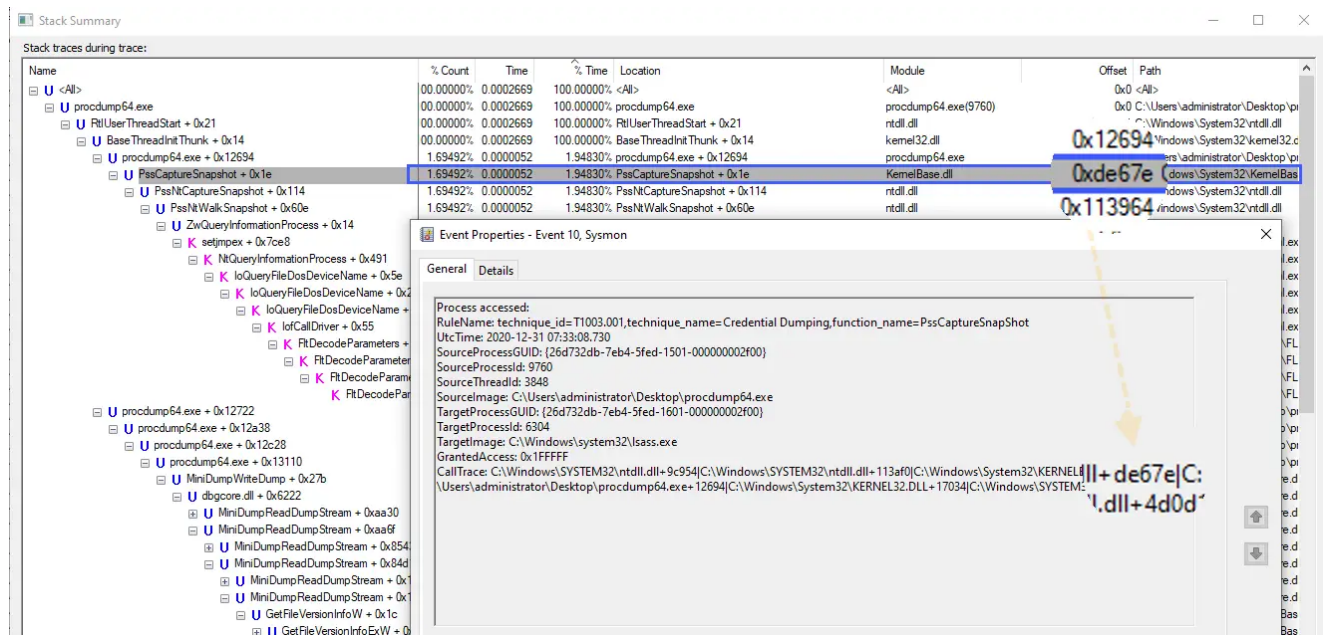
Using WinDbg, we need to either start the process we need to monitor via the debugger or start the process first and then attach a debugger to it. After doing so, a breakpoint needs to be set on the CreateUserProcess API call: `bp ntdll!ZwCreateUserProcess`



The screenshot above illustrates a debugger running, in this case attached to the WINWORD.exe process, with the breakpoint highlighted above it.

The "Stack" window shows us the same values as would be found in the CallTrace field of the Sysmon ProcessAccess event, however, we can see that WinDbg has translated the function calls for us.

The Process Monitor "Stack Summary" can also be used to translate call trace values:



In the screenshot above we can see the PssCaptureSnapshot function address in both the Process Monitor window as well as the Sysmon ProcessAccess CallTrace field.

Those who are more proficient in reverse engineering may have more comprehensive and automated ways of translating these function calls. For a reversing newbie like myself however, the methods above worked as – at the very least – a proof of concept.

At this point, we now know what the Sysmon ProcessAccess CallTrace field is and we also know how to translate the function calls contained within this field.

We can now look at a few examples of how this data can be used within the Sysmon configuration file.

Before doing so however, it must be noted that the location of these function calls will most likely change depending on what Windows version is being used. For this post, Windows 10 20H2 Build 19042.685 was used.

Credential Dumping – MiniDumpWriteDump

A common technique for dumping credentials is using the MiniDumpWriteDump function

With the following Sysmon config snippet:

```
<CallTrace condition="contains"
name="technique_id=T1003.001,technique_name=Credential
Dumping,function_name=MiniDumpWriteDump">C:\Windows\SYSTEM32\dbgcore.DLL+6cfb</CallTr
ace>
```

We can alert on this specific function call.

As a test, we can use procdump to dump the LSASS process:

The image shows a terminal window and an Event Properties dialog box. The terminal window displays the command `procdump.exe -accepteula -ma lsass.exe lsass.dmp` and its output, including the version of ProcDump (v10.0) and the completion of the dump process. The Event Properties dialog box shows the details of the event, including the process accessed, rule name, and call trace information.

```
C:\Users\administrator\Desktop>procdump.exe -accepteula -ma lsass.exe lsass.dmp

ProcDump v10.0 - Sysinternals process dump utility
Copyright (C) 2009-2020 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

[12:33:33] Dump 1 initiated: C:\Users\administrator\Desktop\lsass.dmp
[12:33:33] Dump 1 writing: Estimated dump file size is 57 MB.
[12:33:33] Dump 1 complete: 58 MB written in 0.1 seconds
[12:33:33] Dump count reached.

C:\Users\administrator\Desktop>
```

Event Properties - Event 10, Sysmon

General Details

Process accessed:
RuleName: technique_id=T1003.001 technique_name=Credential
Dumping,function_name=MiniDumpWriteDump
UtcTime: 2021-01-01 17:33:33.256
SourceProcessGUID: {26d732db-5ced-5fef-d505-000000002f00}
SourceProcessId: 8700
SourceThreadId: 8348
SourceImage: C:\Users\administrator\Desktop\procdump64.exe
TargetProcessGUID: {26d732db-7ca8-5fed-0e00-000000002f00}
TargetProcessId: 824
TargetImage: C:\Windows\system32\lsass.exe
GrantedAccess: 0x1FFFFFFF
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9c474|C:\Windows\SYSTEM32\ntdll.dll+d77fa|C:\Windows\System32\KERNEL32.DLL+1decc|C:\Windows\System32\KERNEL32.DLL+2655e|C:\Windows\SYSTEM32\dbgcore.DLL+99b1|C:\Windows\SYSTEM32\dbgcore.DLL+179b5|C:\Windows\SYSTEM32\dbgcore.DLL+11425|C:\Windows\SYSTEM32\dbgcore.DLL+6222|C:\Windows\SYSTEM32\dbgcore.DLL+6cfb|C:\Users\administrator\Desktop\procdump64.exe+13110|C:\Users\administrator\Desktop\procdump64.exe+12b45|C:\Users\administrator\Desktop\procdump64.exe+12a65|C:\Users\administrator\Desktop\procdump64.exe+12722|C:\Windows\System32\KERNEL32.DLL+17034|C:\Windows\SYSTEM32\ntdll.dll+4d0d1

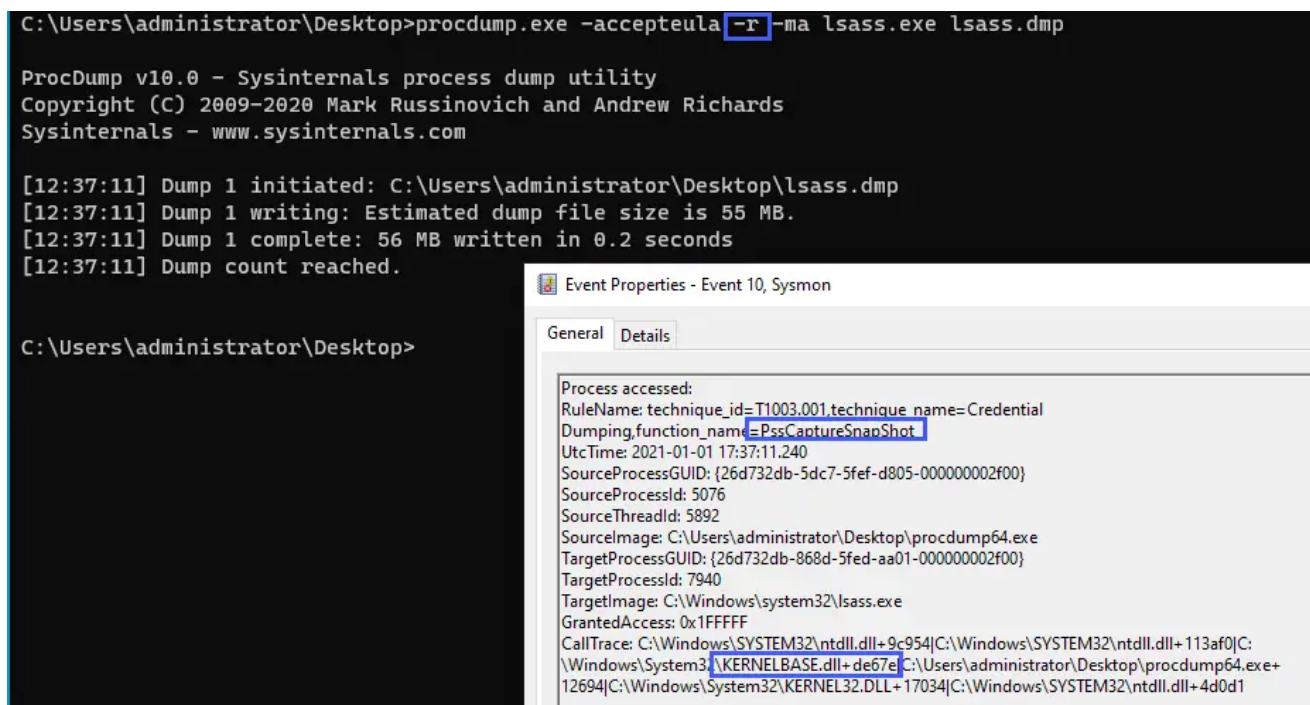
Credential Dumping – PssCaptureSnapshot

Taking a look at another example, we can run procdump with the -r flag which uses the PssCaptureSnapshot function rather than MiniDumpWrite.

We can use the following Sysmon config snippet to alert on this:

```
<CallTrace condition="contains"
name="technique_id=T1003.001,technique_name=Credential
Dumping,function_name=PssCaptureSnapshot">C:\Windows\System32\KernelBase.dll+de67e</C
allTrace>
```

And taking a look at the event generated:



The two above examples cover a few credential theft TTPs, which, as stated in the Sysinternals documentation, is one of the primary use cases for the Sysmon ProcessAccess event.

Using the data contained within the CallTrace field, we can extend the functionality of this event a little bit.

Let's continue on with the examples and look at a few Office Macros TTPs.

We'll be using the awesome "Generate Macro" test provided by Red Canary as a base for the following Macro tests: <https://redcanary.com/blog/testing-initial-access-with-generate-macro-in-atOMIC-red-team/>

WMI Execution

Given the following macro:

```
Sub Calc()
  Set objWMIService = GetObject("winmgmts:
{impersonationLevel=impersonate}!\.\root\cimv2")
  Set objStartup = objWMIService.Get("Win32_ProcessStartup")
  Set objConfig = objStartup.SpawnInstance_
  Set objProcess = GetObject("winmgmts:root\cimv2:Win32_Process")
  errReturn = objProcess.Create("calc", Null, objConfig, intProcessID)
End Sub
```

Combined with the following Sysmon config snippets:

```
<CallTrace condition="contains" name="technique_id=T1047,technique_name=Windows Management Instrumentation,function_name=ProviderExecMethod">C:\Windows\SYSTEM32\framedynos.dll+2b496</CallTrace>
<CallTrace condition="contains" name="technique_id=T1047,technique_name=Windows Management Instrumentation,function_name=CWbemProviderGlueExecMethodAsync">C:\Windows\SYSTEM32\framedynos.dll+2cb3e</CallTrace>
```

We can alert on the function calls used for WMI process execution:

```
Process accessed:
RuleName: technique_id=T1047,technique_name=Windows Management
Instrumentation,function_name=ProviderExecMethod
UtcTime: 2021-01-01 17:42:50.767
SourceProcessGUID: {26d732db-7caa-5fed-6600-000000002f00}
SourceProcessId: 4192
SourceThreadId: 8632
SourceImage: C:\Windows\system32\wbem\wmiprvse.exe
TargetProcessGUID: {26d732db-5f1a-5fef-f505-000000002f00}
TargetProcessId: 1596
TargetImage: C:\Windows\system32\calc.exe
GrantedAccess: 0x1FFFFFFF
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d8a4|C:\Windows\System32\KERNELBASE.dll+32d2c|C:\Windows\System32\KERNELBASE.dll+7a023|C:\Windows\System32\KERNEL32.DLL+1db20|C:\Windows\system32\wbem\cimwin32.dll+48ea9|C:\Windows\system32\wbem\cimwin32.dll+4badc|C:\Windows\system32\wbem\cimwin32.dll+48b27|C:\Windows\system32\wbem\cimwin32.dll+4959f|C:\Windows\system32\wbem\cimwin32.dll+49f71|C:\Windows\SYSTEM32\framedynos.dll+2b496|C:\Windows\SYSTEM32\framedynos.dll+2cb3e|C:\Windows\system32\wbem\wmiprvse.exe+10dat|C:\Windows\system32\wbem\wmiprvse.exe+108df|C:\Windows\System32\RPCRT4.dll+78963|C:\Windows\System32\RPCRT4.dll+1c873|C:\Windows\System32\combase.dll+b384c|C:\Windows\System32\RPCRT4.dll+5a4db|C:\Windows\System32\combase.dll+91e73|C:\Windows\System32\combase.dll+91bfe|C:\Windows\System32\combase.dll+b9356|C:\Windows\System32\combase.dll+21a13|C:\Windows\System32\combase.dll+ad7fd|C:\Windows\System32\combase.dll+77aa8|C:\Windows\System32\combase.dll+f8958
```

Note that in this case, the parent process for calc.exe is wmiprvse.exe and not WINWORD.exe as this particular macro breaks the typical 'Word spawns a process' lineage.

The Sysmon config above that alerts on the ProviderExecMethod function call will also fire if WMI is used to launch processes in other ways, such as PowerShell, and is not limited to macro executions:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\administrator> Invoke-WmiMethod -Path win32_process -Name create -ArgumentList notepad.exe
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS    :
__DYNASTY       : __PARAMETERS
__RELPATH       :
__PROPERTY_COUNT : 2
__DERIVATION    : {}
__SERVER        :
__NAMESPACE     :
__PATH          :
ProcessId       : 692
ReturnValue     : 0
PSComputerName  :
```

```
PS C:\Users\administrator>
```

Event Properties - Event 10, Sysmon

General Details

Process accessed:
 RuleName: technique_id=T1047,technique_name=Windows Management
 Instrumentation,function_name=ProviderExecMethod
 UtcTime: 2021-01-01 17:47:09.915
 SourceProcessGUID: {26d732db-7caa-5fed-6600-00000002f00}
 SourceProcessId: 4192
 SourceThreadId: 8632
 SourceImage: C:\Windows\system32\wbem\wmiprvse.exe
 TargetProcessGUID: {26d732db-601d-5fef-0206-00000002f00}
 TargetProcessId: 692
 TargetImage: C:\Windows\system32\notepad.exe
 GrantedAccess: 0x1FFFFFFF
 CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d8a4|C:\Windows\System32\KERNELBASE.dll+32d2c|C:\Windows\System32\KERNELBASE.dll+7a023|C:\Windows\System32\KERNEL32.DLL+1db20|C:\Windows\system32\wbem\cimwin32.dll+48ea9|C:\Windows\system32\wbem\cimwin32.dll+4badc|C:\Windows\system32\wbem\cimwin32.dll+48b27|C:\Windows\system32\wbem\cimwin32.dll+4959f|C:\Windows\system32\wbem\cimwin32.dll+49f71|C:\Windows\SYSTEM32\framedynos.dll+2b496|C:\Windows\SYSTEM32\framedynos.dll+2cb3e|C:\Windows\system32\wbem\wmiprvse.exe+10daf|C:\Windows\system32\wbem\wmiprvse.exe+108df|C:\Windows\System32\RPCRT4.dll+78963|C:\Windows\System32\RPCRT4.dll+1c873|C:\Windows\System32\combase.dll+b384c|C:\Windows\System32\RPCRT4.dll+5a4db|C:\Windows\System32\combase.dll+91e73|C:\Windows\System32\combase.dll+91bfe|C:\Windows\System32\combase.dll+b9356|C:\Windows\System32\combase.dll+21a13|C:\Windows\System32\combase.dll+ad7fd|C:\Windows\System32\combase.dll+77aa8|C:\Windows\System32\combase.dll+f8958

Wscript.Shell

Let's continue on with our examples and look at the WShellExec function which is another popular execution technique, looking at the following macro code:

```
Sub Auto_Open()
    Set WshShell = CreateObject("WScript.Shell")
    Set WshShellExec = WshShell.Exec("cmd.exe /c calc")
End Sub
```

We can alert on the WShellExec function with the following config snippet:

```
<CallTrace condition="contains any"
name="technique_id=T1059.005,technique_name=Command and Scripting
Interpreter,function_name=WShellExec">C:\Windows\System32\wshom.ocx+c8a0;C:\Windows\System32\wshom.ocx+c39d</CallTrace>
```

```

Process accessed:
RuleName: technique_id=T1059.005,technique_name= Command and Scripting
Interpreter,function_name=WSHellExec
UtcTime: 2021-01-01 17:51:15.559
SourceProcessGUID: {26d732db-6112-5fef-2006-000000002f00}
SourceProcessId: 6428
SourceThreadId: 4616
SourceImage: C:\Program Files\Microsoft Office\Root\Office16\EXCEL.EXE
TargetProcessGUID: {26d732db-6113-5fef-2206-000000002f00}
TargetProcessId: 3368
TargetImage: C:\Windows\SYSTEM32\cmd.exe
GrantedAccess: 0x1FFFFFF
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d8a4|C:\Windows\System32\KERNELBASE.dll+32d2c|C:\Windows\System32\KERNELBASE.dll+76516|C:\Windows\System32\KERNEL32.DLL+1cbb4|C:\Program Files\Microsoft Office\Root\Office16\AppV\svSubsystems64.dll+d864e|C:\Program Files\Microsoft Office\Root\Office16\AppV\svSubsystems64.dll+d7883|C:\Program Files\Microsoft Office\Root\Office16\AppV\svSubsystems64.dll+d81da|C:\Program Files\Microsoft Office\Root\Office16\AppV\svSubsystems64.dll+d0e3e|C:\Program Files\Microsoft Office\Root\Office16\AppV\svSubsystems64.dll+d1a87|C:\Windows\System32\wshom.ocx+c39d|C:\Windows\System32\wshom.ocx+c8a0|C:\Windows\System32\OLEAUT32.dll+1ed9f|C:\Windows\System32\OLEAUT32.dll+121d6|C:\Windows\System32\OLEAUT32.dll+f585|C:\Windows\System32\wshom.ocx+e069|C:\Windows\System32\wshom.ocx+26ad|C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL+2d64d0|C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL+38eb2b|C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL+39020a|C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL+382e84|C:\Windows\System32\OLEAUT32.dll+1ed9f|C:\Windows\System32\OLEAUT32.dll+121d6|C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL+1d1d19

```

ShellExecute

Similar to Wscript.Shell, ShellExecute can also be used to launch processes via Office Macros, taking a look at the following macro code:

```

Function ShellExecuteVB()
    Dim objShell
    Set objShell = CreateObject("Shell.Application")
    Call objShell.ShellExecute("notepad.exe", "", "", "open", 1)
End Function

```

And the following config snippet:

```

<CallTrace condition="contains" name="technique_id=T1059.005,technique_name=Command and Scripting Interpreter,function_name=CSShellExecute_ExecuteAssoc">C:\Windows\System32\SHELL32.dll+9b5bd</CallTrace>
<CallTrace condition="contains" name="technique_id=T1059.005,technique_name=Command and Scripting Interpreter,function_name=CSShellExecute_DoExecute">C:\Windows\System32\SHELL32.dll+a3b9</CallTrace>

```

Gives us the following results:


```
Process accessed:
RuleName: technique_id=T1059.005 technique_name=Command and Scripting
Interpreter,function_name=CSShellExecute_ExecuteAssoc
UtcTime: 2021-01-01 17:58:13.906
SourceProcessGUID: {26d732db-62b1-5fef-3d06-000000002f00}
SourceProcessId: 5080
SourceThreadId: 168
SourceImage: C:\Program Files\Microsoft Office\Root\Office16\WINWORD.EXE
TargetProcessGUID: {26d732db-62b5-5fef-4606-000000002f00}
TargetProcessId: 8792
TargetImage: C:\Windows\System32\notepad.exe
GrantedAccess: 0x1FFFFF
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d8a4|C:\Windows\System32\KERNELBASE.dll+32d2c|C:\Windows\System32\KERNELBASE.dll+76516|C:\Windows\System32\KERNEL32.DLL+1cbb4|C:\Program Files\Microsoft Office\Root\Office16\AppVIsvSubsystems64.dll+d864e|C:\Program Files\Microsoft Office\Root\Office16\AppVIsvSubsystems64.dll+d7883|C:\Program Files\Microsoft Office\Root\Office16\AppVIsvSubsystems64.dll+d81da|C:\Program Files\Microsoft Office\Root\Office16\AppVIsvSubsystems64.dll+d0e3e|C:\Program Files\Microsoft Office\Root\Office16\AppVIsvSubsystems64.dll+d1a87|C:\Windows\SYSTEM32\windows.storage.dll+1a0d3d|C:\Windows\SYSTEM32\windows.storage.dll+15d60c|C:\Windows\SYSTEM32\windows.storage.dll+158dd4|C:\Windows\SYSTEM32\windows.storage.dll+15770b|C:\Windows\SYSTEM32\windows.storage.dll+15738d|C:\Windows\SYSTEM32\windows.storage.dll+15c435|C:\Windows\SYSTEM32\windows.storage.dll+158bac|C:\Windows\SYSTEM32\windows.storage.dll+15db77|C:\Windows\System32\SHELL32.dll+9b5bd|C:\Windows\System32\SHELL32.dll+ae3b9|C:\Windows\System32\SHELL32.dll+78c6d|C:\Windows\System32\shcore.dll+2d999|C:\Windows\System32\KERNEL32.DLL+17034|C:\Windows\SYSTEM32\ntdll.dll+4d0d1
```

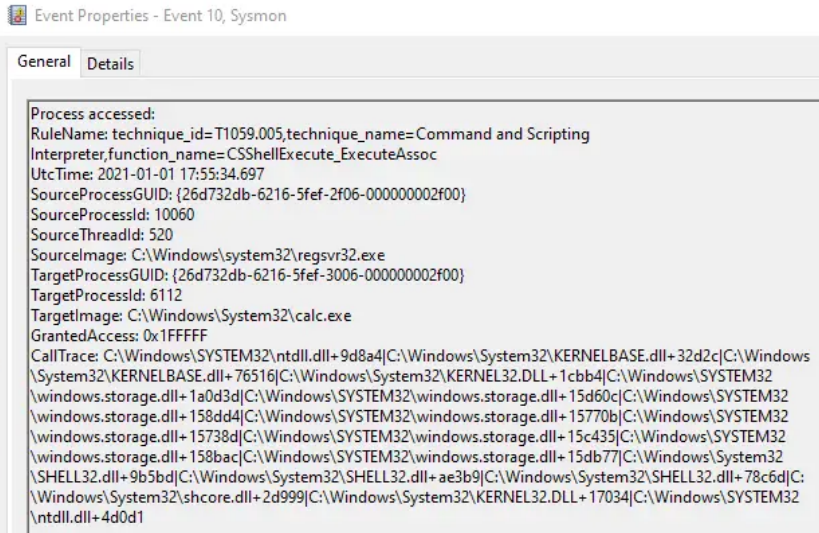
As with some of our previous examples, this function call is not unique to Office macros and will fire whenever the particular function is called, the screenshot below illustrates the same event firing for a sct scriplet executed via regsvr32:

```

C:\Users\administrator\Desktop\Tests>type test.sct
<?XML version="1.0"?>
<scriptlet>
<registration
  progid="TESTING"
  classid="{A1112221-0000-0000-3000-000DA00DABFC}" >
  <script language="JScript">
    <![CDATA[
      var foo = new ActiveXObject("WScript.Shell") Run("calc.exe");
    ]]>
  </script>
</registration>
</scriptlet>
C:\Users\administrator\Desktop\Tests>regsvr32 /s /i test.sct

C:\Users\administrator\Desktop\Tests>

```



Special shout out here to [@spottheplanet](#) and [ired.team](#) which were used as a reference for some of the techniques discussed here.

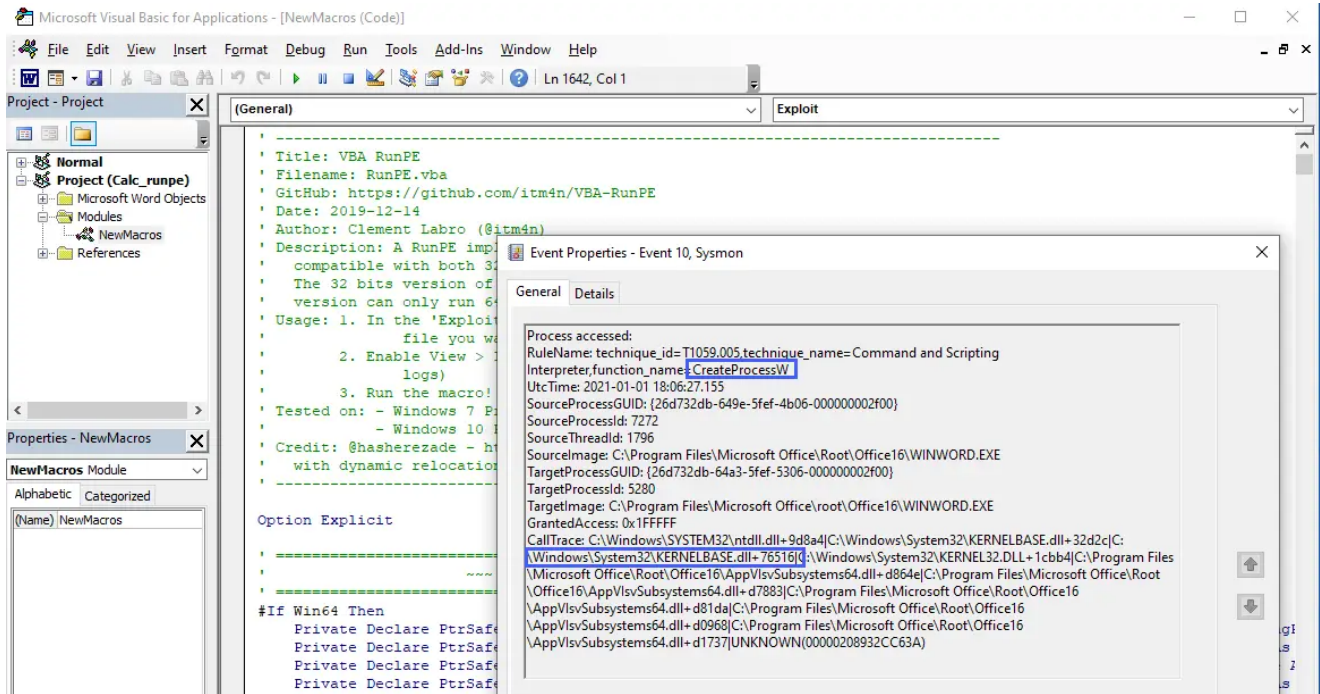
OpenProcess – Office Application

Looking at the final example, we can use our methodology of looking at function calls and combine that with Sysmon Rule Groups, the below example looks at an Office product calling the CreateProcessW function which in my example was used to alert on a VBA RunPE macro:

```

<Rule groupRelation="and" name="technique_id=T1059.005,technique_name=Command and Scripting Interpreter,function_name=CreateProcessW">
  <SourceImage condition="contains">C:\Program Files\Microsoft Office\Root\Office16</SourceImage>
  <CallTrace condition="contains">C:\Windows\System32\KERNELBASE.dll+76516</CallTrace>
</Rule>

```



Putting it Together

The config snippets outlined above followed the naming convention used by [Olaf Hartong's Sysmon Modular](#) project with an added field called `function_name` – if we send this data to a SIEM like Splunk, we can then do a bit of data massaging like so:

```
| eval RuleNameSplit = split(RuleName, ",")
| eval technique_name=mvindex(RuleNameSplit, 2)
| eval function_name=split(technique_name, "=")
| eval function_name_only=mvindex(function_name, 1)
```

And if we combine this with a simple `stats` command:

```
index=sysmon
| eval RuleNameSplit = split(RuleName, ",")
| eval technique_name=mvindex(RuleNameSplit, 2)
| eval function_name=split(technique_name, "=")
| eval function_name_only=mvindex(function_name, 1)
| stats values(function_name_only) BY SourceImage, TargetImage
```

We can get a nice view of the functions called by the tests we went through above:

SourceImage	TargetImage	values(function_name_only)
C:\Program Files\Microsoft Office\Root\Office16\EXCEL.EXE	C:\Windows\SYSTEM32\cmd.exe	WSHShellExec
C:\Program Files\Microsoft Office\Root\Office16\WINWORD.EXE	C:\Program Files\Microsoft Office\root\Office16\WINWORD.EXE	CreateProcessW
C:\Program Files\Microsoft Office\Root\Office16\WINWORD.EXE	C:\Windows\System32\notepad.exe	CSShellExecute_ExecuteAssoc
C:\Users\administrator\Desktop\procdump64.exe	C:\Windows\system32\lsass.exe	MiniDumpWriteDump PssCaptureSnapShot
C:\Windows\system32\regsvr32.exe	C:\Windows\System32\calc.exe	CSShellExecute_ExecuteAssoc
C:\Windows\system32\wbem\wmiiprvse.exe	C:\Windows\system32\calc.exe	ProviderExecMethod
C:\Windows\system32\wbem\wmiiprvse.exe	C:\Windows\system32\notepad.exe	ProviderExecMethod

Conclusion

The aim of this post was to highlight how the data contained within the Sysmon ProcessAccess CallTrace field can be translated and used to aid detection and hunting efforts.

The efficacy of this technique is somewhat blunted by the fact that the addresses of the function calls will change with Windows OS and software updates, combined with the manual process involved in translating the addresses into function names. However, despite these barriers, the use of translated function calls in the CallTrace field may be able to help with Sysmon configuration file tuning, especially when used in Sysmon RuleGroups. In addition, extracting additional context from data already being generated is also an added benefit.

Additional utility can also be extracted from these events when paired with other Sysmon telemetry such as ImageLoads and ProcessCreate events.

Appendix – Sample Sysmon Config Used for Post

For the purposes of this post, a bare-bones configuration file was used. If you want to tinker around with these events, feel free to test the following configuration. However, please consider this configuration as untested in a production environment:

```

<Sysmon schemaversion="4.40">
  <HashAlgorithms>*</HashAlgorithms>
  <CheckRevocation/>
  <EventFiltering>
    <RuleGroup name="" groupRelation="or">
      <CreateRemoteThread onmatch="include"></CreateRemoteThread>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <ImageLoad onmatch="include"></ImageLoad>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <ProcessCreate onmatch="include"></ProcessCreate>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <FileCreateTime onmatch="include"></FileCreateTime>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <NetworkConnect onmatch="include"></NetworkConnect>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <ProcessTerminate onmatch="include"></ProcessTerminate>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <DriverLoad onmatch="include"></DriverLoad>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <RawAccessRead onmatch="include"></RawAccessRead>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
      <ProcessAccess onmatch="include">
        <CallTrace condition="contains"
name="technique_id=T1047,technique_name=Windows Management
Instrumentation,function_name=ProviderExecMethod">C:\Windows\SYSTEM32\framedynos.dll+
2b496</CallTrace>
        <CallTrace condition="contains"
name="technique_id=T1047,technique_name=Windows Management
Instrumentation,function_name=CWbemProviderGlueExecMethodAsync">C:\Windows\SYSTEM32\fr
amedynos.dll+2cb3e</CallTrace>
        <CallTrace condition="contains any"
name="technique_id=T1059.005,technique_name=Command and Scripting
Interpreter,function_name=WSHellExec">C:\Windows\System32\wshom.ocx+c8a0;C:\Windows\S
ystem32\wshom.ocx+c39d</CallTrace>
        <CallTrace condition="contains"
name="technique_id=T1059.005,technique_name=Command and Scripting
Interpreter,function_name=CSShellExecute_ExecuteAssoc">C:\Windows\System32\SHELL32.dl
l+9b5bd</CallTrace>
        <CallTrace condition="contains"
name="technique_id=T1059.005,technique_name=Command and Scripting
Interpreter,function_name=CSShellExecute_DoExecute">C:\Windows\System32\SHELL32.dll+a
e3b9</CallTrace>
      <Rule groupRelation="and" name="technique_id=T1059.005,technique_name=Command
and Scripting Interpreter,function_name=CreateProcessW">

```

```

        <SourceImage condition="contains">C:\Program Files\Microsoft
Office\Root\Office16</SourceImage>
        <CallTrace
condition="contains">C:\Windows\System32\KERNELBASE.dll+76516</CallTrace>
        </Rule>
        <CallTrace condition="contains"
name="technique_id=T1003.001,technique_name=Credential
Dumping,function_name=MiniDumpWriteDump">C:\Windows\SYSTEM32\dbgcore.DLL+6cfb</CallTr
ace>
        <CallTrace condition="contains"
name="technique_id=T1003.001,technique_name=Credential
Dumping,function_name=PssCaptureSnapshot">C:\Windows\System32\KernelBase.dll+de67e</C
allTrace>
        </ProcessAccess>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <FileCreate onmatch="include"></FileCreate>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <RegistryEvent onmatch="include"></RegistryEvent>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <FileCreateStreamHash onmatch="include"></FileCreateStreamHash>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <PipeEvent onmatch="include"></PipeEvent>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <WmiEvent onmatch="include"></WmiEvent>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <DnsQuery onmatch="include"></DnsQuery>
    </RuleGroup>
    <RuleGroup name="" groupRelation="or">
        <FileDelete onmatch="include"></FileDelete>
    </RuleGroup>
</EventFiltering>
</Sysmon>

```

References & Thanks

- <https://github.com/jsecurity101/Windows-API-To-Sysmon-Events>
- <https://github.com/trustedsec/SysmonCommunityGuide>
- <https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon>
- https://zeronights.ru/wp-content/themes/zeronights-2019/public/materials/3_ZN2019__andrej_skablonskijThreatHuntingInCalltrace.pdf
- <https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90010>
- <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>

- <https://docs.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump>
- <https://redcanary.com/blog/testing-initial-access-with-generate-macro-in-atomic-red-team/>
- <https://www.ired.team/offensive-security/code-execution/t1117-regsvr32-aka-squiblydoo>
- <https://github.com/itm4n/VBA-RunPE>
- <https://github.com/olafhartong/sysmon-modular>

Special thank you to <https://twitter.com/TheHack3r4chan> & <https://twitter.com/malwaresoup> from the ThreatHunting Slack for lending their time and expertise to help me talk through the ideas and concepts discussed in this post.



Anton Ovrutsky

Anton is a BSides Toronto speaker, C3X volunteer, and an OSCE, OSCP, CISSP, CSSP certificate holder. Anton enjoys the defensive aspects of cybersecurity and loves logs and queries.

www.lares.com/

