Detecting Parent PID Spoofing - F-Secure Blog

By 21.12.18 8 min. read

Published: 2018-12-21 · Archived: 2026-04-29 02:04:03 UTC

One of the most useful techniques hunt teams can use for detecting anomalous activity is the analysis of parent-child process relationships. However, more capable adversaries can bypass this using Parent PID (PPID) Spoofing allowing the execution of a malicious process from an arbitrary parent process. While this technique itself is not new, having been covered by Cobalt Strike [1] and Didier Stevens [2] [3], very little research has been done in detecting such attacks.

In this blog, we will explore how this technique works and how defenders can utilize Event Tracing for Windows (ETW) to detect this technique. We'll also release a proof of concept PowerShell script to perform PPID spoofing and DLL injection, as well as a Python script that makes use of the pywintrace [8] library to detect this activity.

Why spoof in the first place?

In the past attackers were often able to move through networks undetected, but with the rise of EDR and threat hunting the tables have started to turn. The use of parent-child process analysis in particular has been a useful technique in detecting anomalous activity generated during nearly every phase of a kill chain.

Some examples we use at Countercept:

– Macro payload delivery – WinWord spawning processes

– JS/VBS C# payload delivery – cscript spawning csc

– Lateral movement – services/wmiprvse spawning new processes

This has pushed attackers to re-evaluate their approach and look at techniques like PPID spoofing in order to bypass modern defensive teams.

Spoofing via CreateProcessA

There are a number of different ways to spoof process parents. In this post we're going to focus on one of the simplest and most commonly used techniques involving the API call CreateProcessA.

CreateProcessA, unsurprisingly, lets users create new processes and by default, processes will be created with their inherited parent. However, this function also supports a parameter called "lpStartupInfo" where you can essentially define the parent process you want to use. This feature was first introduced in Windows Vista with the addition of UAC in order to correctly set parents.

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Figure 1

At a deeper technical level, the lpStartupInfo parameter points to a STARTUPINFOEX structure [5].This structure contains an lpAttributeList and using UpdateProcThreadAttribute you can set the process parent via the "PROC_THREAD_ATTRIBUTE_PARENT_PROCESS" attribute.

PROC_THREAD_ATTRIBUTE_PARENT_PROCESS	The *lpValue* parameter is a pointer to a handle to a process to use instead of the calling process as the parent for the process being created. The process to use must have the **PROCESS_CREATE_PROCESS** access right. Attributes inherited from the specified process include handles, the device map, processor affinity, priority, quotas, the process token, and job object. (Note that some attributes such as the debug port will come from the creating process, not the process specified by this handle.)

Figure 2

As an aside, this approach can also be used for privilege escalation. The documentation alludes to this stating that "Attributes inherited from the specified process include handles, the device map, processor affinity, priority, quotas, the process token and job object." Adam Chester has a blog that demonstrates how this can be abused to get SYSTEM in Windows [6].

Have I lied to my parents?

One of the most common ways to gain a foothold on a network is using a malicious macro document. Many payloads will often launch new processes, such as cmd, PowerShell, regsvr32 or certutil; Figure 3 shows an example of this with rundll32 spawning from winword. However, this behavior is relatively anomalous and can easily be detected by most modern blue-teams.

Figure 3

To overcome this issue, an attacker could instead use a VBS macro implementation of the CreateProcessA technique to launch a payload from an expected parent process (e.g. Explorer launching cmd) to blend in with the environment. Figure 4 gives a glimpse into how this could be implemented.

Figure 4

We won't be releasing this VBS code; however, more information can be found here [7].

But, can we take this even further to avoid common Windows utilities completely? One option would be to use some form of DLL or memory injection to load a payload within an already running process.

To illustrate this we created a PowerShell script based on code from Didier Stevens [3] that can be used to create a process with a spoofed parent and then inject a DLL within it.

```
# ppid spoofing and process spoofing happening now. $result1 = [Kernel32]::InitializeProcThreadAttributeList([IntPtr]::Zero, 1, 0, [ref]$lpSize) $sInfoEx.lpAttributeList = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($lpSize) $result1 = [Kernel32]::InitializeProcThreadAttributeList($sInfoEx.lpAttributeList, 1, 0, [ref]$lpSize) $result1 = [Kernel32]::UpdateProcThreadAttribute($sInfoEx.lpAttributeList, 0, 0x00020000, # PROC_THREAD_ATTRIBUTE_PARENT_PROCESS = 0x00020000 $lpValue, [IntPtr]::Size, [IntPtr]::Zero, [IntPtr]::Zero) $result1 = [Kernel32]::CreateProcess($spawnTo, [IntPtr]::Zero, [ref]$SecAttr, [ref]$SecAttr, 0, 0x08080004, #EXTENDED_STARTUPINFO_PRESENT | CREATE_NO_WINDOW | CREATE_SUSPENDED
```

0x00080010, # EXTENDED_STARTUPINFO_PRESENT | CREATE_NEW_CONSOLE (This will show the window of the process) [IntPtr]::Zero, $GetCurrentPath, [ref] $sInfoEx, [ref] $pInfo)

Figure 5

Load and execute dll into spoofed process. $loadLibAddress = [Kernel32]::GetProcAddress([Kernel32]::GetModuleHandle("kernel32.dll"), "LoadLibraryA") $lpBaseAddress = [Kernel32]::VirtualAllocEx($pInfo.hProcess, 0, $dllPath.Length, 0x00003000, 0x4) $result1 = [Kernel32]::WriteProcessMemory($pInfo.hProcess, $lpBaseAddress, (New-Object "System.Text.ASCIIEncoding").GetBytes($dllPath), $dllPath.Length, [ref]0) # $result1 = [Kernel32]::VirtualProtectEx($pInfo.hProcess, $lpBaseAddress, $dllPath.Length, 0x20, [ref]0) $result1 = [Kernel32]::CreateRemoteThread($pInfo.hProcess, 0, 0, $loadLibAddress, $lpBaseAddress, 0, 0)

Figure 6

To demonstrate how such a script could be used to hide activity we looked at commonly running processes on Windows 10. One quite common legitimate relationship we saw was "svchost.exe" launching "RuntimeBroker.exe" (Figure 7).

Figure 7

Using the PowerShell script we were able to mimic this activity and forcibly spawn a legitimate “RuntimeBroker.exe” from “svchost.exe”, then inject and execute our DLL payload.

Figure 8

This vector shows how such techniques can potentially bypass detection rules focused on parent-child relationships.

How can we find the liar?

In the previous section we showed that the CreateProcessA technique can spoof parent IDs and from a blue team perspective if you queried running processes with something like Task Manager or Process Explorer you would see the spoofed IDs. But is there some way to find out the real IDs?

One of the best forensic data sources in Windows is Event Tracing for Windows (ETW). ETW provides a real-time stream of data about events occurring on a system and is something we use within our endpoint agent at Countercept.

The Microsoft-Windows-Kernel-Process provider in particular can give some useful insights into process creation and can help us detect the process ID spoofing. In the example below you’ll see how “rundll32.exe” (PID 5180) is being spawned from “winword.exe” (PID 9224) (Figure 9).

Name	PID	CPU	I/O total r	Private by	User name
System Idle Process	0	82.38		56 kB	NT AUTHO
Registry	88			1.7 MB	
csrss.exe	452			1.56 MB	
wininit.exe	528			1.36 MB	
csrss.exe	544	1.05	1.48 kB/s	1.85 MB	
winlogon.exe	624			2.45 MB	
fontdrvhost.exe	776			3.68 MB	
dwm.exe	400	3.24		136.66 MB	
explorer.exe	4264	2.90	560 B/s	46.97 MB	DESKTOP-
MSASCuiL.exe	9212			1.87 MB	DESKTOP-
vmtoolsd.exe	732	0.13	760 B/s	19.12 MB	DESKTOP-
chrome.exe	8364	0.13	1.03 kB/s	90.25 MB	DESKTOP-
ProcessHacker.exe	6868	0.84		21.17 MB	DESKTOP-
cmd.exe	6256			2.24 MB	DESKTOP-
WINWORD.EXE	9224			34.16 MB	DESKTOP-
rundll32.exe	5180	0.07		1.54 MB	DESKTOP-

Figure 9

Looking into the ETW data collected (Figure 10) you'll see there are multiple ProcessId fields, including the EventHeader ProcessId, as well as the actual event ProcessID and ParentProcessID. Although this is somewhat confusing, reading through the MSDN documentation, we find that the EventHeader ProcessId actually identifies the process that generated the event – i.e. the parent.


```
(1,
{'EventHeader': {'Size': 234, 'HeaderType': 0, 'Flags': 576, 'EventProperty': 0, 'ThreadId': 8956,
'ProcessId': 9224, 'TimeStamp': 131877949259530100, 'ProviderId': '{
            22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716
        }', 'EventDescriptor': {'Id': 1, 'Version': 2, 'Channel': 16, 'Level': 4, 'Opcode': 1, 'Task': 1,
         'Keyword': 9223372036854775824
        }, 'KernelTime': 46, 'UserTime': 47, 'ActivityId': '{
            00000000-0000-0000-0000-000000000000
        }'
    }, 'Task Name': 'PROCESSSTART', 'ProcessID': '5180', 'CreateTime':
    '\u200e2018\u200e-\u200e11\u200e-\u200e27T12: 22: 05.952961900Z', 'ParentProcessID': '9224',
    'SessionID': '1', 'Flags': None, 'ImageName':
    '\\Device\\HarddiskVolume2\\Windows\\System32\\rundll32.exe', 'ImageChecksum': '0x26962',
    'TimeDateStamp': '0x96E0391B', 'PackageFullName': '', 'PackageRelativeAppId': '', 'Description':
    'Process %1 started at time %2 by parent %3 running in session %4 with name %6. '
})
```


Figure 10

And in this legitimate example you'll notice that the EventHeader ProcessId and ParentProcessId correctly match.

In this second example we executed our malicious PowerShell script and spawned a "RuntimeBroker.exe" (PID 4976) from "svchost.exe" (PID 4652).

Name	PID	CPU	I/O total r	Private by	User na
ManagementAgentHost.exe	3172	0.08	1.21 kB/s	5.6 MB	
MsMpEng.exe	3220	0.78	54.41 kB/s	145.77 MB	
svchost.exe	3292			4.29 MB	
svchost.exe	3376			1.95 MB	
svchost.exe	3528			2.43 MB	
svchost.exe	3680			1.33 MB	
svchost.exe	3712			3.43 MB	
dllhost.exe	4148			3.72 MB	
svchost.exe	4208			1.95 MB	
svchost.exe	4652			5.03 MB	DESKTO
RuntimeBroker.exe	4976			1.41 MB	DESKTO
svchost.exe	4696			6.04 MB	DESKTO
svchost.exe				1.72 MB	
ctfmon.exe				5.5 MB	DESKTO
svchost.exe				2.92 MB	
NisSrv.exe				5.33 MB	
msdtc.exe				2.81 MB	
svchost.exe				3.83 MB	
svchost.exe				2.16 MB	
SearchIndexer.ex				32.21 MB	
svchost.exe	8504			1.98 MB	

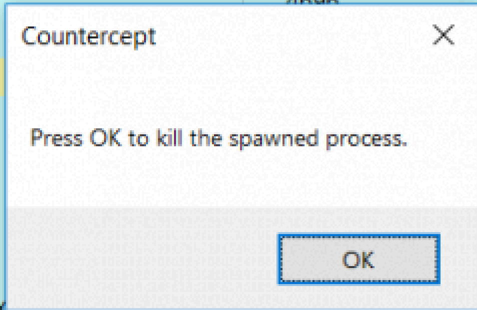

Figure 11

As before, we can see ETW generated a process event (Figure 12); however, this time the EventHeader ProcessId and ParentProcessID are different. And in fact the EventHeader ProcessId shows the real parent, which is “winword.exe” (PID 9224). We’ve just caught someone performing ParentPID spoofing!


```
(1,
{'EventHeader': {'Size': 234, 'HeaderType': 0, 'Flags': 576, 'EventProperty': 0, 'ThreadId': 8956,
'ProcessId': 9224, 'TimeStamp': 131877950582485384, 'ProviderId': '{
        22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716
    }', 'EventDescriptor': {'Id': 1, 'Version': 2, 'Channel': 16, 'Level': 4, 'Opcode': 1, 'Task': 1,
     'Keyword': 9223372036854775824
    }, 'KernelTime': 47, 'UserTime': 47, 'ActivityId': '{
        00000000-0000-0000-0000-000000000000
    }'
}, 'Task Name': 'PROCESSSTART', 'ProcessID': '4976', 'CreateTime':
'\u200e2018\u200e-\u200e11\u200e-\u200e27T12: 24: 18.248513600Z', 'ParentProcessID': '4652',
'SessionID': '1', 'Flags': None, 'ImageName':
'\\Device\\HarddiskVolume2\\Windows\\System32\\RuntimeBroker.exe', 'ImageChecksum': '0x26962',
'TimeDateStamp': '0x96E0391B', 'PackageFullName': '', 'PackageRelativeAppId': '', 'Description':
'Process %1 started at time %2 by parent %3 running in session %4 with name %6. '
})
```


Figure 12

However, as always in threat detection, things aren’t that simple and if you attempt to do this at scale you’re going to see false positives from legitimate spoofing. One common example is User Account Control (UAC), which is used to elevate process privileges. In Windows 10 when UAC executes, the Application Information service (through svchost) is used to launch the elevated process, but will then spoof the parent to show the original caller.

The example below shows how an elevated cmd.exe will show explorer.exe as the parent, when in fact it was svchost.exe.

t	ParentProcessNormalised	C:\Windows\explorer.exe
t	Pid	5472
t	Process	C:\Windows\System32\cmd.exe
t	ProcessNormalised	C:\Windows\System32\cmd.exe
t	RealParentArgs	c:\windows\system32\svchost.exe -k netsvcs -p -s Appinfo

Figure 13

Another false positive we saw involved crash handling with WerFault. In the example below, when MicrosoftEdge crashed svchost was used to launch WerFault.exe and the parent was spoofed as MicrosoftEdge.exe.

t	ParentProcessNormalised	C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe
t	Pid	21704
t	Process	C:\Windows\System32\WerFault.exe
t	ProcessNormalised	C:\Windows\System32\WerFault.exe
t	RealParentArgs	C:\WINDOWS\System32\svchost.exe -k WerSvcGroup

Figure 14

For testing purposes we created a simple proof-of-concept Python script that used pywintrace to log events from ETW, compare PIDs, and then filter results to remove false positives (Figure 15).

```
λ python Detect-PPID-Spoof2.py
Spoofed parent process detected!!!
        RuntimeBroker.exe(24160) is detected with parent svchost.exe(5808)
but originally from parent powershell.exe(11608).
```

Figure 15

The code for the PowerShell spoofing script, as well as the detection script, [can be found on our Github](#).

Conclusion

In this post we've shown how attackers can abuse legitimate Windows functions to fool blue teamers and potentially bypass detection techniques based on parent-child relationships.

From a defensive perspective though, we've shown how analysis of ETW process events can easily highlight anomalous parent spoofing and help discover the true origin of a process. As always this research shows how essential it is for defenders to push the boundaries of current technology and stay one step ahead of attackers at all times.

References

[1] https://blog.cobaltstrike.com/2017/05/23/cobalt-strike-3-8-whos-your-daddy/

[2] https://blog.didierstevens.com/2017/03/20/

[3] https://blog.didierstevens.com/2009/11/22/quickpost-selectmyparent-or-playing-with-the-windows-process-tree/

[4] https://msdn.microsoft.com/en-us/library/windows/desktop/aa363759(v=vs.85).aspx

[5] https://docs.microsoft.com/en-us/windows/desktop/api/winbase/ns-winbase-_startupinfoexa

[6] https://blog.xpnsec.com/becoming-system/

[7] http://www.pwncode.club/2018/08/macro-used-to-spoof-parent-process.html

[8] https://github.com/fireeye/pywintrace

Source: https://web.archive.org/web/20200726110643/https://blog.f-secure.com/detecting-parent-pid-spoofing/