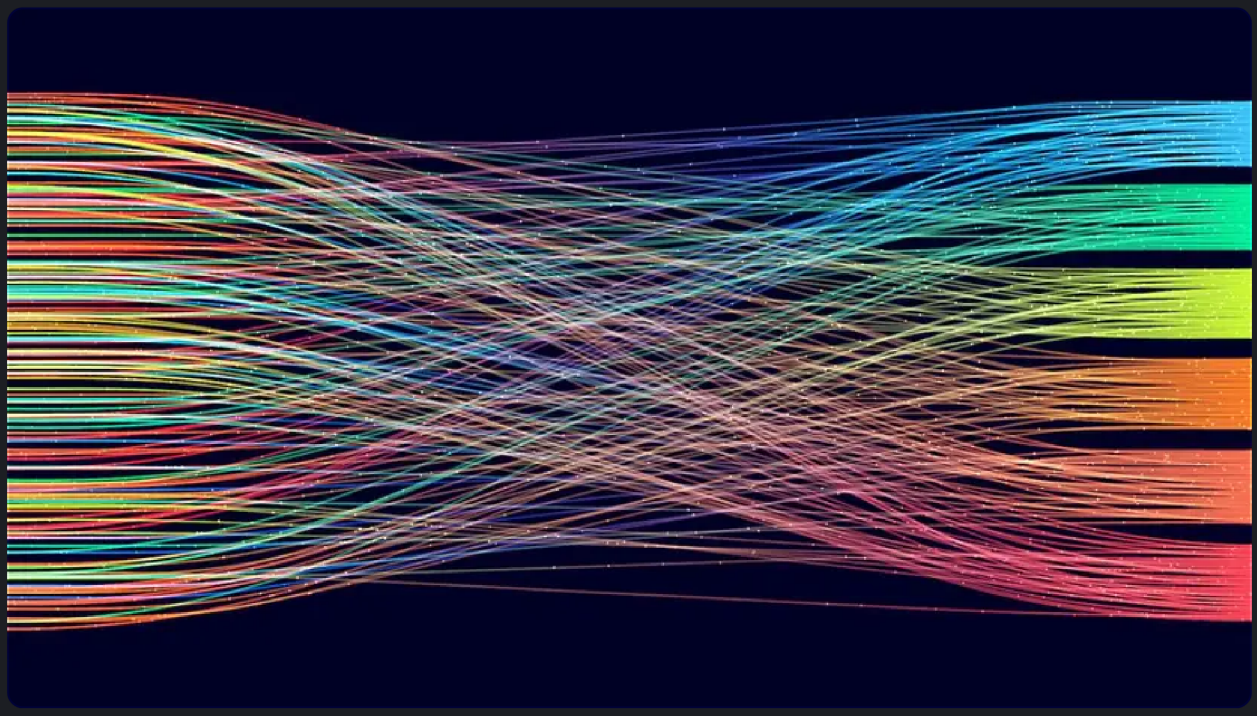
29 MARCH 2023

JOHN UHLMANN

Effective Parenting - detecting LRPC- based parent PID spoofing

Using process creation as a case study, this research will outline the evasion-detection arms race to date, describe the weaknesses in some current detection approaches and then follow the quest for a generic approach to LRPC-based evasion.

17 min read Security research



Adversaries currently utilize **RPC**'s client-server architecture to obfuscate their activities on a host – including **COM** and **WMI** which are both RPC-based. For example, a number of local

RPC servers will happily launch processes on behalf of a malicious client - and that form of defense evasion is difficult to flag as malicious without being able to correlate it with the client.

The above annotated screenshot is the logical process tree after a Microsoft Word macro called three COM objects, each exposing a `ShellExecute` interface and also the WMI `Win32\_Process::Create` method. The WMI call has specialized telemetry that can reconstruct that Microsoft Word initiated the process creation (the blue arrow), but the COM calls don't (the red arrows). So defenders have no visibility that Microsoft Word made a COM call over an RPC call to spawn PowerShell elsewhere on the system.

The defender is left with a challenge to interpretation because of this lack of context - Word spawning PowerShell is a red flag, but is *Explorer* spawning PowerShell malicious, or simply user behavior?

RPC will typically use **LRPC** as the transport for inter-process communication. Using process creation as a case study, this research will outline the evasion-detection arms race to date, describe the weaknesses in some current detection approaches and then follow the quest for a generic approach to LRPC-based evasion.

A Brief History of Child Process Evasion

It is often very beneficial for adversaries to spawn child processes during intrusions. Using legitimate pre-installed system tools to achieve your aims saves on capability development time and can potentially evade security instrumentation by providing a veneer of legitimacy for the activity.

However, for the activity to look plausibly legitimate, the parent process also needs to seem plausible. The classic counter-example is that Microsoft Word spawning PowerShell is highly anomalous. In fact, Elastic SIEM includes a prebuilt rule to detect **suspicious MS Office child processes** and Elastic Endpoint will also **prevent malicious execution**. As documented in the Elastic **Global Threat Report**, suspicious parent/child relationships was one of the three most common defense evasion techniques used by threats in 2022.

Endpoint Protection Platform (EPP) products could prevent the most egregious process parent relationships, but it was the rise of Endpoint Detection and Response (EDR) approaches with pervasive process start logging and the ability to retrospectively hunt that

established a scalable approach to anomalous process tree detection.

Adversaries initially pivoted to evasions using a **Win32 API feature introduced in Windows Vista** to support User Account Control (UAC) that allows a process to specify a different logical parent process to the real calling process. However, **endpoint security could still identify the real parent process** based on the calling process context during the **process creation notification callback**, and **detection rule** coverage was quickly re-established.

New evasion techniques evolved in response, and a common method currently leveraged by adversaries is to indirectly spawn child processes via RPC – including **DCOM** and **WMI** which are both RPC-based. RPC can be either inter-host or simply inter-process. The latter is oxymoronically called Local Remote Procedure Call (LRPC).

The most well-known of these was the `Win32\_Process::Create` WMI method. In order to detect this, Microsoft appears to have explicitly added a new `Microsoft-Windows-WMI-Activity` ETW event in Windows 10 1809. The new event 23 included the client process id - the missing data point needed to associate the activity with a requesting client.

Unfortunately adversaries were quickly able to pivot to alternate process spawning out-of-process RPC servers such as `MMC20.Application::ExecuteShellCommand`. Waiting for Microsoft to add telemetry to dual-purpose out-of-process RPC servers **one-by-one** wasn't going to be a viable detection approach, so last year we set out on a side quest to generically associate LRPC server actions with the requesting LRPC client process.

Detecting LRPC provenance

The majority of previous public RPC telemetry research has focused on inter-host lateral movement – typically spawning a process on a remote host. For example: - **Lateral Movement using the MMC20.Application COM Object**- **Lateral Movement via DCOM: Round 2**- **Endpoint Detection of Remote Service Creation and PsExec** - **Utilizing RPC Telemetry**- **Detecting Lateral Movement techniques with Elastic** - **Stopping Lateral Movement via the RPC Firewall**

The ultimate advice for defenders is typically to monitor RPC network traffic for anomalies or, better yet, to block unnecessary remote access to RPC interfaces with **RPC Filters** (part of the **Windows Filtering Platform**) or specific RPC methods with 3rd party tooling like **RPC Firewall**.

Unfortunately these approaches don't work when the adversary uses RPC to spawn a process elsewhere on the same host. In this case, the RPC transport is typically **ALPC** - monitoring and filtering at the network layer does not then apply.

On the host, detection engineers typically look to leverage telemetry from the inbuilt Event Tracing (including EventLog) in the first instance. If this proves insufficient, then they can investigate custom approaches such as user-mode function hooking or mini-filter drivers.

In the RPC case, `Microsoft-Windows-RPC` ETW events are very useful for identifying anomalous behaviours.

Especially: - Event 5 - `RpcClientCallStart` (GUID InterfaceUuid, UInt32 ProcNum, UInt32 Protocol, UnicodeString NetworkAddress, UnicodeString Endpoint, UnicodeString Options, UInt32 AuthenticationLevel, UInt32 AuthenticationService, UInt32 ImpersonationLevel) - Event 6 - `RpcServerCallStart` (GUID InterfaceUuid, UInt32 ProcNum, UInt32 Protocol, UnicodeString NetworkAddress, UnicodeString Endpoint, UnicodeString Options, UInt32 AuthenticationLevel, UInt32 AuthenticationService, UInt32 ImpersonationLevel)

Additionally, `RpcClientCallStart` is generated by the client and `RpcServerCallStart` by the server so the ETW headers will provide the client and server process ids respectively. Further, there is a 1:1 mapping between endpoint addresses and server process ids. So the server process can be inferred from the `RpcClientCallStart` event.

The RPC interface UUID and Procedure number combined with the caller details are (usually) sufficient to identify intent. For example, RPC interface UUID `{367ABB81-9844-35F1-AD32-98F038001003}` is the Service Control Manager Remote Protocol which exposes the ability to configure Windows services. The 12th procedure in this interface is `RCreateServiceW` which notoriously is the method that PsExec uses to execute processes on remote systems.

For endpoint security vendors, however, there are a few issues to address before scalable robust `Microsoft-Windows-RPC` detections would be possible: 1. RPC event volumes are significant 2. There isn't an obvious mechanism to strongly correlate a client call with the resultant server call 3. There isn't an obvious mechanism to strongly correlate a server call with the resultant server behavior

Let's address these three issues one by one.

LRPC event volumes

There are thousands of LRPC events each second – and most of them are uninteresting. To address the LRPC event volume concern, we could limit the events to just those RPC events that are inter-process (including inter-host). However, this immediately leads to the second concern. We need to identify the client of each server call in order to reduce event volumes down to just those which are inter-process.

Correlating RPC server calls with their clients

Modern Windows RPC has roughly three **transports**: - TCP/IP (nacn_ip_tcp, nacn_http, ncadg_ip_udp and nacn_np over SMB) - inter-process Named Pipes (direct nacn_np) - inter-process ALPC (ncalrpc)

The `RpcServerCallStart` event alone is not sufficient to determine if the call was inter-process. It needs to be correlated against a preceding `RpcCientCallStart` event, and **this correlation** is unfortunately weak. At best you can identify a pair of `RpcServerCall` start/stop events that are bracketed by a pair of `RpcClientCall` events with the same parameters. (Note - for performance reasons, ETW events generated from different threads may arrive out of order). This means that you need to maintain a holistic RPC state - which creates an on-host storage and processing volume concern in order to address the event volume concern.

More importantly though, the `RpcClientCallStart` events are generated in the client process where an adversary has already achieved execution and therefore can be **intercepted with very little effort**. There is little point to implementing a detection for something so trivial to circumvent, especially when there are more effective options.

Ideally, the RPC server would access the client details and directly log this information. Unfortunately, the ETW events don't include this information - which is not surprising since one of the RPC design goals was simplification through abstraction. The RPC runtime (allegedly) can be configured via Group Policy to do exactly this, though. It can store **RPC State Information** which can then be used during debugging to **identify the client caller from the server thread**. Unfortunately the Windows XP era documentation didn't immediately work for Windows 10.

It did provide a rough outline describing how to address the first two problems: reducing event volumes and correlating server calls to client processes. It is possible to hook the RPC

runtime in all RPC servers, account for the various transports, and then log or filter inter-process RPC events only. (This is likely akin to how **RPC Firewall**handles network RPC - just with local endpoints).

Correlating RPC server calls and resultant behavior

The next problem was how to correctly attribute a specific server call to the resultant server behaviour. On a busy server, how could we tie an opaque call to the `ExecuteShellCommand` method to a specific process creation event? And what if the call came from script-based malware and was further wrapped under a method like `IDispatch::Invoke` ?

We didn't want to have to inspect the RPC parameter blob and individually implement parsing support for each abusable RPC method.

Introducing ETW's ActivityId

Thankfully, Microsoft had already thought of this scenario and **provides ETW tracing guidance** to developers.

They suggest that developers generate and propagate a unique 128-bit `ActivityId` between related ETW events to enable end-to-end tracing scenarios. This is typically handled automatically by ETW for events generated on the same thread as the value is stored in thread local storage. However, the developer must manually propagate this ID to related activities performed by other threads... or processes. As long as the RPC Runtime and all Microsoft RPC servers had followed ETW tracing best practices, we should finally have the end-to-end correlation we want!

It was time to break out a decompiler (we like Ghidra but there are many options) and inspect rpcrt4.dll. By looking at the first parameter passed to `EventRegister` calls, we can see that there are three ETW GUIDs in the RPC runtime. These GUIDs are defined in a contiguous block and helpfully came with public symbols.

These GUIDs correspond to `Microsoft-Windows-RPC` , `Microsoft-Windows-Networking-Correlation` and `Microsoft-Windows-RPC-Events` respectively. Further, the RPC runtime helpfully wraps calls to `EventWrite` in just two places.

The first call is in `McGenEventWrite\_EtwEventWriteTransfer` and looks like this:

```
`EtwEventWriteTransfer` (RegHandle, EventDescriptor, NULL, NULL, UserDataCou
```

The NULL parameters mean that `ActivityId` will always be the configured per-thread `ActivityId` and `RelatedActivityId` will always be excluded in events logged by this code path.

The second call is in `EtwEx\_tidActivityInfoTransfer` and looks like this:

```
`EtwEventWriteTransfer` (Microsoft_Windows_Networking_CorrelationHandle, Eve
```

This means that `RelatedActivityId` will only ever be logged in `Microsoft-Windows-Networking-Correlation` events. RPC Runtime `ActivityId` s are (predominantly) created within a helper function that ensures that this correlation is always logged.

Decompilation also revealed that the RPC runtime allocates ETW `ActivityId` s by calling `UuidCreate` , which generates a random 128-bit value. This is done in locations such as `NdrAysncClientCall` and `HandleRequest` . In other words, the client and server both individually allocate `ActivityId` s. This isn't unsurprising because the DCE/RPC specification doesn't seem to include a transaction id or similar construct which would allow the client to propagate an ActivityId to the server. That's okay though: we're only currently missing the correlation between server call and the resultant behaviour. Also we don't want to trust any potentially tainted client-supplied information.

So now we know exactly how RPC intends to correlate activities triggered by RPC calls- by setting the per-thread ETW `ActivityId` and by logging RPC ActivityId correlations to `Microsoft-Windows-Networking-Correlation` . The next question is whether the Microsoft RPC interfaces that support dual-purpose activities, such as process spawning, propagate the `ActivityId` appropriately.

We looked at the execution traces for the four indirect process creation examples from our initial case study. In each one, the RPC request was received on one thread, a second thread handled the request and a third thread spawned the process. Other than the timing, there appeared to be no possible mechanism to link the activities.

Unfortunately, while the RPC subsystem is well behaved, most RPC servers aren't – though this likely isn't entirely their fault. The `ActivityId` is only preserved per-thread so if the server uses a worker thread pool (as per Microsoft's **RPC scalability** advice) then the

causality correlation is implicitly broken.

Further, kernel ETW events seem to universally log an `ActivityId` of `{00000000-0000-0000-0000-000000000000}` – even when the thread has a (user-mode) `ActivityId` configured. It is likely that the kernel implementation of `EtwWriteEvent` simply does not query the `ActivityId` which is stored in user-mode thread local storage.

This observation about kernel events is a showstopper for a generic approach based around ETW. Almost all of the interesting resultant server behaviors (process, registry, file etc) are logged by kernel ETW events.

A new approach was necessary. It isn't scalable to investigate individual ETW providers in dual-purpose RPC servers. (Though the `Microsoft.Windows.ShellExecute` TraceLogging provider looked interesting). What would Microsoft do?

What would Microsoft do?

More specifically, how does Microsoft populate the `ClientProcessId` in the `Microsoft-Windows-WMI-Activity` ETW event 23 (aka `Win32\_Process::Create`)?

```
`task_023` (UnicodeString CorrelationId, UInt32 GroupOperationId, UInt32 Ope
```

Unlike RPC, WMI natively supports end-to-end tracing via a `CorrelationId` which is a GUID that the WMI client passes to the server at the WMI layer so that WMI operations can be associated. However, for security use cases, we shouldn't blindly trust client-supplied information for reasons previously mentioned.

But how was Microsoft determining the process id to log and was their approach something that could be replicated for other RPC Servers – possibly via an RPC server runtime hook?

We needed to find out where the data in that field came from. ETW conveniently provides the ability to record a stack trace when an event is generated and the **Sealighter** tool conveniently exposes this capability. Sealighter illustrates which specific ETW Write function is being called from which process.

In this case, the event was actually being written by `ntdll!EtwEventWrite` in the WMI Core Service (svchost.exe -k netsvcs -p -s Winmgmt) – not in the WMI Provider Host

(WmiPrvSE.exe).

Putting a breakpoint on `PublishWin32ProcessCreation`, we see via parameter value inspection that the `ClientProcessId` is passed (on the stack) as the 10th parameter. We can then look at `InspectWin32ProcessCreateExecution` to determine how the value that is passed in is determined.

A roughly tidied Ghidra decompilation of `InspectWin32ProcessCreateExecution` might resemble this:

We can see that the client process id comes from the `CWbemNamespace` object. Searching for reference to this structure field, we find that it is only set in `CWbemNamespace::Initialize`. Our earlier stack trace started in `wbemcore!CCoreQueue` and this initialization appears to have occurred prior to queuing. So we could statically search for all locations where the initialization occurs or dynamically observe the actual code paths taken.

We know that this activity is being initiated over RPC, so one approach would be to place breakpoints on RPC send/receive functions in the client and server. An alternative might be to fire up Wireshark and examine the packet capture of the entire interaction when it occurs in cleartext over the network. We learned somewhat late in our research that Microsoft had excellent documentation for the **WMI Protocol Initialization** that explained much of this and might have saved a little time.

We took the first approach. The second parameter to `InspectWin32ProcessCreateExecution` is an `IWbemContext` – which allows the caller to provide additional information to providers. This is how the parameters to `Win32\_Process::Create` are being passed. What if the first parameter was related to the WMI Client passing additional context to the WMI Core?

`IWbemLevel1Login::NTLMLogin` stood out in the call traces as a good place to start looking.

And right next to its COM interface UUID was IWbemLoginClientID[Ex] which had a very interesting `SetClientInfo` call, which was documented on MSDN:

The WMI client calls `wbemprox!SetClientIdentity` which looks roughly like this:

`IWbemLoginClientIDEx` is currently undocumented, but we can infer the parameters from the values passed.

At this point, it looks like the client process is passing `ClientMachineName`, `ClientMachineFQDN`, `ClientProcessId` and `ClientProcessCreationTime` to the WMI Core. We can confirm this by changing the values and seeing if the ETW event logged by the WMI Core changes.

Using WinDbg, we set up a couple quick patches to the WMI client process and then spawned a process via WMI:

```
windbg> bp wbemprox!SetClientIdentity+0xff "eu @rdx \"SPOOFED....\"; gc"
windbg> bp wbemprox!SetClientIdentity+0x1c4 "r r9=0n1337; eu @r8 \"SPOOFED.C
PS> ([wmiclass]"ROOT\CIMv2:Win32_Process").Create("calc.exe")
```

Using SilkETW (or another ETW capture mechanism), we see the following event from the server process:

The server is blindly reporting the values provided by the client. This means that this event cannot be relied upon for un-breaking WMI process provenance trees as the adversary can control the client process id. Falsely reporting this information would be an interesting defense evasion, and a tough one to identify reliably.

Further, a remote adversary can actually pass in a `ClientMachine` name equal to the local hostname and this WMI event will mistakenly log IsLocal as true. (See the earlier decompilation of `InspectWin32ProcessCreateExecution`). This will make the event seem like a suspicious local execution rather than lateral movement, and represents another defence evasion opportunity.

So, this isn't an approach that other RPC servers should follow after all.

Conclusion

In trying to generically solve LRPC provenance, we unfortunately demonstrate that the one existing LRPC provenance data point is unreliable. This has been reported to Microsoft where it was assessed as a next-version candidate bug that will be evaluated for future releases.

Our fervent hope is that the ultimate solution involves the creation of a documented API that allows a server LRPC thread to determine the client thread of a connection. This would provide endpoint security products with a reliable mechanism to identify operations being proxied through LRPC calls in an attempt to hide their origin.

More generally though, this research highlights the need for defenders to have a detailed understanding of data provenance. It is necessary but not sufficient to know that the data was logged by a trustworthy source such as the kernel or a server process. In addition, you must also understand whether the data was intrinsic to the event or provided by a potentially untrustworthy client. Otherwise adversaries will exploit the gaps.