Abstracting Scheduled Tasks

Jonathan Johnson · Follow
Published in Posts By SpecterOps Team Members · 9 min read · Mar 15, 2021

Written by: Jonathan Johnson and Matt Hand

Introduction

Capability Abstraction has been adopted as one of the core components of the research phase within the detection engineering process at SpecterOps. This methodology allows for the combination of both static and dynamic analysis to verify a technology's functionality as it pertains to a specific behavior while limiting assumptions that would otherwise be applied within the detection and response pipeline.

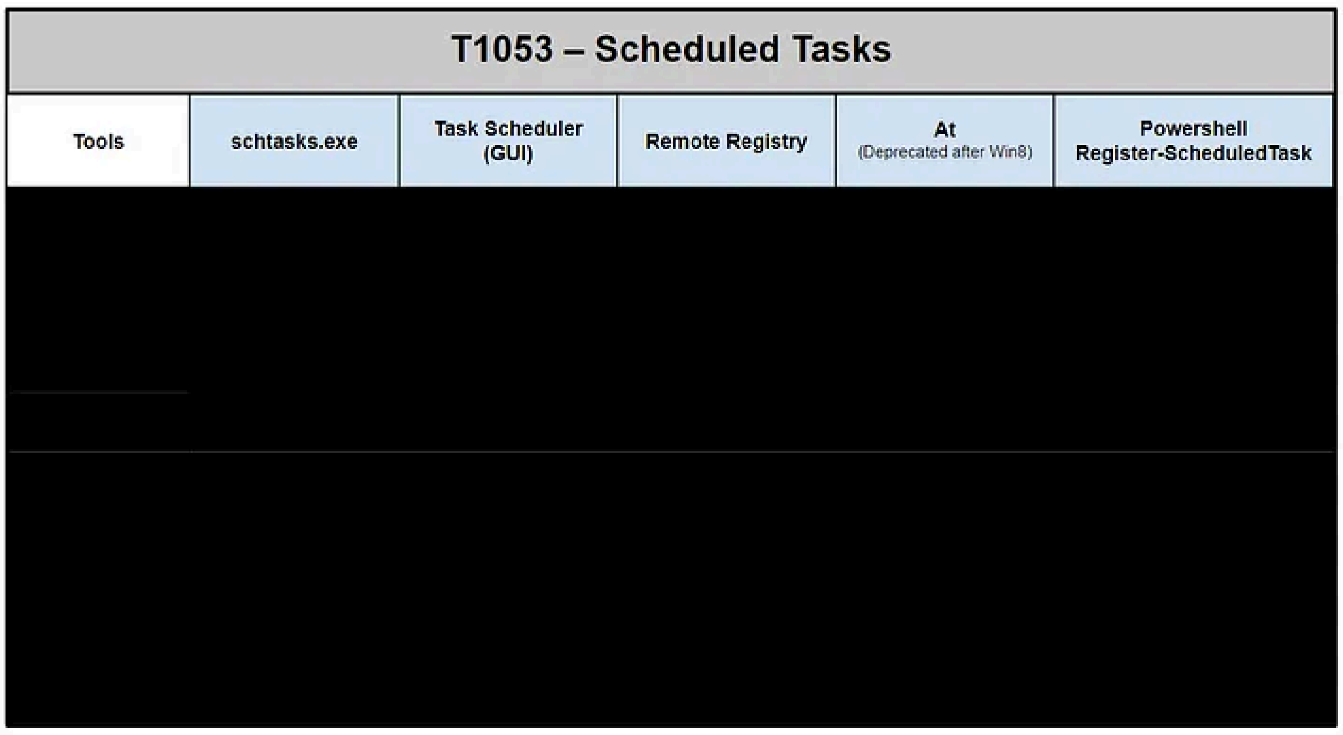
As the team has dived into different behaviors, one that became of interest was Scheduled Tasks. Scheduled tasks allow an attacker to execute code when a specific scheduling condition is met, including both time-based and event-based triggers, and are typically used offensively as a form of persistence on Windows hosts. Additionally, scheduled tasks can be used for privilege escalation in some cases where the attacker can control the target of the trigger itself (i.e. the binary on disk which the scheduled task will execute) or if they can control a task which runs as a more privileged user.

Although this behavior has generally fallen out of favor for offensive use over the past number of years due to increased awareness and wide-scale deployment of detections, it is still actively used by numerous threat actors today, including in SUNSPOT, the implant used during the Solarwinds supply chain compromise. Our interest in this technique was revived after seeing it being actively used in ransomware campaigns with high degrees of success, indicating that our standing detection guidance was insufficient. Because of the significant amount of research previously done in this area, especially around local task creation, we decided to focus on the remote implementations of this behavior.

Also, while this post focuses on scheduled tasks, we more importantly wanted to demonstrate our methodology for performing capability abstractions and show some of the work that goes on behind the scenes to uncover core functionalities behind techniques. Our hope is that you can take some of the tactics outlined in this post and apply them to your internal detection engineering processes.

Starting the Abstraction

Within every abstraction, the first thing to do is to identify the different tools or procedures that could perform the target activity:



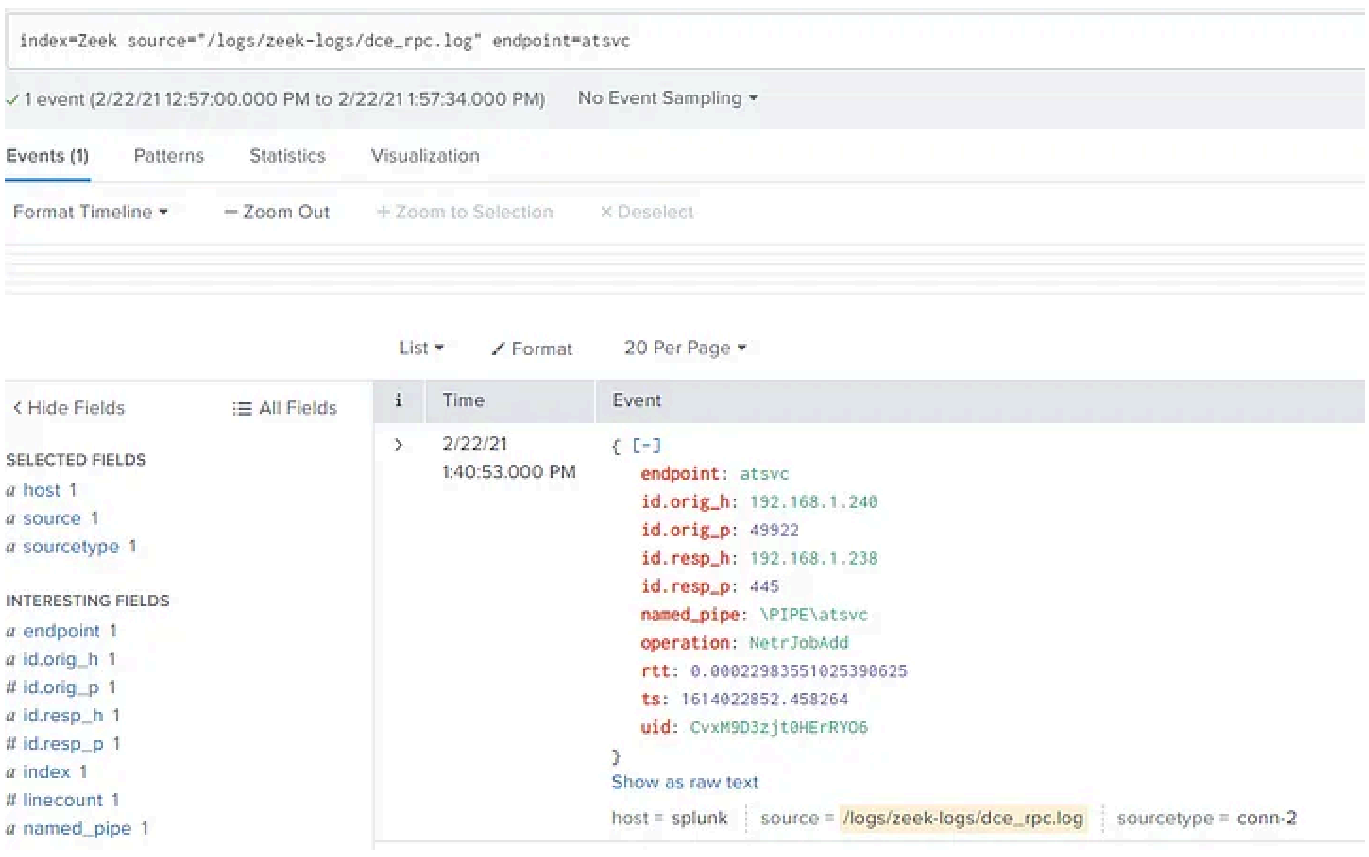
T1053 – Scheduled Tasks					
Tools	schtasks.exe	Task Scheduler (GUI)	Remote Registry	At (Deprecated after Win8)	Powershell Register-ScheduledTask

Keep in mind that there are undoubtedly other tools that create or interact with scheduled tasks, but the goal is to get a diverse grouping that doesn't implement the same functionality in the same way. This approach allows us to remove our tool-specific implementation and instead take a deep dive into the underlying techniques and procedures common amongst all tools. The net effect of this is an understanding of the lowest level of shared functionality with which we can build our detections.

Note: This post will not explore every procedural avenue or tool as shown above in the interest of brevity. Instead, we'll focus on one that was particularly valuable to our research and provided a great deal of insight.

Exploring Remote Task Creation

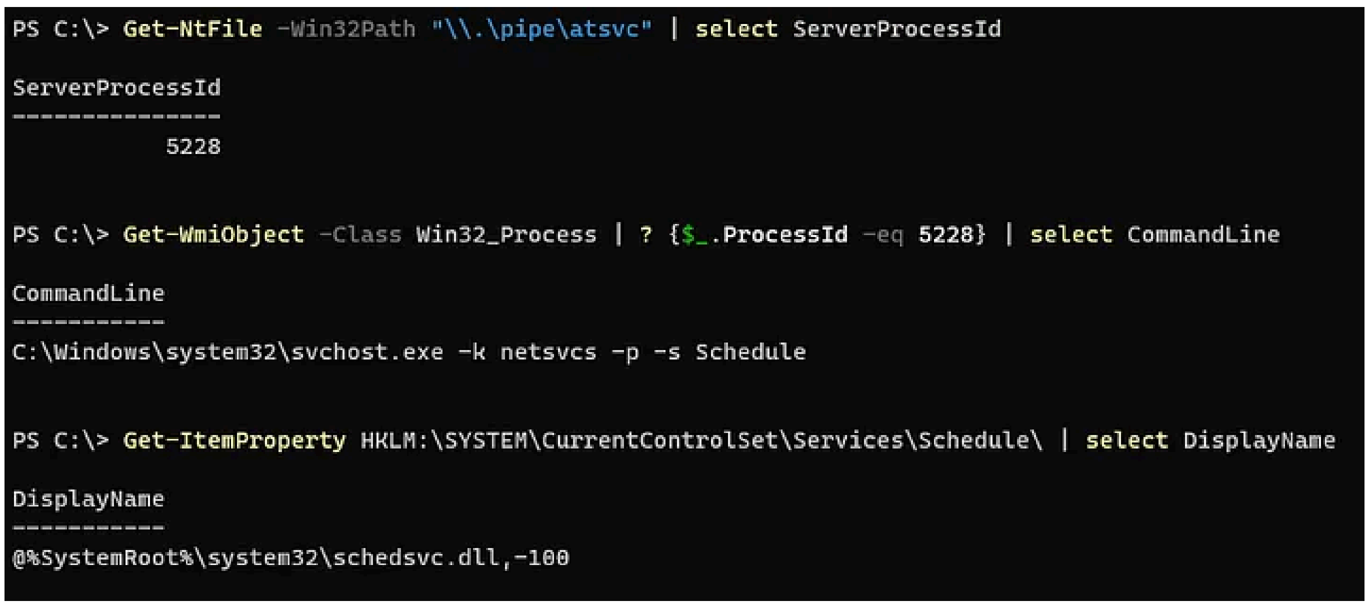
One of the most common technologies that Microsoft uses for communication between operating systems, especially across the machine boundary, is Remote Procedure Call (RPC) and scheduled tasks are no exception. Specifically, the Task Scheduler is backed by the Task Scheduler Remoting Protocol (MS-TSCH). This protocol is backed by three endpoints — ATSvc, SASec, and ITaskSchedulerService. The following screenshot shows RPC telemetry collected via Zeek where the `NetrJobAdd` method is invoked by a remote client and passed to the ATSvc endpoint on another host via its named pipe, `\\.\pipe\atsvc`.



```
index=Zeek source="/logs/zeek-logs/dce_rpc.log" endpoint=atsvc
✓ 1 event (2/22/21 12:57:00.000 PM to 2/22/21 1:57:34.000 PM)    No Event Sampling

2/22/21 1:40:53.000 PM
{ [-]
   endpoint: atsvc
   id.orig_h: 192.168.1.240
   id.orig_p: 49922
   id.resp_h: 192.168.1.238
   id.resp_p: 445
   named_pipe: \PIPE\atsvc
   operation: NetrJobAdd
   rtt: 0.00022983551025390625
   ts: 1614022852.458264
   uid: CvxM9D3zjt0HErRYO6
}
Show as raw text
host = splunk   source = /logs/zeek-logs/dce_rpc.log   sourcetype = conn-2
```

While digging into this a bit deeper, we used NtObjectManager to find that the ATSvc named pipe is served by `taskcomp.dll` which is hosted in the Schedule service, `schedsvc.dll`.



```
PS C:\> Get-NtFile -Win32Path "\\.\pipe\atsvc" | select ServerProcessId

ServerProcessId
---------------
           5228

PS C:\> Get-WmiObject -Class Win32_Process | ? {$_.ProcessId -eq 5228} | select CommandLine

CommandLine
-----------
C:\Windows\system32\svchost.exe -k netsvcs -p -s Schedule

PS C:\> Get-ItemProperty HKLM:\SYSTEM\CurrentControlSet\Services\Schedule\ | select DisplayName

DisplayName
-----------
@%SystemRoot%\system32\schedsvc.dll,-100
```

`Taskcomp.dll` is the "Task Scheduler Backward Compatibility Plug-in" per its file description, which is particularly interesting because this may reference support for the now-deprecated `at.exe` utility. `At.exe` was deprecated in Windows 8 in favor of `schtask.exe` which leverages the newer Task Scheduler 2.0.

Digging into ATSvc

We wanted to investigate how `schedsvc.dll` interacts with `taskcomp.dll` so we loaded it into Ghidra. Immediately, we noticed that `taskcomp` wasn't in the import address table. This means that it must be loaded at a later time — potentially conditionally. We looked for calls to `LoadLibrary()`, which takes a path to the DLL to load as an argument, and found that `taskcomp.dll` was loaded in the `PlugIn::Load` function. Even more interestingly, `taskcomp.dll` is loaded conditionally, meaning that it is only loaded if a specific value is set.

In Ghidra's decompilation for the `Plugin::Load` method, we found that the conditional load is determined by the value stored in an instance of a `JobStore` object plus an offset of `0x40` from the member variable at offset `0x50`. If the value is `0`, an ETW message is generated and sent. If the value is anything other than `0`, the DLL is loaded. After reviewing this function, we wanted to find the source of the value that is being checked.

If we look at the `JobStore` class, there is a function called `InitJobStore` which appears to populate a global instance of the `JobStore` (`m_pCommonStore`) with values. One of the values being populated is the string "`EnableAt`."

The name seemed like it could be related to the `at.exe` deprecation and a quick Google search told us that we were on the right path.

The next step was to verify `schedsvc.dll` was querying the registry value. Rather than comb through Ghidra and deal with weird class member offsets, we opted to just restart the service and capture its behavior with Process Monitor. We implemented a basic filter consisting of the following rules:

- Operation is RegQueryValue
- Path ends with EnableAt
- Process name is svchost.exe

Sure enough, we can see `svchost.exe` querying the value in `EnableAt`.

We opened up the event's call stack and found that the call happens specifically at `schedsvc!ServiceMain+0x5799`.

Back in Ghidra, we navigated to this offset and landed in `JobStore::ReadConfiguration` on a call to `RegQueryExW()`. We saw that the second parameter to `RegQueryExW()`, the registry value to be queried, is derived from the `JobStore` that we just populated. The value that is returned, `lpData`, is then stored `JobStore` at an offset of `0x10 + (int) * 0x18`.

Remember that offset that was checked when deciding whether to load `taskcomp.dll`? Our original thought was that if `uVar2` above is incremented to `2`, then `0x10 + 0x2 * 0x18` comes out to `0x40`, the exact same offset that was being checked. This would have meant that if the value stored in `EnableAt` is not `0`, then `taskcomp.dll` will be loaded.

Unfortunately, our assumption wasn't correct as we observed that `taskcomp.dll` was loaded into `schedsvc.dll` regardless of the registry value.

We hit a bit of a sunk cost trying to chase down the condition for the load of `taskcomp.dll`, so we opted to operate with the knowledge that it is loaded into `schedsvc.dll` under *some* condition and started digging into `taskcomp.dll` to find out how it works.

Since we knew that the DLL was responsible for serving the named pipe, we simply searched for strings containing "atsvc" and found that the named pipe was created in the `StartRpcServer()` function.

This function is called via `CompatibilityAdapter::Init` which is called by `InitializeAdapter()`, an exported function which receives a ULONG as its only parameter. When this function calls `CompatibilityAdapter:Init()`, it passes along this parameter which in turn passes the parameter to a new function, `InitializeNetScheduleApi()`. When it finally reaches this new function, a global variable, `g_AtProtocolEnabled`, is populated with the value in the parameter.

When we looked at the cross references to this global variable, we found that 4 functions prefixed with "`NetrJob`" referenced it, including the `NetrJobAdd()` that we saw during our initial telemetry collection!

Most of these functions check if `g_AtProtocolEnabled` is not `0` before executing its main code. If it is set to `0` an error code of `0x32`, which maps to `ERROR_NOT_SUPPORTED`. This is the error we get when we run `at.exe` on a system with `EnableAt` not set or set to `0`.

Note: The one exception is `NetrJobDel()` which doesn't check the global variable. This allows you to delete scheduled tasks with \`at.exe\` even when the registry value isn't set.

Each of these `NetrJob*()` methods are callable over RPC, as shown in the output from NtObjectManager's `Get-RpcServer` cmdlet.

To validate our assumption that the `g_AtProtocolEnabled` global variable was truly being populated as a result of the `EnabledAt` registry value, we used WinDbg to check the value stored in the variable both when the registry key was set to `1` and `0`.

With this information, we could make the following conclusion:

> at.exe will use RPC to contact `taskcomp.dll` over the `\.\pipe\atsvc` named pipe hosted in `schedsvc.dll` in order to interact with scheduled tasks. `Taskcomp.dll` will only allow interaction if the `g_AtProtocolEnabled` variable is set, which comes from `schedsvc.dll`'s registry query of the `EnableAt` key.

Putting our Abstraction to Work

We learned a few things from out research:

- The `EnableAt` registry value is missing by default. If it were to exist, it should always be set to `0`.
- The named pipe `\\.\pipe\atsvc` will always be present on the system, even if `EnableAt` is `0` or not set. It starts at boot with the Schedule service via `taskcomp.dll`.
- `taskcomp.dll` is ultimately responsible for denying functionality based on the value in `EnableAt`.

- The `NetrJob*()` RPC methods are used to interact with `taskcomp.dll`, so if the `EnableAt` value is set to `1`, correlating only to the execution of `at.exe` could miss executions
- Using `at.exe` to delete scheduled tasks is an edge case that should be covered as it could be considered benign but it subverts the `EnabledAt` restriction

Conclusion

After uncovering this information, we could now finish the abstraction section for the At utility and move on to another procedure. Although we won't be walking through the rest of the map, we wanted to highlight this process as it can be applied to many different scenarios.

Stopping at a certain level and making assumptions can hinder our knowledge about a certain topic or attack, which will funnel into our detection efforts in the future. Having the ability to dive as deep as we can into the technology of interest not only helps build our understanding for the behavior at hand, but our overall knowledge of that technology which can be applied to other actions in the future.

We hope you enjoyed the walkthrough and below is finished abstraction map for scheduled tasks.

-- 1

Written by Jonathan Johnson

Follow

870 Followers · Writer for Posts By SpecterOps Team Members

Principal Security Engineer @Prelude | Windows Internals