A very simple and alternative PID finder

splintercod3.blogspot.com/p/a-very-simple-and-alternative-pid-finder.html

by splinter_code - 5 May 2022

Recently i came across an interesting feature in Windows often used by Ransomware that is the Restart Manager API:

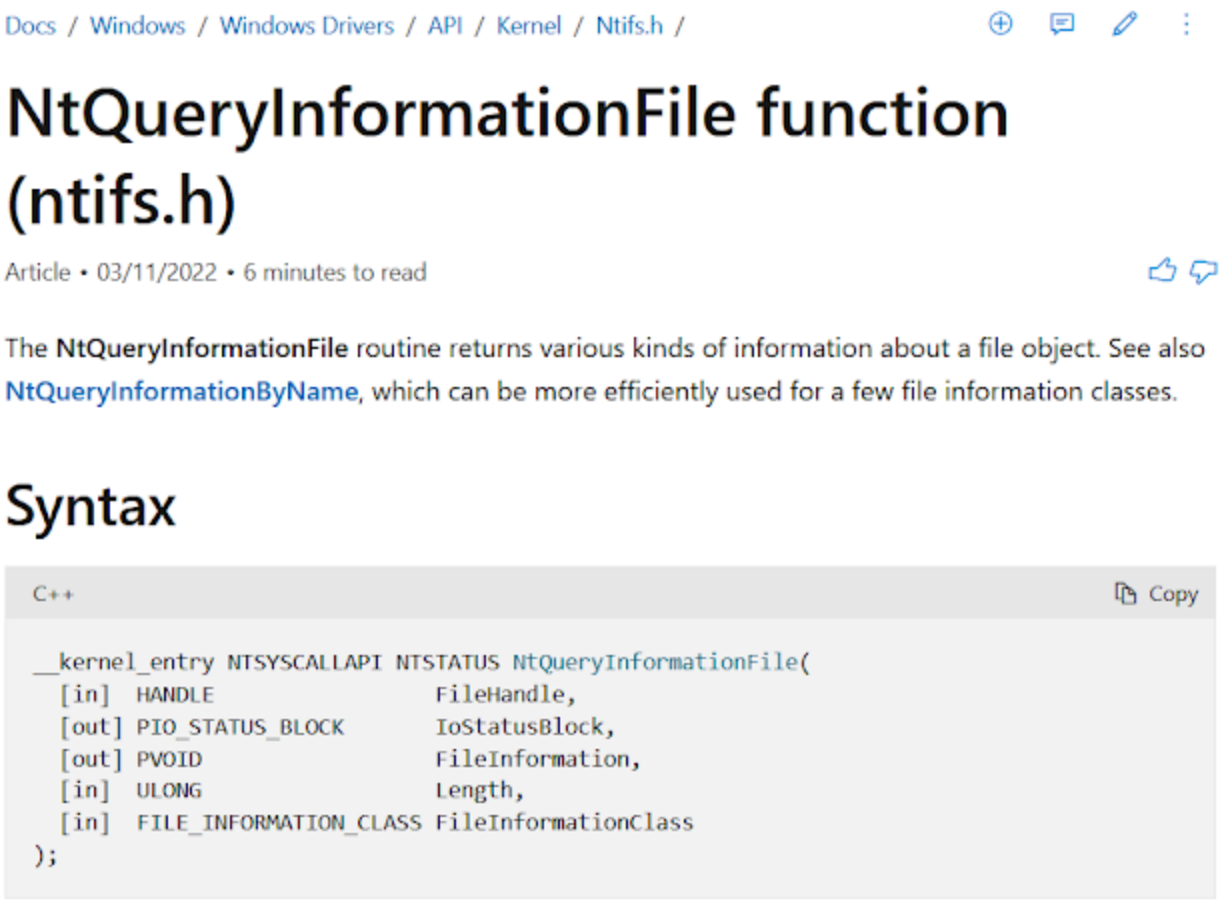
"The primary reason software installation and updates require a system restart is that some of the files that are being updated are currently being used by a running application or service. Restart Manager enables all but the critical applications and services to be shut down and restarted . This frees the files that are in use and allows installation operations to complete."

Of course this is abused in the Ransomware's locker when some process holds lock conditions on files and getting a handle to the file to be encrypted would return an ERROR_SHARING_VIOLATION or ERROR_LOCK_VIOLATION.
So basically what they do is to invoke the Restart Manager API functions to retrieve the list of PIDs that lock a file and then kill them.

My interest peaked in knowing how this happens under the hoods... Firing up Ida and analyzing the function RmGetList (this is one of the Restart Manager API function) i noticed this interesting call stack:

NtQueryInformationFile(hFile, ..., FileProcessIdsUsingFileInformation)
RMRegisteredFile::AffectedPids()
RmFileFactory::UniqueAffectedPids()
CRestartManager::UpdateInternalData()
CRestartManager::GetAffectedApplications()
RmGetList()

So the Restart Manager uses the function NtQueryInformationFile from ntdll with the FILE_INFORMATION_CLASS set to FileProcessIdsUsingFileInformation (47) in order to retrieve a list of PIDs that are using the file specified. Below the function definition:



Docs / Windows / Windows Drivers / API / Kernel / Ntifs.h /

NtQueryInformationFile function (ntifs.h)

Article • 03/11/2022 • 6 minutes to read

The **NtQueryInformationFile** routine returns various kinds of information about a file object. See also NtQueryInformationByName, which can be more efficiently used for a few file information classes.

Syntax

C++ Copy

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationFile(
  [in]  HANDLE                 FileHandle,
  [out] PIO_STATUS_BLOCK       IoStatusBlock,
  [out] PVOID                  FileInformation,
  [in]  ULONG                  Length,
  [in]  FILE_INFORMATION_CLASS FileInformationClass
);
```

Note that this does not apply only on files opened in exclusive access, but for any process using the specified file.
Knowing i could leverage this undocumented functionality in NtQueryInformationFile, a light bulb came into my mind: what if i can use this undocumented functionality and build an alternative PID finder? Ideally it shouldn't use the already known functions CreateToolhelp32Snapshot or NtQuerySystemInformation, otherwise what's the point ?_?
Well, that was a hella good idea. But let's proceed step by step...

Why do you need to find a PID (process id) of a process? Basically it's the main parameter needed for the OpenProcess function in order to get a handle for a process and then do juicy stuff.

From an attacker perspective, the most common scenarios in which you would need to automatically retrieve the pid of a process starting from its name are:

- When you want to steal a SYSTEM token from a privileged process:
  - If you want SeTcbPrivilege you want to target winlogon.exe;
  - If you want SeCreateTokenPrivilege you want to target csrss.exe or lsass.exe;
- When you want to perform a process injection, explorer.exe tells you something? :D
- When you want to dump lsass.exe memory for interesting loot.

By poking around in Process Explorer and running random processes i have identified the following interesting correlation between process name and opened file handle:

- notepad.exe -> C:\Windows\System32\en-US\notepad.exe.mui
- cmd.exe -> C:\Windows\System32\en-US\cmd.exe.mui
- mspaint.exe -> C:\Windows\System32\en-US\mspaint.exe.mui

So we have a way to create a correlation between the process name we want to find a PID for and what is required for the NtQueryInformationFile call (an opened file handle).

But then i realized that most probably the usage of the Image Path of the process would be much more reliable than using .mui files.
Unfortunately, from Process Explorer, i couldn't see any opened handles to the image path of the process itself.

However the NtQueryInformationFile call succedeed anyway in retrieving the right PID of the process!
I guess this occurs because NtQueryInformationFile does not consider a "file use" only a process with a opened file handle, but even if the process has a mapped image/module from the requested path. (Bonus: try to use this function on C:\Windows\System32\ntdll.dll and enjoy the list of all PIDs running on the system ;D)

I have written a very simple POC in order to verify if this method could work at least with the most useful and interesting process names, below the results:

```
C:\Users\splintercode\source\repos\AltPidFinder\x64\Release>AltPidFinder.exe
Pid for process C:\Windows\explorer.exe = 5964
Pid for process C:\Windows\System32\csrss.exe = 536
Pid for process C:\Windows\System32\services.exe = 764
Pid for process C:\Windows\System32\winlogon.exe = 716
Pid for process C:\Windows\System32\lsass.exe = 792
Pid for process C:\Windows\System32\spoolsv.exe = 2668
Pid for process C:\Windows\System32\taskhostw.exe = 3324
Pid for process C:\Windows\System32\dllhost.exe = 4364
Pid for process C:\Windows\System32\RuntimeBroker.exe = 6780
Pid for process C:\Windows\System32\sihost.exe = 3732
```

You can find the POC code here -->
https://gist.github.com/antonioCoco/9db236d6089b4b492746f7de31b21d9d

That's all folks :P