Using process creation properties to catch evasion techniques

microsoft.com/security/blog/2022/06/30/using-process-creation-properties-to-catch-evasion-techniques

June 30, 2022

We developed a robust detection method in Microsoft Defender for Endpoint that can catch known and unknown variations of a process execution class used by attackers to evade detection. This class of stealthy execution techniques breaks some assumptions made by security products and enables attackers to escape antimalware scans by circumventing process creation callbacks using a legacy process creation syscall. Publicly known variations of this class are process doppelganging, process herpaderping, and process ghosting.

Evasion techniques used by attackers often involve running malware within the context of a trusted process or hiding code from filesystem and memory scanners. More sophisticated attackers even carefully choose their process host so that their actions are run by a process that often performs these actions for benign reasons. For example, a browser process communicating with the internet seems completely normal, while an instance of *cmd.exe* doing the same sticks out like a sore thumb. This class of stealthy execution techniques, however, allows malware to create its own malicious process and prevent antimalware engines from detecting it.

This blog post presents our detailed analysis of how this process execution class works and how it takes advantage of Windows functionalities to evade detection. It also presents a peek into the research, design, and engineering concerns that go into the development of a detection method aiming to be as robust and future-proof as possible.

Common classes of stealthy process execution

On Windows systems, most methods attackers use to run code within another process fall within two classes: *process injection* and *process hollowing*. These classes allow attackers to run their code within another process without explicitly creating it from an executable, or making it load a dynamic link library (DLL). Similar classes of techniques are often also called process injection, but this term will be used in a more specific definition for clarity.

Process injection

Process injection, the widest and most common class, consists of different techniques that introduce attacker-supplied executable memory into an already running process. Techniques in this class consist of two main parts:

- **Write primitive:** A Windows API function, or a set of APIs, used to introduce malware into the target process.

- **Execution primitive:** A Windows API method to redirect the execution of the process to the code provided by the attacker.

An example of a classic process injection flow is malware using the *VirtualAllocEx* API to allocate a buffer within a target process, *WriteProcessMemory* to fill that buffer with the contents of a malware module, and *CreateRemoteThread* to initiate a new thread in the target process, running the previously injected code.

Process hollowing

In process hollowing, instead of abusing an already running process, an attacker might start a new process in a suspended state and use a write primitive to introduce their malware module before the process starts running. By redirecting the entry point of the process before unsuspending, the attacker may run their code without using an explicit execution primitive.

Variants (and sometimes combinations) of both classes exist and differ from each other mostly by the APIs being used. The APIs vary because a different function used to achieve the goal of one of the steps may not go through the numerous points at which an endpoint protection product intercepts such behavior, which can break detection logic.

New stealth techniques

In the past few years, stealth techniques from a process execution class have emerged that don't strictly fit into any of the previously mentioned classes. In this class, instead of modifying the memory of an already created (but perhaps not yet executing) process, a new process is created from the image section of a malware. By the time a security product is ready to scan the file, the malware bits aren't there anymore, effectively pulling the rug from under antimalware scanners. This technique requires defenders to use a different detection method to catch attacks that use it. As of today, the following variations of this class are known publicly as the following:

- **Process doppelganging[1]:** Abusing transactional NTFS features to create a volatile version of an executable file used for process creation, with the file never touching the disk.
- **Process herpaderping[2]:** Utilizing a writable handle to an executable file to overwrite the malware bits on disk before antimalware services can scan the executable, but after a process has already been created from the malicious version.
- **Process ghosting[3]:** Abusing a handle with delete permissions to the process executable to delete it before it has a chance to be scanned.

This process execution class, including the variations mentioned above, takes advantage of the way the following functionalities in the operating system are designed to evade detection by security products:

- Antimalware engines don't scan files after every single modification.
- Process creation callbacks, the operating system functionality that allows antimalware engines to scan a process when it's created, is invoked only when the first thread is inserted into a process.
- *NtCreateProcessEx*, a legacy process creation syscall, allows the creation of a process without populating it with any thread.

The following sections explain in more detail how these functionalities are abused.

When are files scanned?

A key feature of this process execution class is circumventing a file scan. Ideally, files are scanned whenever they're modified. Otherwise, an attacker could simply modify an existing file into a malicious one, use it to create a process, and then either revert the file or delete it. So, why aren't files scanned on every file change?

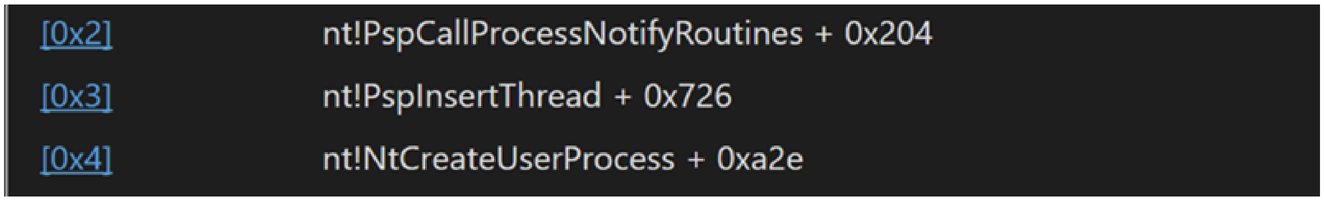
The answer lies in performance concerns. Consider a scenario in which a 1MB file is opened, and it's overwritten by calling an API like *WriteFile* for every byte that needs to be overwritten. While only 1MB would be written to disk, the file would have to be scanned one million times, resulting in ~1 terabyte of data being scanned!

While the example is a good way to assure no detectably malicious content is written to disk, the amount of computing power it will use up makes it an unviable solution. Even a caching solution would simply shift the high resource usage to memory, as a product would need to keep information about the content of every single open file on the machine to be useful.

Therefore, the most common design for file scanning engines ignores the various transient states of the file content and initiates a scan whenever the handle to the file is closed. This is an optimal signal that an application is done modifying a file for now, and that a scan would be meaningful. To determine what the file is about to execute as a process, the antimalware engine scans the file's content at the time of process creation through a process creation callback.

Process creation callbacks in the kernel, such as those provided by the *PsSetCreateProcessNotifyRoutineEx* API, is the functionality in the operating system that allows antimalware engines to inspect a process while it's being created. It can intercept the creation of a process and perform a scan on the relevant executable, all before the process runs.

Process creation notification isn't invoked right when a process creation API is called, but rather when the first thread is inserted into a process. But since *NtCreateUserProcess*, the syscall used by all common high-level APIs to create a process, is designed to do a lot of the work required to create a process in the kernel, the insertion of the initial thread into the created process happens within the context of the syscall itself. This means that the callback launches while the process is still being created, before user mode has a chance to do anything.



```
[0x2]    nt!PspCallProcessNotifyRoutines + 0x204
[0x3]    nt!PspInsertThread + 0x726
[0x4]    nt!NtCreateUserProcess + 0xa2e
```

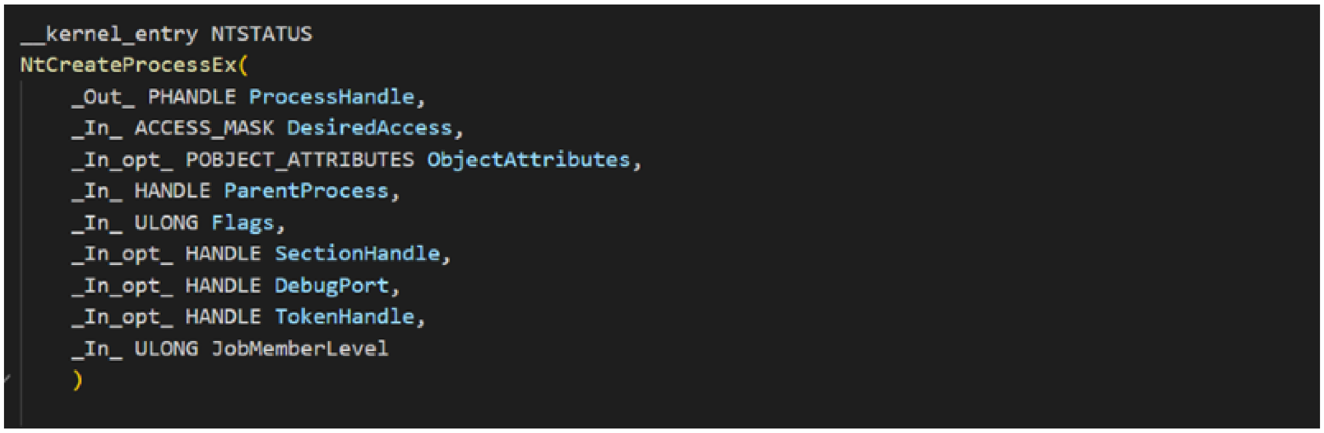
Figure 1. Process creation callbacks being invoked from NtCreateUserProcess

The call stack indicates that in this scenario, *PspCallProcessNotifyRoutines*, the function responsible for invoking process creation callbacks, is called from *PspInsertthread* during the insertion of the initial thread into the process. It also indicates that the subsequent process creation callbacks are all called from within *NtCreateUserProcess*, and that they both finish executing before the syscall returns. This enables the antimalware to scan the process for malware activity as it's created. This works if the process is created using *NtCreateUserProcess*. However, as researchers have found, there are other ways to create a process apart from this syscall.

How are processes created?

The syscall *NtCreateUserProcess* has only been available since the release of Windows Vista. Processes created by the *CreateProcess* API or any API using the *NtCreateUserProcess* syscall only provide the path to the executable. Meanwhile, the kernel opens the file without any share access that could allow modification (no SHARE_WRITE/SHARE_DELETE), creates an image section, and returns to user mode with the process pretty much ready to run (most legitimate Windows processes would require additional work to be done in user mode to operate correctly, but the NtCreateUserProcess syscall does the minimum work needed for a process to execute some code). This means that an attacker doesn't have the time or the capability to modify an executable file after calling *NtCreateUserProcess*, but only before it's scanned.

Versions of the NT kernel prior to the release of Windows Vista used a different syscall called *NtCreateProcessEx*. This function doesn't adhere to the principle of doing a lot of the work in the kernel and in fact delegates a lot of the work normally associated with process creation on modern Windows platforms to user mode.

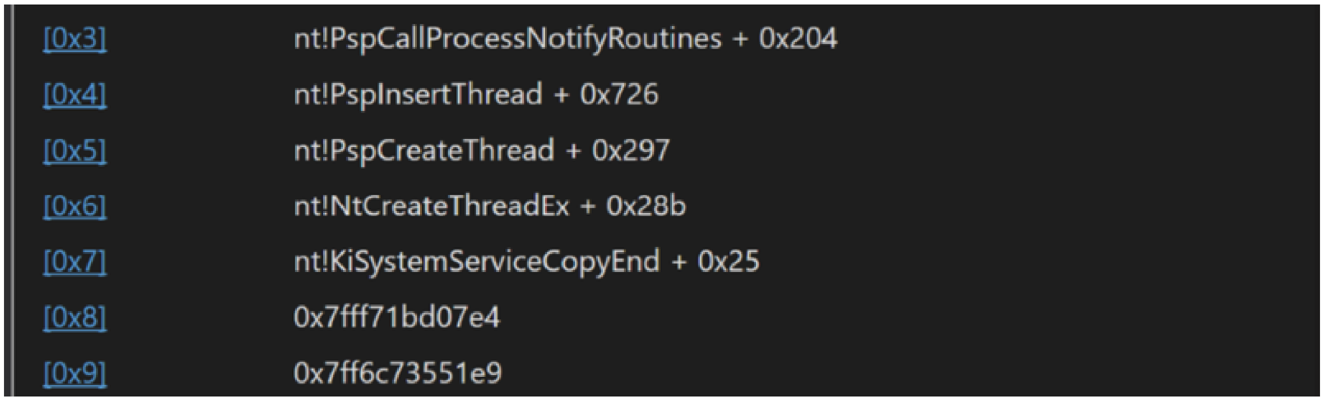


```
__kernel_entry NTSTATUS
NtCreateProcessEx(
    _Out_ PHANDLE ProcessHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ParentProcess,
    _In_ ULONG Flags,
    _In_opt_ HANDLE SectionHandle,
    _In_opt_ HANDLE DebugPort,
    _In_opt_ HANDLE TokenHandle,
    _In_ ULONG JobMemberLevel
    )
```

Figure 2. The function signature of NtCreateProcessEx. Note the absence of a path argument and the presence of SectionHandle.

One difference between the two is that *NtCreateProcessEx* doesn't receive a path to the process executable as an argument, as is the case with *NtCreateUserProcess*. *NtCreateProcessEx* expects the application to open the file on its own and create an image section from that file, which will be used as the main image section of the process, and the handle to which will be passed to *NtCreateProcessEx*.

Also, unlike *NtCreateUserProcess*, *NtCreateProcessEx* creates a process object without populating the process with any threads, and the user application needs to explicitly insert the initial thread into the process using an API like *NtCreateThread*.



```
[0x3]            nt!PspCallProcessNotifyRoutines + 0x204
[0x4]            nt!PspInsertThread + 0x726
[0x5]            nt!PspCreateThread + 0x297
[0x6]            nt!NtCreateThreadEx + 0x28b
[0x7]            nt!KiSystemServiceCopyEnd + 0x25
[0x8]            0x7fff71bd07e4
[0x9]            0x7ff6c73551e9
```

Figure 3. In this callstack, the invocation of PspCallProcessNotifyRoutine and PspInsertThread happens from within NtCreateThreadEx, not from within a process creation syscall.

Combining this information with what we know about process creation callbacks allows us to come up with a generic flow for this stealthy process creation technique:

1. The attacker opens the malware file and brings it into a transient modifiable state (writable without closing a handle, delete pending or an uncommitted transaction, and some other unpublished ones) while having malware content. The attacker doesn't close the file yet.
2. The attacker creates an image section from the file handle using *NtCreateSection(Ex)*.
3. The attacker creates a process using the image section handle as input.

4. The attacker reverts the file's transient state to a benign state (the file is deleted or overwritten, or a transaction is rolled back), and the handle is closed. At this point, the bits of the malware still exist in memory as the image section object is still there, but there is no trace of the malware content on the disk.
5. The attacker inserts the initial thread into the process, and only then will the process creation notification callback for that process be launched. At that point, there is no malware content left to scan.
6. The attacker now runs the malware process without its backing file ever being scanned.

In this generalized flow, a security product should be able to detect any variation of the technique if it can recognize that the process was created using the legacy *NtCreateProcessEx* syscall, which allows an adversary to run the process from a file in a transient state.

Of course, one could circumvent the need for *NtCreateProcessEx* by performing a similar trick with loading DLLs. However, in this scenario, the adversary can either load a new DLL into a process they already have full code execution capabilities without changing its identity, or remotely place the offending DLL into another process, performing what is essentially process injection. In both cases, the technique's effectiveness as an evasion method is greatly diminished.

Detecting legacy process creation

The first anomaly to recognize to detect attacks using this technique is to find out whether a process was created using the legacy *NtCreateProcessEx* syscall.

The simplest way to do so would be to utilize user-mode hooking on the appropriate function in the NTDLL library. However, this approach would be easy to bypass, as it's assumed that the adversary has arbitrary execution capabilities in the process calling the syscall. This means they would be able to unhook any functions intercepted by a security product, or simply directly call the syscall from their own assembly code. Even if the security product was to traverse the user-mode call stack from a process creation callback and check the return address against known values, the product would still be subject to evasion since an attacker could employ some creative pushes and jumps in assembly code to construct a spoofed user-mode call stack to their liking.

To create a robust detection for this behavior, information that can't be modified or spoofed by a user-mode adversary should be used. A good example of this is a Windows file system concept called extra create parameters (ECPs).

ECPs are concepts that allow the kernel or a driver to attach some key-value information to a file create/open operation. The idea is very similar to extended file attributes, but instead of applying to an entire file on disk, ECPs are a transient property related to a specific instance of an open file. This mechanism allows the operating system and drivers to respond to a file being opened under some special circumstances.

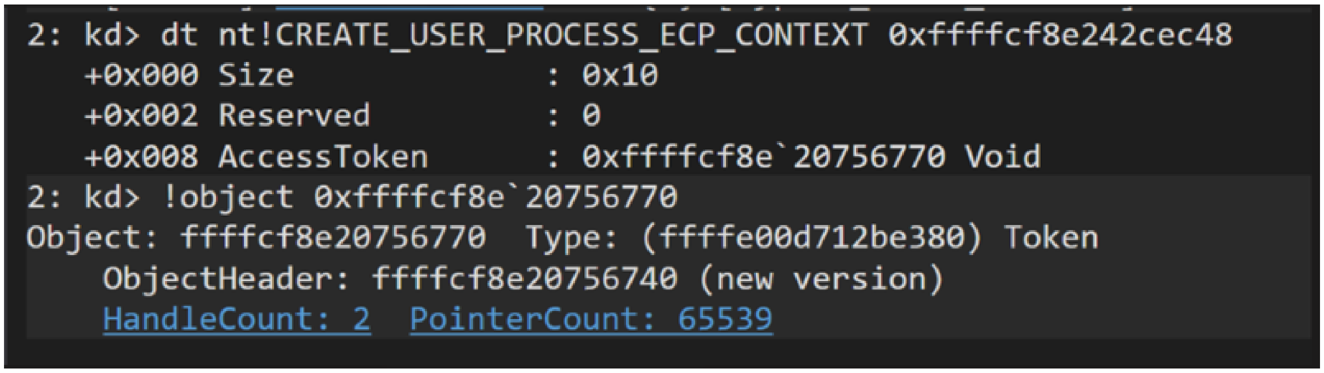
An example of such special circumstances is a file being opened via Server Message Block (SMB). When this happens, an *SRV_OPEN_ECP_CONTEXT* structure is added to the *IRP_MJ_CREATE IRP* with *GUID_ECP_SRV_OPEN* as a key.

This ECP context contains information on the socket used for the communication with the SMB client, the name of the share which has been accessed, and some oplock information. A driver would then be able to use this information to appropriately handle the open operation, which might need some special treatment since the operation happened remotely.

Interestingly, an exported, documented function named *FsRtlIsEcpFromUserMode* exists to determine whether an ECP originated in user mode or kernel mode. This raises the concern that forgetting to use this function in a driver or the OS would cause potential security issues, as a user mode adversary could spoof an ECP. That isn't the case, though, as there is no functionality in the OS which allows a user to directly supply any ECP from user mode. The function itself checks whether a specific flag is set in the opaque ECP header structure, but there exists no code in the OS which can modify this flag.

Using ECPs for process creation API recognition

Starting with Windows 10, a very interesting ECP has been added to the operating system whenever a new process is created using *NtCreateUserProcess*. The *GUID_ECP_CREATE_USER_PROCESS* ECP and its related *CREATE_USER_PROCESS_ECP_CONTEXT* context are applied to the *IRP_MJ_CREATE* operation when the Windows kernel opens the process executable file. This ECP contains the token of the process to be created. In fact, the function used to open the executable path was changed from *ZwOpenFile* to *IoCreateFileEx* specifically to support ECPs on this operation.



```
2: kd> dt nt!CREATE_USER_PROCESS_ECP_CONTEXT 0xffffcf8e242cec48
   +0x000 Size             : 0x10
   +0x002 Reserved         : 0
   +0x008 AccessToken      : 0xffffcf8e`20756770 Void
2: kd> !object 0xffffcf8e`20756770
Object: ffffcf8e20756770  Type: (ffffe00d712be380) Token
    ObjectHeader: ffffcf8e20756740 (new version)
    HandleCount: 2  PointerCount: 65539
```

Figure 4. The CREATE_USER_PROCESS_ECP_CONTEXT

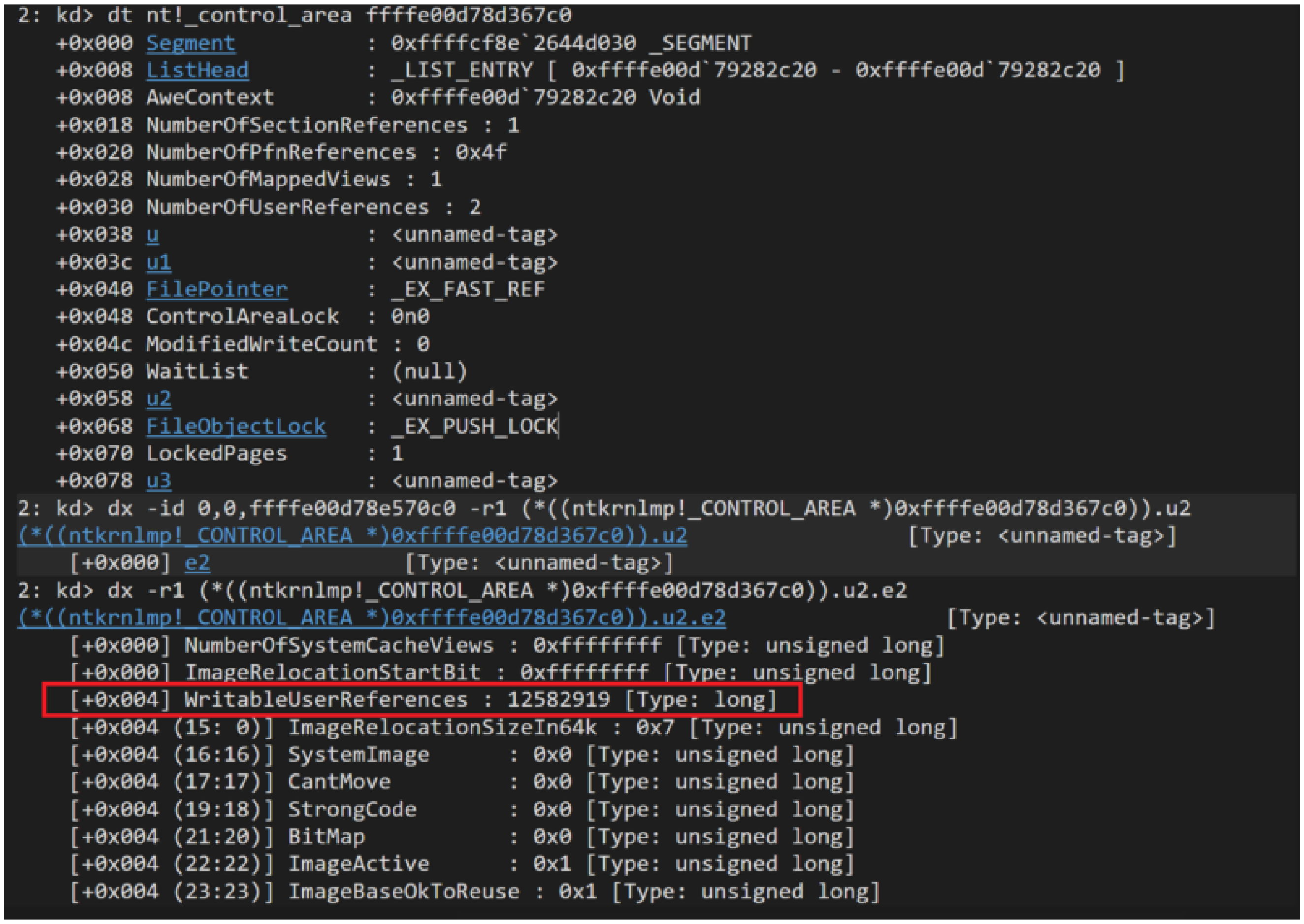
On the other hand, as covered earlier, *NtCreateProcessEx* doesn't open the process executable on its own but instead relies on the user to supply a section handle created from a file opened by the user themselves. Seeing as there is no way for the user to set the process creation ECP on their own handle, any process created using *NtCreateProcessEx* would be missing this ECP on the *IRP_MJ_CREATE* for its main image. Some cases exist in which the ECP wouldn't be present even when the legacy API wasn't used, but those can still be recognized. Barring those cases, the existence of the *CREATE_USER_PROCESS* ECP in the *IRP_MJ_CREATE* operation of the file object related to the main image of the process can now be used to precisely differentiate between processes created by *NtCreateUserProcess* and those created by *NtCreateProcessEx*.

Detecting processes created from files in a transient state

Since it's now possible to check when the legacy process creation API has been used, the next step would be to check if the usage of the legacy process creation API was used to abuse the time-of-check-time-of-use (TOCTOU) issue involving process creation callbacks. This means that the executable image used to create the process has been opened and used in a transient state, which would already be rolled back when it's to be scanned by an antimalware engine. To identify if TOCTOU was abused, it is important to examine the image section of the main executable of the process.

Windows loads executable images into memory and shares their memory between processes using memory sections (also called memory-mapped files). Each *FILE_OBJECT* structure for an open file contains a member called *SectionObjectPointers*, which contains pointers to the data and image section control areas relevant to the file, depending on whether if it has been mapped as a data file or an executable. The bits described by such a section may be backed either by a file on disk or by the page file (in which case the bits of the section won't persist on disk). This property determines whether the mapped section can be flushed and recovered from a file or disk, or simply paged out.

However, an interesting thing happens when the connection between an image section and its backing file is severed. This can happen if, for example, the file is located on a remote machine or some removable storage, *Copy-on-Write* has been triggered, or most importantly, if the file has been somehow modified after the section has been created or could be modified in the future. During such cases, the image section becomes backed by the page file instead of the original file from which it was created.



```
2: kd> dt nt!_control_area ffffe00d78d367c0
   +0x000 Segment          : 0xffffcf8e`2644d030 _SEGMENT
   +0x008 ListHead         : _LIST_ENTRY [ 0xffffe00d`79282c20 - 0xffffe00d`79282c20 ]
   +0x008 AweContext       : 0xffffe00d`79282c20 Void
   +0x018 NumberOfSectionReferences : 1
   +0x020 NumberOfPfnReferences : 0x4f
   +0x028 NumberOfMappedViews : 1
   +0x030 NumberOfUserReferences : 2
   +0x038 u                : <unnamed-tag>
   +0x03c u1               : <unnamed-tag>
   +0x040 FilePointer      : _EX_FAST_REF
   +0x048 ControlAreaLock  : 0n0
   +0x04c ModifiedWriteCount : 0
   +0x050 WaitList         : (null)
   +0x058 u2               : <unnamed-tag>
   +0x068 FileObjectLock   : _EX_PUSH_LOCK
   +0x070 LockedPages      : 1
   +0x078 u3               : <unnamed-tag>
2: kd> dx -id 0,0,ffffe00d78e570c0 -r1 (*((ntkrnlmp!_CONTROL_AREA *)0xffffe00d78d367c0)).u2
(*((ntkrnlmp!_CONTROL_AREA *)0xffffe00d78d367c0)).u2                 [Type: <unnamed-tag>]
    [+0x000] e2              [Type: <unnamed-tag>]
2: kd> dx -r1 (*((ntkrnlmp!_CONTROL_AREA *)0xffffe00d78d367c0)).u2.e2
(*((ntkrnlmp!_CONTROL_AREA *)0xffffe00d78d367c0)).u2.e2                 [Type: <unnamed-tag>]
    [+0x000] NumberOfSystemCacheViews : 0xffffffff [Type: unsigned long]
    [+0x000] ImageRelocationStartBit : 0xffffffff [Type: unsigned long]
    [+0x004] WritableUserReferences : 12582919 [Type: long]
    [+0x004 (15: 0)] ImageRelocationSizeIn64k : 0x7 [Type: unsigned long]
    [+0x004 (16:16)] SystemImage      : 0x0 [Type: unsigned long]
    [+0x004 (17:17)] CantMove         : 0x0 [Type: unsigned long]
    [+0x004 (19:18)] StrongCode       : 0x0 [Type: unsigned long]
    [+0x004 (21:20)] BitMap           : 0x0 [Type: unsigned long]
    [+0x004 (22:22)] ImageActive      : 0x1 [Type: unsigned long]
    [+0x004 (23:23)] ImageBaseOkToReuse : 0x1 [Type: unsigned long]
```

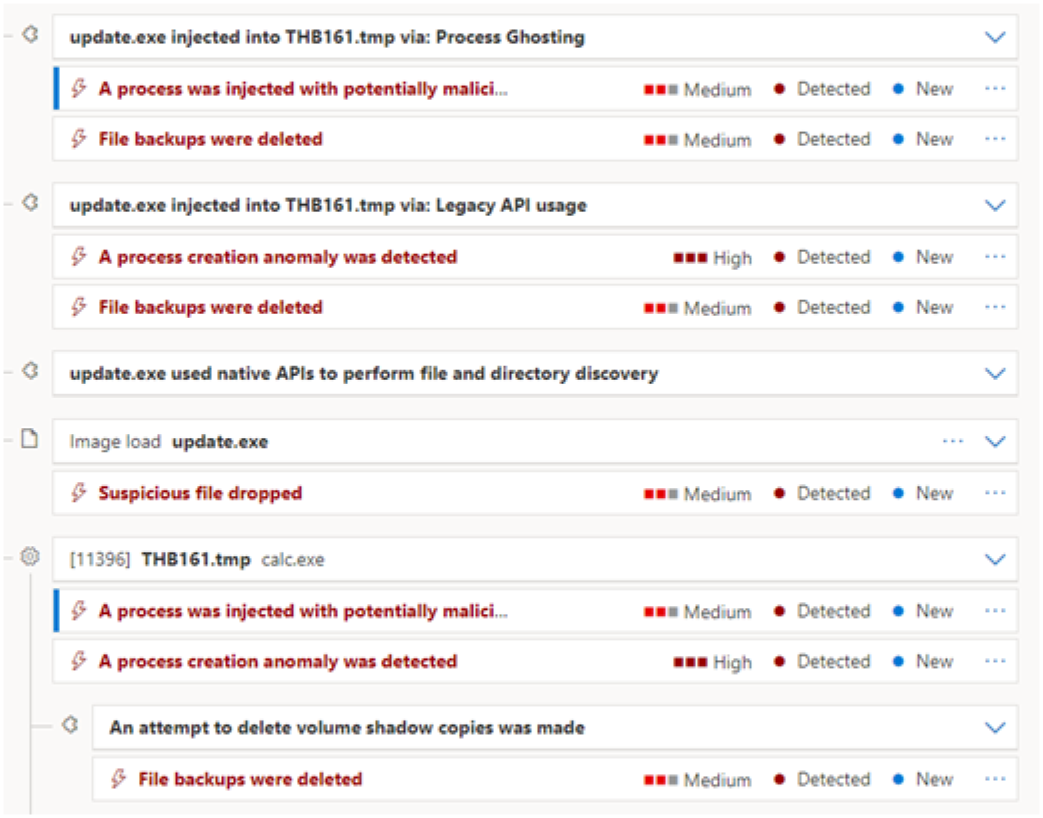
Figure 5. The control area of a section breaking coherency with disk. Note the WritableUserReferences member being set.

The *MmDoesFileHaveUserWritableReferences* function provides the caller with the number of writable (or, more correctly, modifiable) references to the file object of a section and is used by the kernel transaction manager to preserve the atomicity of transactions. Otherwise, a file can be written, deleted, or simply gone when a transaction is to be committed. This function can be used for detection because a non-zero return value means that section coherency has been broken, and the logic switching the backing of the section to the page file has been triggered. This can help determine that the file is in one of the same transient states needed to abuse TOCTOU and evade detection.

Detection through Microsoft Defender for Endpoint

The two primitives discussed earlier can now be combined into detection logic. First, the absence of the *GUID_ECP_CREATE_USER_PROCESS* ECP will verify if the process was created using the legacy API *NtCreateProcessEx*. Then, the function *MmDoesFileHaveUserWritableReferences* checks if the file's image section is backed by the page file, confirming that the process was created while the file is in a transient state. Meeting both conditions can determine that TOCTOU has been abused, whether by any of the published techniques, or a variation of it that uses similar concepts but abuses a functionality built into a driver to create a similar effect.

Microsoft Defender for Endpoint can detect each of the known techniques in this class of stealthy process execution and gives out a specific alert for variations of process ghosting, herpaderping, and doppelganging found in the wild. Apart from the specific alerts for each variation, detections exist for the generalized flow and any abuse of the legacy process creation API, including unpublished variations.



update.exe injected into THB161.tmp via: Process Ghosting

A process was injected with potentially malici...	Medium	Detected	New
File backups were deleted	Medium	Detected	New

update.exe injected into THB161.tmp via: Legacy API usage

A process creation anomaly was detected	High	Detected	New
File backups were deleted	Medium	Detected	New

update.exe used native APIs to perform file and directory discovery

Image load **update.exe**

Suspicious file dropped	Medium	Detected	New

[11396] **THB161.tmp** calc.exe

A process was injected with potentially malici...	Medium	Detected	New
A process creation anomaly was detected	High	Detected	New

An attempt to delete volume shadow copies was made

File backups were deleted	Medium	Detected	New

Figure 6. Microsoft Defender for Endpoint detections for variations of process ghosting, herpaderping, and doppelganging.

This blog post shares Windows internals knowledge and showcases a new detection method in Microsoft Defender for Endpoint that can help prevent detection evasion. Since data and signals from Microsoft Defender for Endpoint also feed into Microsoft 365 Defender, this new detection method further enriches our protection technologies, providing customers a comprehensive and coordinated threat defense against threats.

The stealth execution techniques discussed further prove that the threat landscape is constantly evolving, and that attackers will always look for new avenues to evade detection. This highlights the importance of continuous research on potential attack vectors, as well as future-proof solutions. We hope that the principles presented in this blog post can be used by other researchers in developing similar solutions.

Philip Tsukerman, ***Amir Kutcher***, *and* ***Tomer Cabouly***
Microsoft 365 Defender Research Team

[1] https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf

[2] https://jxy-s.github.io/herpaderping/

[3] https://www.elastic.co/blog/process-ghosting-a-new-executable-image-tampering-attack