What you need to know about Process Ghosting, a new executable image tampering attack

elastic.co/blog/process-ghosting-a-new-executable-image-tampering-attack

June 15, 2021

15 Juni 2021Tech Topics

By
Gabriel Landau
Share

Security teams defending Windows environments often rely on anti-malware products as a first line of defense against malicious executables. Microsoft provides security vendors with the ability to register callbacks that will be invoked upon the creation of processes on the system. Driver developers can call APIs such as PsSetCreateProcessNotifyRoutineEx to receive such events.

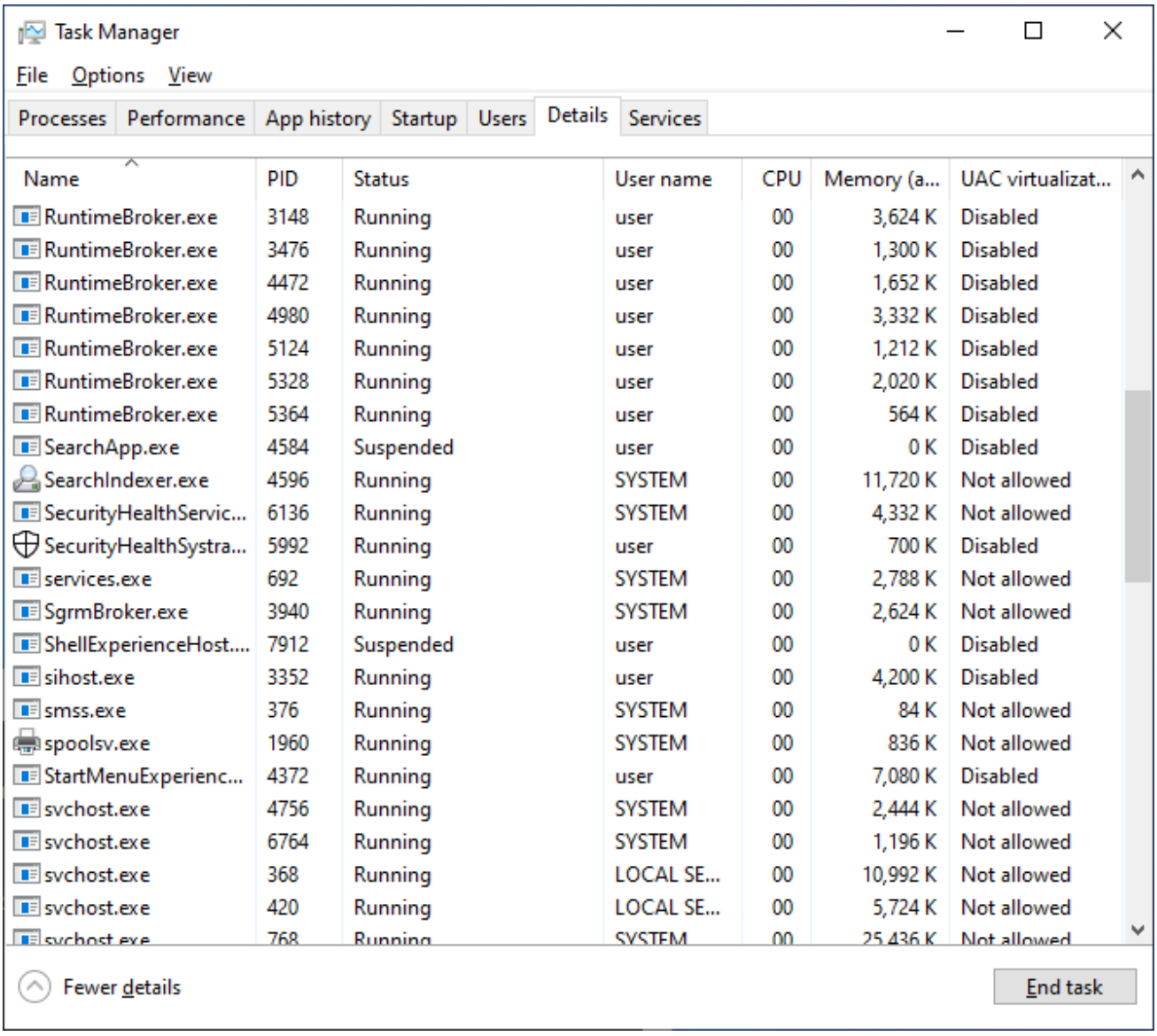
Despite the name, PsSetCreateProcessNotifyRoutineEx callbacks are not actually invoked upon the creation of processes, but rather upon the creation of the first threads within those processes. This creates a gap between when a process is created and when security products are notified of its creation. It also gives malware authors a window to tamper with

the backing executable before security products can scan it. Recent examples of such tampering attacks include Process Doppelgänging and Process Herpaderping, which abuse this behavior to evade security products.

This blog describes a new executable image tampering attack similar to, but distinct from, Doppelgänging and Herpaderping. With this technique, an attacker can write a piece of malware to disk in such a way that it's difficult to scan or delete it — and where it then executes the deleted malware as though it were a regular file on disk. This technique does not involve code injection, process hollowing, or Transactional NTFS (TxF).

The birth of a process

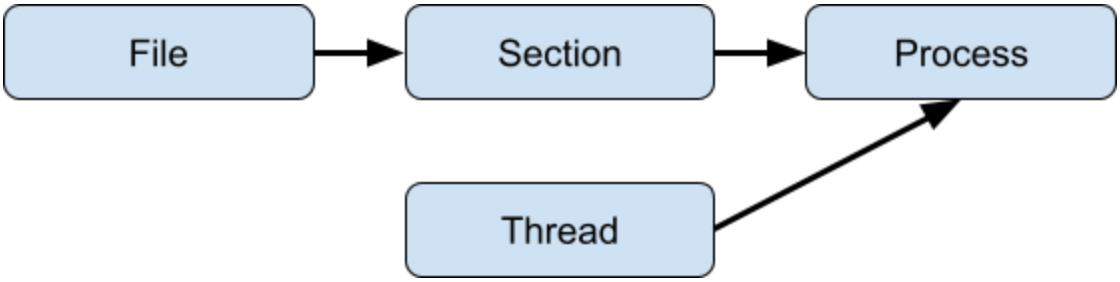
Windows Task Manager shows a list of processes running on the system. Each of these processes is associated with an executable file on disk, such as svchost.exe. This is because Windows launches processes from executable files, usually ending with an EXE file extension.



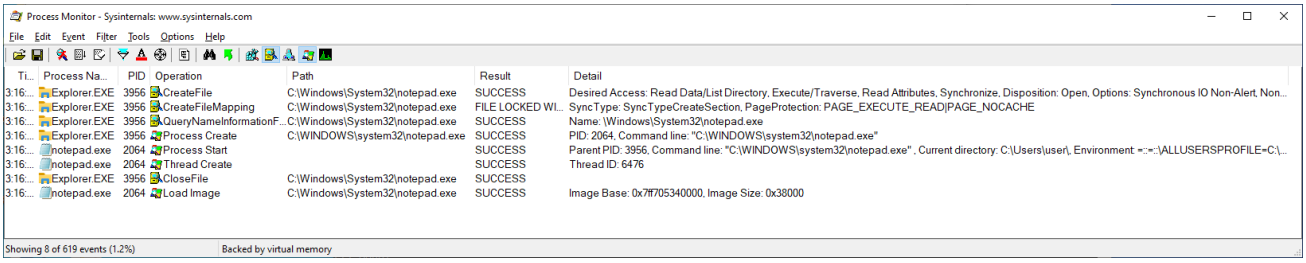
It's important to note that processes are not executables, and executables are not processes. In the example Task Manager above, there are multiple processes launched from RuntimeBroker.exe and svchost.exe.

To launch a new process, a series of steps must occur. In modern versions of Windows, they are typically performed in the kernel by NtCreateUserProcess — however, the individual component APIs (NtCreateProcessEx etc) are still exposed and functional for backwards compatibility purposes. These steps are:

1. Open a handle to the executable to launch. Example: hFile = CreateFile("C:\Windows\System32\svchost.exe")[2]
2. Create an "image" section for the file. A section maps a file, or a portion of a file, into memory. An image section is a special type of section that corresponds to Portable Executable (PE) files, and can only be created from PE (EXE, DLL, etc) files. Example: hSection = NtCreateSection(hFile, SEC_IMAGE)
3. Create a process using the image section. Example: hProcess = NtCreateProcessEx(hSection)
4. Assign process arguments and environment variables. Example: CreateEnvironmentBlock/NtWriteVirtualMemory
5. Create a thread to execute in the process. Example: NtCreateThreadEx



Here is what that looks like in Process Monitor:



explorer.exe launching notepad.exe, as seen in Process Monitor

Processes are launched from executables, but some of the data within the executable file is modified as it is mapped into a process. To account for these modifications, the Windows memory manager caches image sections at the time of their creation. **This means that image sections can deviate from their executable files.**

Scanning processes for malware

Microsoft provides security vendors with the ability to register callbacks that will be invoked upon the creation of processes and threads on the system. Driver developers can call APIs such as PsSetCreateProcessNotifyRoutineEx and PsSetCreateThreadNotifyRoutineEx to receive such events.

Despite the name, PsSetCreateProcessNotifyRoutineEx callbacks are not actually invoked upon the creation of processes, but rather upon the creation of the first threads within those processes. This creates a gap between when a process is created and when security products are notified of its creation. It also gives malware authors a window to tamper with the backing file and section before security products can scan them.

Note how the undocumented process creation API NtCreateProcess takes a section, not file, handle:

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtCreateProcess(
    _Out_ PHANDLE ProcessHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ParentProcess,
    _In_ BOOLEAN InheritObjectTable,
    _In_opt_ HANDLE SectionHandle,
    _In_opt_ HANDLE DebugPort,
    _In_opt_ HANDLE ExceptionPort
    );
```

When a process is launched, security products are provided with the following information about the process being launched:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
  SIZE_T              Size;
  union {
    ULONG Flags;
    struct {
      ULONG FileOpenNameAvailable : 1;
      ULONG IsSubsystemProcess : 1;
      ULONG Reserved : 30;
    };
  };
  HANDLE              ParentProcessId;
  CLIENT_ID           CreatingThreadId;
  struct _FILE_OBJECT *FileObject;
  PCUNICODE_STRING    ImageFileName;
  PCUNICODE_STRING    CommandLine;
  NTSTATUS            CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

Of interest is the FILE_OBJECT, which is the kernel object corresponding to the HANDLE passed to NtCreateSection in the previous section. This FILE_OBJECT typically corresponds to a file on disk, which can be scanned for malware.

Security products may also use filesystem minifilter callbacks, which receive notifications when files are created, interacted with, or closed. The system impact of scanning every single read and write operation can be significant, so files are typically scanned upon open and close for performance reasons.

There are other potential security product interception points that we will not discuss here. See this talk for more information.

Prior work

Process Doppelgänging

Windows Transactional NTFS (TxF) is a mechanism that allows an application to perform a series of filesystem operations as a single atomic transaction, which is then either committed or rolled back. Files can exist within a transaction that, if rolled back, is never visible to the underlying filesystem. Using TxF, it is possible to create an image section from a file within a transaction, then roll back that transaction. It is possible to create a process from such image sections.

Process Herpaderping

After creating the image section, Process Herpaderping uses the existing file handle to overwrite the executable with a decoy PE. While this leaves the decoy on disk, it is different from the one running in memory. The decoy remains on disk throughout the life of the payload process.

Process Reimaging

Process Reimaging exploits a cache synchronization issue in the Windows kernel, causing a mismatch between an executable file's path and the path reported for image sections created from that executable. By loading a DLL at a decoy path, unloading it, then loading it from a new path, various Windows APIs will return the old path. This can fool security products into looking for loaded images at the wrong path.

Ghosting a process

We can build upon Doppelgänging and Herpaderping to run executables that have already been deleted. There are several ways to delete a file on Windows, including:[3]

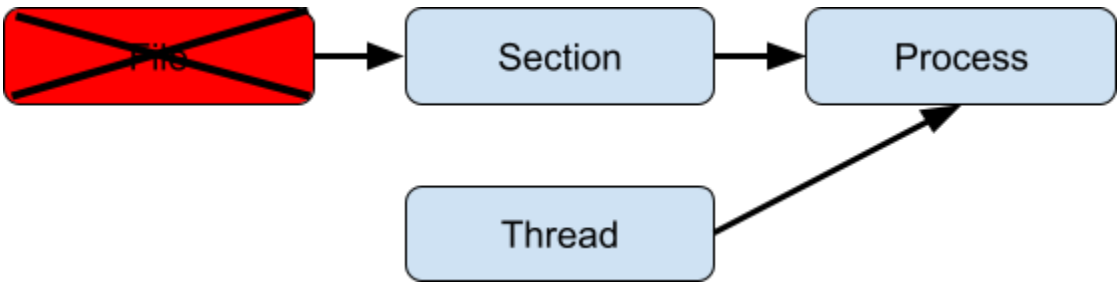
- Create a new file over the old one with the FILE_SUPERSEDE or CREATE_ALWAYS flags set.
- Set the FILE_DELETE_ON_CLOSE or FILE_FLAG_DELETE_ON_CLOSE flags when creating or opening the file.
- Set the DeleteFile field in the FILE_DISPOSITION_INFORMATION structure to TRUE when invoking the FileDispositionInformation file information class via NtSetInformationFile.

Windows attempts to prevent mapped executables from being modified. Once a file is mapped into an image section, attempts to open it with FILE_WRITE_DATA (to modify it) will fail with ERROR_SHARING_VIOLATION. Deletion attempts via FILE_DELETE_ON_CLOSE/FILE_FLAG_DELETE_ON_CLOSE fail with ERROR_SHARING_VIOLATION. NtSetInformationFile(FileDispositionInformation) requires the DELETE access right. Even though the DELETE access right is granted to files mapped to image sections, NtSetInformationFile(FileDispositionInformation) fails with STATUS_CANNOT_DELETE. Deletion attempts via FILE_SUPERCEDE/CREATE_ALWAYS fail with ACCESS_DENIED.

An important note, however, is that this deletion restriction only comes into effect once the executable is mapped into an image section. This means that it is possible to create a file, mark it for deletion, map it to an image section, close the file handle to complete the deletion, then create a process from the now-fileless section. This is Process Ghosting.

The attack flow is:

1. Create a file
2. Put the file into a delete-pending state using NtSetInformationFile(FileDispositionInformation). Note: Attempting to use FILE_DELETE_ON_CLOSE instead will not delete the file.
3. Write the payload executable to the file. The content isn't persisted because the file is already delete-pending. The delete-pending state also blocks external file-open attempts.
4. Create an image section for the file.
5. Close the delete-pending handle, deleting the file.
6. Create a process using the image section.
7. Assign process arguments and environment variables.
8. Create a thread to execute in the process.

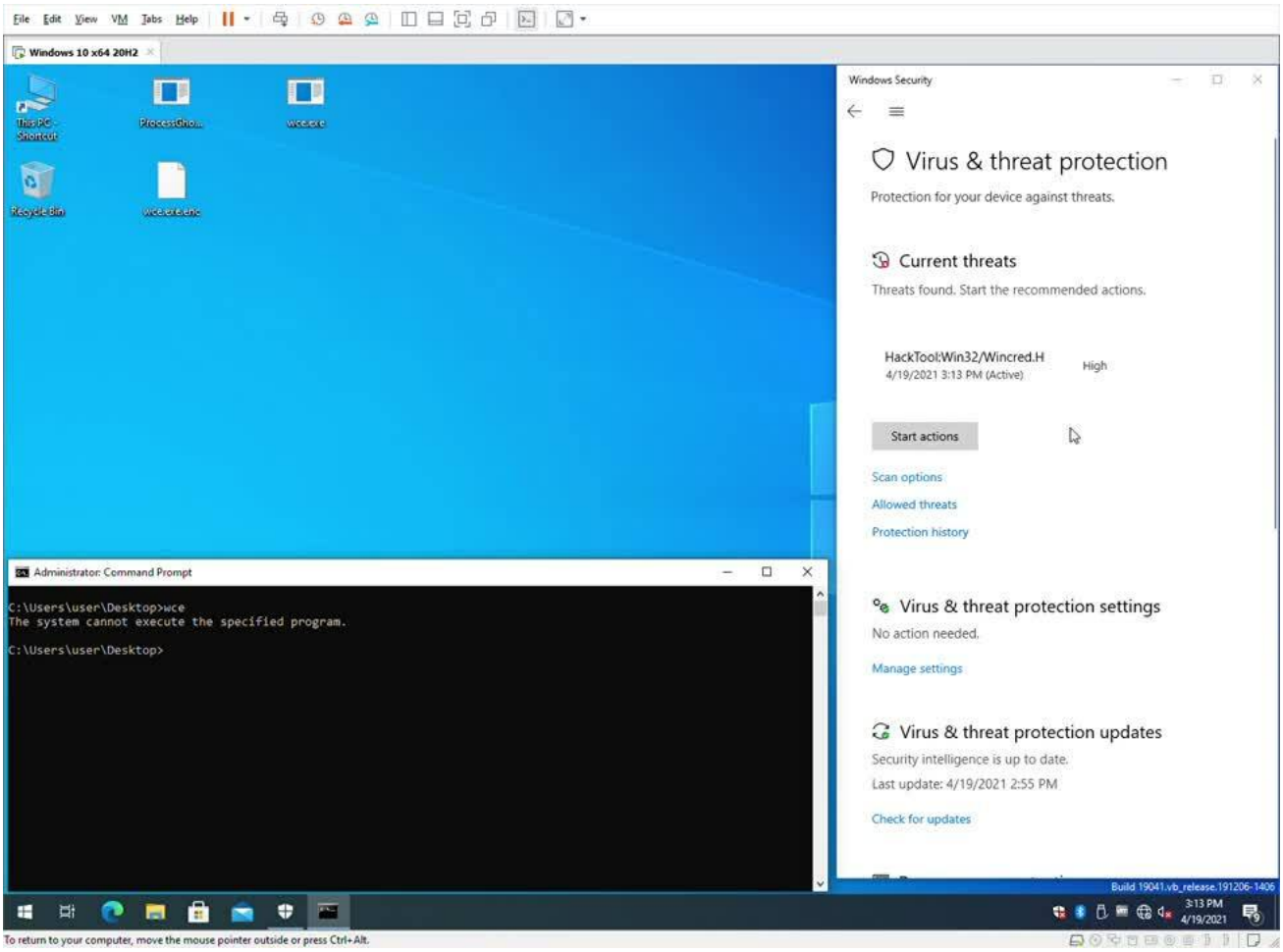


Antivirus callbacks are invoked upon thread creation, which occurs after the file is deleted. Attempts to open the file or perform I/O on the deleted file will fail with STATUS_FILE_DELETED. Attempts to open the file before deletion is complete will fail with STATUS_DELETE_PENDING.

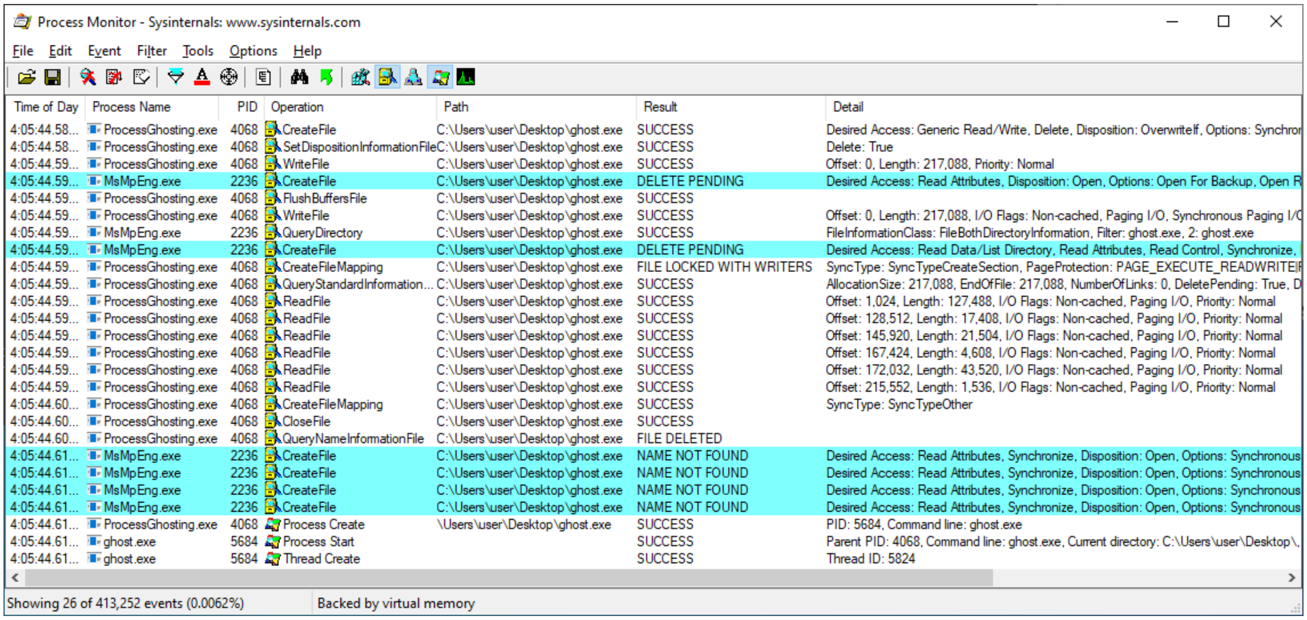
This type of tampering can be applied to DLLs as well, because DLLs are mapped image sections.

Demo

The video below shows Windows Defender detecting and blocking execution of a Potentially Unwanted Program (PUP), Windows Credential Editor, which can be used by attackers for lateral movement. It then shows how Ghosting interferes with Defender's ability to scan and block the PUP.

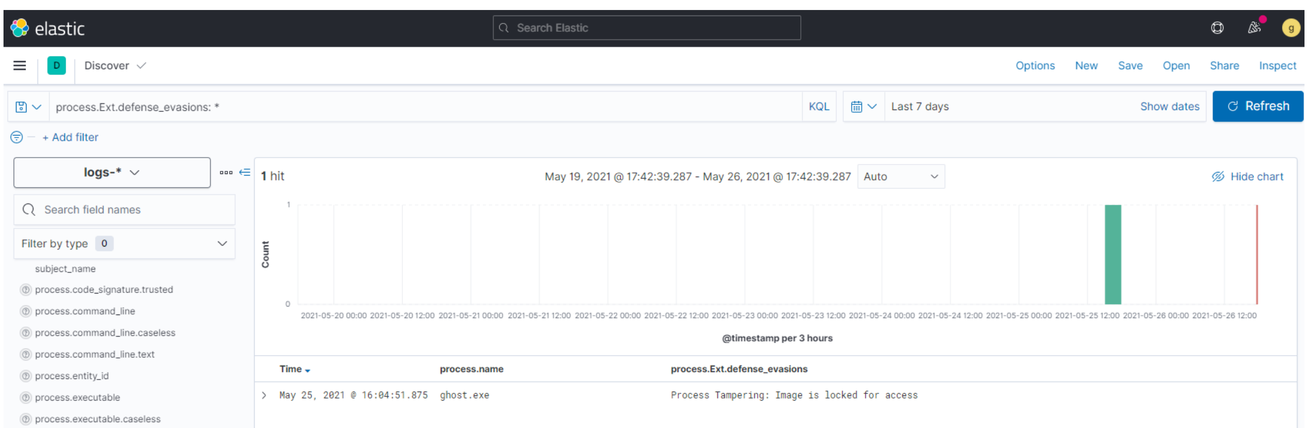


Examining system activity during the demo, we can see Defender initially attempting to open the payload executable to scan it, but failing because the file is in a delete-pending state. Later attempts to open it fail because the file has already been deleted. The payload (ghost.exe) executes without issue.



Detection

Elastic Security detects a variety of process image tampering techniques including Doppelgänging, Herpaderping, and Ghosting. It does this by checking the FILE_OBJECT for abnormalities during the process creation callback. These are reported in process creation events under **process.Ext.defense_evasions**.



Comparing techniques

Building upon a useful table from the Process Herpaderping documentation, we can compare the basic API flow across the various techniques:

Type	Technique
Hollowing	map -> modify section -> execute

Doppelgänging	transact -> write -> map -> rollback -> execute
Herpaderping	write -> map -> modify -> execute -> close
Ghosting	delete pending -> write -> map -> close(delete) -> execute

Conclusion

In this blog, we surveyed the state of the art in Windows executable image tampering attacks, then disclosed a new such attack. We then demonstrated this attack bypassing common security software, and showed how to detect it using freely available software.

To find threats like process tampering in your environment, install the latest version of Elastic Security on Elastic Cloud, and be sure to take advantage of our quick start training to set yourself up for success. Happy hunting!

Responsible disclosure*: We filed a bug report with MSRC on 2021-05-06, including a draft of this blog post, a demonstration video, and source code for a PoC. They responded on 2021-05-10 indicating that this does not meet their bar for servicing, per https://aka.ms/windowscriteria.*

References

1. With some exceptions, such as the System and Registry processes.

2. These examples are pseudocode.

3. https://go.microsoft.com/fwlink/?LinkId=140636 Page 32, "File Deletion Semantics"