Abusing Windows' Implementation of Fork() for Stealthy Memory Operations

billdemirkapi.me/abusing-windows-implementation-of-fork-for-stealthy-memory-operations/

Bill Demirkapi November 26, 2021

Bill Demirkapi

Nov 25, 2021 • 9 min read

Note: Another researcher recently tweeted about the technique discussed in this blog post, this is addressed in the last section of the blog (warning, spoilers!).

To access information about a running process, developers generally have to open a handle to the process through the OpenProcess API specifying a combination of 13 different *process access rights:*

1. `PROCESS_ALL_ACCESS` - All possible access rights for a process.
2. `PROCESS_CREATE_PROCESS` - Required to create a process.
3. `PROCESS_CREATE_THREAD` - Required to create a thread.
4. `PROCESS_DUP_HANDLE` - Required to duplicate a handle using DuplicateHandle.
5. `PROCESS_QUERY_INFORMATION` - Required to retrieve general information about a process such as its token, exit code, and priority class.
6. `PROCESS_QUERY_LIMITED_INFORMATION` - Required to retrieve certain limited information about a process.
7. `PROCESS_SET_INFORMATION` - Required to set certain information about a process such as its priority.
8. `PROCESS_SET_QUOTA` - Required to set memory limits using SetProcessWorkingSetSize.
9. `PROCESS_SUSPEND_RESUME` - Required to suspend or resume a process.
10. `PROCESS_TERMINATE` - Required to terminate a process using TerminateProcess.
11. `PROCESS_VM_OPERATION` - Required to perform an operation on the address space of a process (VirtualProtectEx, WriteProcessMemory).
12. `PROCESS_VM_READ` - Required to read memory in a process using ReadProcessMemory.
13. `PROCESS_VM_WRITE` - Required to write memory in a process using WriteProcessMemory.

The access rights requested will impact whether or not a handle to the process is returned. For example, a normal process running under a standard user can open a SYSTEM process for querying basic information, but it cannot open that process with a privileged access right such as `PROCESS_VM_READ`.

In the real world, the importance of process access rights can be seen in the restrictions anti-virus and anti-cheat products place on certain processes. An anti-virus might register a process handle create callback to prevent processes from opening the Local Security Authority Subsystem Service (LSASS) which could contain sensitive credentials in its memory. An anti-cheat might prevent processes from opening the game they are protecting, because cheaters can access key regions of the game memory to gain an unfair advantage.

When you look at the thirteen process access rights, do any of them strike out as potentially malicious? I investigated that question by taking a look at the drivers for several anti-virus products. Specifically, what access rights did they filter for in their process handle create callbacks? I came up with this subset of access rights that were often directly associated with potentially malicious operations: `PROCESS_ALL_ACCESS`,

`PROCESS_CREATE_THREAD`, `PROCESS_DUP_HANDLE`, `PROCESS_SET_INFORMATION`, `PROCESS_SUSPEND_RESUME`, `PROCESS_TERMINATE`, `PROCESS_VM_OPERATION`, `PROCESS_VM_READ`, and `PROCESS_VM_WRITE`.

This leaves four other access rights that were discovered to be largely ignored:

1. `PROCESS_QUERY_INFORMATION` - Required to retrieve general information about a process such as its token, exit code, and priority class.
2. `PROCESS_QUERY_LIMITED_INFORMATION` - Required to retrieve certain limited information about a process.
3. `PROCESS_SET_QUOTA` - Required to set memory limits using SetProcessWorkingSetSize.
4. `PROCESS_CREATE_PROCESS` - Required to create a process.

These access rights were particularly interesting because if we could find a way to abuse any of them, we could potentially evade the detection of a majority of anti-virus products.

Most of these remaining rights cannot modify important aspects of a process. `PROCESS_QUERY_INFORMATION` and `PROCESS_QUERY_LIMITED_INFORMATION` are purely for reading informational details about a process. `PROCESS_SET_QUOTA` does impact the process, but does not provide much surface to abuse. For example, being able to set a processes' performance limits provides limited usefulness in an attack.

What about `PROCESS_CREATE_PROCESS`? This access right allows a caller to "create a process" using the process handle, but what does that mean?

In practice, someone with a process handle containing this access right can create processes on behalf of that process. In the following sections, we will explore existing techniques that abuse this access right and its undiscovered potential.

Parent Process Spoofing

An existing evasion technique called "parent process ID spoofing" is used when a malicious application would like to create a child process under a different process. This allows an attacker to create a process while having it appear as if it was launched by another legitimate application.

At a high-level, common implementations of parent process ID spoofing will:

1. Call InitializeProcThreadAttributeList to initialize an attribute list for the child process.
2. Use OpenProcess to obtain a `PROCESS_CREATE_PROCESS` handle to the fake parent process.
3. Update the previously initialized attribute list with the parent process handle using UpdateProcThreadAttribute.
4. Create the child process with CreateProcess, passing extended startup information containing the process attributes.

This technique provides more usefulness than just being able to spoof the parent process of a child. It can be used to attack the parent process itself as well.

When creating a process, if the attacker specifies `TRUE` for the InheritHandles argument, all inheritable handles present in the parent process will be given to the child. For example, if a process has an inheritable thread handle and an attacker would like to obtain this handle indirectly, the attacker can abuse parent process spoofing to create their own malicious child process which inherits these handles.

The malicious child process would then be able to abuse these inherited handles in an attack against the parent process, such as a child using asynchronous procedure calls (APCs) on the parent's thread handle. Although this variation of the technique does require that the parent have critical handles set to be inheritable; several common applications, such as Firefox and Chrome, have inheritable thread handles.

Ways to Create Processes

The previous section explored one existing attack that used the high-level kernel32.dll function CreateProcess, but this is not the only way to create a process. Kernel32 provides abstractions such as CreateProcess which allow developers to avoid having to use ntdll functions directly.

When taking a look under the hood, kernel32 uses ntdll functions and does much of the heavy lifting required to perform NtXx calls. OpenProcess uses NtCreateUserProcess, which has the following function prototype:

```
NTSTATUS NTAPI
NtCreateUserProcess (
    PHANDLE ProcessHandle,
    PHANDLE ThreadHandle,
    ACCESS_MASK ProcessDesiredAccess,
    ACCESS_MASK ThreadDesiredAccess,
    POBJECT_ATTRIBUTES ProcessObjectAttributes,
    POBJECT_ATTRIBUTES ThreadObjectAttributes,
    ULONG ProcessFlags,
    ULONG ThreadFlags,
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    PPROCESS_CREATE_INFO CreateInfo,
    PPROCESS_ATTRIBUTE_LIST AttributeList
    );
```

NtCreateUserProcess is not the only low-level function exposed to create processes. There are two legacy alternatives: `NtCreateProcess` and `NtCreateProcessEx`. Their function prototypes are:

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

NTSTATUS NTAPI
NtCreateProcessEx (
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ParentProcess,
    ULONG Flags,
    HANDLE SectionHandle,
    HANDLE DebugPort,
    HANDLE ExceptionPort,
    BOOLEAN InJob
    );
```

NtCreateProcess and NtCreateProcessEx are quite similar but offer a different route of process creation when compared to NtCreateUserProcess.

Forking Your Own Process

A lesser documented *limited* technique available to developers is the ability to fork processes on Windows. The undocumented function developers can use to fork their own process is RtlCloneUserProcess. This function does not directly call the kernel and instead is a wrapper around NtCreateUserProcess.

A minimal implementation of forking through NtCreateUserProcess can be achieved trivially. By calling NtCreateUserProcess with NULL for both object attribute arguments, NULL for the process parameters, an empty (but not NULL) create info argument, and a NULL attribute list; a fork of the current process will be created.

One question that arose when performing this research was: What is the difference between forking a process and creating a new process with handles inherited? Interestingly, the minimal forking mechanism present in Windows does not only include inheritable handles, but *private memory regions* too. Any dynamically allocated pages as part of the parent will be accessible at the same location in the child as well.

Both RtlCloneUserProcess and the minimal implementation described are publicly known techniques for simulating fork on Windows, but is there any use forking provides to an attacker?

In 2019, Microsoft Research Labs published a paper named "A fork() in the road", which discussed how what used to be a "clever hack" has "long outlived its usefulness and is now a liability". The paper discusses several areas, such as how fork is a "terrible abstraction" and how it compromises OS implementations. The section titled "FORK IN THE MODERN ERA" is particularly relevant:

> **Fork is insecure.** By default, a forked child inherits everything from its parent, and **the programmer is responsible for explicitly removing state that the child does not need** by: closing file descriptors (or marking them as close-on-exec), **scrubbing secrets from memory**, isolating namespaces using unshare() [52], etc. From a security perspective, the inherit by-default behaviour of fork violates the principle of least privilege.

This section covers the security risk that is posed by the ability to fork processes. Microsoft provides the example that a forked process "inherits everything from its parent" and that "the programmer is responsible for explicitly removing state that the child does not need". What happens when the programmer is a malicious attacker?

Forking a Remote Process

I propose a new method of abusing the limited fork functionality present in Windows. Instead of forking your *own* process, what if you forked a *remote* process? If an attacker could fork a remote process, they would be able to gain insight into the target process without needing a sensitive process access right such as `PROCESS_VM_READ`, which could be monitored by anti-virus.

With only a `PROCESS_CREATE_PROCESS` handle, an attacker can fork or "duplicate" a process and access any secrets that are present in it. When using the legacy NtCreateProcess(Ex) variant, forking a remote process is relatively simple.

By passing NULL for the SectionHandle and a `PROCESS_CREATE_PROCESS` handle of the target for the ParentProcess arguments, a fork of the remote process will be created and an attacker will receive a handle to the forked process. Additionally, as long as the attacker does not create any threads, *no process creation callbacks will fire*. This means that an attacker could read the sensitive memory of the target and anti-virus wouldn't even know that the child process had been created.

When using the modern NtCreateUserProcess variant, all an attacker needs to do is use the previous minimal implementation of forking your own process but pass the target process handle as a PsAttributeParentProcess in the attribute list.

With the child handle, an attacker could read sensitive memory from the target application for a variety of purposes. In the following sections, we'll cover approaches to detection and an example of how an attacker could abuse this in a real attack.

Example: Anti-Virus Tampering

Some commercial anti-virus solutions may include self-integrity features designed to combat tampering and information disclosure. If an attacker could access the memory of the anti-virus process, it is possible that sensitive information about the system or the anti-virus itself could be abused.

With Process Forking, an attacker can gain access to both private memory and inheritable handles with only a `PROCESS_CREATE_PROCESS` handle to the victim process. A few examples of attacks include:

1. An attacker could read the encryption keys that are used to communicate with a trusted anti-virus server to decrypt or potentially tamper with this line of communication. For example, an attacker could pose as a man-in-the-middle (MiTM) with these encryption keys to prevent the anti-virus client from communicating alerts or spoof server responses to further tamper with the client.
2. An attacker could gain access to sensitive information about the system that was provided by the kernel. This information could include data from kernel callbacks that an attacker otherwise would not have access to from usermode.
3. An attacker could gain access to any handle the anti-virus process holds that is marked as inheritable. For example, if the anti-virus protects certain files from being accessed, such as sensitive configuration files, an attacker may be able to inherit a handle opened by the anti-virus process itself to access that protected file.

Example: Credential Dumping

One obvious target for a stealthy memory reading technique such as this is the Local Security Authority Subsystem Service (LSASS). LSASS is often the target of attackers that wish to capture the credentials for the current machine.

In a typical attack, a malicious program such as Mimikatz directly interfaces with LSASS on the victim machine, however, a stealthier alternative has been to dump the memory of LSASS for processing on an attacker machine. This is to avoid putting a well-known malicious program such as Mimikatz on the victim environment which is much more likely to be detected.

With Process Forking, an attacker can evade defensive solutions that monitor or prevent access to the LSASS process by dumping the memory of an LSASS fork instead:

1. Set debug privileges for your current process if you are not already running as SYSTEM.
2. Open a file to write the memory dump to.
3. Create a fork child of LSASS.
4. Use the common MiniDumpWriteDump API on the forked child.
5. Exfiltrate the dump file to an attacker machine for further processing.

Proof of Concept

A simple proof-of-concept utility and library have been published on GitHub.

Conclusion

Process Forking still requires that an attacker would have access to the victim process in the default Windows security model. Process Forking does not break integrity boundaries and attackers are restricted to processes running at the same privilege level they are. What Process Forking does offer is a largely ignored alternative to handle rights that are known to be potentially malicious.

Remediation may be difficult depending on the context of the solution relying on handle callbacks. An anti-cheat defending a single process may be able to get away with stripping `PROCESS_CREATE_PROCESS` handles entirely, but anti-virus solutions protecting multiple processes attempting a similar fix could face compatibility issues. It is recommended that vendors who opt to strip this access right initially audit its usage within customer environments and limit the processes they protect as much as possible.

Didn't I see this on Twitter yesterday?

> Did you know that it is possible to read memory using a PROCESS_CREATE_PROCESS handle? Just call NtCreateProcessEx to clone the target process (and its entire address space), and then read anything you want from there.😎
>
> — diversenok (@diversenok_zero) November 25, 2021

Yesterday morning, I saw this interesting tweet from @diversenok_zero explaining the same method discussed in this blog post.

One approach I like to take with new research or methods I find that I haven't investigated thoroughly yet is to generate a SHA256 hash of the idea and then post it somewhere publicly where the timestamp is recorded. This way, in cases like this where my research conflicts with what another researcher was working on, I can always prove I discovered the trick on a certain date. In this case, on June 19th 2021, I posted a public GitHub gist of the following SHA256 hash:

```
D779D38405E8828F5CB27C2C3D75867C6A9AA30E0BD003FECF0401BFA6F9C8C7
```

> You can read the memory of any process that you can open a PROCESS_CREATE_PROCESS handle to by calling NtCreateProcessEx using the process handle as the ParentHandle argument and providing NULL for the section argument.

If you generate a SHA256 hash of the quote above, you'll notice it matches the one I publicly posted back in June.

I have been waiting to publish this research because I am currently in the process of responsibly disclosing the issue to vendors whose products are impacted by this attack. Since the core theory of my research was shared publicly by @diversenok_zero, I decided it would be alright to share what I found around the technique as well.