Thread and Process State Change

windows-internals.com/thread-and-process-state-change

By Yarden Shafir

a.k.a: EDR Hook Evasion – Method #4512

Every couple of weeks a new build of Windows Insider gets released. Some have lots of changes and introduce completely new features, some only have minor bug fixes, and some simply insist on crashing repeatedly for no good reason. A few months ago one of those builds had a few surprising changes — It introduced 2 new object types and 4 new system calls, not something that happens every day. So of course I went investigating. What I discovered is a confusingly over-engineered feature, which was added to solve a problem that could have been solved by much simpler means and which has the side effect of supplying attackers with a new way to evade EDR hooks.

Suspending and Resuming Threads – Now With 2 Extra Steps!

The problem that this feature is trying to solve is this: what happens if a process suspends a thread and then terminates before resuming it? Unless some other part of the system realizes what happened, the thread will remain suspended forever and will never resume its execution. To solve that, this new feature allows suspending and resuming threads and processes through the new object types, which will keep track of the suspension state of the threads or processes. That way, when the object is destroyed (for example, when the process that created it is terminated), the system will reset the state of the target process or thread by suspending or resuming it as needed.

This feature is pretty easy to use – the caller first needs to call `NtCreateThreadStateChange` (or `NtCreateProcessStateChange`. Both cases are almost identical but we'll stay with the thread case for simplicity) to create a new object of type `PspThreadStateChangeType`. This object type is not documented, but its internal structure looks something like this:

```
struct _THREAD_STATE_OBJECT
{
    PETHREAD Thread;
    EX_PUSH_LOCK Lock;
    ULONG ThreadSuspendCount;
} THREAD_STATE_OBJECT, *PTHREAD_STATE_OBJECT;
```

`NtCreateThreadStateChange` has the following prototype:

```
NTSTATUS
NtCreateThreadStateChange (
    _Out_ PHANDLE StateChangeHandle,
```

```
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ThreadHandle,
    _In_ ULONG Unused
);
```

The 2 arguments we are interested in are the first one, which will receive a handle to the new object, and the fourth — a handle to the thread that will be referenced by the structure. Any future suspend or resume operation that will be done through this object can only work on the thread that's being passed into this function. `NtCreateProcessStateChange` will create a new object instance, set the thread pointer to the requested thread, and initialize the lock and count fields to zero.

When calling `NtCreateProcessStateChange` to operate on a process, the thread handle will be replaced with a process handle and the object that will be created will be of type `PspProcessStateChangeType`. The only change in the structure is that the `ETHREAD` pointer is replaced with an `EPROCESS` pointer.

The next step is calling `NtChangeThreadState` (or `NtChangeProcessState`, if operating on a process). This function receives a handle to the thread state change object, a handle to the same thread that was passed when creating the object, and an action, which is an enum value:

```
typedef enum _THREAD_STATE_CHANGE_TYPE
{
    ThreadStateChangeSuspend = 0,
    ThreadStateChangeResume = 1,
    ThreadStateChangeMax = 2,
} THREAD_STATE_CHANGE_TYPE, *PTHREAD_STATE_CHANGE_TYPE;

typedef enum _PROCESS_STATE_CHANGE_TYPE
{
    ProcessStateChangeSuspend = 0,
    ProcessStateChangeResume = 1,
    ProcessStateChangeMax = 2,
} PROCESS_STATE_CHANGE_TYPE, *PPROCESS_STATE_CHANGE_TYPE;
```

It also receives an "Extended Information" variable and its length, both of which are unused and must be zero, and another reserved argument that must also be zero. The function will validate that the thread pointed to by the thread state change object is the same as the thread whose handle was passed into the function, and then call the appropriate function based on the requested action – `PsSuspendThread` or `PsMultiResumeThread`. Then it will increment or decrement the `ThreadSuspendCount` field based on the action that was performed. There are `2` limitations enforced by the suspend count:

1. A thread cannot be resumed if the object's `ThreadSuspendCount` is zero, even if the thread is currently suspended. It must be suspended and resumed using the state change API, otherwise things will start acting funny.
2. A thread cannot be suspended if `ThreadSuspendCount` is `0x7FFFFFFF`. This is meant to avoid overflowing the counter. However, this is a weird limitation since `KeSuspendThread` (the internal function called from `PsSuspendThread`) already enforces a suspension limit of `127` through the thread's `SuspendCount` field, and will throw an error `STATUS_SUSPEND_COUNT_EXCEEDED` if the count exceeds that.

So far this works like the classic suspend and resume mechanism, just with a few extra steps. A caller still needs to make an API call to suspend a thread or process and another one to resume it. But the benefit of having new object types is that objects can have kernel routines that get called for certain operations related to the object, such as open, close and delete:

dx (*(nt!_OBJECT_TYPE**)&nt!PspThreadStateChangeType)->TypeInfo
(*(nt!_OBJECT_TYPE**)&nt!PspThreadStateChangeType)->TypeInfo [Type: _OBJECT_TYPE_INITIALIZER]
[+0x000] Length : 0x78 [Type: unsigned short]
[+0x002] ObjectTypeFlags : 0x6 [Type: unsigned short]
[+0x002 (0: 0)] CaseInsensitive : 0x0 [Type: unsigned char]
[+0x002 (1: 1)] UnnamedObjectsOnly : 0x1 [Type: unsigned char]
[+0x002 (2: 2)] UseDefaultObject : 0x1 [Type: unsigned char]
[+0x002 (3: 3)] SecurityRequired : 0x0 [Type: unsigned char]
[+0x002 (4: 4)] MaintainHandleCount : 0x0 [Type: unsigned char]
[+0x002 (5: 5)] MaintainTypeList : 0x0 [Type: unsigned char]
[+0x002 (6: 6)] SupportsObjectCallbacks : 0x0 [Type: unsigned char]
[+0x002 (7: 7)] CacheAligned : 0x0 [Type: unsigned char]
[+0x003 (0: 0)] UseExtendedParameters : 0x0 [Type: unsigned char]
[+0x003 (7: 1)] Reserved : 0x0 [Type: unsigned char]
[+0x004] ObjectTypeCode : 0x0 [Type: unsigned long]
[+0x008] InvalidAttributes : 0x92 [Type: unsigned long]
[+0x00c] GenericMapping [Type: _GENERIC_MAPPING]
[+0x01c] ValidAccessMask : 0x1f0001 [Type: unsigned long]
[+0x020] RetainAccess : 0x0 [Type: unsigned long]
[+0x024] PoolType : PagedPool (1) [Type: _POOL_TYPE]
[+0x028] DefaultPagedPoolCharge : 0x70 [Type: unsigned long]
[+0x02c] DefaultNonPagedPoolCharge : 0x0 [Type: unsigned long]
[+0x030] DumpProcedure : 0x0 [Type: void (__cdecl*)(void *,_OBJECT_DUMP_CONTROL *)]
[+0x038] OpenProcedure : 0x0 [Type: long (__cdecl*)(_OB_OPEN_REASON,char,_EPROCESS *,void *,unsigned long *,unsigned long)]
[+0x040] CloseProcedure : 0x0 [Type: void (__cdecl*)(_EPROCESS *,void *,unsigned __int64,unsigned __int64)]

[+0x048] DeleteProcedure : 0xfffff80265650d20 [Type: void (__cdecl*)(void *)]
[+0x050] ParseProcedure : 0x0 [Type: long (__cdecl*)(void *,void *,_ACCESS_STATE *,char,unsigned long,_UNICODE_STRING *,_UNICODE_STRING *,void *,_SECURITY_QUALITY_OF_SERVICE *,void * *)]
[+0x050] ParseProcedureEx : 0x0 [Type: long (__cdecl*)(void *,void *,_ACCESS_STATE *,char,unsigned long,_UNICODE_STRING *,_UNICODE_STRING *,void *,_SECURITY_QUALITY_OF_SERVICE *,_OB_EXTENDED_PARSE_PARAMETERS *,void * *)]
[+0x058] SecurityProcedure : 0xfffff802656bffd0 [Type: long (__cdecl*)(void *,_SECURITY_OPERATION_CODE,unsigned long *,void *,unsigned long *,void * *,_POOL_TYPE,_GENERIC_MAPPING *,char)]
[+0x060] QueryNameProcedure : 0x0 [Type: long (__cdecl*)(void *,unsigned char,_OBJECT_NAME_INFORMATION *,unsigned long,unsigned long *,char)]
[+0x068] OkayToCloseProcedure : 0x0 [Type: unsigned char (__cdecl*)(_EPROCESS *,void *,void *,char)]
[+0x070] WaitObjectFlagMask : 0x0 [Type: unsigned long]
[+0x074] WaitObjectFlagOffset : 0x0 [Type: unsigned short]
[+0x076] WaitObjectPointerOffset : 0x0 [Type: unsigned short]

`PspThreadStateChangeType` has `2` registered procedures – the security procedure, which is `SeDefaultObjectMethod` and not too interesting to look at in this case as it is the default function, and the delete procedure, which is `PspDeleteThreadStateChange`. This function will get called every time a thread state change object is destroyed, and does a pretty simple thing:

```
NTSTATUS __fastcall PspDeleteThreadStateChange(_THREAD_STATE_OBJECT *ThreadStateObject)
{
  ULONG suspendCount; // er8

  suspendCount = (ULONG)ThreadStateObject->ThreadSuspendCount;
  if ( suspendCount )
    PsMultiResumeThread(ThreadStateObject->Thread, 0i64, suspendCount);
  return ObfDereferenceObjectWithTag(ThreadStateObject->Thread, 'cTsP');
}
```

If the target thread has a non-zero `ThreadSuspendCount`, the function will resume it as many times as it was suspended. As you can imagine, the process state change object also registers a delete procedure, `PspDeleteProcessStateChange`, which does something very similar.

New System Calls == New EDR Bypass

This is a nice, if slightly over-complicated, solution to the problem, but it has the unexpected side-effect of creating new and undocumented APIs to suspend and resume processes and threads. Since suspend and resume are very useful operations for attackers wishing to inject

code, the well-known `NtSuspendThread/Process` and `NtResumeThread/Process` APIs are some of the first system calls that are hooked by security solutions, hoping to detect those attacks.

Having new APIs that perform the same operations without going through the well-known and often-monitored system calls is a great chance for attackers to avoid detection by security solutions that don't keep up with recent changes (though I'm sure all EDR solutions have already started monitoring these new functions and have been doing so since this build was released. Right...?).

There is still a way to keep those same detections without following all of Microsoft's recent code changes – even though this feature adds new system calls, the internal kernel mechanism invoked by them remains the same. And in Windows 10, this mechanism is using a feature whose sole purpose is to help security solutions gain more information about the system and get them away from relying on user-mode hooks – ETW tracing. And more specifically, the Thread Intelligence ETW channel that was added specifically for security purposes. That channel notifies about events that are often interesting to security products, such as virtual memory protection changes, virtual memory writes, driver loads, and, as you probably already guessed, suspending and resuming threads and processes. EDRs that register for these ETW events and use them as part of their detection will not miss any event due to the new state change APIs since these events will be received in either case. Those that don't use them yet should probably open some Jira tickets that will be forgotten until this technique is found in the wild.

1 EDR Bypass + Windows Internals = 2 EDR Bypasses

However, this feature does create another interesting EDR bypass. As I mentioned, the suspended process or thread will automatically be resumed when the state change object gets destroyed. Normally, this would happen when the process that created the object either closes the only handle to it or exits – this automatically destroys all open handles held by the process. But an object only gets destroyed when all handles to it are closed and there are no more references to it. This means that if another process has an open handle to the state change object it won't get destroyed when the process that created it exits, and the suspended process or thread won't be resumed until the second process exits. This shouldn't happen under normal circumstances, but if a process duplicates its handle to a state change object into another process, it can safely exit without resuming the suspended process or thread.

But why would a process want to do that?

The ETW events that report that a process is being suspended or resumed contain a process ID of the process that performed the action – this way the EDR that consumes the event can correlate different events together and attribute them to a potentially malicious process. In this case, the PID would be the ID of the process in whose context the action happened. So let's say we create a process that suspends another process through a state change object,

then duplicates the handle into a third process and exits. The process state change object doesn't get destroyed et since there is still a running process with an open handle to it. Only when the other process exits, the duplicated handle gets closed and the suspended process gets resumed. But since the resume action happened in the context of the second process, which had nothing to do with the suspend action, that is the PID that will appear in the ETW event.

So, in this proposed scenario, a process will get suspended and later resumed, and ETW events will still be thrown for both actions. But these events will have happened in the context of 2 different processes so they will be difficult to link together, and it will be even more difficult to attribute the resume action to the first process without knowledge of this exact scenario. And we can be even smarter – a lot of security products ignore operations that are attributed to certain system processes. This makes sense, since those processes are not expected to be malicious but might have suspicious-looking activity, so it is easier to ignore them unless there is clear indication of code injection, to avoid false positives.

So we can even choose an innocent-looking Windows process to duplicate our handle into, to maximize the chances that the resume operation will be ignored completely. We just need to find a process that we can open a handle to and that will terminate at some point, to resume our suspended process.

Finally, Code!

In this PoC I simply create 2 notepad.exe processes. One will be suspended using a state change object, and the other will have the handle duplicated inside it. Then the PoC process exits but the suspended notepad remains suspended until the other notepad process is terminated:

```
#include <Windows.h>
#include <stdio.h>

EXTERN_C_START
NTSTATUS
NtCreateProcessStateChange (
    _Out_ PHANDLE StateChangeHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ PVOID ObjectAttributes,
    _In_ HANDLE ProcessHandle,
    _In_ ULONG Unknown
    );

NTSTATUS
NtChangeProcessState (
    _In_ HANDLE StateChangeHandle,
    _In_ HANDLE ProcessHandle,
```

```
    _In_ ULONG Action,
    _In_ PVOID ExtendedInformation,
    _In_ SIZE_T ExtendedInformationLength,
    _In_ ULONG64 Reserved
    );
EXTERN_C_END

int main ()
{
    HANDLE stateChangeHandle;
    PROCESS_INFORMATION procInfo;
    PROCESS_INFORMATION procInfo2;
    STARTUPINFOA startInfo;
    BOOL result;
    NTSTATUS status;

    stateChangeHandle = nullptr;

    ZeroMemory(&startInfo, sizeof(startInfo));
    startInfo.cb = sizeof(startInfo);
    result = CreateProcess(L"C:\\Windows\\System32\\notepad.exe",
                           NULL,
                           NULL,
                           NULL,
                           FALSE,
                           0,
                           NULL,
                           NULL,
                           &startInfo,
                           &procInfo);
    if (result == FALSE)
    {
        goto Exit;
    }
    CloseHandle(procInfo.hThread);
    result = CreateProcess(L"C:\\Windows\\System32\\notepad.exe",
                           NULL,
                           NULL,
                           NULL,
                           FALSE,
                           0,
                           NULL,
                           NULL,
                           &startInfo,
                           &procInfo2);
    if (result == FALSE)
```

```
    {
        goto Exit;
    }
    CloseHandle(procInfo2.hThread);

    status = NtCreateProcessStateChange(&stateChangeHandle,
                                        MAXIMUM_ALLOWED,
                                        NULL,
                                        procInfo.hProcess,
                                        0);
    if (!NT_SUCCESS(status))
    {
        printf("Failed creating process state change. Status: 0x%x\n",
status);
        goto Exit;
    }
    //
    // Action == 0 means Suspend
    //
    status = NtChangeProcessState(stateChangeHandle,
                                  procInfo.hProcess,
                                  ProcessStateChangeSuspend,
                                  NULL,
                                  0,
                                  0);
    if (!NT_SUCCESS(status))
    {
        printf("Failed changing process state. Status: 0x%x\n", status);
        goto Exit;
    }

    result = DuplicateHandle(GetCurrentProcess(),
                             stateChangeHandle,
                             procInfo2.hProcess,
                             NULL,
                             NULL,
                             TRUE,
                             DUPLICATE_SAME_ACCESS);
    if (result == FALSE)
    {
        printf("Failed duplicating handle: 0x%x\n", GetLastError());
        goto Exit;
    }

Exit:
    if (procInfo.hProcess != NULL)
```

```
    {
        CloseHandle(procInfo.hProcess);
    }
    if (procInfo2.hProcess != NULL)
    {
        CloseHandle(procInfo2.hProcess);
    }
    if (stateChangeHandle != NULL)
    {
        CloseHandle(stateChangeHandle);
    }
    return 0;
}
```

Like a lot of other cases, this feature started out as a well-intentioned attempt to solve a minor system issue. But an over-engineered design led to multiple security concerns and whole new EDR evasion techniques which turned the relatively small issue into a much larger one.