

Anatomy of the thread suspension mechanism in Windows (Windows Internals)

ntopcode.wordpress.com/2018/01/16/anatomy-of-the-thread-suspension-mechanism-in-windows-windows-internals/

January 16, 2018

```
NTSTATUS __fastcall PsSuspendProcess(PEPROCESS Process)

PVOID CriticalRegion; // rdi@1
PEPROCESS Process2; // rbp@1
char *RundownProtectValue; // r14@1
PETHREAD CurrentThread; // rax@2
NTSTATUS NtStatus; // ebx@2
PETHREAD CurrentThread2; // rsi@4

CriticalRegion = (PVOID)*MK_FP(__GS__, 0x188i64);
Process2 = Process;
--*((_WORD *)CriticalRegion + 0xF2);
RundownProtectValue = (char *)&Process->RundownProtect;
if ( (unsigned __int8)ExAcquireRundownProtection_0(&Process->RundownProtect) == 1 )
{
    LODWORD(CurrentThread) = PsGetNextProcessThread(Process2, NULL); // enumerate threads of the process
    NtStatus = 0;
    while ( 1 )
    {
        CurrentThread2 = CurrentThread;
        if ( !CurrentThread )
            break;
        PsSuspendThread((__int64)CurrentThread, NULL);
        LODWORD(CurrentThread) = PsGetNextProcessThread(Process2, CurrentThread2);
    }
    ExReleaseRundownProtection_0(RundownProtectValue);
}
else
{
    NtStatus = 0xC000010A; // STATUS_PROCESS_IS_TERMINATING
}
KeLeaveCriticalRegionThread(CriticalRegion);
return NtStatus;
```

Introduction

Process suspension is a technique which is quite well-known, and it is used for a variety of reasons (even by malicious software sometimes). The term “suspension” means “stopping” something, and in-case you did not guess it yet, “process suspension” is a technique to temporarily “stop” a running process. If you are suspended from school then you won’t be attending for the duration you’re suspended for, and when the term is used with a process, the process won’t be carrying out any operations whilst it’s suspended.

When you suspend a process, the threads of the process will be set to a suspended state; the threads of a process are responsible for processing the code belonging to the process – the CPU executes the instructions. Usually, a process will have more than one thread, and this will allow the process to execute several operations at the same time simultaneously. If we were to suspend one of the running threads, then the targeted thread would be postponed from carrying out any operations until it has been resumed. If we were to suspend each thread contained under the process object, then we would have successfully suspended the process! To cut it short, process

suspension is an operation which relies on suspending the threads of a process – this cuts off code execution until we resume the process, which consists of resuming each thread under the process which we put into a suspended state.

When a process is being spawned, there will be a main thread for the process and it will be in a suspended state until initialisation of the newly starting process has been completed. Even if the requester of the process spawn operation does not specify the `CREATE_SUSPENDED` flags, the process will still be started with a suspended state by the Windows loader until initialisation has been successfully completed. When the process is ready to start running its own code due to the Windows loader being finished, it will resume the main thread which is maintained with a suspended state until this point (unless the `CREATE_SUSPENDED` flags were specified by the requester of the process spawn). The resume operation for the main thread at this point in time will lead to a routine known as `NtResumeThread` (NTDLL) being invoked – a system call is performed by the `NtResumeThread` stub present in NTDLL to get the real `NtResumeThread` routine invoked (which resides under kernel-mode memory – `NTOSKRNL` which is the Windows Kernel to be precise).

This article will be broken into separated sections. The first section will be discussing user-mode, and the second section will be discussing kernel-mode. In both sections, suspending and resuming a process' threads will be discussed.

Section 1 – User-Mode

When it comes down to suspending a process from user-mode, you have a few options.

- Invoke `NtSuspendProcess` (NTDLL) which is undocumented. [1]
- Enumerate the threads of the targeted process and invoke `NtSuspendThread` (NTDLL). [2]
- Enumerate the threads of the targeted process and invoke `SuspendThread` (KERNEL32). [3]

[1] – The first method noted, via `NtSuspendProcess`, is the most minimal solution. At the same time however, it is also one of the most unreliable. `NtSuspendProcess` is not officially documented by Microsoft (despite being documented by third-parties for several years), which can only make you wonder why they are yet to document it themselves considering it is so widely exposed to the public already – it would take them barely any time to document it over at the Microsoft Developer Network (MSDN).

The function takes in one parameter only which needs to be the handle to the process being targeted for suspension, and thus the data-type of this singular parameter is of `HANDLE` (`VOID*`).

I've left a type-definition for the `NtSuspendProcess` routine below.

```
typedef NTSTATUS(NTAPI *pNtSuspendProcess)(
    HANDLE ProcessHandle
);
```

When you invoke `NtSuspendProcess` (NTDLL), the system performs a transition operation via a system call to cause `NtSuspendProcess` (NTOSKRNL) to become invoked. We can verify these findings by taking a look at `NTDLL.DLL` for the `NtSuspendProcess` exported routine.

```

; ----- S U B R O U T I N E -----
                                public ZwSuspendProcess
ZwSuspendProcess proc near      ; DATA XREF: .rdata:off_1801465F8↓o
                                ; .pdata:000000018016868C↓o
                                ; NtSuspendProcess
                                mov     r10, rcx
                                mov     eax, 181h
                                test    byte ptr ds:7FFE0300h, 1
                                jnz     short loc_1800A3455
                                syscall
                                retn

; -----
loc_1800A3455:                 ; CODE XREF: ZwSuspendProcess+10↓j
                                int     2Eh
                                ; DOS 2+ internal - EXECUTE COMMAND
                                ; DS:SI -> counted CR-terminated command string
                                retn
ZwSuspendProcess endp

4C 8B D1
88 81 01 00 00
F6 04 25 08 03 FE 7F 01
75 03
0F 05
C3

CD 2E
C3

```

NtSuspendProcess function prologue under NTDLL.

If we remember back to what I have previously said at the start of this article, process suspension works by suspending the threads of the targeted process. How does NtSuspendProcess work then? NtSuspendProcess will lead down a path which has one end-result only. The end-result is the threads of the targeted process being enumerated and each found thread during the enumeration being applied for suspension.

I've created a very straight-forward user-mode snippet based in C on invoking NtSuspendProcess (NTDLL) for demonstration purposes. You'll need to include the <stdio.h> and <windows.h> libraries and add a main entry-point routine to compile and test it out.

```

#define STATUS_INSUFFICIENT_RESOURCES 0xC000009A

typedef _Return_type_success_(return >= 0) LONG NTSTATUS;
typedef NTSTATUS *PNTSTATUS;

#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)

typedef NTSTATUS(NTAPI *pNtSuspendProcess)(
    HANDLE ProcessHandle
);

pNtSuspendProcess fNtSuspendProcess;

BOOLEAN InitializeExports()
{
    HMODULE hNtdll = GetModuleHandle("NTDLL");

    if (!hNtdll)
    {
        return FALSE;
    }

    fNtSuspendProcess = (pNtSuspendProcess)GetProcAddress(hNtdll,
        "NtSuspendProcess");

    if (!fNtSuspendProcess)
    {
        return FALSE;
    }

    return TRUE;
}

NTSTATUS NTAPI NtSuspendProcess(
    HANDLE ProcessHandle
)
{
    if (!fNtSuspendProcess)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fNtSuspendProcess(ProcessHandle);
}

BOOLEAN WINAPI SuspendProcess(
    HANDLE ProcessHandle
)
{
    if (!ProcessHandle)
    {
        return FALSE;
    }

    return (NT_SUCCESS(NtSuspendProcess(ProcessHandle)))
        ? TRUE : FALSE;
}

```

Here is a break-down on how the snippet is supposed to be used/work.

1. The InitializeExports routine (return-type of BOOLEAN) will setup NtSuspendProcess for usage via a dynamic import. pNtSuspendProcess is a type-definition for the function structure, and this is used with a global variable which gets pointed to the address of NtSuspendProcess. This means we can simply treat fNtSuspendProcess like a normal function however it will lead to NtSuspendProcess (NTDLL) invocation since it points to NtSuspendProcess (NTDLL) address.
2. After opening a handle to a process (with at-least the PROCESS_SUSPEND_RESUME access rights present), it can be passed as the parameter for the SuspendProcess routine. This routine has the WINAPI macro however this simply represents __stdcall, and the NTAPI macro also represents __stdcall – __stdcall is a calling convention. Just to be clear and not cause potential confusion, SuspendProcess is not a Win32 API routine, I simply used the WINAPI macro to represent __stdcall because I prefer to do so.
3. The SuspendProcess routine will call the NtSuspendProcess routine. The NtSuspendProcess routine which is manually declared will call fNtSuspendProcess which points to the address of NtSuspendProcess (NTDLL) as previously noted.

[2] – The second method noted, via enumeration of the targeted process' threads and then calling NtSuspendThread on each one, is also potentially unstable due to it being undocumented like the NtSuspendProcess method noted under [1], however it does the job.

NtSuspendThread is not an “officially” documented routine in the same way that NtSuspendProcess isn't either. However, NtSuspendThread does not have a complicated structure. The routine takes in two parameters: HANDLE, and ULONG* (PULONG). The former is for the handle of the thread being targeted by suspension (we must first acquire a handle to the thread we are targeting in the same sense that we must have a handle to the target process before we can use NtSuspendProcess – the handle must have at-least THREAD_SUSPEND_RESUME access rights), and the latter is to do with a counter of suspensions (as far as I am aware – although it is entirely optional and whenever I've needed to use this routine I've never had to make use of the second parameter).

I've left a type-definition for the NtSuspendThread routine below.

```
typedef NTSTATUS(NTAPI *pNtSuspendThread)(
    HANDLE ThreadHandle,
    PULONG PreviousSuspendCount OPTIONAL
);
```

Just like with NtSuspendProcess (NTDLL), when we invoke NtSuspendThread (NTDLL), a system call is performed by the system to transition from user-mode to kernel-mode; the end-result is NtSuspendThread (NTOSKRNL) being invoked. The handy thing about NtSuspendThread though is that we can suspend only X amount of threads, and leave some threads resumed, a perk which does not come with NtSuspendProcess of course.

A quick snippet of how you would go about using NtResumeThread is left below for you to take a quick peek at.

```

typedef NTSTATUS(NTAPI *pNtSuspendThread)(
    HANDLE ThreadHandle,
    PULONG PreviousSuspendCount OPTIONAL
);

pNtSuspendThread fNtSuspendThread;

BOOLEAN InitializeExports()
{
    HMODULE hNtdll = GetModuleHandle("NTDLL");

    if (!hNtdll)
    {
        return FALSE;
    }

    fNtSuspendThread = (pNtSuspendThread)GetProcAddress(hNtdll,
        "NtSuspendThread");

    if (!fNtSuspendProcess ||
        fNtSuspendThread)
    {
        return FALSE;
    }

    return TRUE;
}

NTSTATUS NTAPI NtSuspendThread(
    HANDLE ThreadHandle,
    PULONG PreviousSuspendCount
)
{
    if (!fNtSuspendThread)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fNtSuspendThread(ThreadHandle,
        PreviousSuspendCount);
}

BOOLEAN WINAPI SuspendThreadWrapper(
    HANDLE ThreadHandle
)
{
    if (!ThreadHandle)
    {
        return FALSE;
    }

    return (NT_SUCCESS(NtSuspendThread(
        ThreadHandle,
        NULL))) ? TRUE : FALSE;
}

```

In case you're wondering why I took the leap with the routine naming and used "SuspendThreadWrapper" instead of "SuspendThread", it's because there's already a Win32 API routine called "SuspendThread". The SuspendThread routine is exported by KERNEL32.DLL and it will do the same thing we are doing in our wrapper routine (more-or-less at-least) – it will call NtSuspendThread (NTDLL).

Bear in mind that you will need to have at-least THREAD_SUSPEND_RESUME access rights on the handle you attempt to use with NtSuspendThread. To acquire the handle to the thread, you could use NtOpenThread (NTDLL), or preferably it's Win32 API equivalent which would be OpenThread (KERNEL32).

Unless you need to suspend a certain amount of threads of a process, it is likely going to be more convenient to use NtSuspendProcess. The reason being that NtSuspendProcess (NTOSKRNL – invoked after the NTDLL system call) will call a kernel-mode routine to handle the process suspension, and this routine will automatically enumerate through all the threads of the targeted process and call another kernel-mode routine to handle suspension. Whereas, if you are enumerating the threads and suspending them yourself, you're doing more yourself to replicate the same functionality. Unless a routine like NtOpenProcess/NtSuspendProcess has been hooked and you don't fancy bypassing the set hooks, and NtOpenThread/NtSuspendThread was forgotten about, then you may as well use NtSuspendProcess if you need to suspend a process.

[3] – The third method noted, via enumeration of the targeted process' threads and then calling SuspendThread on each one, is without a doubt the most stable technique. At the same time though, it's a bit more "obvious". You'll neither be able to perform a manual transition from user-mode to kernel-mode to bypass any hooks since it isn't a Nt* routine which is one down-fall – it is still the most documented mechanism for accomplishing process suspension though, and for this reason, it is recommended that you use this technique unless you have specific requirements which prevents you from doing so.

Despite SuspendThread being the most documented mechanism for accomplishing suspension functionality, NtSuspendProcess/NtSuspendThread have been around for an extremely long time, since Windows 2000 I believe. The chances of these routines being deprecated are extremely small, it would be like making NtOpenProcess obsolete, which I am sure is not going to happen any-time soon. They are core routines in the Windows Kernel, so whether you go down the undocumented and less-stable route for this or not, as long as you know how to use the routines properly you likely won't have any issues from patch updates any-time soon to say the least.

For the record, SuspendThread (KERNEL32) will call NtSuspendThread (NTDLL). As expected, NtSuspendThread (NTDLL) will perform a system call and then NtSuspendThread (NTOSKRNL) will be invoked; NtSuspendThread is not exported by the Windows Kernel however it can still be accessed if you can find the address via pattern scanning or the System Service Descriptor Table (SSDT).

Since SuspendThread is documented by Microsoft over at the Microsoft Developer Network (MSDN), I've left the type definition for the routine below along with the link to the official documentation.

```
typedef DWORD(WINAPI *pSuspendThread)(
    HANDLE ThreadHandle
);
```

[https://msdn.microsoft.com/en-us/library/windows/desktop/ms686345\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686345(v=vs.85).aspx)

The routine can be used the same way as the `SuspendThreadWrapper` routine we saw earlier which would call the `NtSuspendThread` proxy routine. The return value for `SuspendThread` (KERNEL32) will be the value returned for the `PreviousSuspendCount` parameter which is the parameter we previously ignored with the `NtSuspendThread` call.

```
DWORD Status = SuspendThread(GetCurrentThread());

if (Status)
{
    printf("Thread suspended\n");
}
```

If the suspension operation is successful, you'll never reach the conditional statement because the thread which is supposed to be processing those instructions will be in a suspended state. The conditional statement won't be reached until the suspended thread has been resumed (by another thread which is running or by another process).

We've talked a bit about how we can suspend a process (and how this relies on targeting the threads) or suspend specific threads, but what about resuming them? We'll move onto this now before progressing to the kernel-mode section of this article.

When it comes down to resuming a process from user-mode, you have a few options.

- Invoke `NtResumeProcess` (NTDLL) which is undocumented. [1]
- Enumerate the threads of the targeted process and invoke `NtResumeThread` (NTDLL). [2]
- Enumerate the threads of the targeted process and invoke `ResumeThread` (KERNEL32). [3]

In-case you're yet to notice, we are doing the same as when we are performing suspension, but in-reverse. For example, our first option for process suspension would be by invoking `NtSuspendProcess` (NTDLL), and our first option for resuming a process would be to invoke `NtResumeProcess` (NTDLL). `NtSuspendProcess`, `NtSuspendThread` and `SuspendThread` all have a "Resume" variant; simply replace the "Suspend" key-word in the function routines with "Resume" and bobs your uncle!

As noted with `NtSuspendProcess` and `NtSuspendThread` about stability, `NtResumeProcess` and `NtResumeThread` are in the same boat; they aren't officially documented by Microsoft and there must be a reason for this. However, worrying about it isn't really something you should do, considering they are core routines used in Windows and the likelihood of them being made obsolete is very low.

I've left an example below for `NtResumeProcess` and `NtResumeThread`. It is more-or-less the same as the process suspension examples, except for resume instead. You can use `ResumeThread` the exact same way you use `SuspendThread`.


```

typedef NTSTATUS(NTAPI *pNtResumeProcess)(
    HANDLE ProcessHandle
);

typedef NTSTATUS(NTAPI *pNtResumeThread)(
    HANDLE ThreadHandle,
    PULONG PreviousSuspendCount OPTIONAL
);

pNtResumeProcess fNtResumeProcess;
pNtResumeThread fNtResumeThread;

BOOLEAN InitializeExports()
{
    HMODULE hNtdll = GetModuleHandle("NTDLL");

    if (!hNtdll)
    {
        return FALSE;
    }

    fNtResumeProcess = (pNtResumeProcess)GetProcAddress(hNtdll,
        "NtResumeProcess");

    fNtResumeThread = (pNtResumeThread)GetProcAddress(hNtdll,
        "NtResumeThread");

    if (!fNtResumeProcess ||
        !fNtResumeThread)
    {
        return FALSE;
    }

    return TRUE;
}

NTSTATUS NTAPI NtResumeProcess(
    HANDLE ProcessHandle
)
{
    if (!fNtResumeProcess)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fNtResumeProcess(ProcessHandle);
}

NTSTATUS NTAPI NtResumeThread(
    HANDLE ThreadHandle,
    PULONG PreviousSuspendCount
)
{
    if (!fNtResumeThread)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fNtResumeThread(ThreadHandle,

```

```

        PreviousSuspendCount);
}

BOOLEAN WINAPI ResumeProcess(
    HANDLE ProcessHandle
)
{
    if (!ProcessHandle)
    {
        return FALSE;
    }

    return (NT_SUCCESS(NtResumeProcess(ProcessHandle)))
        ? TRUE : FALSE;
}

BOOLEAN WINAPI ResumeThreadWrapper(
    HANDLE ThreadHandle
)
{
    if (!ThreadHandle)
    {
        return FALSE;
    }

    return (NT_SUCCESS(NtResumeThread(
        ThreadHandle,
        NULL))) ? TRUE : FALSE;
}

```

We can check to ensure that the following is true with some simple static reverse engineering.

1. SuspendThread (KERNEL32) -> NtSuspendThread (NTDLL)
2. ResumeThread (KERNEL32) -> NtResumeThread (NTDLL)

The first thing we are going to do is retrieve the address of SuspendThread (KERNEL32) and we'll set a break-point. I'm using Visual Studio 2017 and I'll be using the Visual Studio debugger for this task.

```

145 int main(void)
146 {
147     FARPROC SuspendThreadAddress = GetProcAddress(
148         GetModuleHandle("KERNEL32"),
149         "SuspendThread");

```

Snippet for obtaining the

address to SuspendThread (KERNEL32).

When we debug and the break-point is hit, we can step into/over and then check the value stored under the *SuspendThreadAddress* variable.

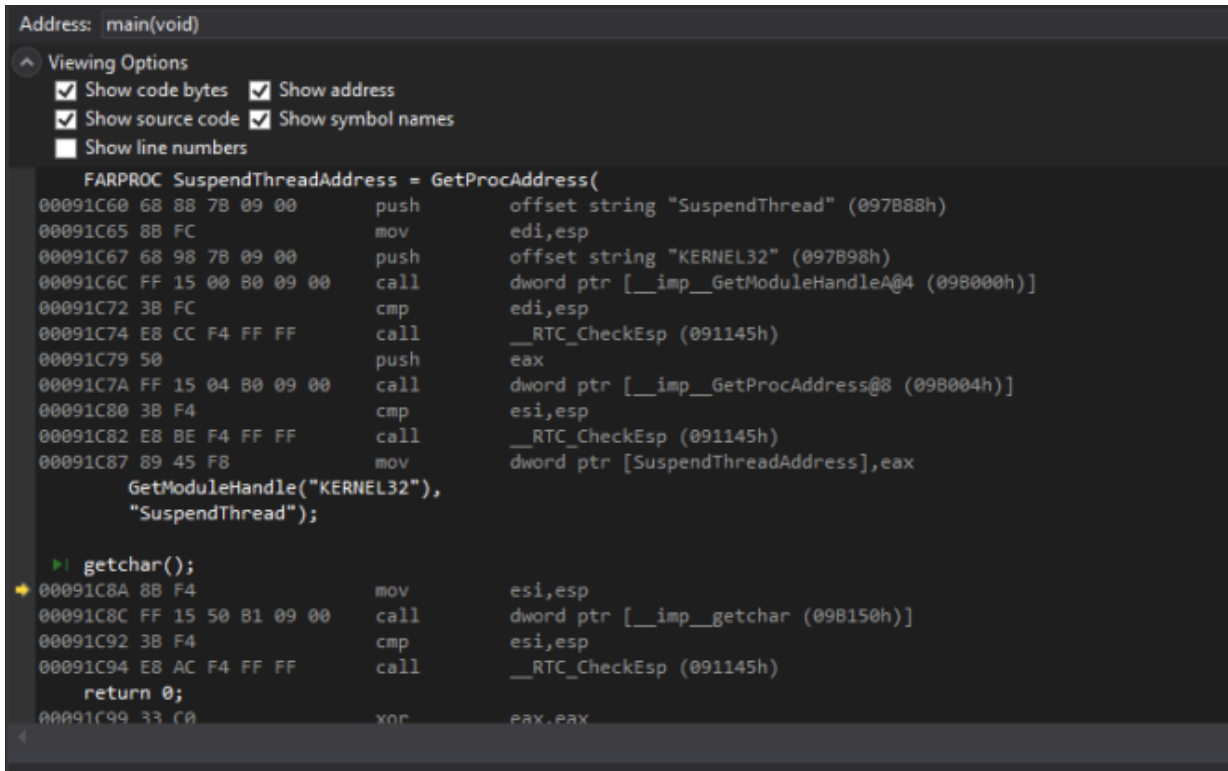
```

146
147 >| FARPROC SuspendThreadAddress = GetProcAddress(
148     GetModuleHandle(SuspendThreadAddress kernel32.dll!0x74536670 (load symbols for additional information)
149     "SuspendThread");

```

Debugging the snippet from above and checking the value held under SuspendThreadAddress after stepping into w/ the debugger.

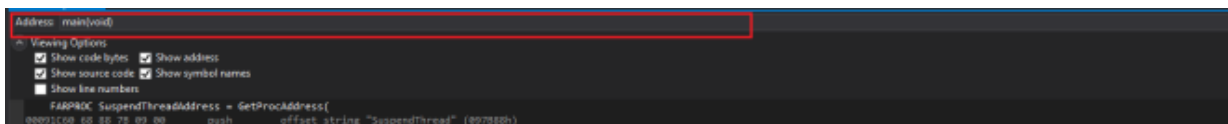
If you go to the top menu in Visual Studio and select **Debug -> Windows -> Disassembly** (Ctrl + Alt + D short-key by default) then you can bring up the Disassembly view. This option won't be possible unless you're currently debugging the program.



An

example of what Visual Studio's Disassembly view looks like.

If you notice at the top of the Disassembly view we have a text label saying "Address" followed by a text-box control. We can put in the address we got back for SuspendThread (KERNEL32) and it will take us to the disassembly at that location; KERNEL32.DLL is loaded under the address space of our currently-debugged process and the SuspendThread address we got given back from GetProcAddress is present under memory for the loaded KERNEL32 module in our process – we can view the disassembly using the Visual Studio debugger.



Highlight of the Address bar on the Disassembly view.

```

Address: 74536670
Viewing Options
  [x] Show code bytes  [x] Show address
  [x] Show source code [x] Show symbol names
  [ ] Show line numbers
74536666 CC          int     3
74536667 CC          int     3
74536668 CC          int     3
74536669 CC          int     3
7453666A CC          int     3
7453666B CC          int     3
7453666C CC          int     3
7453666D CC          int     3
7453666E CC          int     3
7453666F CC          int     3
74536670 8B FF      mov     edi,edi
74536672 55          push   ebp
74536673 8B EC      mov     ebp,esp
74536675 5D          pop     ebp
74536676 FF 25 E0 13 59 74 jmp     dword ptr ds:[745913E0h]
7453667C CC          int     3
7453667D CC          int     3
7453667E CC          int     3
7453667F CC          int     3
74536680 CC          int     3
74536681 CC          int     3
74536682 CC          int     3

```

Disassembly for SuspendThread (KERNEL32).

Now we're viewing the disassembly for the SuspendThread function prologue. Hey! What is going on here? Why are we simply redirecting execution flow to another address via a JMP instruction?

The truth is that the real stub for SuspendThread is now located under another module, named KernelBase.dll (KERNELBASE) since Windows 8. On previous versions of Windows, such as Windows Vista and 7, it would have been under KERNEL32 as it is known to be... But many changes were made in Windows 8 and those changes still haunt us to this day on the latest version of Windows 10. On 32-bit versions of Windows, a 32-bit compiled copy of kernelbase.dll can be found under **SystemDrive:\WINDOWS\System32**, and on 64-bit versions of Windows, a 32-bit compiled copy can be found under **SystemDrive:\WINDOWS\SysWOW64** and a 64-bit compiled copy can be found under **SystemDrive:\WINDOWS\System32**.

kernel.appcore.dll	29/09/2017 14:41	Application extens...	143 KB	
kerberos.dll	29/09/2017 14:41	Application extens...	926 KB	
kernel.appcore.dll	29/09/2017 14:41	Application extens...	54 KB	
kernel32.dll	29/09/2017 14:42	Application extens...	687 KB	
KernelBase.dll	29/09/2017 14:41	Application extens...	2,456 KB	
kernel.dll	29/09/2017 14:41	Application extens...	98 KB	

System32 folder

show-casing the KernelBase.dll is present.

We're going to take a quick peak at KernelBase.dll in Interactive Disassembler (IDA) for static disassembly. We'll start by making sure there's an export available for the SuspendThread routine.

Name	Address	Ordinal	IDA
SuspendThread	0000000180079EA0	1609	

Exports tab showing that SuspendThread is an export of KernelBase.dll.

Well would you look at that! There's an export for SuspendThread under KernelBase.dll. We'll take a look at it now.

```

.text:00000000180079E0 ; ----- S U B R O U T I N E -----
.text:00000000180079E0
.text:00000000180079E0
.text:00000000180079E0 ; DWORD __stdcall SuspendThread(HANDLE hThread)
.text:00000000180079E0 public SuspendThread
.text:00000000180079E0 SuspendThread proc near ; DATA XREF: .rdata:0000000018017ACA4↓o
.text:00000000180079E0 ; .rdata:off_18021C338↓o ...
.text:00000000180079E0 arg_8 = dword ptr 10h
.text:00000000180079E0
.text:00000000180079E0 sub rsp, 28h
.text:00000000180079E04 lea rdx, [rsp+28h+arg_8]
.text:00000000180079E09 call cs:_imp_NtSuspendThread
.text:00000000180079E0F test eax, eax
.text:00000000180079E0B1 js short loc_180079EBC
.text:00000000180079E0B3 mov eax, [rsp+28h+arg_8]
.text:00000000180079E0B7
.text:00000000180079E0B7 loc_180079EBC: ; CODE XREF: SuspendThread+26↓j
.text:00000000180079E0B7 add rsp, 28h
.text:00000000180079E0B8 retn
.text:00000000180079E0B8 ;

```

Disassembly for SuspendThread (KERNELBASE).

Looking at the above function prologue, you may already have noticed the pink styled text which contains the key-word “NtSuspendThread”. Yes, SuspendThread (KERNELBASE) does call NtSuspendThread (NTDLL); after all, **NtSuspendThread (NTDLL) performs the system call** so the functionality (in-which resides in instructions under kernel-mode memory) can really be invoked. The SuspendThread routine will do some other things aside from calling NtSuspendThread, and we’ll note these down now for educational purposes.

```

1 DWORD __stdcall SuspendThread(HANDLE hThread)
2 {
3     DWORD result; // eax@2
4     DWORD v2; // [sp+38h] [bp+10h]@1
5
6     if ( NtSuspendThread(hThread, &v2) < 0 )
7     {
8         BaseSetLastNTErr();
9         result = -1;
10    }
11    else
12    {
13        result = v2;
14    }
15    return result;
16 }

```

Pseudo-code of SuspendThread

(KERNELBASE).

1. Sets up two local variables.
 - o *result* is returned at the end of the routine as the return value for the SuspendThread routine. The caller gets this value returned back to them to determine if the operation was/wasn’t successful.
 - o *v2* is returned at the end of the routine and it will represent the ULONG return for the second parameter of the NtSuspendThread call. If we remember back to NtSuspendThread, there was the PreviousSuspendCount parameter which we were passing as NULL; SuspendThread (KERNELBASE) will have this value returned to the caller if the operation is successful.
2. Invoke NtSuspendThread (NTDLL) and set the value of the <v2> local variable as the PreviousSuspendCount target (return that data to the <v2> variable).
3. If the NtSuspendThread (NTDLL) call does not return STATUS_SUCCESS (which is the same as 0 – 0x00000000) then invoke BaseSetLastNTErr (Win32 API) and set the value of *result* to -1 (which will indicate failure to the caller).

4. If the NtSuspendThread (NTDLL) call does return STATUS_SUCCESS (successful), then set the value of *result* to be returned to the caller as the value held under the *v2* variable.
5. Return back the value of *result*

NOTE: *result* and *v2* aren't the real variable names in the KernelBase.dll source-code. These names are generated automatically by IDA.

It's more or less pretty straight forward and follows this routine for ResumeThread (KERNELBASE) also but we'll look at this as well anyway.

```
1 DWORD __stdcall ResumeThread(HANDLE hThread)
2 {
3     DWORD result; // eax@2
4     DWORD v2; // [sp+38h] [bp+10h]@1
5
6     if ( NtResumeThread(hThread, &v2) < 0 )
7     {
8         BaseSetLastNTErrror();
9         result = -1;
10    }
11    else
12    {
13        result = v2;
14    }
15    return result;
16 }
```

We can see that the same process is being repeated except NtResumeThread (NTDLL) is being invoked instead of NtSuspendThread (NTDLL).

For the record, the BaseSetLastNTErrror routine will invoke RtlSetLastWin32Error. The end-result is the correct error code being set as the "last error" so the caller can invoke GetLastError if the operation fails and acquire additional information regarding what went wrong; this also means that the NTSTATUS error code returned by NtSuspendThread/NtResumeThread is converted to a DOS error code via RtlNtStatusToDosError.

We are going to end this section of the article here and progress on to discussing kernel-mode. The next section will show examples of how to suspend/resume a process from kernel-mode, along with explanations about how NtSuspendProcess/NtSuspendThread/NtResumeProcess/NtResumeThread actually work in the kernel.

Section 2 – Kernel-Mode

In kernel-mode, things a lot different than in user-mode. For starters, you don't have access to the Win32 API in kernel-mode; you can communicate to a user-mode process via Inter-Process Communication (IPC) or perform kernel-mode code injection targeting a user-mode process to get Win32 API calls invoked but this is not the same as directly using such from kernel-mode, which you cannot do.

When working in kernel-mode, you have access to the Native API (NTAPI). The Native API includes routines with an Nt* prefix which are exported by NTDLL (and those routines exported by NTDLL will perform a system call – whereas in kernel-mode you don't need to perform a system call), however you won't have access to all of them by default – there is also the kernel-mode only routines which don't follow the Nt* style.

When a user-mode process directly or indirectly invokes an NTAPI routine and a system call is performed for user-mode to kernel-mode transition, a kernel-mode routine such as KiSystemCall32/KiSystemCall64 (on the latest versions of Windows 10 after the recent patch updates regarding Meltdown, there is now KiSystemCall32Shadow/KiSystemCall64Shadow/KiSystemCall32AmdShadow/KiSystemCall64Shadow) will be invoked. These routines will call other routines, and a kernel-mode routine named KiSystemServiceRepeat will eventually be invoked. The KiSystemServiceRepeat routine will access the System Service Descriptor Table (KeServiceDescriptorTable) which is nothing more than a dispatch table (another saying of an “array”) which contains pointer addresses – each pointer address represents the address of a Native API routine within the address space of NTOSKRNL (the Windows Kernel). There are routines which can be invoked from user-mode via a system call and there are routines which are “kernel-mode only” (and thus cannot be invoked from user-mode via a system call). The routines which can be invoked via a system call have an entry in the KeServiceDescriptorTable, and there's also a “Shadow” version which would be KeServiceDescriptorTableShadow to also allow access to pointer addresses of win32k.sys routines.

This article isn't about going through the process of how a system call works and how the kernel handles them, it's about suspension of processes and how this mechanism works. However, this is all related to the topic because both NtSuspendProcess and NtSuspendThread are not exported by NTOSKRNL. **What does this mean?** It means that by default, you do not have access to either of the routines in kernel-mode when developing a kernel-mode device driver. This can be irritating if you have a genuine reason to suspend a process in kernel-mode, and you may not wish to communicate back down to a user-mode component to invoke NtSuspendProcess (NTDLL)/NtSuspendThread (NTDLL) for you.

There are two options in this scenario if you don't wish to work with a user-mode component (which would be the most stable option available at this moment in time – and I suggest if you do have a user-mode component such as a Windows Service, you should make use of such properly).

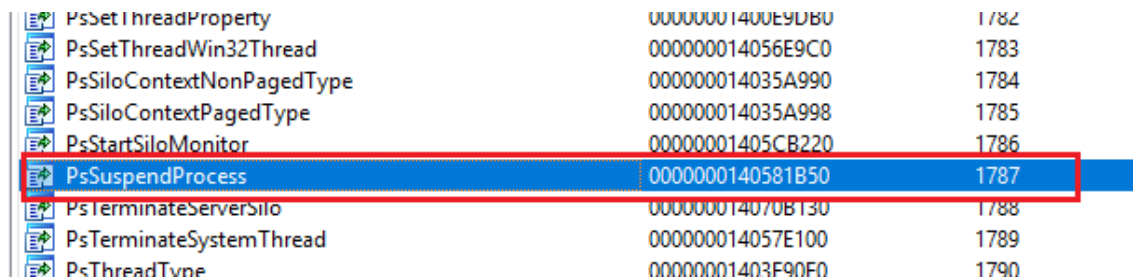
1. Access the KeServiceDescriptorTable manually and use this as an entry-point to locating the address of NtSuspendProcess/NtSuspendThread.
2. Find out if there's an exported kernel-mode routine which can be used for process/thread suspension.

The first method is very straight forward on 32-bit systems because KeServiceDescriptorTable is actually exported by the Windows Kernel for 32-bit systems. This means you can find the address to KeServiceDescriptorTable effortlessly. Sadly, this isn't the case for 64-bit systems; Microsoft implemented a feature called PatchGuard for 64-bit versions of Windows only at the start of Windows Vista and PatchGuard contains a whole wide-range of functionality however one thing they did when they introduced PatchGuard was not export the KeServiceDescriptorTable. Please do not

be confused, accessing the System Service Descriptor Table on a 64-bit system will not cause a BSOD as long as the calculations are correct, however it's more hassle to make use of it due to the fact that it is no longer exported for 64-bit systems.

If you want to go down the route of accessing the System Service Descriptor Table on a 64-bit system, it isn't all that complicated. You can find the address of KiSystemCall64/KiSystemCall64Shadow/KiSystemCall64AmdShadow via the IA32_LSTAR Model Specific Register (MSR) and then you can locate the non-exported kernel-mode routine KiSystemServiceRepeat not far off from the address pointed to by IA32_LSTAR MSR. As we've already established earlier on, the address of the KeServiceDescriptorTable is exposed in the KiSystemServiceRepeat routine. People have been using this technique to locate the SSDT on 64-bit systems for countless years now, but you should avoid doing this unless you really need to, because Microsoft can change something at any time and prevent your source code from working on the updated systems.

Thankfully, the idea noted in our second option is a valid option for us when it comes to process suspension from kernel-mode. As it turns out, there's a routine exported by NTOSKRNL which is called by NtSuspendProcess. The sad part is that there is no exported routine for singular thread suspension, but I suspect most people trying to suspend from kernel-mode will be targeting a whole process and not after suspending X amount of threads under a process only. The exported routine is called PsSuspendProcess, and it isn't officially documented by Microsoft.



PsSetThreadProperty	00000001400E9DB0	1782
PsSetThreadWin32Thread	000000014056E9C0	1783
PsSiloContextNonPagedType	000000014035A990	1784
PsSiloContextPagedType	000000014035A998	1785
PsStartSiloMonitor	00000001405CB220	1786
PsSuspendProcess	0000000140581B50	1787
PsTerminateServerSilo	000000014070B130	1788
PsTerminateSystemThread	000000014057E100	1789
PsThreadTune	00000001403F90F0	1790

Exports of NTOSKRNL – highlighting PsSuspendProcess.

We're going to take a look at how PsSuspendProcess works but we're going to take a look at NtSuspendProcess beforehand. As previously noted, NtSuspendProcess is not exported by NTOSKRNL, however it does have an entry under the System Service Descriptor Table, which will have the pointer address to its routine within the address space of NTOSKRNL – if this wasn't the case then the Windows Kernel wouldn't know the location in-memory of the routine when a system call was performed for it to be invoked.


```

PAGE:0000000140581AD0 ; ----- S U B R O U T I N E -----
PAGE:0000000140581AD0
PAGE:0000000140581AD0
PAGE:0000000140581AD0 ; __int64 __fastcall NtSuspendProcess(HANDLE ProcessHandle)
PAGE:0000000140581AD0 NtSuspendProcess proc near
PAGE:0000000140581AD0
PAGE:0000000140581AD0 var_28 = dword ptr -28h
PAGE:0000000140581AD0 Process = qword ptr 10h
PAGE:0000000140581AD0
PAGE:0000000140581AD0 nov r11, rsp
PAGE:0000000140581AD3 push rbx ; __int64
PAGE:0000000140581AD4 sub rsp, 40h
PAGE:0000000140581AD8 mov rax, gs:188h
PAGE:0000000140581AE1 mov edx, 800h
PAGE:0000000140581AE6 and qword ptr [r11-10h], 0
PAGE:0000000140581AEB and qword ptr [r11-18h], 0
PAGE:0000000140581AF0 mov r8, cs:PsProcessType
PAGE:0000000140581AF7 mov r9b, [rax+232h]
PAGE:0000000140581AFE lea rax, [r11+10h]
PAGE:0000000140581B02 mov [r11-20h], rax
PAGE:0000000140581B06 mov [rsp+48h+var_28], 75537350h ; int
PAGE:0000000140581B0E call ObpReferenceObjectByHandleWithTag
PAGE:0000000140581B13 mov ebx, eax
PAGE:0000000140581B15 test eax, eax
PAGE:0000000140581B17 js short loc_140581B34
PAGE:0000000140581B19 mov rcx, [rsp+48h+Process] ; Process
PAGE:0000000140581B1E call PsSuspendProcess
PAGE:0000000140581B23 mov rcx, [rsp+48h+Process] ; BugCheckParameter2
PAGE:0000000140581B28 mov edx, 75537350h
PAGE:0000000140581B2D mov ebx, eax
PAGE:0000000140581B2F call ObpDereferenceObjectWithTag
PAGE:0000000140581B34 loc_140581B34: ; CODE XREF: NtSuspendProcess+471j
PAGE:0000000140581B34 mov eax, ebx
PAGE:0000000140581B36 add rsp, 40h
PAGE:0000000140581B3A pop rbx
PAGE:0000000140581B3B retn
PAGE:0000000140581B3B NtSuspendProcess endp

```

Disassembly for NtSuspendProcess (NTOSKRNL).

In the NtSuspendProcess routine, there are two important things which will happen.

1. The handle is used to retrieve an object. The handle is passed in to the routine as the parameter which we learnt about during the user-mode section, however the routine doesn't actually send the handle anywhere itself. Instead, it retrieves an object to the process using the handle via an undocumented, non-exported kernel-mode only routine, named ObpReferenceObjectByHandleWithTag. ObpReferenceObjectByHandleWithTag works by accessing an undocumented, non-exported table stored in the kernel; the routine will use other routines such as ExpLookupHandleTableEntry. An object for a process in kernel-mode would be a pointer structure to the EPROCESS (PEPROCESS) kernel-mode structure for that process – the EPROCESS structure contains many fields which stores data about that process. There's also the KPROCESS structure which is the first field of the EPROCESS structure and contains a lot of data about the process in question.
2. Invoke PsSuspendProcess. NtSuspendProcess doesn't really do anything in itself, it just forwards execution control to PsSuspendProcess. The only reason it calls ObpReferenceObjectByHandleWithTag in advance to retrieve an object to the process (PEPROCESS -> pointer structure of EPROCESS to be precise) is because PsSuspendProcess does not accept a handle as the parameter, but it accepts an object instead.

If you go snooping around NtSuspendProcess with static disassembly and try to generate pseudo-code for it, you'll likely get back a messy view which will appear ugly at first. I cleaned up the data-types and variable names a bit so it is a bit more pleasant and understandable when looking at it.

```

1 NTSTATUS __fastcall NtSuspendProcess(HANDLE ProcessHandle)
2 {
3     NTSTATUS NtStatus; // ebx@1
4     _EPROCESS *Process; // [sp+58h] [bp+10h]@1
5
6     NtStatus = ObpReferenceObjectByHandleWithTag(
7         (ULONG_PTR)ProcessHandle,
8         2048,
9         PsProcessType,
10        *(_BYTE *)(*MK_FP(__GS__, 392i64) + 0x232i64),
11        'uSpP',
12        (__int64)&Process,
13        0i64,
14        0i64);
15    if ( NtStatus >= 0 ) // IF NTSTATUS is STATUS_SUCCESS
16    {
17        NtStatus = PsSuspendProcess(Process); // Call PsSuspendProcess (NTOSKRNL)
18        ObfDereferenceObjectWithTag((ULONG_PTR)Process); // Clean-up the referenced object
19    }
20    return NtStatus;
21 }

```

Pseudo-code for NtSuspendProcess (NTOSKRNL).

The following is done in-order.

1. Two local variables are setup. One has a data-type of NTSTATUS and the other has a data-type of PEPROCESS (pointer structure variant of EPROCESS). Of course the variable names in the screen-shot are not the same ones used in the Windows source code, I re-named them. Earlier in previous screen-shots, variable names like “v2” are neither really from the source code. IDA just sets it to these by default and you can change them while reversing.
2. The local variable which has a data-type of NTSTATUS is assigned a value of the return value from the ObpReferenceObjectByHandleWithTag call. This routine is now invoked; ObpReferenceObjectByHandleWithTag returns an NTSTATUS error code.
3. The invocation of ObpReferenceObjectByHandleWithTag will assign a value to the local variable which has a data-type of PEPROCESS (named Process as the variable name by myself). If you look at the 6th parameter for the ObpReferenceObjectByHandleWithTag call, we’re passing a pointer to our PEPROCESS variable – the routine will use this to set the value of our variable in the called routine.
4. If the NtStatus value represents success (NT_SUCCESS -> >= 0) then the routine progresses to call PsSuspendProcess and it will perform a clean-up operation with the object to the process which was previously retrieved via ObfDereferenceObjectWithTag.
5. The NtStatus value is returned by NtSuspendProcess back to the caller.

I think it’s about time we take a look at the famous PsSuspendProcess routine, don’t you?

```

1 NTSTATUS __fastcall PsSuspendProcess(PEPROCESS Process)
2 {
3     PVOID CriticalRegion; // rdi@1
4     PEPROCESS Process2; // rbp@1
5     char *RundownProtectValue; // r14@1
6     PETHREAD CurrentThread; // rax@2
7     NTSTATUS NtStatus; // ebx@2
8     PETHREAD CurrentThread2; // rsi@4
9
10    CriticalRegion = (PVOID)*HK_FP(__GS__, 0x188i64);
11    Process2 = Process;
12    --*(_WORD *)CriticalRegion + 0xF2);
13    RundownProtectValue = (char *)&Process->RundownProtect;
14    if ( (unsigned __int8)ExAcquireRundownProtection_0(&Process->RundownProtect) == 1 )
15    {
16        LODWORD(CurrentThread) = PsGetNextProcessThread(Process2, NULL); // enumerate threads of the process
17        NtStatus = 0;
18        while ( 1 )
19        {
20            CurrentThread2 = CurrentThread;
21            if ( !CurrentThread )
22                break;
23            PsSuspendThread((__int64)CurrentThread, NULL);
24            LODWORD(CurrentThread) = PsGetNextProcessThread(Process2, CurrentThread2);
25        }
26        ExReleaseRundownProtection_0(RundownProtectValue);
27    }
28    else
29    {
30        NtStatus = 0xC000010A; // STATUS_PROCESS_IS_TERMINATING
31    }
32    KeLeaveCriticalRegionThread(CriticalRegion);
33    return NtStatus;
34 }

```

Pseudo-code for PsSuspendProcess (NTOSKRNL).

Here's a break-down of how the routine works, I'll stick to the core parts.

1. The threads of the targeted process are enumerated via PsGetNextProcessThread and a while loop. The way it works is PsGetNextProcessThread will be called to return a PETHREAD (pointer to the ETHREAD structure) object for the first thread found within the targeted process and then an operation using the returned PETHREAD will be performed, followed by another PsGetNextProcessThread call and a re-start of the loop. PsGetNextProcessThread is a non-exported kernel-mode only routine. When there are no more threads to be found, the returned PETHREAD will be nothing (NULL) and this will be caught by the conditional statement which checks if the variable which is supposed to be returned the next PETHREAD (of the next thread to be found), and at this point the break instruction is used to exit the while loop since the operation will have no more business regarding thread enumeration.
2. For each enumerated thread in which a PETHREAD object can be acquired for, the undocumented and non-exported kernel-mode only routine named PsSuspendThread will be called, passing the PETHREAD as a parameter.
3. When an occurrence of a NULL PETHREAD returned value is returned, the operation exits because the loop is exited. Exiting the while loop is followed by the return of the NTSTATUS error code (which could represent success or failure). The return status for PsSuspendProcess will always be STATUS_SUCCESS unless the targeted process is in a state preparing for termination.

We should see what routine PsSuspendThread will call.

```

1  __int64 __fastcall PsSuspendThread(__int64 a1, int *a2)
2  {
3      int *v2; // rdi@1
4      __int64 v3; // rbx@1
5      __int64 v4; // rsi@1
6      __int64 v5; // r14@1
7      signed int v6; // ebx@3
8      int v8; // [sp+20h] [bp-28h]@1
9
10     v2 = a2;
11     v3 = a1;
12     v8 = 0;
13     v4 = *MK_FP(__GS__, 392i64);
14     --*( _WORD * )(v4 + 484);
15     v5 = a1 + 1720;
16     if ( (unsigned __int8)ExAcquireRunDownProtection_0(a1 + 1720) )
17     {
18         if ( *( _DWORD * )(v3 + 1744) & 1 )
19         {
20             v6 = -1073741749;
21         }
22         else
23         {
24             v8 = KeSuspendThread(v3);
25             v6 = 0;
26         }
27         ExReleaseRunDownProtection_0(v5);
28     }
29     else
30     {
31         v6 = -1073741749;
32     }
33     KeLeaveCriticalRegionThread(v4);
34     if ( v2 )
35         *v2 = v8;
36     return (unsigned int)v6;
37 }

```

Pseudo-code for PsSuspendThread (NTOSKRNL).

PsSuspendThread will call KeSuspendThread, and KeSuspendThread returns a status which is set to the value of the second parameter passed in PsSuspendThread as the second parameter. However, PsSuspendProcess doesn't care about the second parameter and thus it doesn't check the return status by KeSuspendThread. PsSuspendProcess will only return STATUS_SUCCESS or STATUS_PROCESS_IS_TERMINATING.

The fuss regarding the acquisition of "run-down protection" is regarding to preventing the thread from being "terminated" during the operation. Such would cause system instability and likely would bug-check the system because the kernel would then be operating with an object which would no longer be deemed valid.

```

1  __int64 __fastcall KeSuspendThread(__int64 a1)
2  {
3      __int64 v1; // rbx@1
4      unsigned __int8 v2; // si@1
5      signed __int64 v3; // r14@1
6      volatile signed __int32 *v4; // rdi@1
7      __int64 v5; // r8@1
8      unsigned int v6; // ebp@1
9
10     v1 = a1;
11     v2 = __readcr8();
12     __writecr8(2ui64);
13     v3 = *MK_FP(__GS__, 32i64);
14     v4 = (volatile signed __int32 *)(a1 + 736);
15     KiAcquireKobjectLockSafe(a1 + 736);
16     v6 = *(_BYTE *)(v1 + 644);
17     if ( v6 == 127 )
18     {
19         _InterlockedAnd(v4, 0xFFFFFFFF);
20         __writecr8(v2);
21         RtlRaiseStatus(0xC000004Ai64);
22     }
23     ++*( _BYTE *)(v1 + 644);
24     if ( !KiSuspendThread(v1, v3, v5) )
25         --*( _BYTE *)(v1 + 644);
26     _InterlockedAnd(v4, 0xFFFFFFFF);
27     KiExitDispatcher(v3);
28     return v6;
29 }

```

Pseudo-code for KeSuspendThread.

Now here is a more interesting part, but it should get a bit more interesting when we move to KiSuspendThread.

KeSuspendThread is invoking a routine called RtlRaiseStatus, but this is only happening depending on a conditional statement. The conditional statement is put in place to determine whether a suspension of a thread is being taken place in-which the maximum suspension count has already been met. If we remember back to NtSuspendThread invocation, we had a parameter regarding the previous suspension count which could be returned to us – SuspendThread (KERNEL32/KERNELBASE) was making use of it happily and returning it as the return value of the routine. Well, it turns out that the maximum suspension count is 127. If the conditional statement is met, a status error code is raised with RtlRaiseStatus. The error code being raised which is displayed as 0xC000004Ai64 actually translates to **0xC000004A** which is the same as **STATUS_SUSPEND_COUNT_EXCEEDED**.

The KiSuspendThread routine is called towards the end of the KeSuspendThread routine. As expected, KiSuspendThread is another undocumented and non-exported kernel-mode only routine. KiSuspendThread is actually a bit more interesting though, because it exposes how the whole suspension mechanism in Windows actually works – you may be very surprised at how minimal it truly is, and it relies on a documented programming technique which most developers will have used both in kernel-mode and user-mode at-least once in their development time.

```

1 char __fastcall KiSuspendThread(__int64 a1, signed __int64 a2, __int64 a3)
2 {
3     signed __int64 v3; // r15@1
4     char v4; // si@1
5     int v10; // eax@3
6     char v11; // r10@5
7     char result; // al@11
8     char v13; // al@13
9     int v14; // eax@21
10    __int64 v15; // rcx@21
11    signed __int64 v16; // rdi@21
12    signed __int64 v17; // r14@21
13    volatile signed __int32 *v18; // rbp@23
14    __int64 v19; // rcx@24
15    _QWORD *v20; // rax@25
16    char v21; // al@35
17    unsigned __int64 v22; // rbp@37
18    unsigned __int64 v23; // rcx@38
19    int v24; // [sp+50h] [bp+8h]@1
20    int v25; // [sp+60h] [bp+18h]@33
21
22    v3 = a2;
23    v4 = 0;
24    v24 = 0;
25    _RBX = a1;
26    while ( 1 )
27    {
28        __asm { lock bts qword ptr [rbx+40h], 0 }
29        if ( !_CF )
30            break;
31        do
32            KeYieldProcessorEx(&v24);
33        while ( *(_QWORD *)(_RBX + 64) );
34    }
35    v10 = *(_DWORD *)(_RBX + 116);
36    if ( _bittest(&v10, 0xEu) )
37    {
38        v4 = 1;
39        if ( *(_DWORD *)(_RBX + 740) )
40        {
41            *(_DWORD *)(_RBX + 740) = 0;
42            v11 = 0;
43            if ( !*( _BYTE *)(_RBX + 730) )
44            {
45                *(_BYTE *)(_RBX + 730) = 1;
46                KiInsertQueueApc(_RBX + 648);
47            }
48        }
49    }
50
51    0000DB17 KiSuspendThread:24

```

Pseudo-code for KiSuspendThread (NTOSKRNL) 1/2.

```

28 __asm { lock bts qword ptr [rbx+40h], 0 }
29 if ( ! CF )
30 break;
31 do
32 KeYieldProcessorEx(&u24);
33 while ( *(_QWORD *)(_RBX + 64) );
34 }
35 u10 = *(_DWORD *)(_RBX + 116);
36 if ( !_bittest(&u10, 0xEu) )
37 {
38 u4 = 1;
39 if ( *(_DWORD *)(_RBX + 740) )
40 {
41 *(_DWORD *)(_RBX + 740) = 0;
42 u11 = 0;
43 if ( !*( _BYTE *)(_RBX + 730) )
44 {
45 *( _BYTE *)(_RBX + 730) = 1;
46 KiInsertQueueApc(_RBX + 648);
47 }
48 if ( KiDisableLightWeightSuspend
49 || *( _BYTE *)(_RBX + 388) != 5
50 || *( _BYTE *)(_RBX + 112) & 7) != 1
51 || (u13 = *( _BYTE *)(_RBX + 3), u13 & 0x40)
52 || u13 < 0
53 || *(_DWORD *)(_RBX + 484)
54 || *( _BYTE *)(_RBX + 390)
55 || *( _BYTE *)(_RBX + 192)
56 || *( _BYTE *)(_RBX + 586)
57 || *( _BYTE *)(*(_QWORD *)(_RBX + 208) + 17i64) != 5 && *( _BYTE *)(*(_QWORD *)(_RBX + 208) + 16i64) != 1 )
58 {
59 if ( u11 )
60 {
61 lobyte(a3) = 2;
62 KiSignalThreadForApc(u3, _RBX + 648, a3);
63 }
64 }
65 else
66 {
67 u14 = (*(_DWORD *)(_RBX + 116) ^ *( _BYTE *)(_RBX + 391) << 18) & 0x40000;
68 *( _BYTE *)(_RBX + 112) = 3;
69 *(_DWORD *)(_RBX + 116) ^= u14;
70 *( _BYTE *)(_RBX + 193) = 1;
71 *(_QWORD *)(_RBX + 64) = 0i64;
72 u15 = *(_QWORD *)(_RBX + 208);
73 u16 = u15 + 17;
74 u17 = u15 + 48i64 * *( _BYTE *)(_RBX + 587);

```

00000000 KiSuspendThread:51

Pseudo-code for KiSuspendThread (NTOSKRNL) 2/2.

Well, would you look at that!

We have a call to `KiInsertQueueApc` which is a step involved in dispatching an Asynchronous Procedure Call (APC). Asynchronous Procedure Calls are used for communication all the time – in fact they have also been abused for thread hijacking as an entry-point for code injection for numerous of years now, and there’s a technique named Atom Bombing which relies on APC injection as well – and the way it works is you target a thread for the APC and you can force the targeted thread to execute the callback routine for the APC event. This in turn, will force the targeted thread to execute the code which you wish it to execute. If we wanted to inject code into a user-mode process from kernel-mode, we could rely on `KeInitializeApc` and `KeInsertQueueApc` – of course this routine is using the non-exported variants though.

The `KiInsertQueueApc` does exactly what the routine name implies. It inserts an APC event into a queue. It pretty much sets up the `PKAPC` structure which is used for the APC dispatch operation – the `PKAPC` structure (pointer to `KAPC` structure) holds data such as the environment for the APC event (e.g. `KernelMode` or `UserMode`), the targeted thread, among other data.

The `KiSignalThreadForApc` also does what the routine name implies. It will signal the thread for the APC event as far as I am aware, and this relies on `KiSignalThread` (another non-exported, undocumented kernel-mode routine). However, a flag named `KiDisableLightWeightSuspend` must be

set to TRUE for the KiSignalThreadForApc operation to occur, among several other conditional statements.

The KiSuspendThread routine will use the ETHREAD structure (PETHREAD because it's a pointer to the ETHREAD structure) for the current thread being put into a "suspended" state.

Under the KTHREAD structure there is data regarding thread suspension, for example, a SuspendCount field. The kernel will update such data regarding thread suspension and then the thread will be held up waiting for the data to be updated again via KeWaitForSingleObject. An Asynchronous Procedure Call is performed so the KeWaitForSingleObject call can be made on the targeted thread and this wait ends when the data is updated again to remove the suspend state.

If you were expecting some sort of super-human mechanism for "thread suspension" then I am sorry for letting you down. The suspension mechanism evolves around waiting for the semaphore data to be updated to indicate the thread should be resumed – the wait is executed in the address space of the targeted process to hold the targeted thread via the APC which was dispatched. It is possible that the semaphore part is not identically correct for the latest versions of Windows, because changes may have been made and I initially remember hearing about this many years ago – and have been unable to completely verify that this operation works like this at-least still – but it makes complete sense and I doubt it is inaccurate.

We have all this talk about thread suspension, but we've forgotten all about thread resuming. It isn't fair for us to give all the attention to PsSuspendProcess, KeSuspendThread and KiSuspendThread... We need to give some love to PsResumeProcess! That's right, there's an exported kernel-mode routine named PsResumeProcess!



Exports for NTOSKRNL, show-casing that PsResumeProcess is an export.

Well this is exciting... I'm going to take a leap with my confidence and estimate what the PsResumeProcess routine will do.

1. Enumerate through all the threads of the targeted process
2. Invoke a routine named KeResumeThread
3. Return an NTSTATUS error code (either **STATUS_SUCCESS** or **STATUS_PROCESS_IS_TERMINATING**).


```

1 NTSTATUS __fastcall PsResumeProcess(PEPROCESS Process)
2 {
3     PVOID CriticalRegion; // rsi@1
4     PEPROCESS Process2; // rbp@1
5     char *RundownProtectValue; // r14@1
6     PETHREAD Thread; // rax@2
7     NTSTATUS NtStatus; // ebx@2
8     PETHREAD Thread2; // rdi@3
9
10    CriticalRegion = (PVOID)*MK_FP(__GS__, 392i64);
11    Process2 = Process;
12    --*((_WORD *)CriticalRegion + 242);
13    RundownProtectValue = (char *)&Process->RundownProtect;
14    if ( (unsigned __int8)ExAcquireRundownProtection_0(&Process->RundownProtect) == 1 )
15    {
16        LODWORD(Thread) = PsGetNextProcessThread(Process2, NULL);
17        NtStatus = 0;
18        while ( 1 )
19        {
20            Thread2 = Thread;
21            if ( !Thread )
22                break;
23            KeResumeThread((__int64)Thread);
24            LODWORD(Thread) = PsGetNextProcessThread(Process2, Thread2);
25        }
26        ExReleaseRundownProtection_0(RundownProtectValue);
27    }
28    else
29    {
30        NtStatus = STATUS_PROCESS_IS_TERMINATING; // STATUS_PROCESS_TERMINATING
31    }
32    KeLeaveCriticalRegionThread(CriticalRegion);
33    return NtStatus;
34 }

```

Pseudo-code for PsResumeProcess (NTOSKRNL).

Looks like my estimation was right. Some may call me a cheater but that will just be the jealousy talking!

The PsResumeProcess routine will be called by NtResumeProcess, and we can verify these findings by checking the routine disassembly.

```

PAGE:000000014057B308 ; NTSTATUS __fastcall NtResumeProcess(HANDLE ProcessHandle)
PAGE:000000014057B308 NtResumeProcess proc near
PAGE:000000014057B308
PAGE:000000014057B308 var_28          = dword ptr -28h
PAGE:000000014057B308 Process        = qword ptr 10h
PAGE:000000014057B308
PAGE:000000014057B308          mov     r11, rsp
PAGE:000000014057B308          push  rbx          ; __int64
PAGE:000000014057B308          sub     rsp, 40h
PAGE:000000014057B310 ; 5:  NTSTATUS = ObpReferenceObjectByHandleWithTag(
PAGE:000000014057B310 ; 6:  (ULONG_PTR)ProcessHandle,          // Process Handle
PAGE:000000014057B310 ; 7:  2048,                             // Tag
PAGE:000000014057B310 ; 8:  PsProcessType,                   // Object
PAGE:000000014057B310 ; 9:  *(_BYTE *)(&HK_FP(__GS__, 392i64) + 0x232i64), // ObjectType
PAGE:000000014057B310 ;10:  'uSsP',                           // Access Mode
PAGE:000000014057B310 ;11:  (__int64)&Process,
PAGE:000000014057B310 ;12:  0i64,
PAGE:000000014057B310 ;13:  0i64);                             // Object Information
PAGE:000000014057B310          mov     rax, gs:188h
PAGE:000000014057B319          mov     edx, 800h
PAGE:000000014057B31E          and     qword ptr [r11-10h], 0
PAGE:000000014057B323          and     qword ptr [r11-18h], 0
PAGE:000000014057B328          mov     r8, cs:PsProcessType
PAGE:000000014057B32F          mov     r9b, [rax+232h]
PAGE:000000014057B336          lea    rax, [r11+10h]
PAGE:000000014057B33A          mov     [r11-20h], rax
PAGE:000000014057B33E          mov     [rsp+48h+var_28], 75537350h ; int
PAGE:000000014057B346          call   ObpReferenceObjectByHandleWithTag
PAGE:000000014057B34B          mov     ebx, eax
PAGE:000000014057B34D ;14:  if ( NTSTATUS >= 0 )
PAGE:000000014057B34D          test   eax, eax
PAGE:000000014057B34F          js     short loc_14057B36C
PAGE:000000014057B351 ;16:  NTSTATUS = PsResumeProcess(Process);
PAGE:000000014057B351          mov     rcx, [rsp+48h+Process] ; Process
PAGE:000000014057B356          call   PsResumeProcess
PAGE:000000014057B35B ;17:  ObpDereferenceObjectWithTag((ULONG_PTR)Process);
PAGE:000000014057B35B          mov     rcx, [rsp+48h+Process] ; BugCheckParameter2
PAGE:000000014057B360          mov     edx, 75537350h
PAGE:000000014057B365          mov     ebx, eax
PAGE:000000014057B367          call   ObpDereferenceObjectWithTag
PAGE:000000014057B36C ;19:  return NTSTATUS;
PAGE:000000014057B36C loc_14057B36C:          ; CODE XREF: NtResumeProcess+47↑j
PAGE:000000014057B36C          mov     eax, ebx
PAGE:000000014057B36E          add     rsp, 40h
PAGE:000000014057B372          pop     rbx
PAGE:000000014057B373          retn
PAGE:000000014057B373 NtResumeProcess endp
0050AB08 000000014057B308: NtResumeProcess

```

Disassembly of NtResumeProcess (NTOSKRNL).

I highlighted in red the call instruction being used for ObpReferenceObjectByHandleWithTag and PsResumeProcess. The only difference between NtSuspendProcess and NtResumeProcess is that the former will be calling PsSuspendProcess and the latter will be calling PsResumeProcess.

Since we're looking at those Nt* stubs, we may as well take a look at NtSuspendThread and NtResumeThread briefly to see if they will call.

```

1  __int64 __fastcall NtSuspendThread(ULONG_PTR BugCheckParameter1, unsigned __int64 a2)
2  {
3      _DWORD *v2; // rbx@1
4      ULONG_PTR v3; // r10@1
5      char v4; // r9@1
6      _DWORD *v5; // rcx@3
7      __int64 result; // rax@6
8      unsigned int v7; // edi@7
9      int v8; // [sp+70h] [bp+18h]@7
10     ULONG_PTR BugCheckParameter2; // [sp+78h] [bp+20h]@6
11
12     v2 = (_DWORD *)a2;
13     v3 = BugCheckParameter1;
14     v4 = *(_BYTE *)(*HK_FP(__GS__, 392i64) + 562i64);
15     if ( v4 && a2 )
16     {
17         v5 = (_DWORD *)4294901760;
18         if ( a2 < 0x7FFFFFFF0000i64 )
19             v5 = (_DWORD *)a2;
20         *v5 = *v5;
21     }
22     result = ObpReferenceObjectByHandleWithTag(
23         v3,
24         2,
25         PsThreadType,
26         v4,
27         1968403280,
28         (__int64)&BugCheckParameter2,
29         0i64,
30         0i64);
31     if ( (signed int)result >= 0 )
32     {
33         v7 = PsSuspendThread(BugCheckParameter2, &v8);
34         ObpDereferenceObjectWithTag(BugCheckParameter2);
35         if ( v2 )
36             *v2 = v8;
37         result = v7;
38     }
39     return result;
40 }

```

Pseudo-code for NtSuspendThread (NTOSKRNL).

NtSuspendThread will just lead down a path of PsSuspendThread, which is also called in PsSuspendProcess for each found thread during the enumeration operation. Nothing we've not seen done before.

What about NtResumeThread though? We really need to stop giving all the attention and love to thread suspension and treat thread resuming the same or it might turn rogue out of anger! 😊

```

1 | int64 __fastcall NtResumeThread(ULONG_PTR BugCheckParameter1, unsigned __int64 a2)
2 | {
3 |     _DWORD *v2; // rbx@1
4 |     ULONG_PTR v3; // r10@1
5 |     char v4; // r9@1
6 |     _DWORD *v5; // rcx@3
7 |     __int64 result; // rax@6
8 |     int v7; // [sp+70h] [bp+18h]@7
9 |     ULONG_PTR BugCheckParameter2; // [sp+78h] [bp+20h]@6
10 |
11 |     v2 = (_DWORD *)a2;
12 |     v3 = BugCheckParameter1;
13 |     v4 = *(_BYTE *)(&MK_FP(__GS__, 392i64) + 562i64);
14 |     if ( v4 && a2 )
15 |     {
16 |         v5 = (_DWORD *)4294901760;
17 |         if ( a2 < 0x7FFFFFFF0000i64 )
18 |             v5 = (_DWORD *)a2;
19 |         *v5 = *v5;
20 |     }
21 |     result = ObpReferenceObjectByHandleWithTag(
22 |         v3,
23 |         4096,
24 |         PsThreadType,
25 |         v4,
26 |         1968403280,
27 |         (__int64)&BugCheckParameter2,
28 |         0i64,
29 |         0i64);
30 |     if ( (signed int)result >= 0 )
31 |     {
32 |         PsResumeThread(BugCheckParameter2, &v7);
33 |         ObfDereferenceObjectWithTag(BugCheckParameter2);
34 |         if ( v2 )
35 |             *v2 = v7;
36 |         result = 0i64;
37 |     }
38 |     return result;
39 | }

```

Pseudo-code for NtResumeThread (NTOSKRNL).

NtResumeThread will simply call PsResumeThread, which is also called by PsResumeProcess during the thread enumeration operation. Nothing too interesting here either sadly because we've already seen it all.

We've done a lot of discussing and not a lot of programming so it's time we brought back some C code. This time, the difference is that the example C source-code will be for kernel-mode software and not for user-mode software; there are some pre-requisites before you can start developing kernel-mode software (e.g. kernel-mode device drivers), however I shouldn't have to lay these out because you shouldn't be attempting such a task without having some background in Windows kernel-mode software engineering in the first place.

For tutorials sake, I will note the following.

1. Download the latest version of [Visual Studio](#) (Visual Studio 2017 at the time of writing this) and install it.
2. Download [Windows 10 SDK](#) (latest version) and install it.
3. Download [Windows Driver Kit](#) (WDK – latest version) and install it.

You can check the following resource to help get into kernel-mode development, and official from Microsoft themselves: <https://docs.microsoft.com/en-us/windows-hardware/drivers/develop/>

The example source code is going to more-or-less replicate NtSuspendProcess/NtResumeProcess; no access to the System Service Descriptor Table required.

1. We do not have access to ObpReferenceObjectByHandle, but we do have access to ObReferenceObjectByHandle. ObReferenceObjectByHandle will call ObpReferenceObjectByHandle and it will allow us to obtain an object to the process by providing a valid handle to it.
2. PsSuspendProcess/PsResumeProcess.

Let's get on with it!

First things first, we need to setup our type-definitions. I'm using a header file specifically for the C file which is going to contain this.

```
#define PROCESS_SUSPEND_RESUME 0x0800

typedef NTSTATUS(NTAPI *pPsSuspendProcess)(
    PEPROCESS Process
);

typedef NTSTATUS(NTAPI *pPsResumeProcess)(
    PEPROCESS Process
);
```

The reason I've also defined **PROCESS_SUSPEND_RESUME** is because these "specific" access rights aren't available in the WDK libraries by default, and we need that access right for process suspension/resume operations. We don't need to have more privileges therefore we shouldn't try to gain such.

I got the PROCESS_SUSPEND_RESUME definition from MSDN: <https://msdn.microsoft.com/en-gb/library/windows/desktop/ms684880>

We already know that PsSuspendProcess and PsResumeProcess take in only one parameter of data-type PEPROCESS because of earlier when we took a look at the routines with static reverse engineering and saw what the NtSuspendProcess/NtResumeProcess routines were doing before invoking the Ps* routines. Because of ObpReferenceObjectByHandleWithTag, we know that the parameter is of type PEPROCESS (EPROCESS*).

The next thing we need to do is retrieve the address to PsSuspendProcess and PsResumeProcess. Since PsSuspendProcess and PsResumeProcess are both exported by NTOSKRNL, this will be very simple for us to do. There's a routine named MmGetSystemRoutineAddress.

```
PVOID MmGetSystemRoutineAddress(
    _In_ PUNICODE_STRING SystemRoutineName
);
```

According to the Microsoft documentation, this routine takes in one parameter which is of PUNICODE_STRING data-type.

We'll continue by doing the following.

1. Setup a routine to return us the address via MmGetSystemRoutineAddress
2. Setup the addresses for PsSuspendProcess and PsResumeProcess

```
pPsSuspendProcess fPsSuspendProcess;
pPsResumeProcess fPsResumeProcess;

PVOID ReturnSystemRoutineAddress(
    WCHAR *RoutineName
)
{
    UNICODE_STRING RoutineNameUs = { 0 };

    if (!RoutineName)
    {
        return 0;
    }

    RtlInitUnicodeString(&RoutineNameUs,
        RoutineName);

    return MmGetSystemRoutineAddress(&RoutineNameUs);
}

NTSTATUS InitializeExports()
{
    fPsSuspendProcess = (pPsSuspendProcess)ReturnSystemRoutineAddress(
        L"PsSuspendProcess");

    fPsResumeProcess = (pPsResumeProcess)ReturnSystemRoutineAddress(
        L"PsResumeProcess");

    if (!fPsSuspendProcess ||
        !fPsResumeProcess)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return STATUS_SUCCESS;
}
```

That should do the trick. Remember, you will need to make sure the InitializeExports routine is called before you attempt to make use of the fPsSuspendProcess/fPsResumeProcess variables being setup, otherwise they will point to NULL and you'll cause a bug-check crash due to dereferencing a NULL pointer.

The next thing we need to do is setup our wrapper routines to invoke PsSuspendProcess and PsResumeProcess. This isn't a necessity of course, you could just call fPsSuspendProcess/fPsResumeProcess or whatever you named those global variables to point to the correct address with the function type-definition set to it, but I find it easier to manage when you have a wrapper routine for each one – this means if there is ever an issue with the call itself, you can patch the issue by changing one routine instead of many. It also looks more appealing to me to have wrapper routines, so this is what I'll be doing in this article.

All we need to do is have two routines which return an NTSTATUS error code (STATUS_SUCCESS or another error, this will be the value returned by PsSuspendProcess/PsResumeProcess). We will need to check within the routine if the fPsSuspendProcess or fPsResumeProcess variable is NULL or not otherwise if the routine is accidentally called and the address truly is NULL, we'll be dereferencing a NULL pointer and end up bug-checking the system, which would be really silly. If the address is NULL then we can return STATUS_INSUFFICIENT_RESOURCES since this implies we do not have the required resources to complete the operation, which would be truthful because the addresses are a "resource" which are needed to make the suspension/resume operation.

```
NTSTATUS NTAPI PsSuspendProcess(
    PEPROCESS Process
)
{
    if (!fPsSuspendProcess)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fPsSuspendProcess(Process);
}

NTSTATUS NTAPI PsResumeProcess(
    PEPROCESS Process
)
{
    if (!fPsResumeProcess)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    return fPsResumeProcess(Process);
}
```

The last thing we have to do to implement the process suspension/resume functionality in kernel-mode is make our wrapper routines for NtSuspendProcess and NtResumeProcess. *Why think about locating the address to NtSuspendProcess (NTOSKRNL) or NtResumeProcess (NTOSKRNL) at all when you can have your own wrapper routine which does the same thing as the official one? Well, I can think of a few reasons why regarding stability... But don't ruin the moment!*

We have a few approaches for the NtSuspendProcess/NtResumeProcess wrapper.

1. We can use the documented, kernel-mode only routine named PsLookupProcessByProcessId to obtain an object to the targeted process and we can then pass this process object (PEPROCESS) to the PsSuspendProcess/PsResumeProcess wrapper routines.
2. We can accept a handle parameter to the NtSuspendProcess/NtResumeProcess wrapper routines the same way the official NtSuspendProcess/NtResumeProcess routines under NTOSKRNL do, and then we can use the handle to obtain an object to the process.

We're going to go with the second option to keep it closer to the real NtSuspendProcess/NtResumeProcess, but either are fine. Personally, I'd go for using the process object instead of a handle from the start if I was working in kernel-mode, but that's just me.

Thankfully, we have the ability to make use of the `ObReferenceObjectByHandle` routine. It's also officially documented. It isn't the same one used in the `NtSuspendProcess/NtResumeProcess` routines in the Windows Kernel but it leads down the same path so it's perfectly fine.


```

NTSTATUS NTAPI NtSuspendProcess(
    HANDLE ProcessHandle
)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    PEPROCESS Process = 0;

    if (!ProcessHandle)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    NtStatus = ObReferenceObjectByHandle(ProcessHandle,
        PROCESS_SUSPEND_RESUME,
        *PsProcessType,
        KernelMode,
        &Process,
        NULL);

    if (!NT_SUCCESS(NtStatus))
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    NtStatus = PsSuspendProcess(Process);

    ObDereferenceObject(Process);

    return NtStatus;
}

NTSTATUS NTAPI NtResumeProcess(
    HANDLE ProcessHandle
)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    PEPROCESS Process = 0;

    if (!ProcessHandle)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    NtStatus = ObReferenceObjectByHandle(ProcessHandle,
        PROCESS_SUSPEND_RESUME,
        *PsProcessType,
        KernelMode,
        &Process,
        NULL);

    if (!NT_SUCCESS(NtStatus))
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    NtStatus = PsResumeProcess(Process);

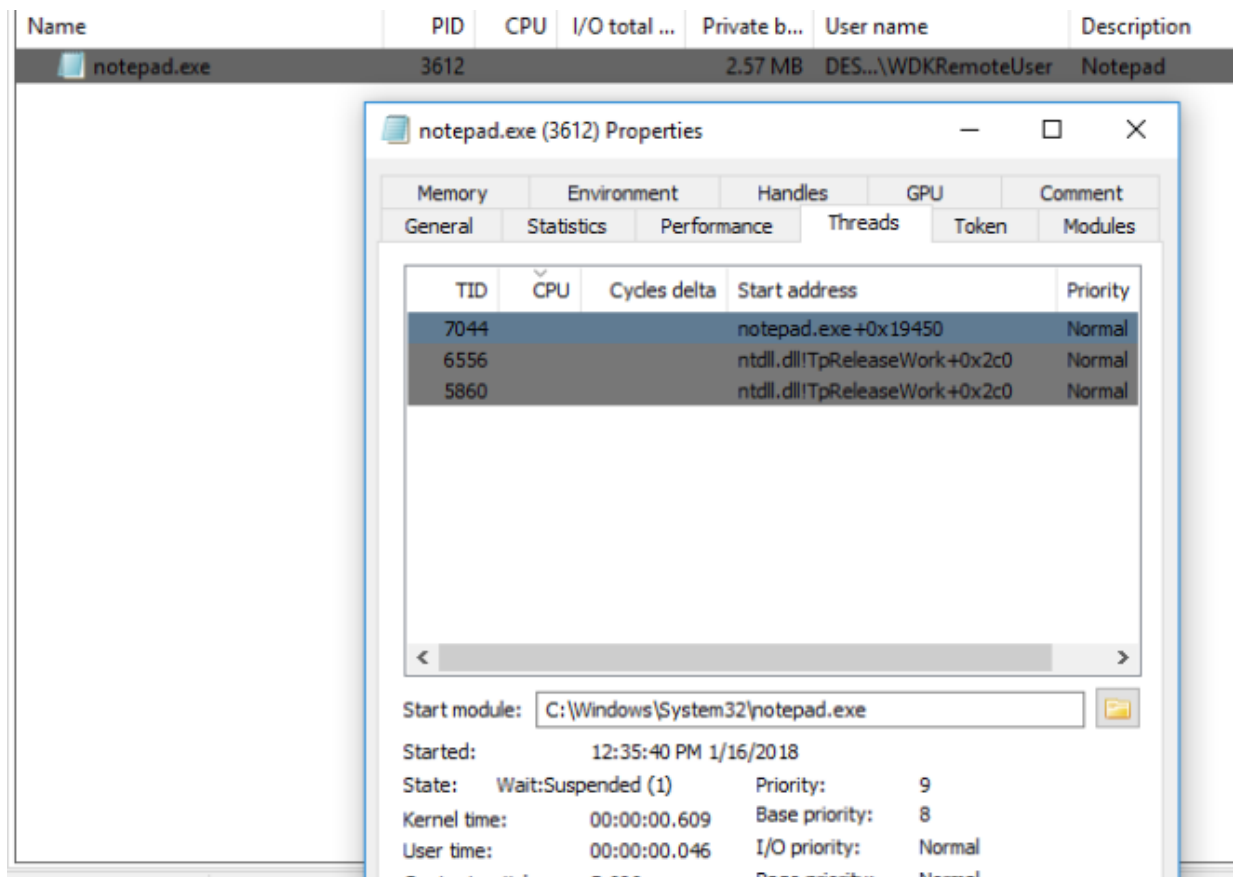
    ObDereferenceObject(Process);
}

```

```
    return NtStatus;  
}
```

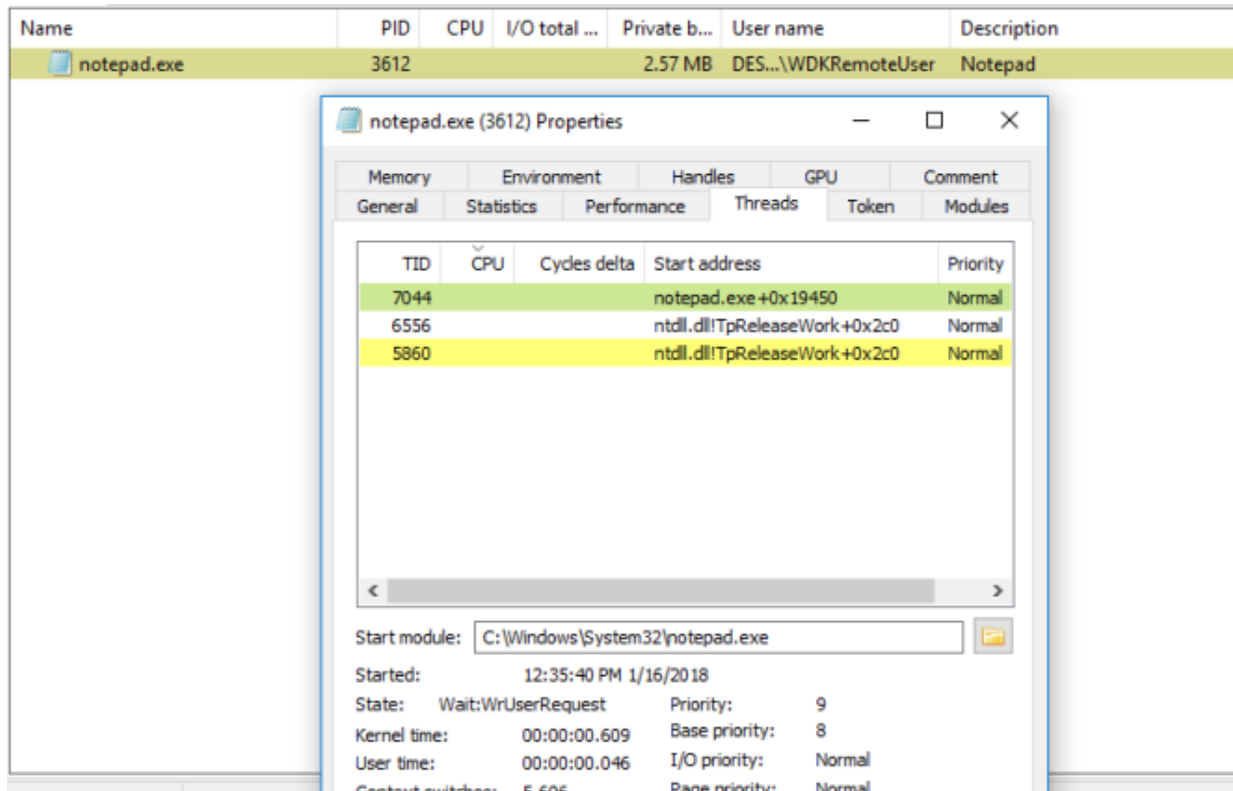
If you're conscious about sticking to as less code as possible for whichever reason, it may be nice for me to comment that you could make another wrapper routine which takes in the process handle and a BOOLEAN flag to determine if the process should be suspended or resumed. Then, depending on the flag, use the returned process object with the PsSuspendProcess/PsResumeProcess wrapper routines. This would prevent you from doing a check-up on the ProcessHandle parameter, calling ObReferenceObjectByHandle and checking the status codes in both routines which would cut down a few lines of code – but the end-result would be identical.

I tested it out by opening a handle to notepad.exe with PROCESS_SUSPEND_RESUME access rights and then passing the handle to the NtSuspendProcess routine. I set my DriverUnload routine to call NtResumeProcess on the process so it would be resumed after the driver was unloaded (experimental purposes).



Testing PsSuspendProcess with a remote kernel-debugger attached under an analysis environment 1/2.

At this moment in time, notepad.exe had been suspended by the experimental kernel-mode device driver.



Testing PsSuspendProcess with a remote kernel-debugger attached under an analysis environment 2/2.

This happened after unloading the driver, because the NtResumeProcess wrapper was called, targeting the notepad.exe suspended process. The process was successfully resumed. =

If you look at the screen-shot in which notepad.exe was in a suspended state, you'll notice the threads were suspended. If those threads weren't suspended, then the process would not be "suspended". After the process has been resumed in the above screen-shot, the threads are "resumed" (aka. "running"/"active").

You can even see in the screen-shot where the process is suspended that the "State" details in the Process Hacker pop-up Properties window (-> Threads tab) is saying "Wait:Suspended (1)" – the thread is being held up the same way a normal developer may call WaitForSingleObject (KeWaitForSingleObject is the kernel-mode equivalent) on a thread of his own in user-mode.

That's all for this post, I'd like to thank you for reading up to this point and hopefully this post was found to be useful!

– Opcode

Twitter: <https://twitter.com/NtOpcode>