APC Series: KiUserApcDispatcher and Wow64

repnz.github.io/posts/apc/wow64-user-apc

Sun, Jun 28, 2020

I recommend to read the previous posts before reading this one:

- User APC API: We discussed the user mode API of user APC
- User APC Internals: We discussed the implementation of user APC in the kernel

Let's continue our discussion about APC internals in windows: This time we'll discuss APC dispatching in user mode and how APC works in Wow64 processes:

- The evolution of KiUserApcDispatcher
- Modifications to APC functions to support Wow64
- Wow64 APC injection techniques

The evolution of KiUserApcDispatcher

NTDLL contains a set of entry points that the kernel uses to run code in user mode like: KiUserExceptionDispatcher, KiUserCallbackDispatcher, ...

In the previous post we saw that User APCs start executing at ntdll!KiUserApcDispatcher in user mode. What is the purpose of KiUserApcDispatcher?

To understand KiUserApcDispatcher, Let's explore the evolution of the implementation of KiUserApcDispatcher from old windows versions to the newest implementation. It's important to say that this function is written in assembly but I created a pseudo-code implementation that will help you understand the logic of this function faster:

```
//
// Windows XP SP3 32 bit OS
// the KiUserApcDispatcher function
//
VOID
KiUserApcDispatcher(
    PPS_APC_ROUTINE ApcRoutine, [esp]
    PVOID SystemArgument1,      [esp+4]
    PVOID SystemArgument2,      [esp+8]
    PVOID SystemArgument3,      [esp+0xc]
    CONTEXT ContextRecord       [esp+0x10]
    )
{
    ApcRoutine(
        SystemArgument1,
        SystemArgument2,
        SystemArgument3
    );

    NtContinue(&ContextRecord, TRUE);
}
```

The arguments of KiUserApcDispatcher are passed on the stack - we know that this function is written in assembly because of the custom calling convention - this function reads the ApcRoutine from `[esp]` while normally the return address is stored in `[esp]` - There's no return address in this case.

The arguments of KiUserApcDispatcher:

1. ApcRoutine: The "NormalRoutine" from the KAPC structure.
2. SystemArgument1-3 - The arguments of the APC
3. ContextRecord - The user mode context that the APC interrupted - In the typical case the instruction pointer is in an alertable wait system call stub, but in case of special user APC it can be anywhere in user mode.

Let's explore the implementation: We can see 2 important functions are called: First, ApcRoutine is called. The "ApcRoutine" argument to NtQueueApcThread / "NormalRoutine" of the KAPC object - This will invoke the actual code of the APC. This is the pointer that was passed to NtQueueApcThread.

As I explained in the first part of the series, All The pending user APCs has to be executed one after another and only when the queue is empty the OS should return to the context in "ContextRecord" - This is handled by "NtContinue":

```
NTSTATUS
NtContinue(
  PCONTEXT ContextRecord,
  BOOLEAN TestAlert
  );
```

This system call receives the ContextRecord from KiUserApcDispatcher. NtContinue is invoked with TestAlert = TRUE, which causes more pending APCs to be execute, if any.

So to summarize, In the original NT design, KiUserApcDispatcher had 2 main responsibilities:

1. Invoke the APC routine
2. Return to kernel mode via NtContinue and invoke more APCs if there are any.
3. If there are no APCs pending, return to the previous context inside the ContextRecord argument.

Now let's see how Wow64 influenced KiUserApcDispatcher in 64 bit operating systems.

WoW64

Code can be pretty different on 64 bit CPUs. For example:

- The pointer size is 8 bytes instead of 4 bytes since the address space is larger.
- Data structure alignment can be different in x64
- The calling convention is different (To utilize the additional registers the x64 CPU provides)
- ....

As you may imagine, the OS and OS APIs had be changed a bit. Most of the API is the same, but the ABI (Application Binary Interface) changed significantly. This raises a certain issue with supporting existing 32 bit applications on 64 bit operating systems. That's why Microsoft created a layer called Wow64.

The main purpose of Wow64 (Windows 32 bit on Windows 64 bit) is to make existing 32 bit executables work both on 32 bit operating systems and 64 bit operating systems. It's implemented as a compatability emulation layer that mostly emulates APIs of Windows that were changed because of the transition to x64 bit code. The OS has many places that had to be changed to adapt to Wow64, As you may imagine, the APC mechanism had to be changed a bit to adapt to Wow64.

A simple explanation of the design of Wow64:

- Most of the user mode libraries are compiled both as 64 bit DLLs and 32 bit DLLs. The 64 bit libraries are stored inside c:\windows\system32 and the 32 bit DLLs are stored in C:\Windows\SysWow64.
- Wow64 supports only user mode code. All kernel mode code is 64 bit.
- There are 2 versions of ntdll that are loaded into a Wow64 process: 64 bit and 32 bit.
- When an application calls a system call wrapper in the 32 bit ntdll (directly or through Win32), wow64cpu is called to change the CPU mode to 64 bit and invoke wow64.dll.

- wow64.dll has a wrapper per system call. The purpose of these wrappers are to translate the parameters for the 64 bit system calls and invoke the corresponding 64 bit NTDLL routine. Most of these wrappers are auto-generated code, but they can have custom implementation like the wrapper for NtQueueApcThread as we'll see soon.
- At the CPU level, the 32 bit / 64 bit mode transition is done by changing the CS segment that is used. This way, the OS changes from long mode (x64 mode) to IA32 compatibility mode. Read (https://wiki.osdev.org/X86-64#How_do_I_enable_Long_Mode_.3F) for details about the CPU mode change. In the security community, this is typically referred to as "Heaven's Gate".
- Many emulation tricks are used to fix assumptions in 32 bit executables. For example, c:\windows\system32 is redirected to c:\windows\syswow64 inside wow64 processes.

There are many more issues regarding Wow64, this is just a simple explanation that will allow us to understand the Wow64 APC.

To read more about Wow64, I recommend you to read a post by Petr Beneš about Wow64 Internals: (https://wbenny.github.io/2018/11/04/wow64-internals.html).

APC Compatability with Wow64

Ok, so imagine you wrote the following 32 bit application:

```
VOID
WINAPI
ApcCode(
        ULONG_PTR dwData
        )
{
        printf("32 bit code!\n");
}

int main()
{
        QueueUserAPC(ApcCode, GetCurrentThread(), 0);

        SleepEx(INFINITE, TRUE);
}
```

Why is there a compatability issue running this application on a 64 bit operating systems? As we saw at the beginning of the article, the execution of an APC starts at ntdll!KiUserApcDispatcher. Note that the kernel does not handle the transition to the Wow64 32 bit mode, so this has to be done in user mode somehow. This means that there are no kernel entry points inside the 32 bit NTDLL and execution begins in the 64 bit NTDLL. I hope you get it now: The 64 bit KiUserApcDispatcher has to handle the transition somehow

before executing the ApcRoutine, which expects to be executed in 32 bit mode. Also, 64 bit code can execute inside a wow64 process. How does the OS know if it needs to execute the APC in 32 bit or 64 bit?

First solution: Until Windows 7

The solution to this issue was implemented long time ago (somewhere around the XP era) and was re-implemented in Windows 7. Let's start by visiting the old implementation and we'll also talk about the modern implementation later.

To understand the solution, let's explore the wow64.dll wrapper of NtQueueApcThread on Windows Vista:

```
//
// In the source code it's probably an array, but I think it's cleaner that way.
//
typedef struct _WOW64_NT_QUEUE_APC_THREAD_ARGS {
    ULONG32 ThreadHandle;
    ULONG32 ApcRoutine;
    ULONG32 SystemArgument1;
    ULONG32 SystemArgument2;
    ULONG32 SystemArgument3;
} WOW64_NT_QUEUE_APC_THREAD_ARGS, *PWOW64_NT_QUEUE_APC_THREAD_ARGS;


//
// Windows Vista Implementation of the NtQueueApcThread compatibility wrapper in
wow64.dll
//
NTSTATUS
whNtQueueApcThread(
    PWOW64_NT_QUEUE_APC_THREAD_ARGS Arguments
    )
{
    PPS_APC_ROUTINE EncodedApcRoutine = (PPS_APC_ROUTINE)Arguments->ApcRoutine;
    PVOID EncodedSystemArgument1 = Arguments->SystemArgument1;

    //
    // - Replace the ApcRoutine with wow64!Wow64ApcRoutine
    // - Encode the real address into SystemArgument1
    //
    if (Arguments->ApcRoutine != NULL){
        EncodedApcRoutine = Wow64ApcRoutine;
        EncodedSystemArgument1 = (PVOID)(Arguments->ApcRoutine << 32 | Arguments-
>SystemArgument1);
    }

    return NtQueueApcThread(
        Arguments->ThreadHandle,
        EncodedApcRoutine,
        EncodedSystemArgument1,
        Arguments->SystemArgument2,
        Arguments->SystemArgument3
    );
}
```

So let's analyze this routine: First, we can see the WOW64_NT_QUEUE_APC_THREAD_ARGS structure. This is a structure that represents the original arguments that were passed to the 32 bit NtQueueApcThread. The 32 bit stdcall calling convention passes parameters on the stack. The "Arguments" pointer is actually a pointer to the stack that contains the original arguments of the call to the 32 bit NTDLL.

Ok, so how does this function solve the issue? As you can see, if Arguments->ApcRoutine is not NULL, it replaces the ApcRoutine with Wow64ApcRoutine. Wow64ApcRoutine is the function inside wow64.dll that prepares the arguments for the APC routine and handles the transition to Wow64 32 bit mode. But wait, what about the original ApcRoutine? The developers took advantage of the bigger size of the pointers (8 bytes instead of 4) and encoded the original APC routine inside the higher DWORD of SystemArgument1.

By the way, If you were wondering why the ContextRecord is passed as an argument for the ApcRoutine, it's to allow Wow64ApcRoutine to restore this context later. Wow64ApcRoutine does not return to the 64 bit KiUserApcDispatcher.

Another function called PsWrapApcWow64Thread was added to the windows kernel to allow drivers to perform this encoding:

```
//
// This function wraps the ApcContext and ApcRoutine for Wow64 processes.
// This function is not that useful. It works only when the current process is a
Wow64 process.
// If the current process is a 64 bit process and the target process is a Wow64
process, this function does not work.
//
NTSTATUS
PsWrapApcWow64Thread(
    __inout PVOID* ApcContext,
    __inout PVOID* ApcRoutine
    );
```

This is a summary of the old flow:

1. A Wow64 application calls the 32 bit ntdll NtQueueApcThread.
2. the 32 bit NtQueueApcThread invokes a wow64 routine that changes the mode to 64 bit.
3. Wow64.dll invokes a per-system call wrapper. In this case, it's whNtQueueApcThread.
4. whNtQueueApcThread replaces the ApcRoutine with Wow64ApcRoutine and encodes the original routine into SystemArgument1.
5. whNtQueueApcThread invokes the real NtQueueApcThread system call.
6. When the APC is delivered, KiUserApcDispatcher invokes the APC routine (Wow64ApcRoutine in this case).
7. Wow64ApcRoutine handles the transition to 32 bit mode and invokes the 32 bit ntdll KiUserApcDispatcher.
8. the 32 bit KiUserApcDispatcher handles the call to the encoded ApcRoutine.

One important thing about this issue is that if you develop a hooking or monitoring software and you hook system calls, you have to make sure to decode the Wow64 APC correctly.

This is a dump of the callstack. It's actually from windows 10 because unfortunately I don't have a vista VM, but the callstack should be pretty similar on Vista.

Wow64 QueueUserAPC Callstack

```
 RetAddr           Call Site
00007ffc`4d70545a ntdll!NtQueueApcThread
00007ffc`4d6f7123 wow64!whNtQueueApcThread+0x2a -> The NtQueueApcThread wrapper.
00000000`77ae1783 wow64!Wow64SystemServiceEx+0x153
00000000`77ae1199 wow64cpu!ServiceNoTurbo+0xb

758122ff ntdll_77af0000!NtQueueApcThread+0xc --> Invoke wow64 to change to 64 bit.
00652537 KERNELBASE!QueueUserAPC+0x4f
00652d53 32bitApc!main+0x47                  --> The main routine of our app.
00652ba7 32bitApc!invoke_main+0x33
00652a3d 32bitApc!__scrt_common_main_seh+0x157
00652dd8 32bitApc!__scrt_common_main+0xd
75f76359 32bitApc!mainCRTStartup+0x8
77b57c24 KERNEL32!BaseThreadInitThunk+0x19
77b57bf4 ntdll_77af0000!__RtlUserThreadStart+0x2f
00000000 ntdll_77af0000!_RtlUserThreadStart+0x1b --> the 32 bit ntdll is executed
now.

00007ffc`4d6fc77a wow64cpu!BTCpuSimulate+0x9  --> At this point, we change to 32 bit
mode to start
00007ffc`4d6fc637 wow64!RunCpuSimulation+0xa      executing the StartAddress of the
thread.
00007ffc`4e1f3f73 wow64!Wow64LdrpInitialize+0x127
00007ffc`4e1e1d75 ntdll!LdrpInitializeProcess+0x186b
00007ffc`4e1917d3 ntdll!_LdrpInitialize+0x50589
00007ffc`4e19177e ntdll!LdrpInitialize+0x3b
00000000`00000000 ntdll!LdrInitializeThunk+0xe --> Beginning of thread execution.
```

Modern solution: From Windows 7

The OS developers were not happy about this implementation. I have a theory about why they changed the implementation, but I'm not really sure about it. The main disadvantages of this scheme are:

1. Both ApcRoutine and SystemArgument1 had to be changed in case of a wow64 APC.
2. Because the address of Wow64ApcRoutine is taken from the source process, it means wow64.dll has to be mapped at the same address globally.
3. To decode the value, we need to get the address of Wow64ApcRoutine so we can test whether it's an encoded value or not.

wow64.dll is a known DLL so practically it's mapped at the same address globally, but maybe it's a constraint that the OS developers didn't want to have. Remember it's just a theory, there could be other reasons, though I could not think of other differences between the

encoding schemes.

The implementation has changed around Windows 7. The main functions that were changed are whNtQueueApcThread and KiUserApcDispatcher:

```
//
// ApcRoutine encoding routines. They are not real functions, so I mark them as
FORCEINLINE.
//
FORCEINLINE
ULONG64
DecodeWow64ApcRoutine(
        ULONG64 ApcRoutine
    )
{
        return (ULONG64)(-((INT64)ApcRoutine >> 2));
}

FORCEINLINE
ULONG64
EncodeWow64ApcRoutine(
        ULONG64 ApcRoutine
    )
{
        return (ULONG64)((-(INT64)ApcRoutine) << 2);
}

//
// Windows 10
//
NTSTATUS
whNtQueueApcThread(
    PWOW64_NT_QUEUE_APC_THREAD_ARGS Arguments
    )
{
    PPS_APC_ROUTINE EncodedApcRoutine = \
        (PPS_APC_ROUTINE)(EncodeWow64ApcRoutine(Arguments->ApcRoutine));

    //
    // Encode only the ApcRoutine
    //
    return NtQueueApcThread(
        Arguments->ThreadHandle,
        EncodedApcRoutine,
        Arguments->SystemArgument1,
        Arguments->SystemArgument2,
        Arguments->SystemArgument3
    );
}


//
// Windows 7 SP1 64 bit 6.1.7601
//
VOID
KiUserApcDispatcher(
    PCONTEXT ContextRecord // rsp
```

```
    )
{
    NTSTATUS Status;
    PPS_APC_ROUTINE ApcRoutine;
    PPS_APC_ROUTINE Wow64DecodedApcRoutine;

    do {
        ApcRoutine = ContextRecord->P4Home;

        //
        // Try to decode the APC routine.
        //
        Wow64DecodedApcRoutine = DecodeWow64ApcRoutine(ApcRoutine);


        //
        // If the result is a 32 bit address, try to invoke the Wow64ApcRoutine.
        //
        if (Wow64DecodedApcRoutine <= 0xFFFFFFFF) {
            if (Wow64ApcRoutine != NULL) {

                PVOID WrappedArgument1 = ((ULONGLONG)Wow64DecodedApcRoutine << 32) |
ContextRecord->P1Home;

                Wow64ApcRoutine(
                    WrappedArgument1,
                    ContextRecord->P2Home,
                    ContextRecord->P3Home,
                    ContextRecord
                );

                RtlRaiseStatus(STATUS_INVALID_PARAMETER);
            }
        } else {
            //
            // If the result is still a 64 bit address, invoke the original
ApcRoutine
            //
            ApcRoutine(
                ContextRecord->P1Home,
                ContextRecord->P2Home,
                ContextRecord->P3Home,
                ContextRecord
            );
        }

        Status = NtContinue(ContextRecord, TRUE);

    } while (Status == STATUS_SUCCESS);

    RtlRaiseStatus(Status);
}
```

The only encoded argument is the ApcRoutine. The encoding is pretty weird, let's try to explain the encoding. The best way to understand this encoding is by looking at the assembly code:

```
whNtQueueApcThread:
    ...
    ;
    ; Encode the ApcRoutine and send to NtQueueApcThread
    ;
    mov edx, [rcx + _WOW64_NT_QUEUE_APC_THREAD_ARGS.ApcRoutine]
    neg rdx
    shl rdx, 2
    ...

KiUserApcDispatcher:
    ...
    ;
    ; Decode the ApcRoutine
    ;
    mov rcx, [rsp + _CONTEXT.P4Home]
    sar rcx, 2
    neg rcx
    ....

    ;
    ; Check if the ApcRoutine is a wow64 routine.
    ; This is done by checking if the decoded value is less than MAX_ULONG
    ;
    shld    rcx, rcx, 32
        test    ecx, ecx
        jz      short Wow64Apc
```

Ok, so the main assumption of this encoding is: All user addresses have their sign bit off. This is because of the page tables structure. One thing I tried to understand is, why not just turn on the sign bit for the Wow64 APC and that's it? I could not figure this out, probably no special reason.

How the encoding really works

I recommend you to skip these details, unless you really need to. It's pretty annoying to understand that.

- If X is a valid user address, decode(X) is a 64 bit APC.
  - This is because user addresses cannot have their sign bit on.
  - The 'sar' instruction will zero the higher 2 bits
  - The 'neg' instruction will cause these 2 bits to be 1's.
- Is there any case that something is wow64 by mistake?
  Considering the fact that the sign bit has to be on, this cannot happen because user APCs has to point to a valid user address.

- If X is less than MAX_ULONG, decode(X) is a 64 bit APC.
  This is caused by the neg instruction, that will turn the zeros in the higher 32 bit to be 1's.

64 bit to Wow64 injection

Ok, So how does the Wow64 encoding influences injection?

Regarding 64 bit -> wow64 injection, we can do one of the following:

1. Run 64 bit APC code inside a Wow64 process. This is the default behavior.
2. Run 32 bit APC code inside a Wow64 process, this can be done by encoding the ApcRoutine.

Example code of a DLL injection below. Remember the APC will run only when the target thread will be alertable.

Look at the code:

```
OpenTargetHandles(
    Args.TargetProcessId,
    Args.TargetThreadId,
    &ProcessHandle,
    &ThreadHandle
);

if (!IsWow64Process(ProcessHandle, &IsWow64)) {
    printf("IsWow64Process Failed. 0x%08X\n", GetLastError());
    exit(-1);
}

if (!IsWow64) {
    printf("The target process is not a wow64 process.\n");
    exit(-1);
}

//
// Ok now we have 2 choices:
//
// - If the DLL is a 32 bit DLL, we need to create a Wow64 APC
// - If the DLL is a 64 bit DLL, we need to create a normal APC to ntdll,
//   because the 64 bit kernel32 is not loaded.
//

if (Is32bitDll(Args.DllPath)) {
    //
    // The DLL we want to load is a 32 bit DLL.
    // First, we need to write the path of the library to the remote process.
    //
    RemoteLibraryPath = WriteLibraryNameToRemote(ProcessHandle, Args.DllPath);

    //
    // To load this library, we can use the 32 bit LoadLibraryA routine inside
kernel32.
    //
    LoadLibraryAWowAddress = QueryWow64LoadLibraryAddress(ProcessHandle);

    //
    // Because the APC needs to run in a Wow64 environment, we need to encode the
routine.
    //
    PPS_APC_ROUTINE ApcRoutine =
(PPS_APC_ROUTINE)EncodeWow64ApcRoutine((ULONG64)LoadLibraryAWowAddress);

    //
    // Use NtQueueApcThread to queue the APC.
    //
    Status = NtQueueApcThread(
        ThreadHandle,
        ApcRoutine,
        RemoteLibraryPath,
```

```
        NULL,
        NULL
    );

    /*
    ntdll has a routine called "RtlQueueApcWow64Thread" which can be used to perform
the encoding.

    Status = RtlQueueApcWow64Thread(
                ThreadHandle,
                LoadLibraryAWowAddress,
                RemoteLibraryPath,
                NULL,
                NULL
            );
    */
}
else {
    //
    // The DLL is a 64 bit DLL and we want to load it to a 32 bit process.
    // We can use ntdll!LdrLoadDll to do it.
    //
    RemoteLibraryPath = WriteUnicodeLibraryNameToRemote(ProcessHandle, Args.DllPath);

    Status = NtQueueApcThread(
        ThreadHandle,
        (PPS_APC_ROUTINE)LdrLoadDllPtr,
        NULL,
        0,
        RemoteLibraryPath
    );
}

if (!NT_SUCCESS(Status)){
    printf("NtQueueApcThread Failed. 0x%08X\n", GetLastError());
    exit(-1);
}
```

Full working code is in my apc research repository: https://github.com/repnz/apc-research/blob/master/x64ToWow64ApcInjector/x64ToWow64ApcInjector.c

Wow64 to 64 bit injection

The issue with queueing a 64 bit apc from a Wow64 process is that the ApcRoutine cannot be saved inside a 4 bytes pointer. The 32 bit NtQueueApcThread routine can receive only a 4 bytes pointer. For certain system calls, there's a solution: The 32 bit NTDLL exports the following functions:

```
NTSTATUS
NTAPI
NtWow64ReadVirtualMemory64(
    HANDLE ProcessHandle,
    PVOID64 BaseAddress,
    PVOID BufferData,
    ULONG64 BufferLength,
    PULONG64 ReturnLength
    );

NTSTATUS
NTAPI
NtWow64WriteVirtualMemory64(
    HANDLE ProcessHandle,
    PVOID64 BaseAddress,
    PVOID BufferData,
    ULONG64 BufferLength,
    PULONG64 ReturnLength
    );

NTSTATUS
NTAPI
NtWow64AllocateVirtualMemory64(
    HANDLE ProcessHandle,
    PVOID64* BaseAddress,
    ULONG64 ZeroBits,
    PULONG64 RegionSize,
    ULONG AllocationType,
    ULONG Protect
    );

......
......
```

These functions allow you to perform operations on a 64 bit process from a 32 bit process. They are not system calls, these functions invoke wrappers in wow64.dll that perform the actual system call. Unfortunately, for NtQueueApcThread there's no such wrapper. If we want to invoke the NtQueueApcThread with a 64 bit pointer, we need to change to 64 bit mode somehow and call the system call. To do this, we can perform a far jump and change the CS segment so the CPU mode will be 64 bit mode ("Heaven's Gate"). Example code can be seen below:

```
__declspec(align(16))
typedef struct _NT_QUEUE_APC_THREAD_ARGS {
        DWORD64 ThreadHandle;
        DWORD64 ApcRoutine;
        DWORD64 SystemArgument1;
        DWORD64 SystemArgument2;
        DWORD64 SystemArgument3;
} NT_QUEUE_APC_THREAD_ARGS, *PNT_QUEUE_APC_THREAD_ARGS;


OpenTargetHandles(Args.TargetProcessId, Args.TargetThreadId, &ProcessHandle,
&ThreadHandle);

//
// Write the path of the DLL to the remote process
//
RemoteLibraryPath = WriteUnicodeLibraryNameToRemote(ProcessHandle, Args.DllPath);

//
// Save the 64 bit arguments on a struct
//
QueueApcArgs.ThreadHandle = (DWORD64)ThreadHandle;
QueueApcArgs.ApcRoutine = x64_GetNtdllProcedure("LdrLoadDll");
QueueApcArgs.SystemArgument1 = 0;
QueueApcArgs.SystemArgument2 = 0;
QueueApcArgs.SystemArgument3 = (DWORD64)RemoteLibraryPath;

//
// Find the address of the system call routine in NTDLL
//
NtQueueApcThreadAddress = x64_GetSyscallAddress("NtQueueApcThread");

//
// Change to 64 bit mode and invoke the system call.
//
Status = x64_InvokeSyscall(NtQueueApcThreadAddress, &QueueApcArgs,
sizeof(QueueApcArgs));
```

To understand more about the implementation, read the code: https://github.com/repnz/apc-research/blob/master/Wow64To64bitInjector/Wow64To64bitInjector.c

The Wow64 ApcRoutine Validation

Microsoft added a validation to prevent a programming error: If you try to queue an APC from a 32 bit process to a 64 bit process and you use a 32 bit address, you'll get this status code:

```
NtQueueApcThreadEx2:
....
//
// Verify that a Wow64 process does not try to queue into a 64 bit process with a 32
bit address.
// This is probably a programming bug in the Wow64 program
//
if (PsIsWow64Process(SourceProcess) && PsIs64BitProcess((PEPROCESS)TargetThread-
>Process)) {
    if (DecodeWow64ApcRoutine(ApcRoutine) <= 0xFFFFFFFF) {
        Status = STATUS_INVALID_HANDLE;
        goto Cleanup;
    }
}
....
```

More about KiUserApcDispatcher

KiUserApcDispatcher has more responsibilities:

1. CFG: Validate that the ApcRoutine is a valid indirect call target
2. Exception Handling: Wrap the ApcRoutine with an exception handler.

Summary

The main takeaways of these articles are:

- KiUserApcDispatcher is the entry point of APCs in user mode.
- In case of Wow64, the APC is encoded so KiUserApcDispatcher could detect and transfer execution to a Wow64 environment.
- If you develop hooking software, make sure you decode the Wow64 apc correctly.
- When we inject to Wow64 process, we can choose whether we want to queue to a 64 bit or 32 bit target code.
- If we want to queue from a Wow64 process to a 64 bit process, we need to switch to long mode and invoke the system call directly.

We finished our adventure in user APCs, In the next posts we'll explore kernel APCs. I hope it was not too long. We will revisit the user APC when we'll talk about special user APC.