APC Series: User APC API

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

Sun, May 17, 2020

Hey! Long time no see.

Coronavirus makes it harder for me to write posts, I hope I'll have the time to write - I have a lot I want to share!

One of the things I did in the last few weeks is to explore the APC mechanism in Windows and I wanted to share some of my findings. The purpose of this series is to allow you to get a systematic understanding of APC internals.

First, I reconstructed the source code of all of the kernel/user functions related to APC in the latest insider build, hoping to get a better understanding of the APC mechanism. (Without using a decompiler, just for fun)

I'll share the sources in the end of the series.

In the APC series we'll talk about the following topics:

- APC usage in User Mode
- APC usage in Kernel Mode
- Internals of User APCs
- Internals of Kernel APCs
- "Alerts" and how they are related to APC.
- How APCs work on wow64
- How to mess with Microsoft's Threat Intelligence APC ETW events, both in kernel mode and in user mode.

- How to unload a driver which uses APCs safely (well, kind of)
- How tools like procmon and process hacker utilize APCs.
- How CET (CPU shadow stack for ROP detection) influenced APC.
- Documented source code of the APC mechanism for the curious people.
- Solutions for “Practical Reverse Engineering” exercises about APC.
- Surprises I leave for future posts.

The first post is about User Mode APCs - This is the simplest part of the series, just to get you started;)

Don’t worry, I’ll share all the annotated source soon, including kernel routines, step by step.

Most of this post was written based on reverse engineering and debugging so it might not be accurate, I tried to test and verify every ‘fact’ I wrote in this post, but if you found any mistake please contact me and I’ll fix it :)

Introduction to APC

APC (Asynchronous Procedure Call) is a mechanism that can be used in Windows to queue a job to be done in the context of a specific thread. This is useful for several things - mainly for asynchronous callbacks - security people know about APC mainly because it’s used by Malware to inject code into a different process - but this is just an abuse of this mechanism.

In kernel mode, people typically should not mess with APCs because the API is undocumented, but security people (including both rootkits and AV developers) use it to inject their code into user mode processes from a kernel driver.

Each thread has 2 queues: One for user mode APCs and the other for kernel mode APCs.

I really discourage the use of APCs from kernel mode if you’re not absolutely sure what you’re doing and even if you’re a super expert nothing prevents Microsoft from breaking your code, that’s true for every undocumented code. However, I think using APCs in certain ways can be relatively safe, if you know what you’re doing. We’ll discuss kernel APCs in future posts in the series.

For example, When you call an asynchronous RPC method, you can specify the address of an APC routine that will execute when the RPC method completes. This is just one example, there are many examples of mechanisms that use APC like NtWriteFile/NtReadFile, IO completion in IRPs, Get/SetThreadContext, SuspendThread, TerminateThread and much more. Also, the scheduler of windows uses APCs too. That’s why I think understanding APCs is important for understanding windows internals - although this is an undocumented feature that people in Microsoft say that “developers should not care about”.

There are 2 types of User Mode APCs:

1. User APC: Normal type of User APCs, Runs only when the thread is alertable (or in specific situations we'll see soon)
2. Special User APC: A relatively new type of APCs that was added in RS5. (undocumented by MS yet)

Just so you get the idea, This is the main API used to queue APCs:

```
DWORD QueueUserAPC(
  PAPCFUNC  pfnAPC,
  HANDLE    hThread,
  ULONG_PTR dwData
);
```

Alertable State

One of the main issues (for malware) with user APC is: the caller thread has to be in an alertable state to receive this call. A thread becomes "Alertable" if it calls the "wait" routines - WaitForSingleObjectEx, SleepEx, etc with Alertable = TRUE. When doing this, Windows may deliver APCs to this thread before returning from these functions. This allows the developer of the program to control in which parts of the program a user APC can be delivered. Another function that can be used to allow pending APCs to execute is NtTestAlert, which we're going to explore in a future post.

Microsoft did not want to "force" a user mode thread to execute an APC. I think (only theory) the main reason is because it may expose this thread to subtle race conditions and deadlocks. Imagine you write a program that has a list that is guarded by a certain lock. In your program, You use async RPC with an APC completion routine. Then, right after the thread acquires the lock, while performing an operation on a list, an APC is forced to run in this thread. Then, the code of the APC tries to acquire the lock and boom deadlock - or worse, the thread will touch the list although the lock is acquired and the code is already inside the critical section. If you deliver APCs only when the thread enters alertable state it can improve this situation by making it less likely to happen (depending on how the developer wrote his code) but it does not prevent it.

Acquiring locks inside an APC is probably a bad idea anyway, but in the kernel microsoft has partially solved this issue - you can prevent APCs to a thread that acquires a certain lock by raising IRQL to APC_LEVEL or disabling APCs via KeEnterCriticalRegion / KeEnterGuardedRegion - sometimes it's required by certain APIs like ExAcquireResourceSharedLite for example. I'll talk about it when I'll explore APC in kernel mode in a future post.

Moreover, It's important to note that while user mode APCs are indeed documented, Microsoft has some recommendations:

- Do not enter alertable state inside an APC - There's nothing preventing an APC from interrupting another user APC if the thread is alertable, so it may cause a stack overflow.
- Microsoft recommends not to queue an APC to a different process, mainly because potential differences between addresses to functions between processes because of relocations and wow64.

I neither agree nor disagree with these recommendations, But when using a certain feature of the OS it's important to understand it deeply and know how it was meant to be used.

APC for injection

Anyway, back to business - Running APCs only when the thread approves it means that if you want to use APCs for injection (from user mode) you have to find an alertable thread or hope the thread will enter an alertable state soon. But, In RS5 Microsoft implemented an interesting mechanism: Special User APCs. This mechanism allows the caller to "force" an APC into a particular thread even if it's not currently alertable, by queueing a Kernel APC that will signal the thread's execution. This kernel APC re-queues a user APC to the same thread. I'll explore the internals of this mechanism in a future post.

Let's talk now about the basic differences between these APCs from an API perspective. Let's say I have 2 processes: "Safe" and "Malicious" process, the "Malicious" process wants to inject code into the "Safe" process using APC. Let's say, the "Safe" process has only one thread which executes the following lines of code:

```
int main() {
    while (1) { Sleep(500); }
    return 0;
}
```

Looks simple, right?

If the injector uses a normal user APC to queue the APC - it will never execute! This is not really surprising because this thread does not enter alertable state. The APC is placed in the queue and never get executed. When the thread is terminated, this user APC is freed.

The Special User APC is a mechanism that was added in RS5 (and exposed through NtQueueApcThreadEx), but lately (in an insider build) was exposed through a new syscall - NtQueueApcThreadEx2. If this type of APC is used, the thread is signaled in the middle of the execution to execute the special APC.

As an attacker, this may sound tempting - but, this is actually pretty dangerous - For example, imagine a thread is in the middle of loading a library (because LoadLibrary was called.) and a special user APC was queued to this thread. As you may know, LoadLibrary touches the loader structures in the PEB and also acquires some locks. Let's say the target address of the APC is LoadLibrary, because the attacker wanted to load his DLL to the remote process. This can cause problems because the same thread is already inside LoadLibrary - This is exactly why Microsoft did not want to allow an APC to run if the thread is not in an alertable state - now the thread is stuck because of a deadlock or the data structures are in an undefined state. This issue sounds rare - but it's actually very dangerous, because LoadLibrary() is not the only function that uses locks. Nevertheless, Special User APCs can be very powerful and useful for attackers.

Generally, if you want to do things right, it's pretty hard to use APC for user mode injection unless you know a bit about the target thread..

Exploring the API

Let's start from the bottom up. the kernel exposes 3 system calls to queue APCs: NtQueueApcThread, NtQueueApcThreadEx and NtQueueApcThreadEx2. QueueUserAPC is a wrapper function in kernelbase that calls NtQueueApcThread. Let's look.

NtQueueApcThread: System Call Layer

```
//
// This is the signature of the first system call that can queue APCs.
// This function is undocumented.
//
// It receives 5 arguments:
//
// ThreadHandle - a HANDLE to the target thread. This HANDLE must have
THREAD_SET_CONTEXT access.
//                This thread can be in a different process, though Microsoft do not
recommend developers to
///               queue an APC to a different process.
//
// ApcRoutine - This is the address of the target routine in the context of the
target process.
//              The parameters of this function can be controlled by
NtQueueApcThread.
//
// SystemArgument1-3 - The first 3 arguments of the ApcRoutine.
//
//
NTSTATUS
NtQueueApcThread(
    IN HANDLE ThreadHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );

//
// This function is the signature of APC functions at the native layer.
// It allows to pass 3 parameters. Moreover, in x64 there's a hidden parameter
// that is passed to the APC function, the ContextRecord,
// which is a pointer to the context of the thread before the execution of the APC.
// I'm not sure if this parameter exists in 32 bit..
//
// Also, something cool about the x64 calling convention - because the caller
allocates / frees
// the parameters, we can call functions with less than 4 parameters like
LoadLibrary() which receives only 1 parameter.
//
//
typedef
VOID
(*PPS_APC_ROUTINE)(
    PVOID SystemArgument1,
    PVOID SystemArgument2,
    PVOID SystemArgument3,
    PCONTEXT ContextRecord
);
```

This API is very simple and easy to use. Let's see an example usage: *removed error handling for simplicity*

Of course I don't recommend to use it in such way in real products. If you're a real developer, read the intro.

```
VOID
QueueLoadLibrary(
    ULONG ProcessId,
    PSTR LibraryName
    )
{
    PVOID RemoteLibraryAddress;
    HANDLE ProcessHandle;
    HANDLE ThreadHandle;
    NTSTATUS Status;

    //
    // Open the process with the required access to allocate and write the library
name.
    //
    ProcessHandle = OpenProcess(
                        PROCESS_QUERY_INFORMATION |
                        PROCESS_VM_OPERATION |
                        PROCESS_VM_WRITE,
                        FALSE,
                        ProcessId
                    );

        RemoteLibraryAddress = WriteLibraryNameToRemote(ProcessHandle, LibraryName);

    //
    // Get a handle to the first thread in the process.
    //
    NtGetNextThread(
        ProcessHandle,
        NULL,
        THREAD_SET_CONTEXT,
        0,
        0,
        &ThreadHandle
    );

    NtQueueApcThread(
        ThreadHandle,
        GetProcAddress(GetModuleHandle("kernel32"), "LoadLibraryA"),
        RemoteLibraryAddress,
        NULL,
        NULL
    );
}
```

Well, this usage is pretty simple, read comments to understand. Note that the signature of LoadLibraryA() is not PPS_APC_ROUTINE, but it’s fine because it’s x64.

QueueUserAPC: KernelBase.dll Layer

Well, Microsoft likes to create wrappers for system calls - so they’ll be able to change the implementation of stuff etc. Microsoft also likes COM and additional DLL loading and redirection mechanisms. The combination of both created the following function: QueueUserAPC.

```
//
// This is the wrapper function implemented in kernelbase.dll to queue APCs. This
function is documented.
// This function has 3 arguments:
//
// pfnAPC - the pointer to the apc routine in the target process context.
//          Note that the signature of this function is different from the signature
in NtQueueApcThread.
//
// hThread - the handle to the target thread. Requires THREAD_SET_CONTEXT.
//
// dwData - the context argument passed to pfnAPC - This is the only argument passed
to pfnAPC.
//
DWORD
QueueUserAPC(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData
    );

//
// This is the signature of the APC Routine if QueueUserAPC is used.
// The only parameter here is the dwData argument from QueueUserAPC.
// You may ask why the signature is different than the signature of PPS_APC_ROUTINE,
we'll see below why.
//
typedef
VOID
(NTAPI *PAPCFUNC)(
    IN ULONG_PTR Parameter
    );


//
// This is the reverse engineered implementation of QueueUserAPC in windows 10
// (In was changed a bit in the latest insider, you'll see below)
//
// This function captures the activation context of the current thread
// and saves it, so it can be inherited by the APC routine.
//
// Activation Contexts are data structures that save configuration for DLL
redirection, SxS and COM.
// To read more about activation context: https://docs.microsoft.com/en-
us/windows/win32/sbscs/activation-contexts.
//
DWORD
QueueUserAPC(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData
    )
```

```
{
    ACTIVATION_CONTEXT_BASIC_INFORMATION Info;
    NTSTATUS Status;

    //
    // Capture the activation context of the current thread.
    //
    Status = RtlQueryInformationActivationContext(
            1,
            NULL,
            NULL,
            ActivationContextBasicInformation,
            &Info,
            sizeof(Info),
            NULL
        );

    if (!NT_SUCCESS(Status)) {
        DbgPrint("SXS: %s failing because RtlQueryInformationActivationContext()
returned status %08lx",
                            "QueueUserAPC", Status);
        BaseSetLastNTError(Status);
        return 0;
    }

    //
    // Forward the call to the actual system call.
    // The ApcRoutine that is used is actually a wrapper function in ntdll, called
"RtlDispatchApc"
    // The purpose of this wrapper function is to use the activation context passed
as a parameter.
    //
    Status = NtQueueApcThread(
            hThread,        // ThreadHandle
            RtlDispatchAPC, // ApcRoutine
            (PPS_APC_ROUTINE)pfnAPC, // SystemArgument1
            (PVOID)dwData, // SystemArgument2
            Info.hActCtx // SystemArgument3
        );

    if (!NT_SUCCESS(Status)) {
        BaseSetLastNTError(Status);
        return 0;
    }

    return 1;
}

//
// This is used as SystemArgument3 if QueueUserAPC
// was used to queue the APC.
//
```

```
typedef union _APC_ACTIVATION_CTX {
    ULONG_PTR Value;
    HANDLE hActCtx;
} APC_ACTIVATION_CTX;


//
// This is the actual APC routine.
// It enables the activation context, calls the user provided routine, and
deactivates the context.
//
VOID
RtlDispatchAPC( // ntdll
    PAPCFUNC pfnAPC,
    ULONG_PTR dwData,
    APC_ACTIVATION_CTX ApcActivationContext
    )
{
    RTL_CALLER_ALLOCATED_ACTIVATION_CONTEXT_STACK_FRAME_EXTENDED StackFrame;

    //
    // Initialize the StackFrame data structure.
    //
    StackFrame.Size =
sizeof(RTL_CALLER_ALLOCATED_ACTIVATION_CONTEXT_STACK_FRAME_EXTENDED);
    StackFrame.Format = 1;
    StackFrame.Extra1 = 0;
    StackFrame.Extra2 = 0;
    StackFrame.Extra3 = 0;
    StackFrame.Extra4 = 0;

    if (ApcActivationContext.Value == -1) {
        pfnAPC(dwData);
        return;
    }

    //
    // Use the activation context of the queuing thread.
    //
    RtlActivateActivationContextUnsafeFast(&StackFrame,
ApcActivationContext.hActCtx);

    //
    // Call the user provided routine.
    //
    pfnAPC(dwData);

    //
    // Pop the activation context from the "activation context stack"
    //
    RtlDeactivateActivationContextUnsafeFast(&StackFrame);
    //
    // Free the handle to the activation context.
    //
    RtlReleaseActivationContext(ApcActivationContext.hActCtx);
}
```

As you can see above, the user provided APC routine of APCs queued using QueueUserAPC is stored in SystemArgument1. This is something important to note for those of you who deal with hooking or monitoring stuff.

NtQueueApcThreadEx: Reusing Kernel Memory

Each time NtQueueApcThread is called, a new KAPC object is allocated in kernel mode (from the kernel pool) to store the data about the APC object. Let's say there's a component that queues a lot of APCs, one after another. This can have performance implications because a lot of non-paged memory is used and also allocating memory takes some time..

In windows 7, Microsoft added a very simple object to kernel mode called the memory reserve object. It allows to reserve memory for certain objects in kernel mode and later when the object is freed use the same memory area to store another objects, thus recuding the number of ExAllocatePool/ExFreePool calls. NtQueueApcThreadEx receives a HANDLE to such object, thus allowing the caller to reuse the same memory.

This is the interface for creating a "Memory Reserve Handle":

```
//
// The memory reserve object currently supports allocating 2 types of objects:
// - User APC
// - Io Completion
//
typedef enum _MEMORY_RESERVE_OBJECT_TYPE {
     MemoryReserveObjectTypeUserApc,
     MemoryReserveObjectTypeIoCompletion
 } MEMORY_RESERVE_OBJECT_TYPE, PMEMORY_RESERVE_OBJECT_TYPE;

//
// The system call to allocate a memory reserve object.
//
NTSTATUS
NtAllocateReserveObject(
    __out PHANDLE MemoryReserveHandle,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in MEMORY_RESERVE_OBJECT_TYPE ObjectType
    );

//
// This is the new system call that was added in Windows 7.
// This system call is the same as NtQueueApcThread, but allows to specify a
MemoryReserveHandle.
// This is a handle to an object allocated using NtAllocateReserveObject.
//
// If the memory is currently in use (for example, the APC object was not freed yet)
you can not reuse the memory.
//
NTSTATUS
NtQueueApcThreadEx(
    IN HANDLE ThreadHandle,
    IN HANDLE MemoryReserveHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );
```

Using this new object, we can save the overhead of the allocation of KAPC objects, example code below:

```
int main(
        int argc,
        const char** argv
        )
{
        NTSTATUS Status;
        HANDLE MemoryReserveHandle;

    //
    // Create the memory reserve handle. This will allocate space for a KAPC object
in kernel mode.
    //
        Status = NtAllocateReserveObject(&MemoryReserveHandle, NULL,
MemoryReserveObjectTypeUserApc);

        if (!NT_SUCCESS(Status)) {
                printf("NtAllocateReserveObject Failed! 0x%08X\n", Status);
                return -1;
        }

        while (TRUE) {
                //
                // Queue the APC to the current thread.
        // Reuse the memory allocated using NtAllocateReserveObject.
                //
                Status = NtQueueApcThreadEx(
                                        GetCurrentThread(),
                                        MemoryReserveHandle,
                                        ExampleApcRoutine,
                                        NULL,
                                        NULL,
                                        NULL
                                );

                if (!NT_SUCCESS(Status)) {
                        printf("NtQueueApcThreadEx Failed! 0x%08X\n", Status);
                        return -1;
                }

                //
                // Enter alertable state to execute the APC.
                //
                SleepEx(0, TRUE);
        }

        return 0;
}

VOID
ExampleApcRoutine(
        PVOID SystemArgument1,
        PVOID SystemArgument2,
        PVOID SystemArgument3
)
{
    //
    // This sleep is not alertable.
    //
        Sleep(500);

        printf("This is the weird loop!\n");
}
```

This mechanism is used by RPC servers to reuse APC objects of completion routines. If you're curious, you can look in rpcrt4!CALL::QueueAPC

NtQueueApcThreadEx: Meet Special User APC

In RS5, Microsoft implemented the Special User APC. I did not find any usage of this functionality in my windows machine. As I said above, the Special User APC can be used to force a thread to execute an APC routine, even if it's not in an alertable state.

When RS5 was released, Microsoft did not want to add another system call (yet) so they changed NtQueueApcThreadEx to support special user APCs by turning the MemoryReserveHandle into a union:

```
//
// This is a new "Flags" enum they added.
// The only supported flag is QueueUserApcFlagsSpecialUserApc.
//
typedef enum _QUEUE_USER_APC_FLAGS {
        QueueUserApcFlagsNone,
        QueueUserApcFlagsSpecialUserApc,
        QueueUserApcFlagsMaxValue
} QUEUE_USER_APC_FLAGS;

//
// This is a union that is used instead of the MemoryReserveHandle argument.
// This is backwards compatible to old usage of the system call.
//
typedef union _USER_APC_OPTION {
        ULONG_PTR UserApcFlags;
        HANDLE MemoryReserveHandle;
} USER_APC_OPTION, *PUSER_APC_OPTION;


//
// Same as before, but MemoryReserveHandle was replaced by UserApcOption.
// This allows the caller to use one of the options.
//
NTSTATUS
NtQueueApcThreadEx(
        IN HANDLE ThreadHandle,
        IN USER_APC_OPTION UserApcOption,
        IN PPS_APC_ROUTINE ApcRoutine,
        IN PVOID SystemArgument1 OPTIONAL,
        IN PVOID SystemArgument2 OPTIONAL,
        IN PVOID SystemArgument3 OPTIONAL
        );
```

This is an example usage of Special User APC:

```
int main(
        int argc,
        const char** argv
        )
{
        PNT_QUEUE_APC_THREAD_EX NtQueueApcThreadEx;
        USER_APC_OPTION UserApcOption;
        NTSTATUS Status;

        NtQueueApcThreadEx = (PNT_QUEUE_APC_THREAD_EX)
(GetProcAddress(GetModuleHandle("ntdll.dll"), "NtQueueApcThreadEx"));

        if (!NtQueueApcThreadEx) {
                printf("wtf, before win7\n");
                return -1;
        }

    //
    // This is a special flag that tells NtQueueApcThreadEx this APC is a special
user APC.
    //
        UserApcOption.UserApcFlags = QueueUserApcFlagsSpecialUserApc;

        while (TRUE) {
                //
                // This will force the current thread to execute the special user
APC,
                // Although the current thread does not enter alertable state.
        // The APC will execute before the thread returns from kernel mode.
                //
                Status = NtQueueApcThreadEx(
                                        GetCurrentThread(),
                                        UserApcOption,
                                        ApcRoutine,
                                        NULL,
                                        NULL,
                                        NULL
                                );

                if (!NT_SUCCESS(Status)) {
                        printf("NtQueueApcThreadEx Failed! 0x%08X\n", Status);
                        return -1;
                }

                //
                // This sleep does not enter alertable state.
                //
                Sleep(500);
        }

    return 0;
}
```

```
VOID
ApcRoutine(
        PVOID SystemArgument1,
        PVOID SystemArgument2,
        PVOID SystemArgument3
        )
{
        printf("yo wtf?? I was not alertable!\n");
}
```

Note that special user APCs can also interrupt the execution of a different thread, this will be explored in future posts.

NtQueueApcThreadEx2: Some new friends in the fast ring

Somewhere around windows insider build 19603, two important functions were added:

1. NtQueueApcThreadEx2: This is a new system call, that allows to pass both UserApcFlags and MemoryReserveHandle. (Well, it won't work becasue some validation checks you don't use both lol.)
2. QueueUserAPC2: This is a new wrapper function in kernelbase.dll, that allows the user to access the special user APC.

The new wrapper function may show Microsoft's intention to document QueueUserAPC2 and allow clients to use it. Well, it can be utilized to signal a thread in the middle of execution, something that can be pretty useful - for example to emulate signaling mechanisms similar to how Linux can signal threads (pthread_cancel).

```
NTSTATUS
NtQueueApcThreadEx2(
    IN HANDLE ThreadHandle,
    IN HANDLE UserApcReserveHandle,
    IN QUEUE_USER_APC_FLAGS QueueUserApcFlags,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );


DWORD
QueueUserApc2(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData,
    QUEUE_USER_APC_FLAGS Flags
    );
```

NtTestAlert

NtTestAlert is a system call that’s related to the alerts mechanism of Windows. This system call can cause execution of any pending APCs the thread has (well, if the thread is not alerted. Don’t confuse an “alerted” thread with an “alertable” thread) We’ll explore the internals of this mechanism in a future post, but it’s important to note that before a thread starts executing it’s Win32 start address (Inside ntdll!_LdrpInitialize) it calls NtTestAlert to execute any pending APCs. Let’s see how we can abuse this:

```
int
main(
        int argc,
        const char** argv
)
{

    //
    // Create the thread as a suspended thread.
    // NtTestAlert is called at the beginning of the lifetime of the new thread,
    // before 'NewThread' is executed.
    //
    // Because we want this NtTestAlert call to cause the thread to execute the APC,
    // we need to queue the APC before this NtTestAlert call is called.
    //
    // This is why we create the thread in a suspended state, then queue the APC,
then
    // resume the thread.
    //
    // This technique can be used with remote threads as well and often used with
CreateProcess(CREATE_SUSPENDED).
    //
        HANDLE ThreadHandle = CreateThread(
                                                        NULL,
                                                        0,
                                                        NewThread,
                                                        NULL,
                                                        CREATE_SUSPENDED,
                                                        NULL
                                                );


        if (!QueueUserAPC(
                ApcRoutine,
                ThreadHandle,
                0
        )) {
                printf("Error queueing APC\n");
        }

        printf("APC Queued. Calling ResumeThread..\n");

        ResumeThread(ThreadHandle);

        WaitForSingleObject(ThreadHandle, INFINITE);

        return 0;
}

VOID
ApcRoutine(
        ULONG_PTR dwData
)
{
        printf("Inside the ApcRoutine.\n");
}


DWORD
NewThread(
        PVOID ThreadArgument
        )
{
        printf("Inside the thread.\n");
        return 0;
}
```

The output is:

```
APC Queued. Calling ResumeThread..
Inside the ApcRoutine.
Inside the thread.
```

By the way, this NtTestAlert call is what allows AVs and rootkits to execute their user APCs when the thread begins - we'll see examples in a future post.

Summary

In the next blog-posts, we'll continue exploring the internals of APCs. I hope I'll have time to write the new post this week.. This is the repo I use to store all the code that is related to this series: apc-research

EDIT: The next blog post is here: User APC Internals

Side-Note

You may ask, what got me into exploring APCs?

Well, I'm always excited about exploring and reverse engineering mechanisms of Windows. After seeing a tweet about the addition of special user APCs, I got super excited. The last time APC internals were documented was long time ago, I think this series can be helpful for people that want to explore windows internals. Stay tuned! This is just the beginning!!

@0xrepnz