An interesting thing I saw many years ago was a plug-in for an old gaming client that added a lot of functionality. Some of the things were simple changes such as just modifying the resources to give the client a sleeker look. Other things were more interesting though, and included adding custom menus, both to the menu bar and context menus. This was interesting since it everything was contained in separate DLLs, which were loaded by the process on startup (parts of the executable were rewritten to jump to loader code). When finding out how this works, I found that it was pretty easy to do. Injected/loaded DLLs are within the address space of the process, so doing all of this is more or less similar to how it would be done when normally developing a GUI application with the Windows API. Unfortunately, this technique does not work as widespread as it did back then. Custom GUI APIs and new additions to Windows such as floating and dockable menus make this technique in its current form less useful. I wanted to put the code that I wrote several years ago on here for archive purposes. The code is old and could possibly be written to be a bit cleaner.

```
#include <Windows.h>

typedef struct _PROCESSWNDINFO {
    HWND hWnd;
    HMENU hMenuBar;
    HMENU hAddedMenu;
    LONG_PTR PrevWndProc;
} PROCESSWNDINFO, *LPPROCESSWNDINFO;

const DWORD MENUITEM_ID = 1234;
PROCESSWNDINFO g_WindowInfo;

LRESULT CALLBACK SublassWndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam) {
    switch(Msg) {
        case WM_COMMAND:
            switch(wParam) {
            case MENUITEM_ID:
                MessageBox(NULL, L"Added Handler!", L"New item",
MB_ICONASTERISK);
                break;
            }
        break;

    default: break;
    }
    return CallWindowProc((WNDPROC)g_WindowInfo.PrevWndProc, hWnd, Msg,
wParam, lParam);
}

BOOL CALLBACK EnumWindowProc(HWND hWnd, LPARAM processId) {
    const INT WINDOW_LENGTH = 32;
    WCHAR windowClass[WINDOW_LENGTH] = {0};
    GetClassName(hWnd, windowClass, sizeof(windowClass));
    if(wcscmp(windowClass, L"GDI+ Hook Window Class") == 0)
        return TRUE;
    DWORD windowProcessId = 0;
    (VOID)GetWindowThreadProcessId(hWnd, &windowProcessId);
    if(windowProcessId == (DWORD)processId) {
        g_WindowInfo.hWnd = hWnd;
```

```
        return FALSE;
    }
    return TRUE;
}

BOOL GetWindowProperties(LPPROCESSWNDINFO windowInfo) {
    EnumWindows((WNDENUMPROC)EnumWindowProc, (LPARAM)GetCurrentProcessId());
    if(windowInfo->hWnd != NULL) {
        windowInfo->hMenuBar = GetMenu(g_WindowInfo.hWnd);
        windowInfo->PrevWndProc = GetWindowLongPtr(windowInfo->hWnd,
GWLP_WNDPROC);
        return TRUE;
    }
    return FALSE;
}

HMENU AddNewMenu(LPPROCESSWNDINFO windowInfo, LPCWSTR title) {
    HMENU hNewMenu = CreateMenu();
    if(hNewMenu != NULL && windowInfo->hMenuBar != NULL)
        if(AppendMenu(windowInfo->hMenuBar, MF_STRING | MF_POPUP,
(UINT_PTR)hNewMenu, title) != 0)
            return hNewMenu;
    return NULL;
}

BOOL AddNewMenuItem(LPPROCESSWNDINFO windowInfo, HMENU hMenu, LPCWSTR title,
const DWORD id) {
    BOOL ret = FALSE;
    if(hMenu != NULL)
        ret = AppendMenu(hMenu, MF_STRING, id, title);
    if(windowInfo->hMenuBar != NULL)
        DrawMenuBar(windowInfo->hWnd);
    return ret;
}

int APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
    if(reason == DLL_PROCESS_ATTACH) {
        DisableThreadLibraryCalls(hModule);
        if(GetWindowProperties(&g_WindowInfo) == TRUE) {
            g_WindowInfo.hAddedMenu = AddNewMenu(&g_WindowInfo, L"Test");
            AddNewMenuItem(&g_WindowInfo, g_WindowInfo.hAddedMenu, L"Hello",
MENUITEM_ID);
            g_WindowInfo.PrevWndProc = SetWindowLongPtr(g_WindowInfo.hWnd,
GWLP_WNDPROC, (LONG_PTR)SublassWndProc);
            }
        }
    return TRUE;
}
```

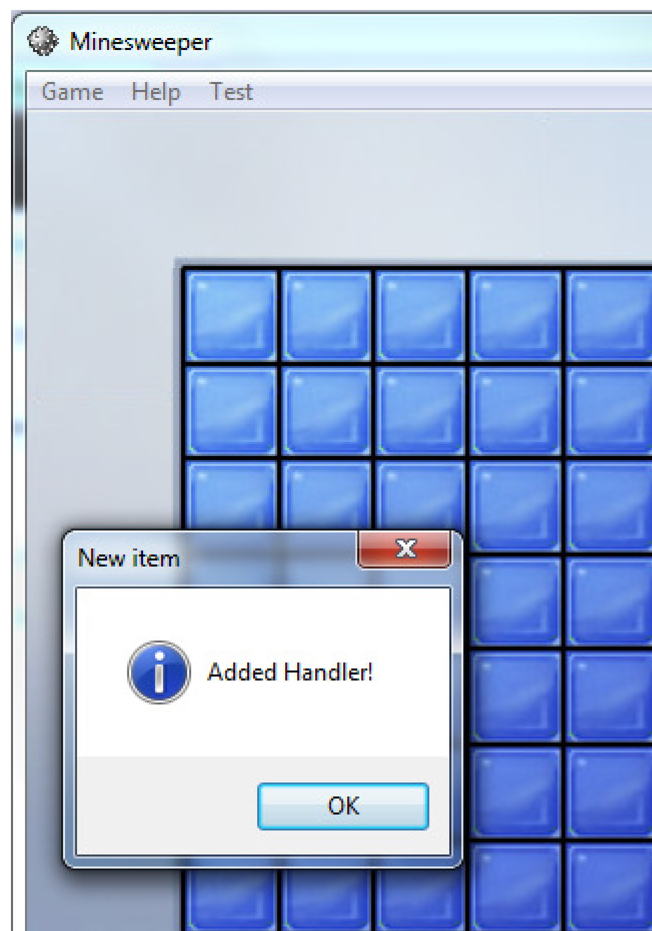
The success of the entire technique relies on the fact that GetMenu returns a valid handle to the menu. If this does not happen (the menu bar is actually not a standard menu), then the result is that nothing will happen. It is still possible to append menus/menu items in the case that the menu is not a standard menu. However, this involves reversing the application to see how menus are implemented and handled, or finding the documentation for the graphics API that is being used if it is available. What the code above does is find the window corresponding to the process

identifier that this DLL is injected to or loaded from. Once the `HWND` is found, it is used to get the handle to the menu with

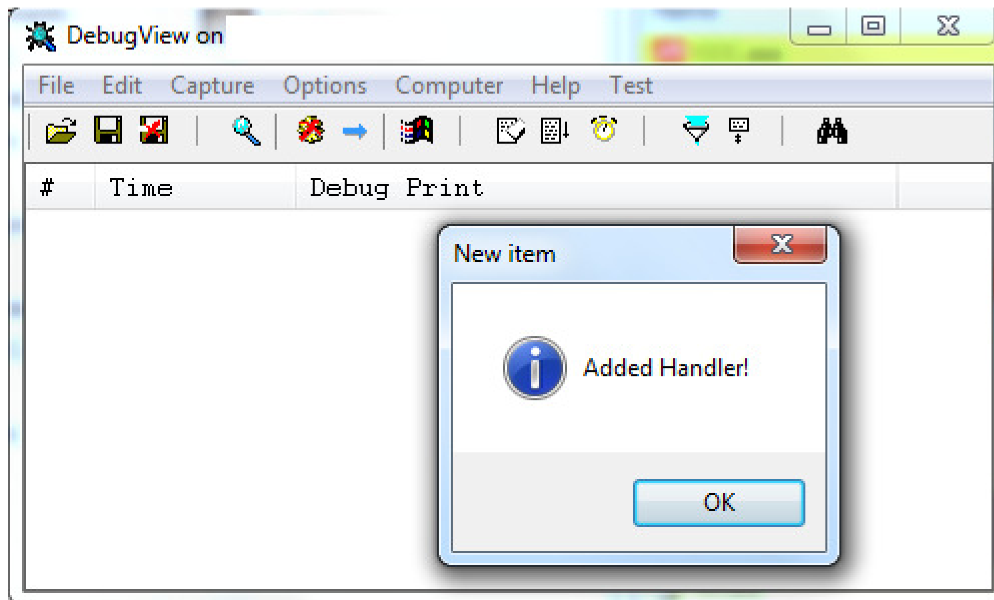
```
windowInfo->hMenuBar = GetMenu(g_WindowInfo.hWnd);
```

This handle is used to append the new menu to the menu bar (with AppendMenu). After that, any additional menu items can be appended to that added menu. `SetWindowLongPtr` is used to set a new window procedure, with the old one being stored to be called later. The handler for the menu items can be implemented in this callback like normal, with control being passed to the original window procedure at the end. One good thing about this technique is that is it done completely through the Windows API, ensuring 32-bit and 64-bit compatibility, ignoring minor details like using `SetWindowLongPtr` instead of `SetWindowLong`. If something like this was to be done through API hooks on the window procedure of the application, then there would be a hassle of finding a 32/64-bit compatible hooking library. This code was (re)tested on Windows 7 64-bit. Screenshots for 64/32-bit test applications are shown below — notice the added "Test" menu.

64-bit Minesweeper application:



32-bit DebugView application:



Source file can be found here
A downloadable PDF of this post can be found here