Desktop Development » Shell and IE programming » IE / Explorer plug-ins

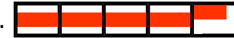
VC7, VC7.1, VC8.0Win2K, WinXP, Win2003, ATL, WTL, STL, Dev

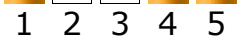
Popup Blocker

By **John Osborn**

Version 3 of the BHO Popup Blocker written in ATL

Posted:	**5 Jun 2003**
Updated:	**7 Jul 2003**
Views:	**252,529**
Bookmarked:	**135 times**

44 votes for this article. 

Popularity: 7.89 Rating: **4.80** out of 5 

Download source files - 104 Kb

Download setup.msi - 1126 Kb

Introduction

My last article demonstrated how to create a basic Popup Blocker using a BHO coded in ATL. This month a number of new features have been added that makes Popup Blocker more competitive with commercial applications for which you actually have to pay money:

- The sound and hotkeys are now user selectable.
- There is an optional visual indicator that a popup window has been blocked.
- A white list has been added to allow popups for an entire website or domain.
- A session history of websites where popups were blocked has been added.
- There is an easy way to set pertinent IE settings to block videos and animations, and to disable the script debugger.
- A way to disable Macromedia Flash animations has been added.
- You no longer have to hold down the hotkey when clicking a link that opens a new window.

Several new ATL classes, DHTML, the DOM and event model will be discussed. Also, the Popup Blocker menu is added to the IE Tools menu, and WTL is used to create a tabbed options dialog. There is a lot of material so lets get started.

Creating the Options Dialog

The first thing we need is an options dialog, not only to make it easier for the user, but also because later we will want to launch this dialog from the IE Tools menu. We could create this dialog using the ATL dialog classes, or even straight Win32 if you are a glutton for punishment, but I really enjoy working with WTL so I decided to go that route. Besides, I also wanted to eventually use some other WTL classes like `CHyperLink` and some convenient classes for common dialog controls. By the way, WTL is really cool because it is so simple, small and light-weight, and you don't have to lug around any huge DLLs. It's not yet officially supported by Microsoft, but who cares when, like ATL, you can often fix it yourself if it's broken (as we will also demonstrate). I use it instead of MFC whenever I can.

That being said, right away we run into a small problem with WTL and attributed code, which is that WTL expects a `_Module` variable that was part of ATL 3. When using ATL 7 and attributed code, that variable is nowhere to be found. The work-around is to define our own `_Module` variable in *stdafx.h*:

```
#define _Module (*_pModule)
#include <atlmisc.h>
#include <atlctl.h>
...
#undef _Module
```

If Microsoft ever gets around to officially supporting WTL, little things like that should go away.

One problem with the Popup Blocker version 1, is that it is accessible exclusively from the IE context menu. This is not always acceptable because there may be conflicts with other programs that use the context menu, or an embedded object (like Macromedia Flash) with its own context menu can occupy the entire web page. So clearly we need another way to control Popup Blocker if the context menu is turned off or unavailable. I noticed that some other applications place a shell icon on the tray, but I wanted to avoid that because the tray is often over-used and over-crowded. I decided to add Popup Blocker to the IE Tools menu because it is discrete, yet always accessible -- even without a mouse. Each menu item you add to the IE menu bar must be handled by a separate class, so, in order to avoid the extra work, I wanted to add only one menu item that pulled up the options dialog and have ALL options on that dialog. Looking at our original menu, the options dialog would therefore have to handle:

- Enable
- Play sound
- Visit Osborn Technologies
- About

This can be easily handled using the `WTL::CPropertySheet`. We can create a `WTL::CPropertyPage` for options like 'Enable' and 'Play sound', and create another page for 'About'. The class definition for a bare-bones property page is as follows:

```
class CYourPage :
    public CPropertyPageImpl<CYourPage>
{
public:
    enum { IDD = IDD_PROPPAGE_YOUR_RESOURCE_ID };

    BEGIN_MSG_MAP(CYourPage)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        CHAIN_MSG_MAP(CPropertyPageImpl<CYourPage>)
    END_MSG_MAP()

    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&)
    {
        return TRUE;
    }
};
```

Create one or more property pages and add them to the `CPropertySheet` like so:

```
void CPub::EditOptions(UINT nStartPage /*= 0*/)
{
    CPropertySheet            sht(IDS_CONFIGDLG_TITLE, nStartPage);
    CConfigOptionsPage        pageOptions;
    CConfigAdvancedPage        pageAdvanced;
    CConfigWhiteListPage    pageWhiteList;
    CConfigHistoryPage        pageHistory;
    CConfigAboutPage        pageAbout;

    // Initialize the variables on your property pages here
    ...

    // Initialize the property sheet
    sht.m_psh.dwFlags |= PSH_NOAPPLYNOW;
    sht.AddPage(pageOptions);
    sht.AddPage(pageAdvanced);
    sht.AddPage(pageWhiteList);
    sht.AddPage(pageHistory);
    sht.AddPage(pageAbout);

    if (sht.DoModal() == IDOK)
    {
        // Retrieve the new values from the property pages here.
    }
}
```

If you create the property sheet and launch it from the BHO, it will always appear in the upper left corner of the browser. This looks a little odd. It would be better to at least center it over the browser window. To do this you can call `GetPropertySheet().CenterWindow()` from `CPropertyPage::OnInitDialog`. This works really swell if you always use the same start page, but lets say you want to show different pages at startup. You could call `CenterWindow()` from each page, but if the user moves the dialog, then clicks another tab, the dialog will jump back to the center of the browser window. My solution was to create my own property sheet class that saves the start page in a global variable that is accessible from the property pages. Then I compare the start page to the active page at the beginning of `OnInitDialog` and center the window only if they are the same.

```
class CConfigPropertySheet :
    public CPropertySheetImpl<CConfigPropertySheet>
{
public:
    CConfigPropertySheet(UINT nTitleID = NULL,
    UINT uStartPage = 0, HWND hWndParent = NULL)
    : CPropertySheetImpl<CConfigPropertySheet>(nTitleID,
        uStartPage, hWndParent)
    {
        g_nStartPage = uStartPage;
    }

    BEGIN_MSG_MAP(CConfigPropertySheet)
        MESSAGE_HANDLER(WM_COMMAND,
            CPropertySheetImpl<CConfigPropertySheet>::OnCommand)
    END_MSG_MAP()
};
```

```
LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&)
{
    if (g_nStartPage == GetPropertySheet().GetActiveIndex())
        GetPropertySheet().CenterWindow();
    ...
}
```

One handy WTL class is `CWinDataExchange` which is an analog of the MFC data exchange class. Add check boxes for 'Enable' and 'Play sound' to your Options property page resource, then derive the property page class from `CWinDataExchange` and add a DDX map like this:

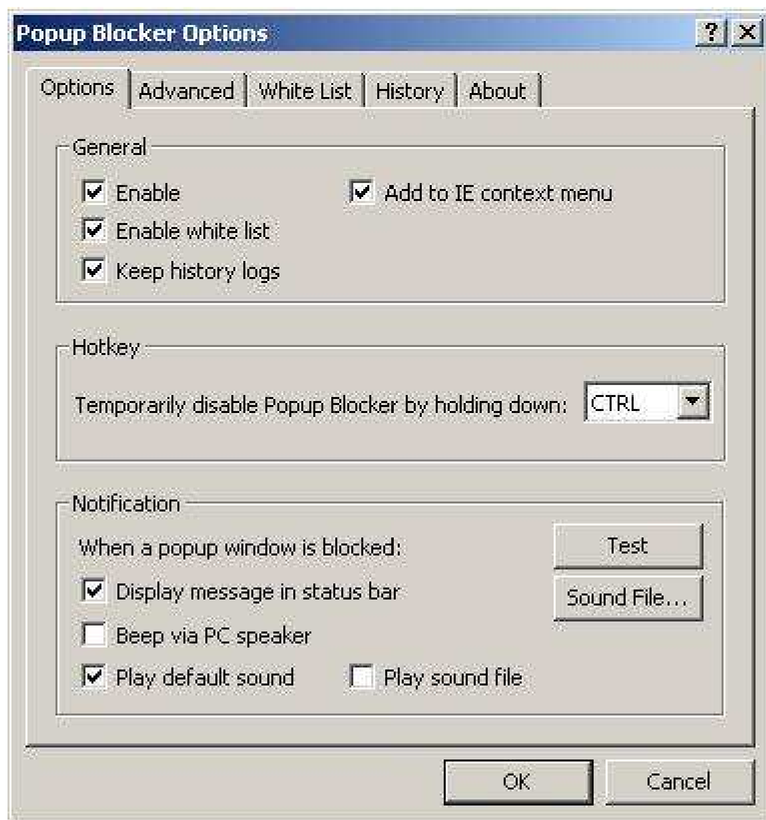
```
class CConfigOptionsPage :
    public CPropertyPageImpl<CCONFIGOPTIONSPAGE>,
    public CWinDataExchange<CCONFIGOPTIONSPAGE>
{
public:

    BEGIN_DDX_MAP(CConfigOptionsPage)
        DDX_CHECK(IDC_ENABLE, m_nEnable)
        DDX_CHECK(IDC_PLAY_DEFAULT_SOUND, m_nPlayDefaultSound)
        ...
    END_DDX_MAP()
}
```

Just like with MFC you call `DoDataExchange(FALSE)` to initialize the controls and `DoDataExchange(TRUE)` to retrieve values from the controls. You can find the definition of `CWinDataExchange` and a list of all the `DDX_*` macros in *atlddx.h*.

Another set of useful WTL classes can be found in *atlctrls.h*. These are wrapper classes for standard and common Windows controls such as `CEdit`, `CComboBox`, `CListViewCtrl`, `CTreeViewCtrl`, etc. Using these classes could not be easier. Attach to your dialog control with the `DDX_CONTROL` macro or simply assign a window handle as I do here:

```
class CConfigOptionsPage :
    public CPropertyPageImpl<CConfigOptionsPage>,
    public CWinDataExchange<CConfigOptionsPage>
{
    // Add a combobox with ID == IDC_HOTKEY to the Options page.
    // Use the WTL CComboBox class to interact with the control.
    CComboBox    m_cbHotkey;

public:
    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&)
    {
        if (g_nStartPage == GetPropertySheet().GetActiveIndex())
            GetPropertySheet().CenterWindow();

        // Attach to the WTL::CComboBox class
        m_cbHotkey = GetDlgItem(IDC_HOTKEY);
        ATLASSERT(m_cbHotkey.IsWindow());

        CString s;
        s.LoadString(IDS_CTRL);
        m_cbHotkey.InsertString(-1, s);    // VK_CONTROL
        s.LoadString(IDS_SHIFT);
        m_cbHotkey.InsertString(-1, s);    // VK_SHIFT
        s.LoadString(IDS_DISABLED);
        m_cbHotkey.InsertString(-1, s);

        if (m_cbHotkey.GetCount() > 0)
        {
            if (m_nHotkey == VK_CONTROL)
                m_cbHotkey.SetCurSel(0);
            else if (m_nHotkey == VK_SHIFT)
                m_cbHotkey.SetCurSel(1);
            else
                m_cbHotkey.SetCurSel(2);
        }

        return 1;  // Let the system set the focus
    }
}
```

My final Options page looks like this:



For the About page, I pretty much did a copy/paste from the original About dialog. However, I did take advantage of the `WTL::CHyperLink` class to add links to my website and email, allowing me to do away with the Visit Osborn Tech context menu item. To the About property page I added static controls for the web link and the E-mail link, then used the `CHyperLink` class to subclass them:

```
class CConfigAboutPage :
    public CPropertyPageImpl<CConfigAboutPage>
{
    CHyperLink m_weblink;
    CHyperLink m_emaillink;

public:
    ...
    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&)
    {
        ...

        m_weblink.SetLabel(_T("www.osborntech.com"));
        m_weblink.SetHyperLink(_T("http://www.osborntech.com"));
        m_weblink.SubclassWindow(GetDlgItem(IDC_WEBLINK));

        m_emaillink.SetLabel(_T("info@osborntech.com"));
        m_emaillink.SetHyperLink(_T("mailto:info@osborntech.com"));
        m_emaillink.SubclassWindow(GetDlgItem(IDC_EMAILLINK));
        return TRUE;
    }
};
```

Here is the final About page:

We can now launch the new Options Dialog from the BHO by tweaking the `CPub::ShowContextMenu` method. We determine which property page is initially displayed by passing the desired startup page to `EditOptions()`:

```
...
else if (nCmd == ID_CTXMENU_OPTIONS)
{
    EditOptions(0);
}
else if (nCmd == ID_CTXMENU_WHITELIST)
{
    EditOptions(2);
}
else if (nCmd == ID_CTXMENU_HISTORY)
{
    EditOptions(3);
}
else if (nCmd == ID_CTXMENU_ABOUT)
{
    EditOptions(4);
}
```

We can prevent more than one Options Dialog from being displayed simultaneously by using a mutex.

```
void CPub::EditOptions(UINT nStartPage /*= 0*/)
{
    // The mutex prevents simultaneous access by
    // multiple instances of the BHO.
    if (WAIT_OBJECT_0 != WaitForSingleObject(m_hMutexOptions, 0))
    {
        AtlMessageBox(NULL, IDS_MSG_OPTIONSDLG, IDS_PROJNAME);
        return;
    }

    ...

    ReleaseMutex(m_hMutexOptions);
}
```

This is a good time to explain something about BHOs in general. The BHO will attach itself to each instance of

IE. Each IE instance may be in a separate process (and a DLL has to be loaded by each process) or several instances of IE may run in the same process. This is very important to understand when dealing with global variables. When several IE instances happen to run in the same process, they load the BHO DLL just once, though each still creates its own instance of a BHO object. Global variables are shared between all BHO instances running in the same process, so you have to use some kind of synchronization when modifying these variables. If you wish to test this, you can launch IE in a separate process each time you launch IE from the desktop, or you can launch another instance of IE in the same process by right-clicking on a link and selecting 'Open in New Window'.

By the way, note the call to `AtlMessageBox` in `EditOptions()`. This is an ATL function that displays a normal message box, but has the added benefit of taking string IDs instead of quoted strings. By placing the strings in the resource file it will be easier to internationalize Popup Blocker at a later date (maybe version 3).

Modifying the IE Tools menu

To make Popup Blocker accessible with the keyboard I decided to add it to the IE Tools menu. This process is very clearly explained in the MSDN article entitled 'Adding Menu Items'. There are three types of things the new menu item can run: scripts, executable files, and COM objects. For Popup Blocker we are going to run a COM object. The *General Steps* described in the MSDN article show the required registry modifications. I have encapsulated those registry changes in the *PopupBlocker.rgs* file.

```
HKLM
{
    NoRemove SOFTWARE
    {
        NoRemove Microsoft
        {
            NoRemove 'Internet Explorer'
            {
                NoRemove Extensions
                {
                    ForceRemove {0D555BC6-E331-48b3-A60E-AAC0DF79438A} =
                        s 'PopupBlocker Menu'
                    {
                        val MenuCustomize = s 'Tools'
                        val CLSID =
                            s '{1FBA04EE-3024-11D2-8F1F-0000F87ABD16}'
                        val MenuText = s 'Popup Blocker'
                        val MenuStatusBar = s 'Show the Popup Blocker menu'
                        val ClsidExtension =
                            s '{93F764AC-24D1-484F-92EA-3C84E31CDF72}'
                    }
                }
            }
        }
    }
}
```

The GUID on the `ForceRemove` line can be any new GUID that you generate with Create GUID utility included in MSVC. `MenuCustomize` (optional) shows that we are adding the menu item to the Tools menu. Tools is the default. You have the choice of Tools or Help, nothing else. `MenuText` is what the user will see when pulling down the Tools menu. `MenuStatusBar` (optional) is what is displayed on the IE status bar when the user highlights our menu item. `ClsidExtension` is the GUID of the COM object we will create next. Notice that these registry settings are placed under HKLM (`HKEY_LOCAL_MACHINE`). Depending on your application you could also place them under HKCU (`HKEY_CURRENT_USER`); however, we want them under HKLM because the BHO is registered under HKLM.

The COM object should be created as an ATL Simple Object using the wizard. In the wizard be sure to select the `IObjectWithSite` option because we will need to access the running instance of IE. Once the object is created we also need to implement `IOleCommandTarget`. Your class declaration should look something like this:

```
// IToolsMenu
[
    object,
    uuid("6ED8119A-E0C0-4F30-84B6-5294CFADC3DC"),
    dual,    helpstring("IToolsMenu Interface"),
    pointer_default(unique)
]
__interface IToolsMenu : IDispatch
{
};

// CToolsMenu

[
    coclass,
    threading("apartment"),
    aggregatable("never"),
    vi_progid("OsbornTech.PopupBlocker.ToolsMenu"),
    progid("OsbornTech.PopupBlocker.ToolsMenu.1"),
    version(1.0),
    uuid("93F764AC-24D1-484F-92EA-3C84E31CDF72"),
    helpstring("ToolsMenu Class")
]
class ATL_NO_VTABLE CToolsMenu :
    public IObjectWithSiteImpl<CTOOLSMENU>,
    public IToolsMenu,
    public IOleCommandTarget
{
    ...
    //
    // IOleObjectWithSite Methods
    //
    STDMETHOD(SetSite)(IUnknown *pUnkSite)
    {
        return S_OK;
    }

    //
    // IOleCommandTarget
    //
    STDMETHOD(QueryStatus)(
        /*[in]*/ const GUID *pguidCmdGroup,
        /*[in]*/ ULONG cCmds,
        /*[in,out][size_is(cCmds)]*/ OLECMD *prgCmds,
        /*[in,out]*/ OLECMDTEXT *pCmdText)
    {
        return S_OK;
    }

    STDMETHOD(Exec)(
        /*[in]*/ const GUID *pguidCmdGroup,
        /*[in]*/ DWORD nCmdID,
        /*[in]*/ DWORD nCmdExecOpt,
        /*[in]*/ VARIANTARG *pvaIn,
        /*[in,out]*/ VARIANTARG *pvaOut)
    {
        return S_OK;
    }
    ...
}
```

Notice that the `CToolsMenu` GUID is the one written to the registry under `ClsidExtension`. At this point, if you compile Popup Blocker and run IE, you should see our new menu item inserted under the IE Tools menu. All we have to do is add some code so that it actually does something.

However, there is a rather substantial problem we have to solve first. That is how to find some way for this `CToolsMenu` object to communicate with the BHO. Both are attached to the same running instance of IE, but there is no obvious way to get them to talk to each other. My idea was to create a DOM extension with the BHO and let the `CToolsMenu` object talk to the BHO via `window.external` just as you would with a piece of script.

To accomplish this, we need to get hold of the running instance of IE under `SetSite`. This is where MSDN falls down. The MSDN article describes how to get `IShellBrowser`, it doesn't say anything about how to get `IWebBrowser2` other than it can be done. Well, thank you very much. After a little digging I found that you have to use `IServiceProvider::QueryService()` to get a pointer to the browser (see Q257717). Once we have `IWebBrowser2` it is a simple matter to drill down to `IHTMLWindow2::get_external` to get the external dispatch which we squirrel away in a member variable.

```
STDMETHOD(SetSite)(IUnknown *pUnkSite)
{
    if (!pUnkSite)
    {
        ATLTRACE(_T("SetSite(): pUnkSite is NULL\n"));
    }
    else
    {
        // Get the web browsers external dispatch interface.

        // Get the service provider from the site
        CComQIPtr<IServiceProvider><ISERVICEPROVIDER,
                        &IID_IServiceProvider> spProv(pUnkSite);
        if (spProv)
        {
            CComPtr<IServiceProvider><ISERVICEPROVIDER> spProv2;
            HRESULT hr = spProv->QueryService(SID_STopLevelBrowser,
                IID_IServiceProvider, reinterpret_cast<void **>(&spProv2));
            if (SUCCEEDED(hr) && spProv2)
            {
                CComPtr<IWebBrowser2><IWEBBROWSER2> m_spWebBrowser2;
                hr = spProv2->QueryService(SID_SWebBrowserApp,
                     IID_IWebBrowser2,
                     reinterpret_cast<void **>(&m_spWebBrowser2));
                if (SUCCEEDED(hr) && m_spWebBrowser2)
                {
                    CComPtr<IDISPATCH> spDisp;
                    hr = m_spWebBrowser2->get_Document(&spDisp);
                    if (SUCCEEDED(hr))
                    {
                        CComQIPtr<IHTMLDocument2><IHTMLDOCUMENT2,
                                      &IID_IHTMLDocument2> spHTML(spDisp);
                        if (spHTML)
                        {
                            CComPtr<IHTMLWindow2><IHTMLWINDOW2> spWindow;
                            spHTML->get_parentWindow(&spWindow);
                            if (spWindow)
                                spWindow->get_external(&m_spExtDisp);
                        }
                    }
                }
            }
        }
    }
    return S_OK;
}
```

To make the actual call to the external dispatch we need to finish implementing IOleCommandTarget. The only method we are concerned with is IOleCommandTarget::Exec() which gets called with nCmd == 0 when our menu item is selected. We haven't yet created the DOM extension (that's next), but when we do we will add a command called EditOptions which will take no parameters and return no values. To make the call, we get the dispatch ID of the EditOptions method in the BHO and then call Invoke.

```
STDMETHOD(Exec)(
    /*[in]*/ const GUID *pguidCmdGroup,
    /*[in]*/ DWORD nCmdID,
    /*[in]*/ DWORD nCmdExecOpt,
    /*[in]*/ VARIANTARG *pvaIn,
    /*[in,out]*/ VARIANTARG *pvaOut)
{
    if (nCmdID == 0)
    {
        // User selected our menu item from the IE Tools menu.
        // Call the DOM extension in the BHO.
        if (m_spExtDisp)
        {
            DISPID dispid;
            OLECHAR FAR* szMember = L"EditOptions";

            HRESULT hr = m_spExtDisp->GetIDsOfNames(IID_NULL, &szMember, 1,
                LOCALE_SYSTEM_DEFAULT, &dispid);
            if (SUCCEEDED(hr))
            {
                DISPPARAMS DispParams;
                VARIANT varResult;
                EXCEPINFO  ExcepInfo;
                UINT uArgErr;

                VariantClear(&varResult);
                memset(&DispParams, 0, sizeof(DISPPARAMS));
                memset(&ExcepInfo, 0, sizeof(EXCEPINFO));

                m_spExtDisp->Invoke(dispid, IID_NULL,
                    LOCALE_SYSTEM_DEFAULT, 0,
                    &DispParams, &varResult, &ExcepInfo, &uArgErr);
            }
        }
    }
    return S_OK;
}
```

Of course, none of this will work until we create the DOM extension with an EditOptions method, which is done in the BHO.

Creating the DOM Extension

The DOM extension gives us the ability to add new commands to IE that may be called from script or from anything else (like CToolsMenu) which can get a pointer to the document window. When we call IHTMLWindow2::get_external() in CToolsMenu::SetSite(), IE makes a call to the CPub::GetExternal() method in the BHO to see if we want to supply an external dispatch interface. In version 1, we just return S_FALSE from GetExternal(), but in version 2 we will return the dispatch interface of our DOM extension. Create the DOM extension by using the wizard to add another ATL Simple Object and add a method to this object called EditOptions.

```
// IPubDomExtender
[
    object,
    uuid("F3777260-7308-464A-BAA2-CC492C0CE7D2"),
    dual,    helpstring("IPubDomExtender Interface"),
    pointer_default(unique)
]
__interface IPubDomExtender : IDispatch
{
    [id(1), helpstring("method EditOptions")] HRESULT EditOptions();
};

// CPubDomExtender

[
    coclass,
    threading("apartment"),
    aggregatable("never"),
    vi_progid("OsbornTech.PubDomExtender"),
    progid("OsbornTech.PubDomExtender.1"),
    version(1.0),
    uuid("83EC9074-6CBA-43E8-B7E0-6A3809C4A958"),
    helpstring("OsbornTech PubDomExtender Class")
]
class ATL_NO_VTABLE CPubDomExtender :
    public IPubDomExtender
{
public:
    CPubDomExtender() :
        m_pPub(NULL)
    {
    }


    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
    STDMETHOD(EditOptions)()
    {
        if (m_pPub)
            m_pPub->EditOptions();
        return S_OK;
    }

public:
    CPub* m_pPub;
};
```

In `CPub::Invoke()` add the following code under `DISPID_DOCUMENTCOMPLETE`:

```
// Create a CPubDomExtender object that will be used by
// window.external
CComObject<CPUBDOMEXTENDER>* pDomExt;
hr = CComObject<CPUBDOMEXTENDER>::CreateInstance (&pDomExt);
ATLASSERT(SUCCEEDED(hr));
if (SUCCEEDED(hr))
{
    CComQIPtr spDisp = pDomExt;
    m_spExtDispatch = spDisp;
    pDomExt->m_pPub = this;
}
```

Then change the `CPub::GetExternal()` method to pass back this `IDispatch` pointer.

```
STDMETHOD(GetExternal)(IDispatch** ppDispatch)
{
    // IE will call this when the user asks for the external dispatch
    // of the window, either via script (window.external) or via
    // the Tools menu.
    if (m_spExtDispatch)
        return m_spExtDispatch.CopyTo(ppDispatch);

    if (m_spDefaultDocHostUIHandler)
        return m_spDefaultDocHostUIHandler->GetExternal(ppDispatch);
    return S_FALSE;
}
```

Once you get all this to compile, the Options Dialog should be displayed when Popup Blocker is selected from the IE Tools menu. You can set a break point in `CToolsMenu::SetSite()` and a break point in `CPub::GetExternal()` to see the order of events when the menu item is selected.

Session history

The purpose of the session history is to keep track of all sites where popups have been blocked during the current session. The only tricky part involved here is sharing the site list between all instances of the BHO. Like everything else, there are a number of ways to do this. In this case, I chose to use a memory mapped file as a way of sharing data between processes.

If you don't know what memory mapped files are, you can get a quick over view from 'About File Mapping' in MSDN. Here is an even quicker overview: once you map a file you get a pointer to the data, that you can treat just like any other `char*`. You can do `strcpy`, `strcmp`, etc. as if you were working with a normal C-string. The sharing part comes in where you can give this map a name. Then from another process you can open the map by supplying the name, and *presto*, you have a string pointer to the same data. I left out some stuff, but that's the gist of it.

Since I am keeping the site list only for the current session, I don't want to keep the file. When the last BHO terminates, the memory mapped file should get deleted. ATL supplies a `CAtlTemporaryFile` class that fits the bill. A `CAtlTemporaryFile` automatically deletes itself when it is closed.

ATL comes with another nifty class called `CAtlFileMapping` which is a thin wrapper around the Win32 file mapping functions. This class hides some of the details of creating the file mapping, and supplies a method for opening a named file mapping. Of course, this class would be even niftier if it didn't have one glaring omission: while you can open a named file mapping there is no way to create one using this class. Out of the box, the `CAtlFileMapping::MapFile()` method does not take a name parameter. So let's take the bull by the horns and create a new "fixed" class. What I did was copy the whole *atlfile.h* and create *atlfile_fixed.h* which contains the same class, except I changed the namespace from `ATL` to `PUB`. Then I changed the parameter list of the `MapFile` method like so:

```
// FIXED:  Added the pszName parameter
HRESULT MapFile(
    HANDLE hFile,
    SIZE_T nMappingSize = 0,
    ULONGLONG nOffset = 0,
    DWORD dwMappingProtection = PAGE_READONLY,
    DWORD dwViewDesiredAccess = FILE_MAP_READ,
    LPCTSTR pszName = NULL) throw()
```

Down in the guts of `MapFile` I changed the call to `CreateFileMapping` from:

```
m_hMapping = ::CreateFileMapping(hFile, NULL, dwMappingProtection,
    liFileSize.HighPart, liFileSize.LowPart, 0);
```

to

```
m_hMapping = ::CreateFileMapping(hFile, NULL, dwMappingProtection,
    liFileSize.HighPart, liFileSize.LowPart, pszName);
```

There, all fixed. Now to use our class instead of the normal ATL class you simply specify the PUB namespace like this:

```
PUB::CAtlFileMapping<TCHAR> g_filemapBlockedList;
```

In DllMain under DLL_PROCESS_ATTACH I first try to open an existing file mapping with the name _T("PubBlockedSiteList"). If that fails, I create a new mapping on a temporary file.

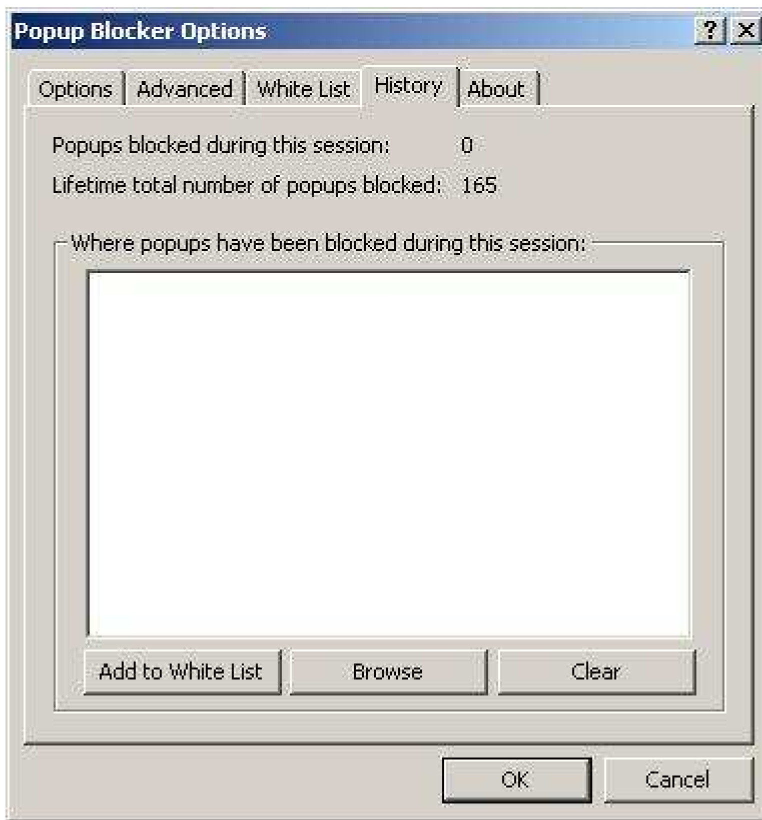
```
// File mapping is another way of sharing data between processes.
// Popup Blocker keeps a list of sites where popups have been blocked
// during the current session.  This list is shared between all
// instances of the BHO.
HRESULT hr = E_FAIL;
try
{
    hr = g_filemapBlockedList.OpenMapping(_T("PubBlockedSiteList"),
                                                             SHMEMSIZE);
    if (FAILED(hr))
    {
        CAtlTemporaryFile f;
        hr = f.Create(NULL, GENERIC_READ | GENERIC_WRITE);
        if (SUCCEEDED(hr))
        {
            f.SetSize(SHMEMSIZE);
            hr = g_filemapBlockedList.MapFile(f, SHMEMSIZE,
                 0, PAGE_READWRITE,
                 FILE_MAP_ALL_ACCESS, _T("PubBlockedSiteList"));
        }
    }
}
catch(...)
{
    hr = E_FAIL;
}
```

Normally the temporary file would be deleted when it went out of scope, but it will remain open as long as there is an open mapping -- even if you explicitly close it! The file is eventually unmapped in DllMain under DLL_PROCESS_DETACH, which will close the mapping. The temporary file, however, will remain open as long as another BHO still has the file mapped. The data in the mapped file is simply a string consisting of URLs delimited by newline characters. I created an UpdateStats method that appends a new URL to the file and increments counters that keep track of the number of popups blocked:

```
void CPub::UpdateStats(BSTR bsUrl)
{
    InterlockedIncrement((LPLONG)&g_dwBlockedSession);
    InterlockedIncrement((LPLONG)&g_dwBlockedTotal);

    if (WAIT_OBJECT_0 != WaitForSingleObject(m_hMutex, 1000))
        return;

    try
    {
        CRegKey rk;
        DWORD dwErr = rk.Open(HKEY_CURRENT_USER, g_sCoRegKey + g_sPubRegKey);
        if (dwErr == ERROR_SUCCESS)
            rk.SetDWORDValue(_T("BlockedTotal"), g_dwBlockedTotal);

        if (g_bEnableHistory)
        {
            CString sUrl = CW2T(bsUrl);
            sUrl += _T("\n");

            TCHAR* pchData = (TCHAR*)g_filemapBlockedList.GetData();
            if (pchData && _tcsstr(pchData, sUrl) == NULL)
            {
                // Not already added
                int nBufLen = (int)g_filemapBlockedList.GetMappingSize();
                int nDataLen = lstrlen(pchData);
                int nUrlLen = sUrl.GetLength();

                if (nDataLen + nUrlLen < nBufLen)
                {
                    nDataLen = lstrlen(pchData);
                    memcpy(pchData + nDataLen, (LPCTSTR)sUrl, nUrlLen);
                }
            }
        }
    }
    catch(...)
    {
    }

    ReleaseMutex(m_hMutex);
}
```

Notice that this code is protected by a mutex that prevents multiple BHOs from changing the global data simultaneously. I could have also placed the counters inside the mutex protected code, but I wanted to use `InterlockedIncrement` just to show another way of doing it. `InterlockedIncrement` would normally just work for threads within the same process, but since these variables are within shared memory (see `#pragma data_seg(".SHARED")`), it works in this case too.

To show the contents of the mapped file, I added the history page to the Options Dialog.



In `OnInitDialog` of the history property page, I parse the list of URLs and stuff them into the listbox.

```
LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&)
{
    if (g_nStartPage == GetPropertySheet().GetActiveIndex())
        GetPropertySheet().CenterWindow();

    CString s;
    s.Format(_T("%ld"), m_dwBlockedSession);
    SetDlgItemText(IDC_BLOCKED_SESSION, s);

    s.Format(_T("%ld"), m_dwBlockedTotal);
    SetDlgItemText(IDC_BLOCKED_TOTAL, s);

    m_lbHistory = GetDlgItem(IDC_HISTORY);

    ATLTRY(s = (TCHAR*)g_filemapBlockedList.GetData());
    if (!s.IsEmpty())
    {
        int nPos = 0;
        CString sUrl = s.Tokenize(_T("\n"), nPos);
        while (sUrl != "")
        {
            CUrl url;
            if (url.CrackUrl(sUrl, ATL_URL_DECODE))
            {
                if (url.GetScheme() == ATL_URL_SCHEME_HTTP ||
                    url.GetScheme() == ATL_URL_SCHEME_HTTPS)
                {
                    // Get the domain without the scheme
                    CString sDomain = url.GetHostName();
                    sDomain += url.GetUrlPath();
                        // Strip off the file name
                    int idx = sDomain.ReverseFind(_T('/'));
                    if (idx > 0)
                        sDomain = sDomain.Left(idx);
                    sDomain.MakeLower();
                    sDomain.Trim();

                    if (!sDomain.IsEmpty())
                        m_lbHistory.InsertString(0, sDomain);
                }
            }
            sUrl = s.Tokenize(_T("\n"), nPos);
        }
    }

    if (m_lbHistory.GetCount() > 0)
        m_lbHistory.SetCurSel(0);

    return TRUE;
}
```

Here you will note the use of another cool ATL class called `CUrl` which is really useful for parsing URLs. In this case I use it to check that the URL is correctly formed (otherwise `CrackUrl` fails) and that the scheme is supported (either HTTP or HTTPS). Then I strip the scheme and path before inserting it into the combobox.

The white list

The white list is one of the more interesting parts of version 2. The purpose of the white list is to keep a set of URLs for which popups should be allowed. Again, this is data that must be shared between all instances of the BHO and ideally there should be only one copy in memory. While this could be done any number of ways, I chose to use `IPersistFile` through a simple file moniker mostly because it was suggested by someone commenting on version 1 (that would be you, Heath).

First a (very) little background on monikers. The white list is stored in a disk file 'object'. The disk file object has a display name something like *C:\My Documents\WhiteList.pub*. The display name tells us something about the location of the file object, but nothing about how to access the data it contains. This is where monikers come in. You can think of a moniker as an intelligent name (indeed moniker means name) that encapsulates the intelligence for accessing the object it represents. So really a moniker is an object used to name and access another object. A moniker can represent many different things -- not just files -- but I want to keep this simple so I won't go into the details here. You can refer to 'Inside OLE' by Kraig Brockschmidt for just about everything you would ever want to know about monikers. In our case we are just interested in a

simple file moniker that represents our white list disk file.

Since we are going to use a file moniker, we need to create an object that implements `IPersistFile` and that knows how to load our white list file. First we add a new ATL Project to our solution, and to that project we add a new ATL Simple Object and change the object class so that it implements `IPersistFile`. We end up with something like this:

```
// IWhiteList
[
    object,
    uuid("A40E7A73-615A-450B-A302-2A316E9C9892"),
    dual, helpstring("IWhiteList Interface"),
    pointer_default(unique)
]
__interface IWhiteList : IDispatch
{
};



// CWhiteList

[
    coclass,
    threading("apartment"),
    support_error_info("IWhiteList"),
    aggregatable("never"),
    vi_progid("OsbornTech.PopupBlocker.WhiteList"),
    progid("OsbornTech.PopupBlocker.WhiteList.1"),
    version(1.0),
    uuid("52F0D70A-DBE1-4D79-AA46-1AD947CB6BCC"),
    helpstring("PopupBlocker WhiteList Class")
]
class ATL_NO_VTABLE CWhiteList :
    public IWhiteList,
    public IPersistFile
{
public:
    CWhiteList() :
    {
    }

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:

    // IPersistFile
    STDMETHOD(GetClassID)(LPCLSID)       { return E_NOTIMPL; }
    STDMETHOD(IsDirty)()                 { return E_NOTIMPL; }
    STDMETHOD(Save)(LPCOLESTR, BOOL)     { return E_NOTIMPL; }
    STDMETHOD(SaveCompleted)(LPCOLESTR) { return E_NOTIMPL; }
    STDMETHOD(GetCurFile)(LPOLESTR*)     { return E_NOTIMPL; }
    STDMETHOD(Load)(LPCOLESTR, DWORD);
};
```

Notice that the only method that we implement is `IPersistFile::Load()`, which will get called with the display name of the white list file when COM loads our object. By now you should be asking yourself, how is COM going to load our object? How is it that COM is able to associate our white list file with this object we just created? COM does this through a process known as binding. When we call `CoGetObject` from Popup Blocker, the white list file name is converted into a moniker that identifies the white list file and then binds to the `CWhiteList` object identified by the moniker. The moniker figures out that the white list is a file and calls `GetClassFile`. `GetClassFile` attempts to match various bits in the file (the white list file header in our case) against a pattern in the registry which also contains an entry for the GUID of our `CWhiteList`

object.

For this to work, we need to create a white list file that contains a header with some unique bits in it. I created a file called *PubWhiteList.pwl* contains nothing but `#jpo` in the first line (the header). As white listed URLs are added, they will be listed in plain text on subsequent lines, one URL per line. Then we need to add a registry entry that links those unique bits to the GUID of the `CWhiteList` object. I used the *PubWhiteList.rgs* to do this.

```
HKCR
{
    NoRemove FileType
    {
        ForceRemove {52F0D70A-DBE1-4D79-AA46-1AD947CB6BCC} =
            s 'PopupBlocker WhiteList'
        {
            0 = s '0,4, FFFFFFFF, 236A706F'
        }
    }
}
```

The pattern in the registry is identified as:

```
regdb key = offset, cb, mask, value
```

In other words, if the first four bytes of the file are `0x23 0x6A 0x70 0x6F` (`#jpo`), then the file should be associated with the object identified by the GUID `{52F0D70A-DBE1-4D79-AA46-1AD947CB6BCC}` (our `CWhiteList` object). Once the moniker makes this association, it calls `CoCreateInstance` with the GUID, requests the `IPersistFile` interface, and calls `IPersistFile::Load`, passing the name of the white list file. The moniker then calls `IPersistFile::QueryInterface(IID_IWhiteList)` and passes the pointer back to Popup Blocker as the out-parameter of `CoGetObject`. Having completed its task, the moniker then gets completely out of the way. This is the how `CoGetObject` is called in Popup Blocker:

```
::CoGetObject(m_bsWhiteListPath, 0, __uuidof(IWhiteList),
                                      (void**)&g_spWhiteList);
```

`m_bsWhiteListPath` is the display name of the white list file, `IWhiteList` is the interface we are requesting, and `g_spWhiteList` is object returned by the moniker.

But we are not yet done with the white list object. One of our requirements is that there be only one copy of the data in memory. As it stands, one white list object will be instantiated for each BHO. What we want is ONE white list object that all BHO instances use. To accomplish this, we register the white list object in the Running Object Table (ROT).

```
STDMETHODIMP CWhiteList::Load(LPCOLESTR pszFileName, DWORD grfMode)
{
    ATLTRACE(_T("CWhiteList::Load\n"));
    m_sFileName = pszFileName;

    // Load the white list file data
    HRESULT hr = LoadListFromFile();
    if (SUCCEEDED(hr))
    {
        // Register ourselves in the ROT to ensure only one instance // of
        this class per file is running at a time.
        CComPtr<IRUNNINGOBJECTTABLE> pROT;
        HRESULT hr = GetRunningObjectTable(0, &pROT);
        if (SUCCEEDED(hr))
        {
            CComPtr<IMONIKER> pmk;
            hr = CreateFileMoniker(pszFileName, &pmk);
            if (SUCCEEDED(hr))
                pROT->Register(0, GetUnknown(), pmk, &m_dwCookie);
        }
    }

    return S_OK;
}
```

Here we simply create a File Moniker, which is a standard moniker type, based on the white list file name and register the moniker in the ROT. Then the next time an instance of Popup Blocker calls `CoGetObject`, before creating a new `CWhiteList` object, COM first checks the ROT to see if one is already running and, if so, returns a pointer to that one instead. Pretty slick. You can verify that the white list File Moniker is registered in the ROT using the IROTVIEW utility that comes with MSVC.

What remains to be done is to add methods to the `CWhiteList` object that allow access to the data. The most interesting method is the `IWhiteList::Find()`, which takes a URL and returns `TRUE` if it was found in the white list.

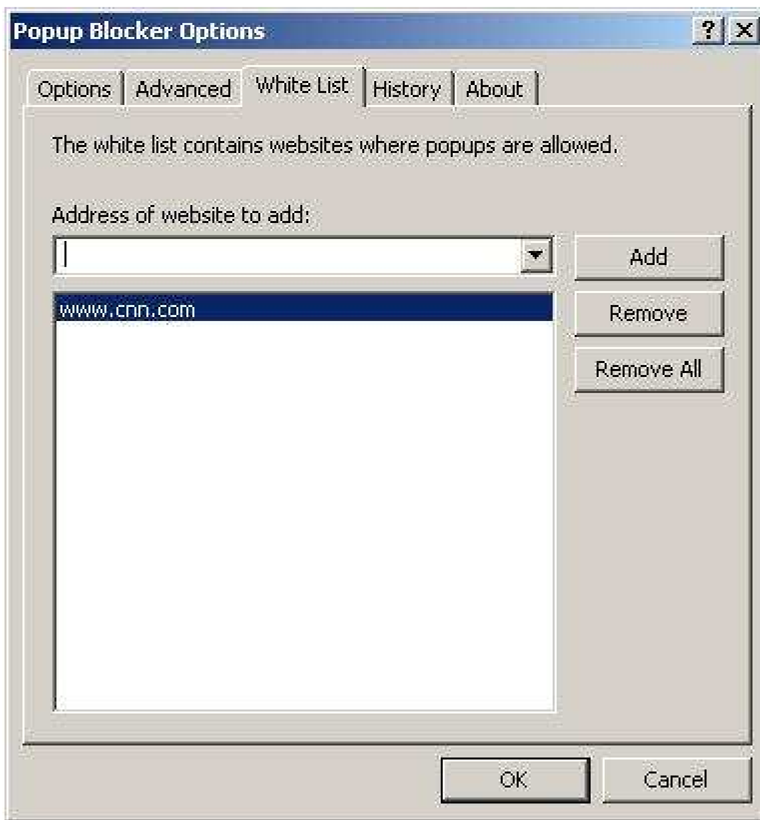
```
STDMETHODIMP CWhiteList::Find(BSTR bsUrl, VARIANT_BOOL* pbFound)
{
    USES_CONVERSION;

    *pbFound = VARIANT_FALSE;

    CString sUrl = W2T(bsUrl);
    sUrl.MakeLower();

    ATLTRACE(_T("Find: %s\n"), (LPCTSTR)sUrl);

    // We are expecting a fully formed URL like http://blah...
    // Make a half-hearted attempt at fixing the URL if it
    // doesn't look right.  Sometimes you will get something
    // like javascript:makeWin2('http://blah...').
    int i = sUrl.Find(_T("http://"));
    if (i > 0)
    {
        sUrl = sUrl.Mid(i);
    }
    else if (i < 0)
    {
        i = sUrl.Find(_T("https://"));
        if (i > 0)
            sUrl = sUrl.Mid(i);
        else if (i < 0)
            sUrl = _T("http://") + sUrl;
    }

    CUrl url;
    if (url.CrackUrl(sUrl, ATL_URL_DECODE))
    {
        CString s = url.GetHostName();
        if (lstrcmp(url.GetUrlPath(), _T("/")) != 0)
            s += url.GetUrlPath();

        std::set<CString>::const_iterator itr;

        while (1)
        {
            if ((itr = m_setDomain.find(s)) != m_setDomain.end())
            {
                *pbFound = VARIANT_TRUE;
                break;
            }

            int i = s.ReverseFind(_T('/'));
            if (i < 0)
                break;
            s = s.Left(i);
        }
    }
    return S_OK;
}
```

I store the white list in memory in a `std::set` because sets are optimized for fast lookups. The find algorithm goes like this: Given a URL like *http://some.domain.name/popups/ad1/123456.ext*, we strip off the scheme to yield *some.domain.name/sub1/sub2/123456.ext*. Then we enter a loop where perform the following lookups in order:

```
some.domain.name/sub1/sub2/123456.ext
some.domain.name/sub1/sub2
some.domain.name/sub1
some.domain.name
```

If at any point we have a successful lookup, we set found `TRUE` and break out. The search order is important because you may not want to white list the entire site (*some.domain.name*), but more specifically one page within a site (*some.domain.name/sub1*).

The White List property page on the Options Dialog looks like this:



The combobox allows the user to type in a site and, if dropped down, lists all sites enumerated by `IUrlHistoryStg2` that have a scheme of HTTP or HTTPS.

```
BOOL LoadHistory()
{
    m_cbDomain.ResetContent();

    // We need this interface for getting the history.
    // Load the correct Class and request IUrlHistoryStg2
    CComPtr<IURLHISTORYSTG2> spHistory;
    HRESULT hr = CoCreateInstance(CLSID_CUrlHistory,
                 NULL, CLSCTX_INPROC_SERVER,
                 IID_IUrlHistoryStg2,
                 reinterpret_cast<VOID **>(&spHistory));
    if (SUCCEEDED(hr))
    {
        CComPtr<IENUMSTATURL> spEnum;
        hr = spHistory->EnumUrls(&spEnum);
        if (SUCCEEDED(hr))
        {
            spEnum->Reset();

            STATURL stat = {NULL};
            stat.cbSize = sizeof(STATURL);

            ULONG nFetched = 0;
            do
            {
                hr = spEnum->Next(1, &stat, &nFetched);
                if (SUCCEEDED(hr) && nFetched)
                {
                    CString sUrl = CW2T(stat.pwcsUrl);
                    sUrl.MakeLower();

                    // Attempt to fix url
                    int idx = sUrl.Find(_T("http://"));
                    if (idx > 0)
                        sUrl = sUrl.Mid(idx);

                    CUrl url;
                    if (url.CrackUrl(sUrl, ATL_URL_DECODE))
                    {
                        if (url.GetScheme() == ATL_URL_SCHEME_HTTP ||
                            url.GetScheme() == ATL_URL_SCHEME_HTTPS)
                        {
                            // Get the domain without the scheme
                            CString s = url.GetHostName();
                            s += url.GetUrlPath();

                            // Strip off the file name
                            int idx = s.ReverseFind(_T('/'));
                            if (idx > 0)
                                s = s.Left(idx);
                            s.MakeLower();

                            // Don't add the domain to the combobox if it
                            // already exists in the combobox or whitelist.
                            if (m_cbDomain.FindStringExact(-1, s) < 0 &&
                                m_lbWhiteList.FindStringExact(-1, s) < 0)
                            {
                                m_cbDomain.AddString(s);
                            }
                        }
                    }

                    // free memory allocated for us
                    CoTaskMemFree(stat.pwcsTitle);
                    CoTaskMemFree(stat.pwcsUrl);
                    stat.pwcsTitle = NULL;
                    stat.pwcsUrl = NULL;
                }
            }
            while (nFetched);
        }
    }
    return TRUE;
}
```

The intention was to make it easier for the user to find a recently visited site, but if everyone else's history is

as long as mine, it may be too difficult/confusing to be of much use. In the next version it may be worthwhile to use a date limit and not insert in the combobox anything older than, say, three days.

Event handling

In version 2 I also demonstrate how to sink `HTMLWindowEvents2` and `HTMLDocumentEvents2`. Mostly I added these sinks to show how it is done (which is pretty much exactly like `DWebBrowserEvents2` was done in version 1), so they are not used to do very much at this point. However, I do use `HTMLDocumentEvents2::onclick` to determine when the user has clicked on a link and then to allow a new window to open.

```
case DISPID_HTMLDOCUMENTEVENTS2_ONCLICK:
    ATLTRACE(_T("HTMLDocumentEvents::onclick fired\n"));
    {
        CComPtr<IDISPATCH> spDisp;
        spDisp = pDispParams->rgvarg[0].pdispVal;
        if (spDisp)
        {
            CComQIPtr<IHTMLEVENTOBJ &IID_IHTMLEventObj,> spEventObj(spDisp);
            if (spEventObj)
            {
                CComPtr<IHTMLELEMENT>
                spElem; HRESULT  hr=
                spEventObj->get_srcElement(&spElem);
                if (SUCCEEDED(hr) &&spElem)
                {
                    CComBSTR bsTagName;
                    while (1)
                    {
                        spElem->get_tagName(&bsTagName);
                        bsTagName.ToUpper();

                        if (bsTagName == L"BODY")
                            break;    // did not click a link

                        if (bsTagName == L"A" ||
                            bsTagName == L"AREA" ||
                            bsTagName == L"INPUT" ||
                            bsTagName == L"IMG")
                        {
                            *m_pbUserClickedLink = TRUE;
                            break;
                        }

                        CComPtr<IHTMLELEMENT> spParentElem;
                        hr = spElem->get_parentElement(&spParentElem);
                        if (FAILED(hr) || !spParentElem)
                            break;
                        spElem = spParentElem;
                    }
                }
            }
        }
    }
    break;
```

When the user clicks the document, I retrieve the source element from the event object and check if it is a link (or an image, area, etc). If so, I set a flag and break out of the loop. If not, I get the source elements parent and try again, looping like this until I have checked all the nested elements. Back in `CPub::Invoke` I check this flag to determine whether I will allow a new window to open:

```
case DISPID_NEWWINDOW2:
    ATLTRACE(_T("(%ld) DISPID_NEWWINDOW2\n"), ::GetCurrentThreadId());
    {
        BOOL bSiteWhiteListed = TRUE;

        CComBSTR bsUrl;
        HRESULT hr = m_spWebBrowser2->get_LocationURL(&bsUrl);
        if (SUCCEEDED(hr) && bsUrl.Length() > 0)
            bSiteWhiteListed = IsSiteWhiteListed(bsUrl);

        if (m_bBlockNewWindow &&
            !bSiteWhiteListed &&
            !m_bUserClickedLink &&
            IsEnabled())
        {
            ATLTRACE(_T("(%ld) Blocked a popup\n"), ::GetCurrentThreadId());

            // Some links open a new window and will be blocked.
            // Give some indication to the user as to what is
            // happening; otherwise, the link will appear to be
            // broken.
            NotifiyUser();

            // Update statistics
            UpdateStats(bsUrl);

            // Set the cancel flag to block the popup
            pDispParams->rgvarg[0].pvarVal->vt = VT_BOOL;
            pDispParams->rgvarg[0].pvarVal->boolVal = VARIANT_TRUE;
        }

        m_bBlockNewWindow = TRUE;     // Reset
        m_bUserClickedLink = FALSE;     // Reset
    }
    break;
```

Checking for the mouse click goes a long way towards making Popup Blocker as unobtrusive as possible.

`CPub` also implements `IPropertyNotifySink` which catches the `READYSTATE` of the document. When I get `READYSTATE_COMPLETE` the entire document has been downloaded and processed, so at that point I hide Flash animations.

Hiding Flash animations

Popups are just one way website designers compete for our attention. Another is the ubiquitous Flash animation. While Flash technology is pretty cool and it definitely has a place on the web, it can be abused. Everybody who browses the web has seen websites that virtually squirm with Flash animation. So rather than suffer the distraction, I added an optional method to hide Flash animations. The way I do it in Popup Blocker is to wait for the document to be completely loaded and then run through the DOM and hide all the Flash animations that I find.

```
void DisableFlashMoviesHelper(IHTMLDocument2* pDoc)
{
    CComQIPtr<IHTMLDOCUMENT2, &IID_IHTMLDocument2> spHTML(pDoc);
    if (spHTML)
    {
    CComPtr<IHTMLELEMENTCOLLECTION> spAll;
    HRESULT hr = spHTML->get_all(&spAll);
    if (SUCCEEDED(hr) && spAll)
    {
        // Find all the OBJECT tags on the (maybe partially loaded) document
        CComVariant vTagName = L"OBJECT";
        vTagName.ChangeType(VT_BSTR);

            CComPtr<IDISPATCH> spTagsDisp;
        hr = spAll->tags(vTagName, &spTagsDisp);
        if (SUCCEEDED(hr) && spTagsDisp)
        {
        CComQIPtr<IHTMLELEMENTCOLLECTION,
                  &IID_IHTMLElementCollection> spTags(spTagsDisp);
        if (spTags)
        {
            long nCnt;
            hr = spTags->get_length(&nCnt);
            if (SUCCEEDED(hr))
            {
            for (long i = 0; i < nCnt; i++)
            {
                CComVariant varIdx;
                V_VT(&varIdx) = VT_I4;
                V_I4(&varIdx) = i;

                CComPtr<IDISPATCH> spTagDisp;
                hr = spTags->item(varIdx, varIdx, &spTagDisp);
                if (SUCCEEDED(hr) && spTagDisp)
                {
                CComQIPtr<IHTMLOBJECTELEMENT,
                          &IID_IHTMLObjectElement> spObject(spTagDisp);
                if (spObject)
                {
                    CComBSTR bsClassID;
                    hr = spObject->get_classid(&bsClassID);
                    if (SUCCEEDED(hr) && bsClassID)
                    {
                    bsClassID.ToUpper();
                    if (bsClassID ==
                        L"CLSID:D27CDB6E-AE6D-11CF-96B8-444553540000")
                    {
                        // This is a flash activex control.  Resize
                        // and hide it so we don't have to look at it.
                        CComQIPtr<IHTMLElement> spElem(spTagDisp);
                        if (spElem)
                        {
                        CComBSTR bs = L"hidden";
                        CComPtr<IHTMLSTYLE> spStyle;
                        hr = spElem->get_style(&spStyle);
                        if (SUCCEEDED(hr) && spStyle)
                        {
                            spStyle->put_visibility(bs);
                            spStyle->put_pixelHeight(0);
                            spStyle->put_pixelWidth(0);
                        }
                        }
                        ATLTRACE(_T("Deleted flash animation\n"));
                    }
                        }
                }
                }
            }
            }
        }
        }

        // Find all the EMBED tags on the (maybe partially loaded) document
        vTagName = L"EMBED";
        vTagName.ChangeType(VT_BSTR);

        spTagsDisp.Release();
        spTagsDisp;
```

The flash animations can be placed on a web page either using an `OBJECT` tag or an `EMBED` tag, so we have to check for both. If the `OBJECT` tag was used to specify the ActiveX control then we can use the class ID to determine if it is a Flash control. If the `EMBED` tag was used, I examine the extension (*.swf*) of the source file. If a Flash control is found, I use inline CSS to resize and hide it. Another way to do it, as suggested by Ferdo, is to simply delete it. Yet another way to do it is to get the Flash controls `IDispatch` pointer and call its `Stop` and `Rewind` methods. The advantage of doing it that way is that you can still see the Flash control, but without the Flash, if you know what I mean.

You will notice that the `DisableFlashMoviesHelper` routine listed above takes an `IHTMLDocument2` as a parameter. This is because this routine may be called multiple times on one webpage due to nested frames, each of which has its own document. Probably the most logical way to handle the nested frames, is to use recursion.

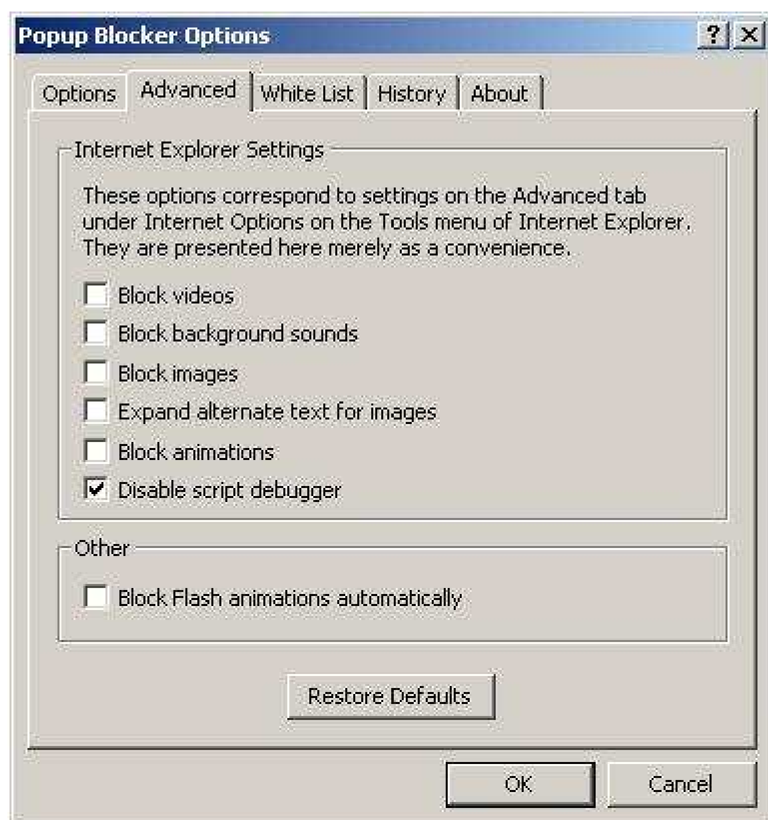
```
void DisableFlashMoviesRecursively(IHTMLDocument2* pDoc)
{
    CComQIPtr<IHTMLDOCUMENT2 &IID_IHTMLDocument2,> spHTML(pDoc);
    if (spHTML)
    {
        // Disable flash for the this document
        DisableFlashMoviesHelper(spHTML);

        // Disable flash for any embedded documents
        CComPtr<IHTMLELEMENTCOLLECTION> spAll;
        HRESULT hr = spHTML->get_all(&spAll);
        if (SUCCEEDED(hr) && spAll)
        {
            long nCnt =
        0; HRESULT  hr=


            spAll->get_length(&nCnt); if  (SUCCEEDED(hr)
            && nCnt  >0)
            { for (long i =

                  0;  i <  nCnt; i++)
                {
                    CComVariant varIdx;
                    V_VT(&varIdx) = VT_I4;
                    V_I4(&varIdx) = i;

                    CComPtr<IDISPATCH> spElemDisp;
                    hr = spAll->item(varIdx, varIdx, &spElemDisp);
                    if (SUCCEEDED(hr) && spElemDisp)
                    {
                        CComQIPtr<IHTMLFRAMEBASE2 &IID_IHTMLFrameBase2,>
                                     spFrame(spElemDisp);
                        if (spFrame)
                        {
                            // This is a frame or iframe element.
                            // Disable flash in its document.
                            CComPtr<IHTMLWINDOW2> spWin;
                            hr = spFrame->get_contentWindow(&spWin);
                            if (SUCCEEDED(hr) && spWin)
                            {
                                CComPtr<IHTMLDOCUMENT2> spDoc;
                                hr = spWin->get_document(&spDoc);
                                if (SUCCEEDED(hr) && spDoc)
                                {
                                    DisableFlashMoviesRecursively(spDoc);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

void CPub::DisableFlashMovies()
{
    ATLASSERT(m_spWebBrowser2);
    if (!m_spWebBrowser2)
        return;

    CComPtr<IDISPATCH> spDisp;
    HRESULT hr = m_spWebBrowser2->get_Document(&spDisp);
    if (SUCCEEDED(hr) && spDisp)
    {
        CComQIPtr<IHTMLDOCUMENT2 &IID_IHTMLDocument2,> spHTML(spDisp);
        if (spHTML)
            DisableFlashMoviesRecursively(spHTML);
    }
}
```

In the `DisableFlashMoviesRecursively` routine I traverse all the elements of the document looking for frames. If I find one, I drill down to its document and call `DisableFlashMoviesRecursively` recursively. This way we can be absolutely certain to have found all the Flash controls on the page.

One annoyance is that the Flash animations are played as the page loads, and we have to wait for the document to be loaded before we can run through the entire DOM. So animations will play until the final `DISPID_DOCUMENTCOMPLETE` or `READYSTATE_COMPLETE`, which will be most obvious if you have a slow connection. This can be somewhat mitigated by calling `DisableFlashMovies` every `DISPID_DOWNLOADBEGIN` or intermediate `DISPID_DOCUMENTCOMPLETE`, so that the animations are hidden on a frame-by-frame basis. A little ham-fisted but fairly effective. I chose not to do this because I think there is probably a better, less intensive, way to approach the problem, such as keying off an onload or ready state event for each individual Flash control. Personally, I just use 'Delete Flash Animations' on the context menu when I come across a really bad page. Maybe I will revisit this in version 3.

IE settings

There are some pertinent IE settings that turn off videos and animations, and disable the script debugger. These are simply registry settings that are also accessible from the IE Tools /Internet Options /Advanced dialog. I have placed these settings on the Advanced tab of the Options Dialog merely as a convenience.



The most important setting on this page is 'Disable script debugger', which by default should be selected. If you do not disable the script debugger, the BHO will not be able to trap script errors. Refer to IE Internet Options for help with the other settings.

Visual indication

Kiliman suggested that there should be a visual as well as aural indication that a popup has been blocked. Any type of indication should be as discrete as possible because to add distractions to the browsing experience is counter to what we are trying to achieve with Popup Blocker. After considering a number of ways to do it, I settled on simply changing the default status text from 'Done' to 'This page contains popup windows!'. IE shows the default status when it has nothing better to do, so this does not interfere with other messages that IE might send to the status bar.

```
// Show message in status bar
if (g_bStatusMsg && m_spHTMLWindow)
{
    // IE shows default status when there is nothing better to do.
    CComBSTR bsPopupsWarning = L"This page contains popup windows!";
    m_spHTMLWindow->put_defaultStatus(bsPopupsWarning);
}
```

Don't bother with the `IWebBrowser2::StatusText` property, IE will immediately overwrite your text with the `defaultStatus`.

Miscellaneous

One thing in version 1 that needed to be fixed was to allow popups on secure sites (HTTPS). This is because many secure sites use popups to prompt for user identification. To accomplish this I simply white list all secure sites.

This program is not compatible with Windows 98 or 98 SE; however, it should be fairly easy to correct. Since I no longer have a Win98 system I must appeal to the community. If anyone wants to make the corrections for Win98 I will post them here.

<shameless_plug>The latest version can be installed from the web at Osborn Technologies.
</shameless_plug>

Improvements

There is still room for improvement:

1. Find a better way to purge Flash animations, possibly by keying off an event when a Flash object is loaded.
2. Find a way to make the white list history combobox less daunting.

Revision history

- June 6, 2003: Removed code that deletes frames on `E_ACCESSDENIED`, pending further review. Added missing setup project. Added missing binary (*setup.msi*).
- June 7, 2003: Updated to install on Win98 SE.
- June 8, 2003: Added ferdo's code to block resizing by script. Added refresh after changing IE settings on Advanced property page.
- v2.3 (June 10, 2003): Flash now deleted in all frames. Hotkey and white list now disable all functionality.
- v2.4 (June 11, 2003): Fixed error 0x8001010E introduced with white list fix.
- v2.5 (June 17, 2003): Fixed save settings error.
- v3.0 (July 3, 2003): The resize-by-script blocking code has been made more robust (many, many thanks to Ferdinand Oeinck for his work!). A user-selectable hotkey has been added for deleting animated images. Now animated images may be blocked manually with the hotkey or automatically after the page loads, with the option (again thanks to Ferdinand!) of blocking non-cached images. Some problems with catching mouse clicks on elements within frames have been fixed.
- v3.0 (July 8, 2003): Added miers fix for IE5.5 (thank you!). Added a fix to the resizing code (thanks again to Ferdinand!).

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

John Osborn

Member

John is a software consultant specializing in MS Windows. In his limited spare time he enjoys playing tennis, road/mountain biking, snowboarding, speaking French, and finding creative ways to keep deer out of the garden and raccoons off the birdfeeder.

Occupation: Web Developer

Location: United States

Discussions and Feedback

291 messages have been posted for this article. Visit **http://www.codeproject.com/KB/shell/popupblocker2.aspx** to post and view comments on this article, or click **here** to get a print view with messages.

PermaLink | Privacy | Terms of Use
Last Updated: 7 Jul 2003
Editor: Smitha Vijayan

Copyright 2003 by John Osborn
Everything else Copyright © CodeProject, 1999-2009
Web18 | Advertise on the Code Project