Using Debugging Tools for Windows

Annotated x86 Disassembly

The following section will walk you through a disassembly example.

The Source Code

The following is the code for the function that will be analyzed.

```
HRESULT CUserView::CloseView(void)
{
    if (m_fDestroyed) return S_OK;

    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);

    if (m_psv) {
        m_psb->EnableModelessSB(FALSE);
        if(m_pws) m_pws->ViewReleased();

        IShellView* psv;

        HWND hwndCapture = GetCapture();
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }

        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
        NotifyClients(m_psv, NOTIFY_CLOSING);
        m_fRecursing = FALSE;

        m_psv->UIActivate(SVUIA_DEACTIVATE);

        psv = m_psv;
        m_psv = NULL;

        ReleaseAndNull(&_pctView);

        if (m_pvo) {
            IAdviseSink *pSink;
            if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
                if (pSink == (IAdviseSink *)this)
                    m_pvo->SetAdvise(0, 0, NULL);
                pSink->Release();
            }

            fViewObjectChanged = TRUE;
            ReleaseAndNull(&m_pvo);
        }

        if (psv) {
            psv->SaveViewState();
            psv->DestroyViewWindow();
            psv->Release();
        }

        m_hwndView = NULL;
        m_fHandsOff = FALSE;

        if (m_pcache) {
            GlobalFree(m_pcache);
            m_pcache = NULL;
        }

        m_psb->EnableModelessSB(TRUE);

        CancelPendingActions();
    }
```

```
    ReleaseAndNull(&_psf);

    if (fViewObjectChanged)
        NotifyViewClients(DVASPECT_CONTENT, -1);

    if (m_pszTitle) {
        LocalFree(m_pszTitle);
        m_pszTitle = NULL;
    }

    SetRect(&m_rcBounds, 0, 0, 0, 0);
    return S_OK;
}
```

The Assembly Code

This section contains the annotated disassembly example.

Functions with EBP frames will start out the following way:

```
HRESULT CUserView::CloseView(void)
SAMPLE!CUserView__CloseView:
71517134 55              push    ebp
71517135 8bec            mov     ebp,esp
```

This sets up the frame so the function can access its parameters as positive offsets from EBP, and local variables as negative offsets.

This is a method on a private COM interface, so the calling convention is Stdcall. This means that parameters are pushed right to left (in this case, there are none), then the "this" pointer is pushed, then the function is called. So on entry to the function, the stack looks like this:

```
[esp+0] = return address
[esp+4] = this
```

After the above two instructions, the parameters are accessible as:

```
[ebp+0] = previous ebp pushed on stack
[ebp+4] = return address
[ebp+8] = this
```

For a function that uses a standard EBP frame, the first pushed parameter is accessible at [ebp+8]; subsequent parameters are accessible at consecutive higher DWORD addresses.

```
71517137 51              push    ecx
71517138 51              push    ecx
```

This function requires only two local stack variables, so it's smaller to push two dummy registers than to do a "sub esp, 8". The pushed values are then available as [ebp-4] and [ebp-8].

For a function that uses a standard EBP frame, stack local variables are accessible at negative offsets from the EBP register.

```
71517139 56              push    esi
```

Now the compiler saves the registers that are required to be preserved across function calls. Actually, it saves them in bits and pieces, interleaved with the first line of actual code.

```
7151713a 8b7508          mov     esi,[ebp+0x8]     ; esi = this
7151713d 57              push    edi               ; save another registers
```

It so happens that CloseView is a method on ViewState, which is at offset 12 in the underlying object. This method will be referred to as "this," although when there is possible confusion with another base class, it will be more carefully specified as "(ViewState*)this."

```
    if (m_fDestroyed)
7151713e 33ff            xor     edi,edi           ; edi = 0
```

XOR'ing a register with itself is a standard way of zeroing it out.

```
71517140 39beac000000    cmp     [esi+0xac],edi    ; this->m_fDestroyed == 0?
71517146 7407            jz      NotDestroyed (7151714f)  ; jump if equal
```

The **cmp** instruction compares two values (by subtracting them). The **jz** instruction checks if the result is zero, indicating that the two compared values are equal.

The cmp instruction compares two values; a subsequent j instruction jumps based on the result of the comparison.

```
    return S_OK;
71517148 33c0            xor     eax,eax           ; eax = 0 = S_OK
7151714a e972010000      jmp     ReturnNoEBX (715172c1) ; return, do not pop EBX
```

The compiler delayed saving the EBX register until later in the function, so if the program is going to "early-out" on this test, then the exit path needs to be the one that doesn't restore EBX.

```
    BOOL fViewObjectChanged = FALSE;
    ReleaseAndNull(&m_pdtgt);
```

The execution of these two lines of code is interleaved, so pay attention.

```
NotDestroyed:
7151714f 8d86c0000000    lea     eax,[esi+0xc0]    ; eax = &m_pdtgt
```

The **lea** instruction computes the effect address of a memory access and stores it in the destination. The actual memory address is not dereferenced.

The lea instruction takes the address of a variable.

```
71517155 53              push    ebx
```

You should save that EBX register before it is damaged.

```
71517156 8b1d10195071    mov ebx,[_imp__ReleaseAndNull]
```

Since you will be calling **ReleaseAndNull** a lot, it is a good idea to cache its address in EBX.

```
7151715c 50              push    eax               ; parameter to ReleaseAndNull
7151715d 897dfc          mov     [ebp-0x4],edi     ; fViewObjectChanged = FALSE
71517160 ffd3            call    ebx               ; call ReleaseAndNull
    if (m_psv) {
71517162 397e74          cmp     [esi+0x74],edi    ; this->m_psv == 0?
71517165 0f8411010000    je      No_Psv (7151727c) ; jump if zero
```

Remember that you zeroed out the EDI register a while back, and that EDI is a register preserved across function calls (so the call to **ReleaseAndNull** didn't change it). Therefore, it still holds the value zero and you can use it to test for zero quickly.

```
        m_psb->EnableModelessSB(FALSE);
7151716b 8b4638          mov     eax,[esi+0x38]    ; eax = this->m_psb
7151716e 57              push    edi               ; FALSE
7151716f 50              push    eax               ; "this" for callee
71517170 8b08            mov     ecx,[eax]         ; ecx = m_psb->lpVtbl
71517172 ff5124          call    [ecx+0x24]        ; __stdcall EnableModelessSB
```

The above pattern is a telltale sign of a COM method call.

COM method calls are pretty popular, so it's a good idea to learn to recognize them. In particular, you should be

able to recognize the three IUnknown methods directly from their Vtable offsets: QueryInterface=0, AddRef=4, and Release=8.

```
        if(m_pws) m_pws->ViewReleased();
71517175 8b8614010000     mov     eax,[esi+0x114]   ; eax = this->m_pws
7151717b 3bc7             cmp     eax,edi           ; eax == 0?
7151717d 7406             jz      NoWS (71517185) ; if so, then jump
7151717f 8b08             mov     ecx,[eax]         ; ecx = m_pws->lpVtbl
71517181 50               push    eax               ; "this" for callee
71517182 ff510c           call    [ecx+0xc]         ; __stdcall ViewReleased
NoWS:
        HWND hwndCapture = GetCapture();
71517185 ff15e01a5071    call [_imp__GetCapture]    ; call GetCapture
```

Indirect calls through globals is how function imports are implemented in Microsoft® Win32®. The loader fixes up the globals to point to the actual address of the target. This is a handy way to get your bearings when you're investigating a crashed machine. Look for the calls to imported functions and in the target. You'll usually have the name of some imported function, which you can use to figure out where you are in the source code.

```
        if (hwndCapture && hwndCapture == m_hwnd) {
            SendMessage(m_hwnd, WM_CANCELMODE, 0, 0);
        }
7151718b 3bc7             cmp     eax,edi           ; hwndCapture == 0?
7151718d 7412             jz      No_Capture (715171a1) ; jump if zero
```

The function return value is placed in the EAX register.

```
7151718f 8b4e44           mov     ecx,[esi+0x44]    ; ecx = this->m_hwnd
71517192 3bc1             cmp     eax,ecx           ; hwndCapture = ecx?
71517194 750b             jnz     No_Capture (715171a1) ; jump if not

71517196 57               push    edi               ; 0
71517197 57               push    edi               ; 0
71517198 6a1f             push    0x1f              ; WM_CANCELMODE
7151719a 51               push    ecx               ; hwndCapture
7151719b ff1518195071     call    [_imp__SendMessageW] ; SendMessage
No_Capture:
        m_fHandsOff = TRUE;
        m_fRecursing = TRUE;
715171a1 66818e0c0100000180 or    word ptr [esi+0x10c],0x8001 ; set both flags at once

        NotifyClients(m_psv, NOTIFY_CLOSING);
715171aa 8b4e20           mov     ecx,[esi+0x20]    ; ecx = (CNotifySource*)this.vtbl
715171ad 6a04             push    0x4               ; NOTIFY_CLOSING
715171af 8d4620           lea     eax,[esi+0x20]    ; eax = (CNotifySource*)this
715171b2 ff7674           push    [esi+0x74]        ; m_psv
715171b5 50               push    eax               ; "this" for callee
715171b6 ff510c           call    [ecx+0xc]         ; __stdcall NotifyClients
```

Notice how you had to change your "this" pointer when calling a method on a different base class from your own.

```
        m_fRecursing = FALSE;
715171b9 80a60d0100007f   and     byte ptr [esi+0x10d],0x7f
        m_psv->UIActivate(SVUIA_DEACTIVATE);
715171c0 8b4674           mov     eax,[esi+0x74]    ; eax = m_psv
715171c3 57               push    edi               ; SVUIA_DEACTIVATE = 0
715171c4 50               push    eax               ; "this" for callee
715171c5 8b08             mov     ecx,[eax]         ; ecx = vtbl
715171c7 ff511c           call    [ecx+0x1c]        ; __stdcall UIActivate
        psv = m_psv;
        m_psv = NULL;
715171ca 8b4674           mov     eax,[esi+0x74]    ; eax = m_psv
715171cd 897e74           mov     [esi+0x74],edi    ; m_psv = NULL
715171d0 8945f8           mov     [ebp-0x8],eax     ; psv = eax
```

Psv is your first local variable.

```
        ReleaseAndNull(&_pctView);
715171d3 8d466c          lea     eax,[esi+0x6c]    ; eax = &_pctView
715171d6 50              push    eax               ; parameter
715171d7 ffd3            call    ebx               ; call ReleaseAndNull
        if (m_pvo) {
715171d9 8b86a8000000    mov     eax,[esi+0xa8]    ; eax = m_pvo
715171df 8dbea8000000    lea     edi,[esi+0xa8]    ; edi = &m_pvo
715171e5 85c0            test    eax,eax           ; eax == 0?
715171e7 7448            jz      No_Pvo (71517231) ; jump if zero
```

Note that the compiler speculatively prepared the address of the *m_pvo* member because you're going to use it a lot for a while, so having the address handy will result in smaller code.

```
            if (SUCCEEDED(m_pvo->GetAdvise(NULL, NULL, &pSink)) && pSink) {
715171e9 8b08            mov     ecx,[eax]         ; ecx = m_pvo->lpVtbl
715171eb 8d5508          lea     edx,[ebp+0x8]     ; edx = &pSink
715171ee 52              push    edx               ; parameter
715171ef 6a00            push    0x0               ; NULL
715171f1 6a00            push    0x0               ; NULL
715171f3 50              push    eax               ; "this" for callee
715171f4 ff5120          call    [ecx+0x20]        ; __stdcall GetAdvise
715171f7 85c0            test    eax,eax           ; test bits of eax
715171f9 7c2c            jl      No_Advise (71517227) ; jump if less than zero
715171fb 33c9            xor     ecx,ecx           ; ecx = 0
715171fd 394d08          cmp     [ebp+0x8],ecx     ; _pSink == ecx?
71517200 7425            jz      No_Advise (71517227)
```

Notice that the compiler concluded that the incoming "this" parameter was not required (since it long ago stashed that into the ESI register), so it reused the memory as the local variable pSink.

If the function uses an EBP frame, then incoming parameters arrive at positive offsets from EBP and local variables are placed at negative offsets. But, as in this case, the compiler is free to reuse that memory for any purpose.

If you're paying really close attention, you'll see that the compiler could've optimized this code a little better. It could've delayed the **lea edi, [esi+0xa8]** instruction until after the two **push 0x0** instructions above, replacing them with **push edi**. This would've saved two bytes.

```
                if (pSink == (IAdviseSink *)this)
```

These next several lines are to compensate for the fact that in C++, (IAdviseSink *)NULL must still be NULL. So if your "this" is really "(ViewState*)NULL", then the result of the cast should be NULL and not the distance between IAdviseSink and IBrowserService.

```
71517202 8d46ec          lea     eax,[esi-0x14]    ; eax = -(IAdviseSink*)this
71517205 8d5614          lea     edx,[esi+0x14]    ; edx = (IAdviseSink*)this
71517208 f7d8            neg     eax               ; eax = -eax (sets carry if != 0)
7151720a 1bc0            sbb     eax,eax           ; eax = eax - eax - carry
7151720c 23c2            and     eax,edx           ; eax = NULL or edx
```

Although the Pentium has a conditional move instruction, the base i386 architecture does not, so the compiler uses tricks to simulate a conditional move instruction without taking any jumps.

The general pattern for a conditional evaluation is the following:

```
        neg     r
        sbb     r, r
        and     r, (val1 - val2)
        add     r, val2
```

The **neg r** sets the carry flag if **r** is nonzero, because **neg** negates the value by subtracting from zero, and subtracting from zero will generate a borrow (set the carry) if you subtract a nonzero value. It also damages the value in the **r** register, but that's okay because you're about to overwrite it anyway.

Next, the **sbb r, r** instruction subtracts a value from itself, which always results in zero, but it also subtracts the carry (borrow) bit, so the net result is to set **r** to 0 or -1, depending on whether the carry was clear or set, respectively.

Therefore, **sbb r, r** sets **r** to zero if the original value of **r** was zero, or to -1 if the original value was nonzero.

The third instruction performs a mask. Since the **r** register is 0 or -1, "this" serves either to leave **r** zero or to change **r** from -1 to **(val1 - val1)**, since ANDing any value with -1 leaves the original value.

Therefore, the result of "and r, (val1 - val1)" is to set r to 0 if the original value of r was zero, or to "(val1 - val2)" if the original value of r was nonzero.

Finally, you add **val2** to **r**, resulting in **val2** or **(val1 - val2) + val2 = val1**.

So the ultimate result of this series of instructions is to set **r** to **val2** if it was originally zero or to **val1** if it was nonzero.

This is the assembly equivalent of r = r ? val1 : val2.

In this particular instance, you can see that **val2 = 0** and **val1 = (IAdviseSink*)this**. (Notice that the compiler elided the final **add eax, 0** since it has no effect.)

```
7151720e 394508          cmp     [ebp+0x8],eax ; pSink == (IAdviseSink*)this?
71517211 750b            jnz     No_SetAdvise (7151721e) ; jump if not equal
```

Earlier in this section, you set EDI to the addess of the *m_pvo* member. You're going to be using it now. You also zeroed out the ECX register a while back.

```
                    m_pvo->SetAdvise(0, 0, NULL);
71517213 8b07            mov     eax,[edi]         ; eax = m_pvo
71517215 51              push    ecx               ; NULL
71517216 51              push    ecx               ; 0
71517217 51              push    ecx               ; 0
71517218 8b10            mov     edx,[eax]         ; edx = m_pvo->lpVtbl
7151721a 50              push    eax               ; "this" for callee
7151721b ff521c          call    [edx+0x1c]        ; __stdcall SetAdvise
No_SetAdvise:
                pSink->Release();
7151721e 8b4508          mov     eax,[ebp+0x8]     ; eax = pSink
71517221 50              push    eax               ; "this" for callee
71517222 8b08            mov     ecx,[eax]         ; ecx = pSink->lpVtbl
71517224 ff5108          call    [ecx+0x8]         ; __stdcall Release
No_Advise:
```

All these COM method calls should look very familiar.

The evaluation of the next two statements is interleaved. Don't forget that EBX contains the address of **ReleaseAndNull**.

```
            fViewObjectChanged = TRUE;
            ReleaseAndNull(&m_pvo);
71517227 57              push    edi               ; &m_pvo
71517228 c745fc01000000  mov     dword ptr [ebp-0x4],0x1 ; fViewObjectChanged = TRUE
7151722f ffd3            call    ebx               ; call ReleaseAndNull
No_Pvo:
        if (psv) {
71517231 8b7df8          mov     edi,[ebp-0x8]     ; edi = psv
71517234 85ff            test    edi,edi           ; edi == 0?
71517236 7412            jz      No_Psv2 (7151724a) ; jump if zero
            psv->SaveViewState();
71517238 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
7151723a 57              push    edi               ; "this" for callee
7151723b ff5034          call    [eax+0x34]        ; __stdcall SaveViewState
```

Here are more COM method calls.

```
            psv->DestroyViewWindow();
7151723e 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
71517240 57              push    edi               ; "this" for callee
71517241 ff5028          call    [eax+0x28]        ; __stdcall DestroyViewWindow
            psv->Release();
71517244 8b07            mov     eax,[edi]         ; eax = psv->lpVtbl
71517246 57              push    edi               ; "this" for callee
```

```
71517247 ff5008          call    [eax+0x8]         ; __stdcall Release
No_Psv2:
        m_hwndView = NULL;
7151724a 83667c00        and     dword ptr [esi+0x7c],0x0 ; m_hwndView = 0
```

ANDing a memory location with zero is the same as setting it to zero, since anything AND zero is zero. The compiler uses this form because, even though it's slower, it's much shorter than the equivalent **mov** instruction. (This code was optimized for size, not speed.)

```
        m_fHandsOff = FALSE;
7151724e 83a60c010000fe  and     dword ptr [esi+0x10c],0xfe
        if (m_pcache) {
71517255 8b4670          mov     eax,[esi+0x70]    ; eax = m_pcache
71517258 85c0            test    eax,eax           ; eax == 0?
7151725a 740b            jz      No_Cache (71517267) ; jump if zero
            GlobalFree(m_pcache);
7151725c 50              push    eax               ; m_pcache
7151725d ff15b4135071    call    [_imp__GlobalFree]    ; call GlobalFree
            m_pcache = NULL;
71517263 83667000        and     dword ptr [esi+0x70],0x0 ; m_pcache = 0
No_Cache:
        m_psb->EnableModelessSB(TRUE);
71517267 8b4638          mov     eax,[esi+0x38]    ; eax = this->m_psb
7151726a 6a01            push    0x1               ; TRUE
7151726c 50              push    eax               ; "this" for callee
7151726d 8b08            mov     ecx,[eax]         ; ecx = m_psb->lpVtbl
7151726f ff5124          call    [ecx+0x24]        ; __stdcall EnableModelessSB
        CancelPendingActions();
```

In order to call **CancelPendingActions**, you have to move from (ViewState*)this to (CUserView*)this. Note also that **CancelPendingActions** uses the __thiscall calling convention instead of __stdcall. According to __thiscall, the "this" pointer is passed in the ECX register instead of being passed on the stack.

```
71517272 8d4eec          lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
71517275 e832fbffff      call CUserView::CancelPendingActions (71516dac) ; __thiscall
    ReleaseAndNull(&_psf);
7151727a 33ff            xor     edi,edi           ; edi = 0 (for later)
No_Psv:
7151727c 8d4678          lea     eax,[esi+0x78]    ; eax = &_psf
7151727f 50              push    eax               ; parameter
71517280 ffd3            call    ebx               ; call ReleaseAndNull
    if (fViewObjectChanged)
71517282 397dfc          cmp     [ebp-0x4],edi     ; fViewObjectChanged == 0?
71517285 740d            jz      NoNotifyViewClients (71517294) ; jump if zero
       NotifyViewClients(DVASPECT_CONTENT, -1);
71517287 8b46ec          mov     eax,[esi-0x14]    ; eax = ((CUserView*)this)->lpVtbl
7151728a 8d4eec          lea     ecx,[esi-0x14]    ; ecx = (CUserView*)this
7151728d 6aff            push    0xff              ; -1
7151728f 6a01            push    0x1               ; DVASPECT_CONTENT = 1
71517291 ff5024          call    [eax+0x24]        ; __thiscall NotifyViewClients
NoNotifyViewClients:
    if (m_pszTitle)
71517294 8b86800000000   mov     eax,[esi+0x80]    ; eax = m_pszTitle
7151729a 8d9e80000000    lea     ebx,[esi+0x80]    ; ebx = &m_pszTitle (for later)
715172a0 3bc7            cmp     eax,edi           ; eax == 0?
715172a2 7409            jz      No_Title (715172ad) ; jump if zero
        LocalFree(m_pszTitle);
715172a4 50              push    eax               ; m_pszTitle
715172a5 ff1538125071    call   [_imp__LocalFree]
        m_pszTitle = NULL;
```

Remember that EDI is still zero and EBX is still &m_pszTitle, since those registers are preserved by function calls.

```
715172ab 893b            mov     [ebx],edi         ; m_pszTitle = 0
No_Title:
    SetRect(&m_rcBounds, 0, 0, 0, 0);
715172ad 57              push    edi               ; 0
715172ae 57              push    edi               ; 0
```

```
715172af 57              push    edi             ; 0
715172b0 81c6fc000000    add     esi,0xfc        ; esi = &this->m_rcBounds
715172b6 57              push    edi             ; 0
715172b7 56              push    esi             ; &m_rcBounds
715172b8 ff15e41a5071    call   [_imp__SetRect]
```

Notice that you don't need the value of "this" any more, so the compiler uses the **add** instruction to modify it in place instead of blowing another register to hold the address. This is actually a performance win due to the Pentium u/v pipelining, since the v pipe can do arithmetic but not address computations.

```
    return S_OK;
715172be 33c0            xor     eax,eax         ; eax = S_OK
```

Finally, you restore the registers you are required to preserve, clean up the stack, and return to your caller, removing the incoming parameters.

```
715172c0 5b              pop     ebx             ; restore
ReturnNoEBX:
715172c1 5f              pop     edi             ; restore
715172c2 5e              pop     esi             ; restore
715172c3 c9              leave                   ; restores EBP and ESP simultaneously
715172c4 c20400          ret     0x4             ; return and clear parameters
```

Built on Monday, July 02, 2001