# What can go wrong when you mismatch the calling convention?

devblogs.microsoft.com/oldnewthing/20040115-00

January 15, 2004

Raymond Chen

Believe it or not, calling conventions is one of the things that programs frequently get wrong. The compiler yells at you when you mismatch a calling convention, but lazy programmers will just stick a cast in there to get the compiler to "shut up already".

And then Windows is stuck having to support your buggy code forever.

### The window procedure

So many people misdeclare their window procedures (usually by declaring them as __cdecl instead of __stdcall), that the function that dispatches messages to window procedures contains extra protection to detect incorrectly-declared window procedures and perform the appropriate fixup. This is the source of the mysterious 0xdcbaabcd on the stack. The function that dispatches messages to window procedures checks whether this value is on the stack in the correct place. If not, then it checks whether the window procedure popped one dword too much off the stack (if so, it fixes up the stack; I have no idea how this messed up a window procedure could have existed), or whether the window procedure was mistakenly declared as __cdecl instead of __stdcall (if so, it pops the parameters off the stack that the window procedure was supposed to do).

### DirectX callbacks

Many DirectX functions use callbacks, and people once again misdeclared their callbacks as __cdecl instead of __stdcall, so the DirectX enumerators have to do special stack cleanup for those bad functions.

### IShellFolder::CreateViewObject

I remember there was one program that decided to declare their CreateViewWindow function incorrectly, and somehow they managed to trick the compiler into accepting it!

```
class BuggyFolder : public IShellFolder ... {
 ...
 // wrong function signature!
 HRESULT CreateViewObject(HWND hwnd) { return S_OK; }
}
```

Not only did they get the function signature wrong, they returned S_OK even though they failed to do anything! I had to add extra code to clean up the stack after calling this function, as well as verify that the return value wasn't a lie.

**Rundll32.exe entry points**

The function signature required for functions called by rundll32.exe is documented in this Knowledge Base article. That hasn't stopped people from using rundll32 to call random functions that weren't designed to be called by rundll32, like user32 LockWorkStation or user32 ExitWindowsEx.

Let's walk through what happens when you try to use rundll32.exe to call a function like ExitWindowsEx:

The rundll32.exe program parses its command line and calls the ExitWindowsEx function on the assumption that the function is written like this:

```
void CALLBACK ExitWindowsEx(HWND hwnd, HINSTANCE hinst,
       LPSTR pszCmdLine, int nCmdShow);
```

But it isn't. The actual function signature for ExitWindowsEx is

```
BOOL WINAPI ExitWindowsEx(UINT uFlags, DWORD dwReserved);
```

What happens? Well, on entry to ExitWindowsEx, the stack looks like this:

| |
|---|
| .. rest of stack .. |
| nCmdShow |
| pszCmdLine |
| hinst |
| hwnd |
| return address <- ESP |

However, the function is expecting to see

| |
|---|
| .. rest of stack .. |
| dwReserved |
| uFlags |
| return address <- ESP |

What happens? The hwnd passed by rundll32.exe gets misinterpreted as uFlags and the hinst gets misinterpreted as dwReserved. Since window handles are pseudorandom, you end up passing random flags to ExitWindowsEx. Maybe today it's EWX_LOGOFF, tomorrow it's EWX_FORCE, the next time it might be EWX_POWEROFF.

Now suppose that the function manages to return. (For example, the exit fails.) The ExitWindowsEx function cleans two parameters off the stack, unaware that it was passed four. The resulting stack is

| .. rest of stack .. | |
|---|---|
| nCmdShow | (garbage not cleaned up) |
| pszCmdLine | <- ESP (garbage not cleaned up) |

Now the stack is corrupted and really fun things happen. For example, suppose the thing at ".. rest of the stack .." is a return address. Well, the original code is going to execute a "return" instruction to return through that return address, but with this corrupted stack, the "return" instruction will instead return to a command line and attempt to execute it as if it were code.

**Random custom functions**

An anonymous commenter exported a function as __cdecl but treated it as if it were __stdcall. This will seem to work, but on return, the stack will be corrupted (because the caller is expecting a __stdcall function that cleans the stack, but what it gets is a __cdecl funcion that doesn't), and bad things will happen as a result.

Okay, enough with the examples; I think you get the point. Here are some questions I'm sure you're asking:

**Why doesn't the compiler catch all these errors?**

It does. (Well, not the rundll32 one.) But people have gotten into the habit of just inserting the function cast to get the compiler to shut up.

Here's a random example I found:

```
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg,
   WPARAM wParam, LPARAM lParam);
```

This is the incorrect function signature for a dialog procedure. The correct signature is

```
INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg,
    WPARAM wParam, LPARAM lParam);
```

You start with

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
          hWnd, DlgProc);
```

but the compiler rightly spits out the error message

```
error C2664: 'DialogBoxParamA' : cannot convert parameter 4
from 'LRESULT (HWND,UINT,WPARAM,LPARAM)' to 'DLGPROC'
```

so you fix it by slapping a cast in to make the compiler shut up:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
          hWnd, reinterpret_cast<DLGPROC>(DlgProc));
```

“Aw, come on, who would be so stupid as to insert a cast to make an error go away without actually fixing the error?”

Apparently everyone.

I stumbled across this page that does exactly the same thing, and this one in German which gets not only the return value wrong, but also misdeclares the third and fourth parameters, and this one in Japanese. It’s as easy to fix (incorrectly) as 1-2-3.

**How did programs with these bugs ever work at all? Certainly these programs worked to some degree or people would have noticed and fixed the bug. How can the program survive a corrupted stack?**
I’ll answer this question tomorrow.

Raymond Chen

**Follow**