You can read a contract from the other side

devblogs.microsoft.com/oldnewthing/20031226-00

December 26, 2003

Raymond Chen

An interface is a contract, but remember that a contract applies to both parties. Most of the time, when you read an interface, you look at it from the point of view of the client side of the contract, but often it helps to read it from the server side.

For example, let's look at the interface for control panel applications.

Most of the time, when you're reading this documentation, you are wearing your "I am writing a Control Panel application" hat. So, for example, the documentation says

> When the controlling application first loads the Control Panel application, it retrieves the address of the **CPlApplet** function and subsequently uses the address to call the function and pass it messages.

With your "I am writing a Control Panel application" hat, this means "Gosh, I had better have a function called CPlApplet and export it so I can receive messages."

But if you are instead wearing your "I am hosting a Control Panel application" hat, this means, "Gosh, I had better call GetProcAddress() to get the address of the application's CPlApplet function so I can send it messages."

Similarly, under the "Message Processing" section it lists the messages that are sent from the controlling application to the Control Panel application. If you are wearing your "I am writing a Control Panel application" hat, this means "Gosh, I had better be ready to receive these messages in this order." But if you are wearing your "I am hosting a Control Panel application" hat, this means "Gosh, I had better send these messages in the order listed."

And finally, when it says "the controlling application release the Control Panel application by calling the FreeLibrary function," your "I am writing a Control Panel application" hat says "I had better be prepared to be unloaded," whereas your "I am hosting a Control Panel application" hat says, "This is where I unload the DLL."

So let's try it. As always, start with our scratch program and change the WinMain:

```
#include <cpl.h>
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
                   LPSTR lpCmdLine, int nShowCmd)
{
  HWND hwnd;
  g_hinst = hinst;
  if (!InitApp()) return 0;
  if (SUCCEEDED(CoInitialize(NULL))) {/* In case we use COM */
      hwnd = CreateWindow(
          "Scratch",                      /* Class Name */
          "Scratch",                      /* Title */
          WS_OVERLAPPEDWINDOW,            /* Style */
          CW_USEDEFAULT, CW_USEDEFAULT,   /* Position */
          CW_USEDEFAULT, CW_USEDEFAULT,   /* Size */
          NULL,                           /* Parent */
          NULL,                           /* No menu */
          hinst,                          /* Instance */
          0);                             /* No special parameters */

      if (hwnd) {
        TCHAR szPath[MAX_PATH];
        LPTSTR pszLast;
        DWORD cch = SearchPath(NULL, TEXT("access.cpl"),
                     NULL, MAX_PATH, szPath, &pszLast);
        if (cch > 0 && cch < MAX_PATH) {
          RunControlPanel(hwnd, szPath);
      }
    }
    CoUninitialize();
  }
  return 0;
}
```

Instead of showing the window and entering the message loop, we start acting like a Control Panel host. Our victim today is access.cpl, the accessibility control panel. After locating the program on the path, we ask RunControlPanel to do the heavy lifting:

```
void RunControlPanel(HWND hwnd, LPCTSTR pszPath)
{
  // Maybe this control panel application has a custom manifest
  ACTCTX act = { 0 };
  act.cbSize = sizeof(act);
  act.dwFlags = 0;
  act.lpSource = pszPath;
  act.lpResourceName = MAKEINTRESOURCE(123);
  HANDLE hctx = CreateActCtx(&act);
  ULONG_PTR ulCookie;
  if (hctx == INVALID_HANDLE_VALUE ||
      ActivateActCtx(hctx, &ulCookie)) {
    HINSTANCE hinstCPL = LoadLibrary(pszPath);
    if (hinstCPL) {
      APPLET_PROC pfnCPlApplet = (APPLET_PROC)
        GetProcAddress(hinstCPL, "CPlApplet");
      if (pfnCPlApplet) {
        if (pfnCPlApplet(hwnd, CPL_INIT, 0, 0)) {
          int cApplets = pfnCPlApplet(hwnd, CPL_GETCOUNT, 0, 0);
          //  We're going to run application zero
          //  (In real life we might show the user a list of them
          //  and let them pick one)
          if (cApplets > 0) {
            CPLINFO cpli;
            pfnCPlApplet(hwnd, CPL_INQUIRE, 0, (LPARAM)&cpli);
            pfnCPlApplet(hwnd, CPL_DBLCLK, 0, cpli.lData);
            pfnCPlApplet(hwnd, CPL_STOP, 0, cpli.lData);
          }
        }
        pfnCPlApplet(hwnd, CPL_EXIT, 0, 0);
      }
      FreeLibrary(hinstCPL);
    }

    if (hctx != INVALID_HANDLE_VALUE) {
      DeactivateActCtx(0, ulCookie);
      ReleaseActCtx(hctx);
    }
  }
}
```

Ignore the red lines for now; we'll discuss them later.

All we're doing is following the specification but reading it from the host side. So we load the library, locate its entry point, and call it with CPL_INIT, then CPL_GETCOUNT. If there are any control panel applications inside this CPL file, we inquire after the first one, double-click it (this is where all the interesting stuff happens), then stop it. After all that excitement, we clean up according to the rules set out for the host (namely, by sending a CPL_EXIT message.)

So that's all. Well, except for the red parts. What's that about?

The red parts are to support Control Panel applications that have a custom manifest. This is something new with Windows XP and is documented in MSDN here.

If you go down to the "Using ComCtl32 Version 6 in Control Panel or a DLL That Is Run by RunDll32.exe" section, you'll see that the application provides its manifest to the Control Panel host by attaching it as resource number 123. So that's what the red code does: It loads and activates the manifest, then invites the Control Panel application to do its thing (with its manifest active), then cleans up. If there is no manifest, CreateActCtx will return INVALID_HANDLE_VALUE. We do not treat that as an error, since many programs don't yet provide a manifest.

Exercise: What are the security implications of passing NULL as the first parameter to SearchPath?

Raymond Chen

Follow