Tales of Application Compatibility

These stories, unlike the ones in Chapter 13, aren't really trying to teach you anything. They're just interesting little stories. Often, telling these little vignettes around the lunch table was our last grasp on sanity.

The tools of application compatibility

To set some of these stories into context, I'll start with some background on the application compatibility infrastructure, which has evolved over the decades. At first, the only way to address a compatibility problem was to change the core operating system so that the compatibility fix applied to all programs. If one program relied on a feature behaving a particular way, then the feature had to continue behaving that way for all programs. Part of the reason was that there simply wasn't any infrastructure available at the time to target a particular program, and part of the reason was the principle, "Well, if we found one program doing it, there are probably lots of others that do it, too." The application compatibility team tests thousands of applications, but that's still only a small fraction of the total number of applications out there.

In Windows 3.0, the operating system gained the capability of tailoring its behavior according to whether the running application was designed for Windows 3.0 or an earlier version of Windows. The general principle was still, "If one program relies on a particular behavior, then there are probably others." But on occasion, the old behavior was so undesirable that having the operating system behave in different ways depending on how the application was written was the lesser of two evils.

Windows 3.1 contained many additional changes, and with those changes came new compatibility issues. For example, some programs were passing invalid parameters and relying on some quirk of how Windows behaved when faced with particular types of abuse. A program might have passed invalid flags, that is, flags with no defined meaning, in Windows 3.0. A function may have accepted the flags `0x0001` and `0x0002`, say, but a program passed `0x0005` instead. In Windows 3.0, the function interpreted the value as `0x0001 | 0x0004` and ignored the `0x0004`. Come Windows 3.1, the value `0x0004` gained a particular meaning, and this old program now was receiving some new, unwanted behavior. The solution was to ignore the value `0x0004` if the program was designed for Windows 3.0.

Often, however, disabling a feature for all Windows 3.0 programs would have undermined the feature too severely. For example, a few programs could not cope with improvements that had been made to Windows printing. If only the target operating system version number were checked, then no Windows 3.0 programs would have benefited from these improvements. Instead, the specific programs that had trouble with optimized printing were tagged, and the improvements were disabled only for those programs.

In Windows 95, a new tool joined the application compatibility arsenal: the patch. If a program was doing something wrong in such a way that the operating system couldn't really work around the problem, then there was an option in the loader to detect the program and apply a set of patches to the in-memory image to fix the problem. This was a very powerful technique, but it came with many caveats. First, a patch applies only to a particular version of a program. When a new version comes out, the code is different and applying the patch would end up modifying the wrong thing. Care, therefore, had to be

taken to ensure that the program being patched really was the one the patch was designed for. Furthermore, the program's vendor had to be contacted, and permission obtained, to modify its program. Some vendors would change their program without updating the version number, making the development of a patch even trickier, which is why the vendor was asked to provide a list of all known versions of its program that had this problem so that a patch could be developed for each version. And, of course, the details of the patch were given to the vendors so that they could fix their program and thus prevent future versions from having the same problem. This last step is essential, because the patch can't apply to versions that didn't exist when the patch was developed.

Windows 2000 took another major step forward with the introduction of shims. The idea behind shims is to remove application-specific compatibility hacks from the core operating system and move them to a supporting library. When the loader detects that an application requires special compatibility behavior, the corresponding shim is loaded, and the shim inserts itself between the application and the operating system, thereby allowing the shim to provide whatever compatibility behaviors are necessary. For example, one useful shim is known as HeapPadAllocation; it is applied to programs that have heap buffer overrun bugs. The shim intercepts calls to the `HeapAllocate` function and adds a specified amount to the requested size. That way, when the program overruns a buffer, it merely corrupts the padding rather than corrupting the next heap block.

With all these tools available, the compatibility team faces a decision once a compatibility problem has been analyzed and understood: They need to evaluate whether this is a problem specific to one application or whether the problem is likely to affect many other applications as well. If the problem is peculiar enough that only that one program suffers from it, then a shim would be a suitable solution. On the other hand, if they conclude that the problem will probably afflict many programs (which, given the tip-of-the-iceberg principle, is the more likely alternative), then the next decision is whether a fix would be harmful to unaffected programs if it were applied generally. In many cases, applying the fix to the core operating system doesn't harm the other

programs in the system. The operating system is just "more compatible than you might have expected." But in other cases, such as the HeapPadAllocation example, applying the fix to all programs would be harmful. In those cases, a shim would be warranted.

Note also that all the application compatibility tools apply on the process level. If a problem exists in an extension mechanism, such as an Explorer shell extension or an Internet Explorer Web browser extension or simply a COM object, then an application-level fix would not accomplish anything. For the Web browser extension, the compatibility fix would be applied to Internet Explorer, in which case there's no point making the fix to the window manager if you can just put it into Internet Explorer itself. With a COM object, you're in even worse shape, because COM objects by their nature can be loaded into *any* process. Your only choice is to put the compatibility fix into the core operating system.

That's a quick overview of the application compatibility toolbox. The following anecdotes cover a wide time frame, and knowing what compatibility tricks were available at the time of each particular problem may help you understand why Windows settled on the solution it did. Otherwise, you may ask questions akin to this one overheard at a showing of the movie *Apollo 13*: "Why didn't they just use the Space Shuttle to rescue the astronauts?"

Contacting a vendor about a bug in its application

THE DETAILS OF the conversation are imaginary, but the point is real.

"Hello?"

"Hello, this is XYZ from Microsoft Windows application compatibility. We found a bug in your application that renders it unusable, and we added a workaround to Windows 95 so that your program still works."

"That's great, thanks! Bye-bye." *Click.*

Ring-ring.

"Hello?"

"Hi again, um, didn't you want to know what the bug was?"

"Why does it matter? You fixed it. Thanks! You guys are awesome."

"Right, well, you see, the workaround is applied only to the current version of your program. Windows won't apply the workaround to future versions of your program, so once you ship your next version, it'll crash."

"Oh. Um, hang on, let me get one of our programmers on the line."

You really don't get their attention until you mention that their program will eventually stop working.

When I was working on application compatibility for MS-DOS-based programs (games mostly), I would often call the vendor to let the company know that its program wasn't working on Windows 95. Many of the vendors replied, "Oh, we don't support Windows."

Rolling your own version of standard system functions

We received reports that a popular software development library was failing to run on Windows 95. Upon closer inspection, we found the reason: The program wanted to look at the system configuration file that was responsible for Windows device drivers, known as `system.ini`. Instead of using the `GetPrivateProfileString` function to read strings from that file, the program opened the file and parsed it manually. Unfortunately, what the authors of the program failed to take into account was that `GetPrivateProfileString` uses a case-insensitive comparison to locate the section. Their version used a case-sensitive comparison. The result was that the program failed to locate the `[386Enh]` section of the configuration file. The fix was to tweak Windows 95 Setup so that it used exactly the capitalization that the library expected.

This wasn't the only program to try its hand at parsing the `system.ini` file manually. One card game tried to make changes to the `system.ini` file but ended up destroying it. The game read the `system.ini` file a line at a time into an 80-character buffer, made any necessary changes, and wrote the result to a temporary file. If any line contained more than 80 characters, the buffer

overflowed and corrupted the next variable on the stack, which happened to be the name of the temporary file!

Once the changes were made, the program deleted the `system.ini` file and renamed the temporary file to `system.ini`. But the rename operation failed because the name of the temporary file was corrupted by the extra-long line. The result: a system with no system configuration file.

In other words, installing this program rendered your system unbootable.

The fix from the operating system side was to go through all the components of the system that used the `system.ini` file and make sure none of them ever wrote lines longer than 80 characters.

Why was a card game editing the driver configuration file, anyway? It turns out that the program was installing a 32-bit device driver to play "high-quality sound out the PC speaker." Spot the oxymoron. What's more, the driver the program was installing is one that I recognized as having "correctness issues" on Windows 95. As one example, the driver manipulates the timer chip on the motherboard to get the high-resolution timing necessary to do its "high-quality sound." But it forgot to set things back the way it found them, so once this game started, even after you quit the game, system animations and audio playback (from your actual sound card) were all messed up.

Bypassing the operating system altogether

A POPULAR VIDEO playback program didn't want the computer to suspend while a movie was playing, so the program issues an `int 15h` call into the BIOS to disable Advanced Power Management (APM) and issues a matching enable call when the movie is finished. This, despite the power management messages that already existed, such as the `WM_POWER` message, which allows an application to veto a suspend.

Notice that the program ignores whether APM was already disabled when it comes time to reenable power management. As a result, this code was designed to override the user's decision whether or not to enable power management. Even if APM were turned off before movie playback, the program turns it back on afterward.

If you dig into the APM documentation, it states that you are not allowed to issue the "disable" call until you have first issued the "connect" call. This program never issues a "connect." And this bug actually saves them, for when the program is run on Windows 3.1, all the calls to enable and disable merely fail with the error "not connected." The feature never worked.

But on Windows 95, the program crashes because Windows 95 comes with native support for APM. The power management driver connects to the APM BIOS in 32-bit mode at startup. This time, when the program issues its "disable" call, this does not result in a "not connected" error, because APM is indeed connected. It's not connected to the program, though; it's connected to the power management driver, but the BIOS has no way of knowing who is issuing the request.

And it so happens that some BIOSes get confused when 16-bit code issues an APM call when APM is connected to a 32-bit client—so confused that they hang the machine hard. The bug was not in Windows 95, but rather in the interaction between two other components; Windows 95 served merely as the catalyst.

Further adventures in overflowing buffers

Multiple programs from a leading productivity application vendor were discovered to pass a buffer size of 198 whenever they called the `LoadString` function, regardless of the actual buffer size, even for buffers only 4 or 8 characters in size. Luckily for these programs, they always allocated just enough space for the actual string they wanted.

The Windows 95 team members discovered this problem after they made some changes to the `LoadString` function, part of which involved using the "unused" parts of the buffer. They were forced to tweak their changes so as not to trust the buffer size if the "expected OS version" of the application was less than 4.0.

The "erroneously sized buffer" problem is quite common. Programs would pass a buffer of a particular size, but specify that the size was some other (larger) size, and they would get away with it because the result did fit in the buffer. But when the system decided to use the full buffer temporarily (say, to

build a result, and then chop it back down to size before returning it), the programs crashed because they overreported the buffer size.

Another program tried to read the name of the current screen saver from the `system.ini` file, but if the screen saver's name was more than twelve characters long (i.e., longer than 8.3 format), the buffer overflowed. This is why all the system screen savers in Windows 95 have short filenames.

Another category of problem is reading past the end of a buffer. A crash in a shell extension for a portable music player was traced to computing the correct buffer length and then willfully overflowing it. When the shell connects to the shell extension, one of the parameters is a data structure that describes the location of the shell extension in the namespace. This particular shell extension wants to make a copy of this data structure for safekeeping. It so happens that the data structure is in the form of a packed array of variable-length structures: Each structure comes with a size prefix, and the array ends with an element whose length is zero. (Those familiar with shell programming will recognize this as an item ID list.)

The function that copies the data structure goes something like this (the names have been changed, of course):

```
LPITEMIDLIST CopyIDList(IMalloc *pmalloc, LPITEMIDLIST pidlIn)
{
 DWORD cbPidl = GetIDListSizeWithExtra(pidlIn);
 LPITEMIDLIST pidlNew = pmalloc->Alloc(cbPidl);
 if (pidlNew) memcpy(pidlNew, pidlIn, cbPidl);
 return pidlNew;
}
```

This seems a perfectly reasonable way of copying a variable-length structure. You compute its length, allocate some memory to accommodate its size, and then copy … wait a minute! What is this "with extra"?

```
DWORD GetIDListLengthWithExtra(LPITEMIDLIST pidl)
{
 DWORD cb =0;
 if (pidl) {
   /* this code is correct, with one exception noted below */
   while (pidl->mkid.cb) {
     cb += pidl->mkid.cb;    // add the size of this element
```

```
    // move to next element
    pidl = (LPITEMIDLIST)((LPBYTE)pidl + pidl->mkid.cb);
   }
   cb += 3; // Here's the extra!
 }
 return cb;
}
```

Let's look at those three extra bytes. Two of the bytes are expected; they are the size of the zero-terminator itself. But that third extra byte is gratuitous. As a result, the `CopyIDList` function allocates one extra byte and copies one extra byte. Copying one extra byte means running off the end of the buffer, and if you are sufficiently unlucky, that extra byte will cross a page boundary and the shell extension will crash.

In other words, the `CopyIDList` function is the moral equivalent of this incorrect string-duplication function:

```
LPSTR BadStrDup(LPCSTR pszSource)
{
 DWORD length = strlen(pszSource) + 2;
 PSTR pszResult = malloc(length);
 if (pszResult) memcpy(pszResult, pszSource, length);
 return pszResult;
}
```

Adding one to the return value of the `strlen` function is expected, since that accounts for the null terminator. But adding two? That just causes a buffer overrun during the `memcpy`.

When the operating system version number changes

THERE ARE MANY hazards to changing the reported operating system version number. We'll start with some examples. Time and again, we found programs that checked the Windows version number like this:

```
  UINT Ver = GetVersion();
  UINT MajorVersion = LOBYTE(uVer);
```

```
  UINT MinorVersion = HIBYTE(uVer);
  if (MajorVersion < 3 || MinorVersion < 10) {
   Error("This program requires Windows 3.1");
  }
```

Now consider what happens when the version number is reported as 4.0. The major version check passes, but the minor version check fails, because 0 is less than 10.

It would be one thing if the program merely displayed an error and exited. But some programs crashed so badly that it's clear that the code path was never even tested. Here's one example from a leading disk utility program:

```
int MainProgram()
{
  HINSTANCE hinstDll1, hinstDll2;

  if (CheckWindowsVersion()) {
    hinstDll1 = LoadLibrary("DLL1");
    hinstDll2 = LoadLibrary("DLL2");
    ... normal program operation ...
  } else {
    ... display error message ...
  }
  if (hinstDll2) FreeLibrary(hinstDll2);
  if (hinstDll1) FreeLibrary(hinstDll1);
  return 0;
}
```

If this program decides that it doesn't like the Windows version, it crashes instead of exiting cleanly. If you run it under the checked build of Windows, you get the ominous message

```
fatl K16 GlobalFree of SS. This call won't return.
```

before the program finally explodes. The reason, as you may have noticed, is that wherever the Windows version check fails, two uninitialized variables are passed to the `FreeLibrary` function. But the value was uninitialized in a consistent manner: The code that calls the `MainProgram` function also does other preparatory work and the memory that would become the `hinstDll1` variable has been used previously to hold a value corresponding to the program's

own stack. In other words, this program was doomed from the start; the code in the "`else`" branch had never been tested.

As we saw in Chapter 13, this type of bug was so prevalent that we gave up trying to identify every program that had the problem and just decided, "Fine. If anybody asks, say that the Windows version is 3.95."

MS-DOS applications had their own problems with changing version numbers. Such problems are surprising, considering that MS-DOS went through several major versions before it was finally retired. For example, one home productivity suite used the MS-DOS major version number to index a jump table but never did a range check on the version. The table went up to five. When the program was run under MS-DOS 6.0, it jumped to a random address and crashed.

Fortunately, MS-DOS had a utility known as `SETVER`, which could be used to instruct the operating system to report a false version number to particular programs, so it was a simple matter of adding the program to the list. The hard part is finding all those programs ourselves so that our customers wouldn't have to. The development team knows how to read assembly language and debug applications to determine where the compatibility problem lies and determine the appropriate course of action to get the application back on its feet. Your typical end users do not have this skill and have no interest in learning it. They just want to get their work done.

Changing the reported operating system version number is always a very tense but necessary step in shipping a new operating system. You throw the switch, knowing full well that hundreds of applications that had been running just fine will start to fall over, and you've just signed up the product team for thousands more hours in the application compatibility labs.

On the importance of bug-for-bug compatibility

When you changed the size of various window elements, the Windows 3.1 Control Panel broadcast the expected `WM_WININICHANGE` message but also

mistakenly sent out a spurious WM_SYSCOLORCHANGE message (which normally is sent to indicate that the user changed the *colors* of window elements), even though no colors changed. The Windows 95 Display Control Panel fixed this bug, only for us to find that a top-selling productivity application was relying on the old buggy behavior. The program ignored the WM_WININICHANGE message entirely, keying off the spurious WM_SYSCOLORCHANGE message to know when system metrics had changed.

Once this was discovered, the Windows 95 Display Control Panel had to add the extra unnecessary WM_SYSCOLORCHANGE broadcast to remain bug-for-bug compatible with Windows 3.1.

"Out of memory" is the generic error message

One of the common threads we've seen in application compatibility work is that if a program doesn't know what went wrong, it just assumes it's out of memory. For example, one company's productivity suite puts up "Out of memory" errors if you don't have a printer installed. But the strangest example of an unexpected "Out of memory" message I can think of is a handful of programs that display an "Out of memory" error because your scroll bars are too small.

Why did too-small scroll bars generate an "Out of memory" message? These programs have their own custom scroll bar window classes but choose the size of their custom scroll bars to match the system scroll bars. The reason that these programs use a custom scroll bar window class is that their developers enhanced the scroll bars with additional buttons that go next to the up and down arrows, and the images that go on these buttons are stored as resources at a variety of sizes. To load the correct-sized bitmap, the developers take the scroll bar size, do some math to convert it to a resource identifier, and load the appropriate bitmap. If your scroll bars are too narrow, then the result of the computation is a value that doesn't exist in the resources, and the call to LoadBitmap fails. The program interprets this failure as a low-memory condition, unaware that the real reason was that it passed a resource identifier that simply doesn't exist.

The fix: Change the Windows 95 Display Control Panel so that it doesn't let you set your scroll bar width so narrow that programs would encounter this problem.

Another example of "Out of memory" being used as a generic error message was found in another program from that same productivity suite. The program tries to read the name of a helper DLL from the registry, but if you choose a typical installation, this helper DLL is not installed. The program forgets to check the error code, though, and assumes the function succeeded. It then appends "32.DLL" to the end of what it thought was the string it retrieved from the registry but which is in reality just stack garbage, resulting in something like `y^ÅDP°32.DLL`. (Lucky for the developers, there is a garbage zero so that appending the string doesn't overflow the buffer.) The program then takes this garbage string and passes it to the `LoadLibrary` function, which rightfully fails.

You can see where this is headed. The program checks the error code set by the `LoadLibrary` function to see if it's on a list of "approved" errors; if not, the program changes it to `ERROR_NOT_ENOUGH_MEMORY`. Because "Out of memory" is the generic error message.

Misusing file format documentation

During the development of Windows 95, we discovered many programs whose developers confused the file format documentation with the interface documentation and decided that the best way to install programs into Program Manager was not to use the formal interface, but rather to manipulate the contents of the `PROGMAN.INI` file and `GRP` files directly. When Windows 95 replaced Program Manager with Explorer, these programs found themselves modifying the internal data structures of a program that no longer was running. If the programs had used the documented interface, they would have worked with Explorer just fine.

One program tried to use the documented interface, but didn't quite follow the protocol properly. The interface for creating items in Program Manager,

known as Dynamic Data Exchange, begins with a search for available providers, but the program decided that going through that process was too much work. Instead, it performed a `FindWindow` looking for Program Manager and assumed that that window was the provider. In Windows 95, there was no Program Manager anymore; the provider was Explorer. The developer who was responsible for implementing the Program Manager interface in Explorer summed it up in one sentence: "These people should not be allowed to write code."

To work around these problems with programs hard-coding Program Manager, another decoy was created: Explorer named its desktop window "Program Manager" so that it could act as the decoy for all these programs that really wanted to talk to Program Manager. This decoy quacked just enough like Program Manager to keep those programs happy.

But sometimes programs would go too far. One such program would locate the Program Manager window and minimize it. But when the desktop is standing in as a decoy for Program Manager, the program ends up minimizing the desktop, which is kind of exciting. (The fix was to add code to the desktop so that it ignores attempts to minimize it.)

Bypassing the operating system for no apparent reason

We found several programs that groveled directly into the `WIN.INI` file looking for fonts, and if they couldn't find their font, they died. The thing is, the use of the `WIN.INI` file to record the list of installed fonts is an implementation detail. If you want to look for fonts, just use the `EnumFonts` function.

Hunting around in the `WIN.INI` file has another downside: That's not where font list was stored anymore! In Windows 95, this information migrated to the registry, so programs that went spelunking in `WIN.INI` found nothing. To keep those programs happy, Windows 95 detects that a program is trying to get font information the wrong way and converts it to the right way. One program even insisted that TrueType fonts be reported as `.FON` files even though nearly all of them had shifted to `.TTF`, so the compatibility hack also

changes the names of the files as reported to the program. The program doesn't want to access the file; it just wants to check that the font it wants exists, but it checks with a hard-coded filename instead of using the face name.

Of course, the desire to bypass the operating system interface extends well past Windows 95. In Windows NT, we found a program whose menus appeared in an unreadably small font. The reason was that the program wanted to use the system menu font, but instead of using `SystemParametersInfo` with the `SPI_GETNONCLIENTMETRICS` flag to obtain that font, it went straight to the registry and read the binary data. What the program didn't realize was that Windows 95 and Windows NT store the information in different formats. It read the value from the registry and interpreted the value according to the Windows 95 format, unaware that Windows NT uses a different format.

Violating generally accepted laws of causality

Sometimes you run into a compatibility issue so awful you consider it a miracle that any programs run at all.

The setup for a major software productivity suite attempts to replace a system file with the version from the CD-ROM, even though the CD version is older than the version already on the system. The setup program doesn't bother checking whether the existing version is newer. Fine, Windows file protection is already on the lookout for programs that do this, and it'll restore the good copy of the file once the setup program finishes. That's not the bug.

If the file the program wants to replace is in use, then the program goes into an alternate installation mode, using the `MOVEFILE_DELAY_UNTIL_REBOOT` option to the `MoveFileEx` function so that the file will be replaced at the next restart of the system:

```
// Simplified for expository purposes
MoveFileEx("sysfile.new", "sysfile.dll", MOVEFILE_DELAY_UNTIL_REBOOT);
CopyFile("D:\\CDROM\\INSTALL\\sysfile.dll", "sysfile.new");
```

That's right; the program tries to move a file that doesn't exist yet. This actually worked on Windows NT because Windows NT's `MoveFileEx` function did not validate the existence of the source file if you passed the `MOVEFILE_DELAY_UNTIL_REBOOT` flag. The call failed on beta versions of Windows 2000 because of tighter parameter validation, and the program treated the failure of `MoveFileEx` as fatal and aborted the install.

The Windows 2000 version of `MoveFileEx` had to make a special exemption for delayed-moves of files that didn't exist. It allows them to succeed, in the hope that by the time the system reboots, the file will actually exist.

The strangest way of checking for success

I'm not quite sure what the developers were thinking, but there was a multimedia title that failed to run because of the way the program checked whether its calls succeeded: Instead of testing the `MMRESULT` against `MMSYSERR_NOERROR`, it asked the multimedia system to convert the error number to a readable text string and compared the string against "The specified command complete successfully." Well, actually, it only checked whether the first sixteen characters were "The specified co"—I guess somebody doing a code review decided to do some performance optimization.

Of course, this technique doesn't work very well if the system changes the precise wording of its error messages or if the end user is running a non-English version of Windows.

An even stranger-than-strangest way of checking for success

Okay, I lied. There's an even stranger way of checking for success. At least the program that converted the error code to text checked the error code. Another program used the Media Control Interface (MCI) to play videos and

tested for success by ignoring the return value entirely. Instead of checking the result of the `MCIWndOpen` call, the program retrieved the window text of the MCI window and compared it against the string "No Device" to determine whether the operation succeeded.

In Windows 95, the title of the window changed at a different point in the initialization of the MCI window, and that program found itself unable to open any files because it checked the title too soon.

Sometimes you're better off not checking for success

Meanwhile, there was a bug reported against a multimedia travel title. If you navigated through the program and asked to see a video of a particular point of interest, nothing happened.

Upon closer investigation, the reason was very simple: The vendor forgot to include the video on the CD.

I'm not sure why multimedia titles seem to be the class of programs that has the most problems with error checking.

How you can take a one-line function and break it

One day, one of my colleagues came into my office and said, "Hey, you want a sound card?" My computer didn't have a sound card, so I accepted his offer.

Turns out that it was a trick question. He gave me the sound card because he couldn't get it to work with Windows 95. Now I was the sucker with the broken sound card. It wasn't long before I saw why he wanted to be rid of it: It crashed the system randomly. Luckily, my computer ran the debugging version of Windows, so at least I had the proper setup to try to track down the problem.

The sound card driver took a one-line function that ships in the Device Driver Kit (DDK) and somehow managed to break it. The original function, called at hardware interrupt time, looks like this in the DDK:

```
void FAR PASCAL midiCallback(NPPORTALLOC pPortAlloc, WORD msg,
                             DWORD dwParam1, DWORD dwParm2) {
  if (pPostAlloc->dwCallback)
    DriverCallBack(pPortalloc->dwCallback, HIWORD(pPortalloc->dwFlags),
                   pPortalloc->hMidi, msg, dwParam1, dwParam2);
}
```

Their version of the function looked like this:

```
void FAR PASCAL midiCallback(NPPORTALLOC pPortAlloc, WORD msg,
                             DWORD dwParam1, DWORD dwParm2) {
  char szBuf[80];
  if (pPostAlloc->dwCallback) {
    wsprintf(szBuf, " Dc(hMidi=%X,wMsg=%X)", pPortalloc->hMidi, msg);
#ifdef DEBUG
    OutputDebugString(szBuf);
#endif
    DriverCallBack(pPortalloc->dwCallback, HIWORD(pPortalloc->dwFlags),
                   pPortalloc->hMidi, msg, dwParam1, dwParam2);
  }
}
```

Not only is there leftover debug stuff in retail code, but it is calling a non-interrupt-safe function at hardware interrupt time. If the `wsprintf` function ever gets discarded, the system will take a segment-not-present fault inside a hardware interrupt, which leads to a pretty quick death. Even when you get lucky, the function modifies the high word of extended registers, whereas the hardware interrupt handler saved only the low word. Doing this across a hardware interrupt means that whatever code was interrupted is in for a rude awakening when the hardware interrupt returns. (The fix for this driver was to patch out the call to `wsprintf`.)

And somehow this sound card managed to receive an award from a major PC magazine.

What's worse, the vendor of the sound card wanted to have its updated drivers included with Windows 95. When we received the drivers from the

vendor, they still had that crashing bug, the very bug we had persuaded the vendor's engineers to acknowledge months earlier when they granted permission to patch their driver. As you might suspect, we rejected the "updated" drivers.

The intimate parasite of Sound Recorder

The multimedia team found a "Learn to speak English" program for native speakers of another language, let's say, Italian. The program was a very intimate parasite of 16-bit Sound Recorder. The more you studied it, the deeper the problem went.

The first problem was relatively straightforward: The program wants to launch Sound Recorder and uses the 16-bit filename, `SOUNDREC.EXE`. Windows 95 converted the Sound Recorder to a 32-bit program and correspondingly renamed it to `SNDREC32.EXE` to be consistent with Windows NT. A simple matter of renaming the binary back to its 16-bit name took care of this. One layer of the onion has been peeled.

After fixing that first problem, the program still didn't work. After launching Sound Recorder, the program goes looking for the window by its caption. First, it tries the English title (in case you're running English Windows), then the Italian title (since most of their customers are running Italian Windows). Hey, at least the program doesn't assume English exclusively. Of course, native Italian speakers who live in France and wish to learn English are out of luck, but that's not our problem. Windows 95 changed the window caption to include the name of the file being recorded—this change made the program unable to find the copy of Sound Recorder it had just launched.

We changed the 16-bit Sound Recorder so that its window caption exactly matches the name it had in Windows 3.1, and the program finally manages to run. And then we found that the program never even uses Sound Recorder.

If it never uses Sound Recorder, why does it care so much about it?

Well, it does use Sound Recorder, but only under certain conditions. For normal usage, it uses a custom control to play sounds, but if you don't have a sound card, it ships the SPEAKER.DRV driver, which plays sound out the tinny PC speaker. This is a synchronous driver, and the program's custom control doesn't support synchronous drivers, so the programmers hunted around for something that does, and they found Sound Recorder. Send some keystrokes to Sound Recorder to control playback, and there you have your synchronous playback control.

But if you watch carefully, you can see a copy of Sound Recorder flash onto the screen for a split second, and then it turns into a sliver at the left-hand side of the screen. What's going on?

The programmers didn't want that copy of Sound Recorder to be visible. But because they didn't know how to hide a window, they did the next best thing: They moved it off the screen. Well, mostly off the screen. Because saying that it "moves it off the screen" gives the program too much credit. What the programmers actually did was simulate mouse clicks and mouse motion in order to *drag* the Sound Recorder window across the screen until it has gone as far to the left as it can, at which point they figure, "Well, it's off to the side. Hopefully, nobody will notice."

One way to make sure users follow the instructions

THE ROOMMATE OF one of our developers bought a productivity application that kept crashing when he launched it. The developer and his colleagues went through some of the standard troubleshooting steps—changing video drivers, trying it on another machine—but nothing worked. Having run out of ideas, they called the vendor, and they got an explanation.

The vendor's product support told him that if you don't fill in all three fields in the setup program (name, organization, and serial number), the setup completes fine, but the program crashes when you try to run it. And this is not by design as a form of copy protection. It's a known bug.

Masking a bug by introducing an even worse bug

A LARGE, WELL-KNOWN company wrote a program that it sends to its major customers so that the customers can place orders and track the status of the order by dialing into one of the company's mainframes scattered throughout the country. (This was in the pre-Internet days, of course.) The company's version 1.00 had a serious bug: The programmers forgot to disable the ninety-day time bomb. After ninety days, the program didn't simply stop working, though. It intentionally crashed your entire machine. Version 1.01 came out shortly thereafter.

A beta site reported that when it ran the program, the Windows 95 machine died horribly if the testers placed a certain type of complicated order. The first step was getting the company to send the application compatibility team a copy of the program and an account number so that the program can be put through its paces. Somehow, a key piece of information was lost in this process: Unknown to the development team, the account number the vendor sent was not a test account but a real live account! It took two weeks to straighten that one out, and a lot of people doubtless gained a few extra gray hairs in the process.

Anyway, back to the bug. Sure enough, the problem was reproduced in our compatibility labs. But in fact, the problem occurred even with Windows 3.1. So it wasn't a Windows 95 compatibility bug.

Still, you're probably wondering what the problem was. It so happened that when you placed that special type of complicated order, the window focus ended up in a different location from where it normally would be, but the program assumed that this focus was always on the list of unprocessed selections. As a result, the program crashed with a null pointer access.

Okay, but why did that crash the entire machine? Well, whoever wrote the program saw that the program was crashing a lot and couldn't figure out why, so the person installed a systemwide fault hook to mask the problem. (Sixteen-bit Windows did not have structured exception handling. If you

wanted to trap faults, you had to install a systemwide fault hook and then, in the hook, decide whether the particular fault is the one you're interested in.) The programmer set a flag at the beginning of the function that tended to crash a lot and cleared it when the function exited. The fault handler checks the flag, and if set, it performs the equivalent of a `longjmp` back to the function. The function tries to clean up from the mess; it misses a few places, so there's a small memory leak, but at least it doesn't look like the program crashed.

But what if the flag isn't set? If a fault happens anywhere outside that function, the fault handler tries to clean up anyway by freeing all the libraries that the program uses. There's a bit of a problem, though. System fault hooks need to be written in assembly language because the parameters are passed to them in a very unusual way. Too bad the programmer wrote it in C++, indeed, not just C++ but C++ in an application framework. The framework "helpfully" made the function a 16-bit DLL callback function, which means that it loads the `DS` register (which points to the data the program was accessing when the fault occurred) on entry and restores it on exit.

That's where the problem lies. The fault handler frees all the libraries that the program was using; if the fault happened inside the program, then on entry to the fault handler, the `DS` register points to the program's data. After the fault handler frees all the libraries that the program was using, it restores the `DS` register as part of the function exit. But we freed the program's data; the attempt to restore the `DS` register raises a nested fault. This invokes the fault handler, and the nightmare starts all over again.

But wait, it gets even better (or worse, depending on your point of view). In addition to unloading the libraries that the program was using, it also unloaded USER, KERNEL, and GDI, the three modules central to 16-bit Windows. As a result, each time through the recursion, the reference count on each of those modules decrements, and eventually they are freed one by one. Ultimately, the recursive death gets so bad that kernel is forced to step in and display a fatal error message. But it can't, because the fatal error message is in USER, which itself already got unloaded. Now in a horrific panic, Windows finally puts up a blue screen error and forces a reboot.

One of those cases where the remedy is far worse than the original problem.

(As a final remark: If you call the company's product support and describe a problem you've encountered with the program, the response is always, "Yeah, we're really sorry about that bug.")

If you don't have the right object, just cast it

One program was kind of confused. It wanted to extend a shell property sheet, so it installed a shell extension. A shell extension adds a property sheet page by creating the page with the `CreatePropertySheetPage` function and passing the resulting `HPROPSHEETPAGE` to the callback function provided by the `IShellPropSheetExt::AddPages` method. At least that's the theory.

This particular shell extension must have missed that little detail of how to create property sheet pages, so instead of calling the `CreatePropertySheetPage` function, it just took a `PROPSHEETPAGE` structure and cast it to a `HPROPSHEETPAGE`. By an amazing stroke of luck, the program actually ran on Windows 95 because the internal layout of an `HPROPSHEETPAGE` happens to be 95% compatible with that of a `PROPSHEETPAGE`; the remaining 5% resulted in error messages in the debugging version of Windows, which the authors of this program no doubt ignored.

On Windows NT and Internet Explorer 5, the structures are significantly different, and the difference caused the program to crash. As a result, the common controls library that comes with Internet Explorer 5 contains a special check to see whether the `HPROPSHEETPAGE` it was passed really is an `HPROPSHEETPAGE`. If it turns out to be one of these `PROPSHEETPAGE`s masquerading as an `HPROPSHEETPAGE`, the common controls library converts it to a genuine `HPROPSHEETPAGE` by calling the `CreatePropSheetPage` function—the very function the program was supposed to have called in the first place.

But wait. If this method didn't work on Windows NT, how did the program ever ship in the first place? Simple. If you try to install this program on Windows NT, an error message pops up that reads, "Your version of Windows does not have the Explorer shell supported by Program X. Program X cannot

be installed." In other words, the program never supported Windows NT and didn't even try.

So much for convergence.

Locating items by blind counting

A PROBLEM WAS reported on a program that tries to be an Explorer clone with a tree view on the left-hand side showing the folders in the system and a content viewer on the right-hand side. In the program's toolbar are buttons that quickly move you to each of the drives in the system. But when you run the program under Windows 2000, the buttons take you to random folders instead of to the root of the corresponding drive. What is happening?

Let's say you clicked on the button that says "Go to the C: drive." The program responds by starting at the Desktop folder, navigating to the first item under the desktop, which it "knows" is the My Computer icon, then navigating to the third item under My Computer, which it "knows" is your C: drive.

Except that starting in Windows 2000, the order of the icons on the desktop changed. My Documents is the first icon on the desktop, and My Computer moved to the second position. Now, the program goes to the first item under the Desktop folder and finds My Documents instead of My Computer. Undaunted, it continues to the third child under My Documents and concludes that this is your C: drive.

The fix for this was to add an application compatibility flag that forced the order of items on the Desktop folder to place My Computer first.

Passing the wrong "this" pointer

DURING THE DEVELOPMENT of Windows 2000, we found a program that passed the wrong `this` pointer to the `IUnknown::Release` method. It turns out that you have to do quite a bit of work to get this wrong. The C++ language passes the `this` parameter automatically, which removes the opportunity to

get it wrong. The program must have been written in C, and the authors must have avoided the interface helper macro `IShellFolder_Release`, because that macro also passes the correct `this` parameter automatically:

```
#define IShellFolder_Release(This)      \
        (This)->lpVtbl -> Release(This)
```

No, the program must have written out the `Release` call manually:

```
IShellFolder *psfDesktop;
IShellFolder *psfMyComputer;
...
psfDesktop->lpVtbl->Release(psfMyComputer); // oops
```

Notice that it calls the Release method on the Desktop folder, but passes a pointer to the My Computer folder as the `this` pointer. There lies madness. (Maybe they thought it would release both objects with a single call?)

A "signature" field was added to the Desktop folder, and a special check was added to the Release method to check the signature to ensure that the `this` pointer really is the Desktop folder. If the signature is missing, then the call is ignored. So not only does the program no longer crash, it actually runs better than it did before! (Before, it just corrupted some memory.)

Success through sheer luck

One of the optional fields in a `PROPSHEETPAGE` structure is the `pfnCallback`, which provides a function that the property sheet manager will call at various stages in the property sheet lifetime. Originally, the only two messages were `PSPCB_CREATE` and `PSPCB_RELEASE`. In Windows 2000, a new message, `PSPCB_ADDREF`, was added.

When we added the `PSPCB_ADDREF` message, we found that a very popular server program crashed. On closer inspection, we found that its callback function went like this:

```
UINT CALLBACK PropSheetPageProc(HWND hwnd, UINT uMsg, LPPROPSHEETPAGE
                                ppsp)
{
```

```
 DoSomething();
 return 0;
}
```

Notice that the function completely ignores the message number! In fact, the developers were only interested in PSPCB_CREATE, but they did it on the release, too. So how did this ever work? Why did sending the PSPCB_ADDREF message put it over the edge?

It so happens that what the callback function did in the DoSomething() function was install a thread hook. Even before the addition of the PSPCB_ADDREF message, the callback installed two thread hooks by mistake. When the property sheet is closed, the program cleans up one hook but leaves the second (mistaken) one installed. Luckily for the program, it also exits the thread immediately afterward. As a result, the leaked hook applies to a thread that no longer exists and therefore never fires. The program got lucky.

But this doesn't explain why it crashes once we add the PSPCB_ADDREF message. All you would expect is that the program would install three hooks (two inadvertently), but that the second accidental hook would be leaked just like the first one. Why the crash?

The answer is that the program creates its property sheet pages on one thread but displays them on another. The PSPCB_ADDREF message is sent when the property sheet page is created (and therefore sent on the thread that called CreatePropertySheetPage), as opposed to the other two, which are called on the thread that is displaying the property sheet (one when the dialog is created and the other when the page is destroyed). And that's the key. The thread that creates the property sheet page is a persistent thread, not a temporary one. Therefore, the hook that was inadvertently installed by the program when it received the PSPCB_ADDREF message refers to a thread that is still alive. Once that thread does something that triggers the hook, the program crashes.

The fix is to send the PSPCB_ADDREF message only if the dwSize member of the PROPSHEETPAGE structure is greater than or equal to PROPSHEETPAGE_V2_SIZE, thereby indicating that the program was specifically designed for the Windows 2000 property sheet manager and therefore will not do something stupid in response to the new PSPCB_ADDREF message.

Getting away with a corrupted stack

The creators of one program decided that creating a shortcut on the desktop wasn't annoying enough. Instead, they created a shell namespace extension on the desktop. But not a full shell namespace extension. Rather, the icon on the desktop implements just barely enough of the shell namespace requirements that double-clicking the icon launches the associated program. On Windows 2000, double-clicking the icon caused Explorer to crash. The reason was that the shell extension implemented the `IShellFolder::CreateViewObject` method with the wrong function signature. The declaration should have been

```
STDAPI CShellFolder::CreateViewObject(
    HWND hwnd, REFIID riid, void **ppv)
```

but instead, the programmers responsible for this shell extension wrote

```
STDAPI CShellFolder::CreateViewObject(HWND hwnd)
{
 return S_OK;
}
```

Observe first that the function returns success without returning an output pointer in the `*ppv` parameter, which means that the caller thinks the call succeeded yet received garbage as the result. But that's assuming the caller even gets that far. Since the parameter lists don't match, the function returns to the caller with a corrupted stack.

The incorrect implementation of the `IShellFolder::CreateViewObject` method was masked in earlier versions of the shell because the shell had been compiled with frame pointers. Frame pointers are a redundant technique for keeping track of where a function's parameters are, but it was used in the shell because it made debugging much easier. Debuggers back in 1993 were not as sophisticated as they are now. Nowadays, frame pointer omission doesn't give debuggers much of a hiccup at all. When the shell was compiled with frame pointers, the safety net of the frame pointer forgave limited classes of stack corruption, this being one of them.

In Windows 2000, the frame pointer omission optimization was enabled in the shell, and this shell extension ended up crashing Explorer because the frame pointer that covered the mistake no longer existed. The fix was to change the way the shell called the `IShellFolder::CreateViewObject` method. Instead of just calling it as a normal method call

```
hr = psf->CreateViewObject(hwnd, IID_IShellView, (void**)&psv);
```

the call was wrapped inside a helper function:

```
hr = SafeCreateViewObject(psf, hwnd, IID_IShellView, (void**)&psv);

STDAPI SafeCreateViewObject(IShellFolder *psf, HWND hwnd,
                            REFIID riid, void **ppv)
{
 *ppv = NULL; // in case shell extension forgets
 _asm mov ebx, esp // protect against wrong calling convention
 HRESULT hr = psf->CreateViewObject(hwnd, riid, ppv);
 _asm mov esp, ebx // restore stack pointer
 // cover for shell extensions that return S_OK but *ppv == NULL
 if (SUCCEEDED(hr) && *ppv == NULL) hr = E_NOINTERFACE;
 return hr;
}
```

This helper function keeps an eye out for three different bugs common in shell extensions. First, it sets the output pointer to `NULL` before even calling the `IShellFolder::CreateViewObject` method because, as we just saw, the shell extension that corrupted the stack forgot to initialize it to anything. Second, the function uses inline assembly to recover from the stack corruption caused by shell extensions that misdeclared the function prototype for the `IShellFolder::CreateViewObject` method. And third, it covers for shell extensions that return success without returning an interface pointer, turning the success back into the failure code it should have been.

But why did this program install a shell namespace extension when a desktop shortcut would have done just as well? My only guess as to why programmers went to all this trouble was that on earlier versions of Windows, deleting namespace items from the desktop was harder than just deleting a file. In other words, this program just wanted to be extra annoying.

Relying on spurious or obsolete messages

YOU MIGHT NOTICE that you will often get `WM_PAINT` messages where the `rcPaint` is empty. In other words, you're being asked to paint nothing. Why does the window manager send you meaningless messages?

Because oodles of programs rely on these meaningless messages. Earlier versions of Windows sent paint messages at particular points during the program's execution, but which as the result of changes to the window manager no longer require painting. Programs designed for earlier versions of Windows relied on those messages—if they didn't get the messages, the programs produced incorrect output or even crashed, because they were counting on the messages to tell them when it's time to initialize a variable or allocate an object. Suppress the message because it's no longer needed (from the window manager's point of view), and these programs end up using something before it's been created, dividing by zero, or positioning their screen elements incorrectly. For example, if you launched one of these programs as a minimized window, then right-clicked on its taskbar, the program might crash because its system menu handler divided by a global variable whose default value is zero, but which gets set to a nonzero value during `WM_PAINT` handling. To avoid this problem, the window manager sends these messages to windows even though they appear to accomplish nothing.

Other examples of spurious messages that are nevertheless sent for compatibility reasons are the messages related to the appearance of minimized windows in Windows 3.1 and earlier. In earlier versions of Windows, there was no taskbar. Instead, minimized programs appeared as an icon on the desktop, and programs received special messages like `WM_ICONERASEBKGND` and `WM_PAINTICON`, which allowed them to customize the appearance of their minimized icons. Windows 95 continued to send these messages to programs designed for earlier versions of Windows, even though there was nothing to erase or paint, because many programs (some by major manufacturers) relied on those messages to trigger essential program functionality.

Identifying dependencies on message ordering is probably the hardest part of maintaining compatibility when making changes to the window manager. Thousands of messages are sent during the lifetime of a program, and any of them can cause the program to keel over if you send it at precisely the wrong time, send it out of order, or fail to send it at all.

Using the order in which items paint to mask your bugs

The Rebar control is a comparatively little-known control that acts as a container for toolbars or similar controls and allows the user to use drag-and-drop to rearrange them. Windows XP uses it, for example, in Explorer's toolbar, where the Rebar control contains a menu band, the forward and back buttons, and the address bar, all of which can be moved and resized. In Windows 2000, we added a call to the `UpdateWindow` function to the Rebar control after it repositioned the children so that the pixels were nice and fresh as you fiddled the size. This made resizing look snappier and more responsive. But unfortunately, it also created problems.

There were many programs that used a Rebar control but forgot to set the `WS_CLIPCHILDREN` style on the Rebar's parent window. Omitting the `WS_CLIPCHILDREN` style means that the parent can paint over its children, and that's what these programs did, entirely inadvertently. During a resize, the Rebar would repaint itself with the new configuration, and then the parent window would erase its background, thereby erasing the Rebar that had just been resized. Result: a blank window. Without the call to `UpdateWindow`, the parent window would erase its background first (corrupting the Rebar), and then the Rebar would get its `WM_PAINT` message and paint itself properly.

The fix was to remove the call to the `UpdateWindow` function, returning the Rebar to its previous behavior. Though the result wasn't as snappy as it could be, that's the price you pay for compatibility.

Accidentally freeing yourself

A MAJOR NETWORKING provider wrote a shell extension that it attaches to the property sheet of network drives so that the provider can display additional information about the server. And starting with Windows 2000, Explorer crashed as soon as you dismissed the property sheet. Here's why: The provider's shell extension class went something like this:

```
ShellExtension::~ShellExtension()
{
 --g_cObjects;
 CoFreeUnusedLibraries();
}
```

Ignore the multithreaded race condition in the `--g_cObjects` for now. The reason for the crash is that the extension calls `CoFreeUnusedLibraries` in its destructor. When the last shell extension is destroyed, the call to `CoFreeUnusedLibraries` will call the DLL's `DllCanUnloadNow`, which will return `S_OK` because the DLL object count is now zero. OLE will then proceed to unload the DLL, and when the call to `CoFreeUnusedLibraries` returns, you find yourself trying to execute code from a DLL that is no longer loaded, which crashes. In other words, this code was fatally flawed in its design—the code is intentionally trying to crash itself. Yet this code ran fine on Windows 95 and Windows NT. How is that?

Recall that the shell in Windows 95 and Windows NT used "mini-COM" rather than the real COM. Consequently, the load of the shell extension was done by mini-COM, and the call to `CoFreeUnusedLibraries` did not unload the shell extension DLL, because COM didn't know about it. But the shell's mini-COM was discarded starting in Windows 2000; the shell used real COM exclusively. As a result, when this shell extension called `CoFreeUnusedLibraries`, it ended up freeing itself.

The fix was to teach the shell to recognize this particular shell extension, and when the shell creates the extension, it does an extra `LoadLibrary` on the DLL, so that when COM unloads it in response to `CoFreeUnusedLibraries`, the

DLL stays in memory. Yes, it means that the DLL, once loaded, will never be unloaded, but it's better than crashing.

This is why the documentation for the `CoFreeUnusedLibraries` points out that applications—and not extensions—call this function, because the EXE of a running program will not be unloaded.

Relying on a function that does not modify unused output parameters

THE FIRST EXAMPLE in this category comes from a program that passed the wrong flags and worked by mistake. In fact, it did so many things wrong, it's amazing that the program ever worked! The program starts by requesting the icons for various items in the shell so that it can use them in its tree view.

```
SHFILEINFO sfi;
// assume that the program has already used the SHGetFileInfo
// function
// to obtain the folder icon in sfi.iIcon
LPITEMIDLIST pidl;
SHGetFileInfo("x", FILE_ATTRIBUTE_DIRECTORY, &sfi, sizeof(sfi),
              SHGFI_USEFILEATTRIBUTES | SHGFI_OPENICON | SHGFI_ICON);
OpenIcon = sfi.iIcon;
```

The program asks for an open folder, but passes the `SHGFI_ICON` flag instead of the `SHGFI_SYSICONINDEX` flag. As a result, the `SHGetFileInfo` function returns an `HICON` in the `hIcon` member rather than the system image list index in the `iIcon` member, and by a stroke of good fortune on the part of this program, the `SHGetFileInfo` function does not modify the `iIcon` member. Consequently, the value that was left over from a previous call to the `SHGetFileInfo` function remains there, and the program ends up using the folder icon when it really wanted the open folder icon. This bug can even be seen in the product's advertising screen shot! The screen shot shows a tree view with a folder node selected and expanded, but the icon for the expanded folder shows a closed folder rather than an open one. The bug was staring them in the face the whole time, and no one noticed.

This program ran great on Windows 95 but displayed garbage on Windows NT. Why was that?

On Windows 95, the `SHGetFileInfo` function is implemented as an ANSI function, and the result of the function call was stored directly into the buffer provided by the caller. On Windows NT, however, the `SHGetFileInfo` function is implemented as a Unicode function, and the ANSI version is merely a thin wrapper. This wrapper converts the filename from ANSI to Unicode and then calls the Unicode `SHGetFileInfo` function with a temporary Unicode `SHFILEINFO` structure. When the Unicode function returns, the ANSI version of `SHGetFileInfo` converts the temporary Unicode `SHFILEINFO` structure back to ANSI, putting the result in the buffer provided by the caller. This is a standard pattern for many ANSI wrapper functions: Convert the incoming parameters to Unicode, call the Unicode function, then convert the Unicode result back to ANSI.

Perhaps you can now figure out why the program got a garbage icon on Windows NT. The temporary Unicode `SHFILEINFO` structure was not initialized, since the `SHFILEINFO` is an output-only structure. (Starting in Windows 2000, the `SHFILEINFO` structure actually is an in/out parameter if you pass the `SHGFI_ATTR_SPECIFIED` flag, but this story takes place in the Windows NT era, when the structure was indeed output-only.) After the function returned, the wrapper function copied the Unicode `SHFILEINFO` structure to the ANSI structure, including the garbage values that the function didn't set, because the caller didn't ask for them to be set to anything in particular. As a result, the `iIcon` received by the program is stack garbage.

The fix is to modify the wrapper so it copies the caller's `iIcon` member to the temporary Unicode `SHFILEINFO` structure, so that when it is copied back, the original value is restored if the `SHGetFileInfo` function didn't change it.

(Those with more experience with the `SHGetFileInfo` function may have noticed another bug: Since the program inadvertently requested an `HICON` instead of a system image list index, it neglected to destroy the returned icon, thereby leaking it.)

Failing to set all your output parameters

The IDataObject::GetData method returns the requested data in a structure known as STGMEDIUM. One of the members of that structure is called pUnkForRelease, which describes how the memory described by the STGMEDIUM structure should be freed. The full details of how this member works aren't important; what's important here is that in the case where the STGMEDIUM represents a block of memory, the pUnkForRelease member can be set to a non-NULL value, in which case the STGMEDIUM is freed by releasing that pointer. If the member is NULL, then the STGMEDIUM is freed by freeing the underlying memory directly. (This is all handled automatically by the ReleaseStgMedium function.)

One shell extension provided a data object which failed to set the pUnkForRelease member to any particular value; as a result, the value of the member upon return was garbage, and when the shell wanted to free the memory in the STGMEDIUM, it tried to call IUnknown::Release on a garbage pointer and crashed. The shell extension presumably worked in the past only because the memory that was being used for the pUnkForRelease member was zero by sheer luck. Perturb the system in the slightest manner, and uninitialized stack garbage changes, resulting in the return of a nonzero garbage value in the pUnkForRelease.

The fix for the shell is to set the pUnkForRelease explicitly to NULL before calling the IDataObject::GetData method. That way, when a shell extension forgets to set the member to any particular value, the value that is returned ends up being NULL, which is what the shell extension probably wanted in the first place.

Passing garbage to a function and expecting something reasonable to happen

The SHGetPathFromIDList function takes what is known as an *absolute pidl*. A pidl (pronounced "piddle") is a data structure used to describe an object

in the shell namespace. My Computer has a pidl, as does the Control Panel, as does every file on your machine. You can think of a pidl as a sort of filename, but one that can be used to describe things that aren't files. As with filenames, there are two types of pidls: relative and absolute. Relative pidls describe a shell object relative to a folder in the same way that a relative filename describes a file relative to a current directory. Absolute pidls describe a shell object relative to the desktop, which acts as the root of the shell namespace.

It so happens that one shell extension didn't quite understand the distinction between relative and absolute pidls, because it passed pidls relative to the shell extension's root folder (rather than relative to the desktop) to the `SHGetPathFromIDList` function. It's as if you asked somebody, "How do I get from City Hall to your house?" and the person gave you directions starting at the town library instead. "Go one block south, turn left, go straight, turn right at the donut shop, go to the third house on the right." The `SHGetPathFromIDList` tried to interpret the block of memory as if it were an absolute pidl, but because of the faulty "directions," it wound up at the completely wrong place. But by some miracle, there was still a house there, and `SHGetPathFromIDList` returned a path to what it found. The result was garbage, of course, since the directions had the wrong starting point, but it turns out that it was always the same garbage, and the program relied on the fact that the garbage contained a particular character. (Indeed, all the program cared about was that the garbage returned by the function contained that one character! It didn't use the return value for anything else.)

In Windows 2000, the `SHGetPathFromIDList` function noticed that the directions it got were nonsense. "There's no donut shop at the corner. Something's wrong." As a result, it returned failure. Unfortunately, the shell extension responded to the inability to get a path to garbage by failing to initialize itself. The fix was to detect the specific pidl that the program passed and return the same garbage that previous versions of Windows did.

A different program passed a malformed pidl to the `SHGetPathFromIDList` function. Instead of passing an absolute pidl, it passed two absolute pidls glued together. It's as if the same person who was asked for directions gave the directions and then repeated them: "Go one block south, turn left, go straight, turn right at the donut shop, go to the third house on the right, go one block

south, turn left, go straight, turn right at the donut shop, go to the third house on the right." Again, the program managed to work in earlier versions of Windows because by some miracle, there was actually a house there when this confused set of directions was followed to completion, and what's more, the house that it found looked similar enough to the house it wanted that the program was satisfied with the result! However, it so happens that the shell would crash if you passed a specific type of invalid "double-directions," so in Windows 2000, the crashing bug was fixed so that the shell would detect the invalid directions and return failure. Which, of course, led to the program's crashing.

In other words, the program was relying on a crashing bug in the shell.

Again, the fix was to detect this program and reintroduce the behavior it was relying on (without the crash) just so it could get what it wants.

Shell extensions that mess up IClassFactory::CreateInstance

The IClassFactory::CreateInstance method is called to create an object; you'd think this would be a relatively straightforward process, but it's disappointing how often people get it wrong.

The final two parameters to the `IClassFactory::CreateInstance` method are the interface being requested and an output pointer. The interface parameter specifies the interface that the caller will use to communicate with the created object but does not have any effect on the object itself. In other words,

```
IInterface *p = NULL;
hr = pFactory->CreateInstance(NULL, IID_IInterface, &p);
```

is equivalent to

```
IInterface *p = NULL;
IUnknown *punk;
hr = pFactory->CreateInstance(NULL, IID_IUnknown, &punk);
if (SUCCEEDED(hr)) {
 hr = punk->QueryInterface(IID_IInterface, &p);
 punk->Release();
}
```

Put another way, when you create an object and specify an interface, it's the same as creating the object with a generic interface (`IUnknown`) and then asking the object for the specified interface.

In theory, the `IClassFactory::CreateInstance` method could have hard-coded the interface ID to `IID_IUnknown`, but the interface is passed to permit two optimizations:

- It cuts the number of calls to the class factory in half. This is a performance win if the class factory is slow to access. For example, it may reside in a different apartment, in another process, or even on a different computer entirely.
- It permits the class factory itself to reject the creation of an object if an unsupported interface is requested. This is a performance win if the object takes a long time to create. If the caller is asking for an interface that the object doesn't support anyway, there's no point in creating the object, only to have the call to `IUnknown::QueryInterface` fail.

One shell extension didn't quite understand the equivalence of these two forms of call to `IClassFactory::CreateInstance`. If you called the shell extension to create an object and asked for the `IShellFolder` interface,

```
IShellFolder *psf;
hr = CoCreateInstance(clsidExtension, NULL, CLSCTX_INPROC,
                      IID_IShellFolder, (void**)&psf);
```

then the object was created and the `IShellFolder` interface returned. However, if you changed `IShellFolder` to any other interface, the call not only failed, but also crashed Explorer by raising a debugging exception.

The first attempt to fix this problem was to change the way the shell creates shell namespace extensions. Instead of passing the desired interface, it always creates them with the `IShellFolder` interface and manually converts them to the actual interface desired with a separate call to `IUnknown::QueryInterface`:

```
HRESULT SHCreateNamespaceExtension(REFCLSID rclsid, REFIID riid,
void **ppv)
{
 IShellFolder *psf;
 HRESULT hr = CoCreateInstance(rclsid, NULL, CLSCTX_INPROC,
                               IID_IShellFolder, (void**)&psf);
 if (SUCCEEDED(hr)) {
  hr = psf->QueryInterface(riid, ppv);
  psf->Release();
 } else {
  *ppv = NULL;
 }
 return hr;
}
```

In other words, the function undoes the optimization that was the whole point of passing the interface as an explicit parameter.

While this fix did avoid the crash, the performance of the shell extension became insufferable because creating the object was indeed a very expensive operation. Opening the shell extension slowed performance from two seconds to thirty seconds because the shell was querying for interfaces other than `IShellFolder` rather often. As a result, the many copies of the underlying object were being created and immediately thrown away.

Thus, a second round of fixes was required. An application compatibility flag was invented just for this shell extension to tell the shell that the object supports only `IShellFolder`, and that if the shell wanted anything other than `IShellFolder`, it should just fail immediately instead of creating the object and waiting for the failure to come out of the `IUnknown::QueryInterface` method.

There was another program whose implementation of `IClassFactory::CreateInstance` created an instance of the folder if the requested interface was `IShellFolder`. But if the caller asked for any other interface, it returned `S_OK` and a `NULL` pointer. That is, the shell extension flat-out lied. The status code said, "Yes, I created your object," but the resulting pointer was invalid. Explorer now checks both that the call to `IClassFactory::CreateInstance` succeeded and that the returned pointer is non-`NULL`.

Who would host a shell extension other than Explorer?

MORE THAN ONCE we've run across shell namespace extensions that assumed that they were hosted in Explorer. While Explorer is the most common host for shell namespace extensions, it is hardly the only one. The common file open and save dialogs host the shell namespace, and any application can use the shell namespace directly if it so chooses.

One of the methods that a shell namespace extension implements is the `IShellFolder::EnumObjects` method. This method is called to enumerate the contents of a folder, and one of the parameters to this method is a window handle the method can use if it needs to interact with the user, for example, if it needs to prompt for a password.

One shell extension took this window handle and used it as the starting point for a search through the window hierarchy looking for a Rebar control. Once the shell extension found one, it reached inside, grabbed what it thought was the Address bar control, and hid the control.

The authors of this shell extension didn't realize that the window handle that is passed to the `IShellFolder::EnumObjects` method can be `NULL` to indicate that any user interaction should be suppressed. One of the functions that the shell extension used to hunt through the window hierarchy was `FindWindowEx`. As luck would have it, if you ask `FindWindowEx` to search starting from `NULL`, it searches through all the top-level windows in the system. As a result, the shell extension would sometimes stumble across the Rebar control that belonged to a completely unrelated window and try to hide the Address bar in the other process! It did this by sending Rebar control messages across processes. Since different processes have separate address spaces, the pointers in the message end up being garbage in the destination process. The result is that whenever the shell asked the shell extension to enumerate its contents silently (for example, when the shell wants to populate an auto-complete dropdown list), a random program in the system crashed.

The fix was to detect this shell extension and make sure never to pass it `NULL` as the window handle for enumeration.

But why did this shell extension want to hide its Address bar, anyway? It so happens that the authors of the shell extension didn't implement the `IShellFolder::ParseDisplayName` method. Among the things this method can be called on to do is taking something the user has typed into the Address bar and producing the object with that name. Since the shell extension didn't implement this method, the authors didn't want the user to be able to type anything into the Address bar (thereby exposing their incomplete shell folder implementation). They figured it was easier to write code to hunt for and hide the Address bar than to implement the method in the first place.

Another program took the window handle passed to the `IContextMenu::InvokeCommand` method (which is used to execute commands on context menus) and assumed that it was always the main Explorer window. It groveled into undocumented data associated with the window, treated what it found as a pointer, and dereferenced it. The authors of this program apparently never tried right-clicking an item in the folder tree and choosing a command from that popup menu, because if they did, they would have found that their shell extension crashed instantly.

Punishing users who aren't administrators

IT IS WELL known that many poorly written programs require administrator privileges for no good reason outside of laziness on the part of the programmer. Typically, however, most programs will at least report the problem before anything bad happens. But not all programs.

One program requires write access to a registry key in the local machine section of the registry, a key that requires administrator privileges to modify it. If the program fails to get write access to that key, it doesn't display any error message. It just runs normally, but if you try save your work, the program actually deletes the file!

Implementing only part of an interface and ensuring nobody tries anything more

When you use drag-and-drop to copy a nonfile object to the desktop or to a file folder, the shell follows the standard OLE protocol of using the `IDataObject` interface to mediate the data transfer. Programs that offer objects for drag-and-drop have a variety of options available to them for how they wish to offer the data. One of the mechanisms available is the `IStream` interface, which exposes a block of bytes that the consumer can either `IStream::Read` from or `IStream::Write` to. Of course, when an object is dropped onto Explorer, Explorer is not going to use the `IStream::Write` method, because it wants to make a copy of the file being dragged, not modify it.

As an optimization, Windows 95 used one of the other methods of the `IStream` interface known as `IStream::CopyTo`. This method allows a stream to take over the job of copying itself to another location, in case there were some special optimizations the stream could perform to make the copy go faster. For example, the stream knows the ideal buffer size that should be used, and it can use its internal buffers directly to avoid transferring the data through a temporary buffer that a typical read/write loop would require.

In Windows 2000, the code in the shell that handled items that were dropped onto Explorer folders changed from the `IStream::CopyTo` method to a read/write loop. The code read from the stream with `IStream::Read` and wrote to the destination (usually a file) with `IStream::Write`. The reason for this was multifold. First, by controlling the copy loop, it became possible to use the expected size of the stream to provide feedback about how many bytes had been copied and approximately when the copy would be done. But the more important reason for controlling the read/write loop directly was that it allowed the copy operation to be canceled, because the shell would check whether the Cancel button was pressed between each chunk of data copied. By comparison, the `IStream::CopyTo` method provided no way to cancel the operation.

One program decided to cut a few corners in its `IStream` implementation. Since it knew that the shell used the `IStream::CopyTo` method to copy the stream, it implemented only that method and failed all the others. Well, not quite. It did implement `IStream::Read`, but the implementation was simply to hang and never complete the operation. As a result, when this program was run on Windows 2000, attempting to drag an object out of the program and drop it onto an Explorer folder locked up both the program (which was hung in its `IStream::Read` method) as well as the folder you dropped the object onto (since it was waiting for the `IStream::Read` to complete).

Fixing this problem was tricky, because the shell needed to predict the future! It needed to know whether calling `IStream::Read` was going to hang and therefore should switch to `IStream::CopyTo`. The program itself didn't help any. Its data object didn't support the `IPersist` interface, which is a common way for objects to identify themselves. The shell was forced to rely on circumstantial evidence: The data object the program provided did not specify in its object descriptor the number of bytes in the stream, nor did the program implement the `IStream::Stat` method, which would normally also provide that same information. If the data object was that lacking in helpful information, the shell looked around to see whether that particular program was running, and if so, the shell played it safe and assumed that the data object might have a faulty `IStream::Read` and switched to the old `IStream::CopyTo` method.

As a result, if you had this program running and you used drag-and-drop to drag an object out of some other program onto an Explorer window, there was a chance that Explorer would neither let you cancel the operation nor give you any copy progress out of fear that the object came from the buggy program. That program has shipped two new major versions since the application compatibility workaround was added to the shell. I doubt that many people are using the older version of the program, but the compatibility workaround remains in the system just in case.

Calling unimplemented functions and expecting them to fail

We received a report that a program was stuck in an infinite loop trying to enumerate the contents of the desktop in order to populate its tree view. The loop went like this:

```
IEnumIDList *penum; // assume initialized
LPITEMIDLIST pidl;
ULONG ulFetched;
int count = 0;
// keep fetching items
while (penum->Next(1, &pidl, &ulFetched) == NULL) {
 ... do stuff...
 pMalloc->Free(pidl);  // free the item we enumerated
 penum->Reset();       // restart the enumeration
 penum->Skip(++count); // skip over ++count items
}
```

This program decided to take what would normally be a simple $O(n)$ enumeration and turn it into an $O(n^2)$ enumeration by restarting the enumeration. It's like counting the number of people waiting in line by starting at the head and counting "one." Then returning to the head and counting "one, two." Then returning to the head and counting "one, two three," and so on, each time counting one more item than last time, until you finally reach the end of the line.

Yet it managed to work okay on Windows 95 because the shell in Windows 95 did not implement the `IEnumIDList::Reset` or `IEnumIDList::Skip` methods; calling either method got you `E_NOTIMPL` back. As a result, the program got its $O(n)$ behavior after all, because the attempts to go back to the start of the line and then walk forward both failed.

Windows 2000 added support for the `IEnumIDList::Reset` method (though not for `IEnumIDList::Skip`), which sent this code into an infinite loop because it would enumerate an item, rewind to the beginning of the list, attempt to skip over the item (which failed), then repeat. As a result of the rewind, it ended up enumerating the first item over and over again.

A special workaround was added to detect this application and return the `IEnumIDList::Reset` method to its `E_NOTIMPL` status. This keeps the program happy with respect to the file system enumerator, but it remains at the mercy of third-party shell namespace extensions, which might or might not implement various optional portions of the `IEnumIDList` interface. (But at least Windows 2000 didn't make it any worse; the program had problems with those shell namespace extensions on Windows 95, too.)

Mishandling integer ordinal classes

COMPUTER VENDORS TYPICALLY include on their computers several preinstalled programs. One of these programs caused Explorer to start crashing on Windows XP seemingly at random. (This program is so awful, the vendor even includes instructions on how to uninstall it in the company's online knowledge base.) The reason for this behavior is that the program simply doesn't know what to do with integer ordinal window classes. The program installs a system hook that is triggered whenever a window is created, and the hook goes roughly like this:

```
case HCBT_CREATEWND:                    // a window has been created
 CBT_CREATEWND *pccw = (CBT_CREATEWND *)lParam;
// get the class name
 LPSTR pszClassName = pccw->lpcs->lpszClassName;

// Hey, let's spew debug information even in the release version!
OutputDebugString(pszClassName);

if (IS_INTRESOURCE(pszClassName)) { // if it is a MAKEINTRESOURCE(n)
 char szBuf[256];
 GetClassName((HWND)lParam, szBuf, 100); // get its name
 if (strcmp(szBuf, "#32770")) ... // dialog box
}

if (lstrcmpi(pszClassName, "CabinetWClass") == 0) ...
if (lstrcmpi(pszClassName, "InternetConnectionWizardWClass") == 0)
...
... etc...
```

The program wants to take action for particular window classes (for what reason, I don't know and I'm afraid to find out). The authors of the program noticed that sometimes the class name is not a string but is rather an ordinal, so they convert it to a string by fetching the class name from the window handle. This is a valid technique, although rather inefficient, for one could simplify the test to

```
if (IS_INTRESOURCE(pszClassName)) {
 if (pszClassName == WC_DIALOG) ...
}
```

thereby avoiding the calls to `GetClassName` and `strcmp` and taking a large buffer off the stack. But that's not a bug, just an inefficiency.

The bug is that the authors forgot to wrap the second half of the function inside an "`else`" block. If some other ordinal class name is used, the test against the dialog class fails and control falls through to the string comparisons, resulting in a crash when a nondialog ordinal window class appears. It so happens that Windows XP uses a few more ordinal window classes internally to manage the appearance of unresponsive windows. As a result, when a window stops processing messages and Windows displays "Not responding" in the title bar, it creates one of these helper windows with an ordinal window class, and the program's hook crashes Explorer.

Being content to look good

After installing a particular program, users found that double-clicking a folder made the "Open With" dialog appear rather than the folder open. The reason is that the authors of the program didn't quite understand what it meant when the registry editor program said

```
(default)   REG_SZ (value not set)
```

Now, you and I know that this means that the default value for the key does not have a value. But the authors of the program thought it meant that they should set the default value to the string `(value not set)`. As a result, they

wrote `(value not set)` to `HKEY_CLASSES_ROOT\Folder\Shell` in a failed attempt to let the system choose the default operation for double-clicking a folder.

You're always a day away

After you install one particular program, it becomes impossible to upgrade to Windows XP. Instead, Windows XP Setup keeps reporting that existing software is not fully installed and that you have to restart the computer to allow it to finish installing. Yet no matter how many times you restart the computer, the message never goes away.

The reason for the false report is that the program writes itself into the `RunOnce` key. Each time it runs, it re-adds itself to the `RunOnce` key. So it doesn't really run once; rather, it runs all the time, but one run at a time. Since the `RunOnce` key is used for programs to finalize their installation, the continuous presence of an entry in the `RunOnce` key causes Windows XP Setup to conclude that it was put there by a Setup application that needs to do "one last thing" to complete the install.

The solution was to teach Windows XP Setup to ignore this particular program if it appears in the `RunOnce` key.

Memory will always be there when you need it

There are many examples of programs using memory after freeing it, but programs that use memory without even allocating it take the issue to a new level.

A DLL that offers COM objects exports a function called `DllGetClassObject`. Because a DLL can typically provide multiple types of objects, one of the parameters to the `DllGetClassObject` function is the CLSID that COM is interested in. Many DLLs cache the CLSID so that they can remember

what object they are creating. That's not particularly unreasonable. However, we found a DLL that didn't so much cache the CLSID as it did cache the pointer to the CLSID. In other words, it was using memory beyond the lifetime of the function call. Normally, of course, in the absence of indications to the contrary, the parameters of a function are guaranteed valid only for the lifetime of that function call. Once the call returns, the caller can use the memory for something else.

Yet this particular shell extension managed to survive because when it used that cached pointer, the memory still contained its original value. The caller hadn't used it for anything else yet.

When support for COM+ was added, COM changed the way the memory for the CLSID parameter was allocated as well as its lifetime. As a result, the shell extension discovered that by the time it went to check that cached pointer, the memory contained a value different from what it had when the `DllGetClassObject` function was called, and the shell extension treated this as a fatal error.

Sometimes a program shoots itself in the foot. One game parsed its command line into buffers it allocated on the stack and then used those buffers long after the function that did the parsing returned. And yet the program managed to survive, using pointers into freed stack memory, because none of the functions it called in the interim used very much stack—until Windows XP added support for side-by-side assemblies to the `LoadLibrary` function. The new `LoadLibrary` function uses more stack and eventually spills into the space that the program had been using on borrowed time. When the program then goes to look at its command line, it gets garbage.

If it has eight bytes, it must be a dialog box

A SERIES OF Explorer crashes was eventually traced back to a popular program that assumes that any window with eight bytes must be a dialog box.

```
hwnd = GetForegroundWindow();

if (GetWindowLong(hwnd, DWL_DLGPROC)) {
    SetWindowLong(hwnd, DWL_DLGPROC, MyDialogProc);
} else {
    SetWindowLong(hwnd, GWL_WNDPROC, MyWndProc);
}
```

The program wanted to subclass the foreground window and do it differently, depending on whether the window was a dialog box or a plain window. It's not clear to me why the authors cared about the difference, because a dialog box is a special case of a plain window; whatever programmers do for plain windows should work for dialog boxes, too. Plus, there is the very dodgy assumption that you can just grab a random window (which possibly belongs to another process!) and subclass it.

The program doesn't bother checking that the window is a dialog box before trying to read the `DWL_DLGPROC` from the window; it just assumes that if it can read the `DWL_DLGPROC`, then the window must be a dialog box. Since `DWL_DLGPROC` has the numeric value of 4 (on 32-bit Windows), an attempt to read the `DWL_DLGPROC` long will succeed for any window with eight or more window bytes, whether it is a dialog box or not. As a result, if the window that the program stumbles across happens to have eight or more window bytes, the program assumes it's a dialog box, even though it isn't.

Explorer had to work around this by making sure all its windows had fewer than eight window bytes.

Programs that rely on bugs in the operating system

This is a tricky one. If you have a bug that a program depends on, is it still a bug?

The status bar painting bug

The status bar control has two modes, simple mode and complex mode. Complex mode is the one you see most of the time, where the status bar is

broken into sections, each of which can display a different message. A status bar in simple mode, by comparison, displays a single line of text. Simple mode is typically used only temporarily, for example, to display brief help messages as you browse a program's menu.

It turns out that the status bar control had a cosmetic bug: If you changed the simple text while the status bar was in complex mode, the simple text would be drawn on the screen briefly. It would get cleaned up at the next paint cycle, but until then you had the wrong text on the status bar. Fixing this bug was a simple matter, but it didn't take long for the consequences of even the simplest bug fix to become apparent.

The status bar of a program from a major software publisher stopped working. This program changes its status bar text by changing the simple text, even if the status bar is in complex mode. The programmers must have noticed that the effect lasted only until the next paint cycle, so they applied a workaround, which was something equivalent to the following:

```
BOOL CWinApp::OnIdle()
{
 SendMessage(hwndStatus, SB_SETTEXT, SB_NOBORDERS | SB_POPOUT |
             SB_SIMPLEID,  szStatusBarText);
 return 0;
}
```

In other words, the programmers made sure that the program set the status bar text at every idle opportunity. That way, after the status bar "corrupted" their simple text (by painting the correct complex text), the program would set it back and the bug would paint the text for them.

The fix had to be undone, and the original bug restored. The flickering of the status bar is now a feature.

The folder leak

Another example of a program's reliance on a bug is a shell extension that relied on a memory leak in Explorer. In Windows 98, the internal cache used by the tree view in the left-hand side of the Explorer window was changed to avoid various multithreading race conditions that plagued the original

Windows 95 version. However, the cache was a bit too greedy and often held on to folders forever instead of releasing them when they hadn't been used in a while. The bug was subsequently fixed in Windows 2000.

Surprisingly, a program that won a "Best Internet Utility" award installed a shell namespace extension that relied on this memory leak. After fixing the leak, the namespace extension failed to update itself correctly, because it was relying on those leaked folders to do some of the state management. The fix in this case was to keep track of which folders in the folder cache came from this program's shell extension and to continue to leak them.

The incorrect daylight saving time computation

You may not remember it, but a bug fix was issued for the Microsoft C runtime library that would incorrectly determine the start date of daylight saving time by a week. The first year affected by this bug was 2001, wherein the `cvtdate` function erroneously started daylight saving time on April 8 rather than April 1. A corrected version of the C runtime was distributed, and the application compatibility bug reports weren't far behind.

A shell namespace extension that was declared "one of the best products of the year" by a major computer magazine suddenly stopped working. There was no crash, no error message, but clicking the icon in Explorer did nothing. Upon closer investigation, the namespace extension's `CreateInstance` method had an uninitialized variable bug three levels deep in its object creation. The variable happened to be an `HRESULT` whose value was consistently `0xFFFFFFFF`, stack garbage left over from a previous function call into the C runtime library. Since this value had the high bit set, it was treated as a COM failure code, and the error propagated back to the `CreateInstance` function, which returned the failure back to OLE and ultimately to Explorer.

As a result of the changes to the `cvtdate` function, a particular local variable had the value –1 when the function exited rather than the value of 0, which it did previously. The stack space previously used by that local variable was subsequently used (without ever being initialized) by the shell extension.

Unfortunately, not much could be done for this application. Fixing the bug would mean rolling back the fix to daylight saving time.

The accidental bind

One popular productivity suite was found to have arrows on all the icons in its quick-launch-type window. Upon closer investigation, the reason was that the program obtained the icons from the shell namespace in a manner similar to the following:

```
pidlShortcut = the full pidl to the shortcut;
pidlShortcutTarget = the full pidl to the shortcut target;

IShellFolder *psf;
if (psfDesktop->BindToObject(pidlShortcutTarget,
                             NULL, IID_IShellFolder, &psf)) {
  GetIconFrom(pidlShortcutTarget);
  psf->Release();
} else {
  GetIconFrom(pidlShortcut);
}
```

The call to the `BindToObject` method asks the shell namespace to produce a folder object for the item specified by the `pidlShortcutTarget`. In other words, it asks for the folder object corresponding to the target of the shortcut. In the case of a shortcut to an executable, the shortcut is something like "`Program.lnk`" and the target is "`Program.exe`". Windows 2000 had a bug in which asking for the folder object that corresponded to an executable actually succeeded. The result wasn't particularly useful, because the object is not actually a folder; most subsequent method calls would just fail with errors like "path not found," but the bind nevertheless succeeded. Windows XP fixed this bug so that attempting to bind to an executable as a folder failed.

With this change in behavior, the program ends up going down the "`else`" branch and gets the icon from the shortcut. And the icon for the shortcut contains an arrow.

The program seemed to want to do something special for shortcuts to folders, but it could have tested for folder-ness in a much simpler way, namely, by calling the `IShellFolder::GetAttributesOf` method and checking for the `SFGAO_FOLDER` attribute.