14:08 Control Panel Vulnerabilities

by Geoff Chappell

Back in 2010, as what I then feared might be "the last new work that I will ever publish," I wrote *The CPL Icon Loading Vulnerability*[18] about what Microsoft called a Shortcut Icon Loading Vulnerability.[19] You likely remember this vulnerability. It was notorious for having been exploited by the Stuxnet worm to spread between computers via removable media. Just browsing the files on an infected USB drive was enough to get the worm loaded and executing.

Years later, over drinks at a bar in the East Village, I brought up this case to support a small provocation that the computer security industry does not rate the pursuit of detail as highly as it might—or even as highly as it likes to claim. Thus did I recently reread my 2010 article, which I always was unhappy to have put aside in haste, and looked again at what others had written. To my surprise—or not, given that I had predicted "the defect may not be properly fixed"—I saw that others had revisited the issue too, in 2015 while I wasn't looking. As reported by Dave Weinstein in *Full details on CVE-2015-0096 and the failed MS10-046 Stuxnet fix*,[20] Michael Heerklotz showed that Microsoft had not properly fixed the vulnerability in 2010. Numerous others jumped on the bandwagon of scoffing at Microsoft for having needed a second go. I am writing about this vulnerability now because I think we might do well to have a *third* look!

Don't get too excited, though. It's not that Microsoft's second fix, of a DLL Planting Remote Code Execution Vulnerability,[21] still hasn't completely closed off the possibilities for exploitation. I'm not saying that Microsoft needs a third attempt. I will show, however, that the exploitation that motivated the second fix depends on some extraordinarily quirky behaviour that this second fix left in place. It is not credibly retained for backwards compatibility. That it persists is arguably a sign that we still have a long way to go for how the computer security industry examines software for vulnerabilities and for how software manufacturers fix them.

CVE-2010-2568

You'd hope that Stuxnet's trick has long been understood in detail by everyone who ever cared, but let's have a quick summary anyway. Among the browsed files is a shortcut (`.LNK`) file that presents as its target a Control Panel item whose icon is to be resolved dynamically. Browsing the shortcut induces Windows to load and execute the corresponding CPL module to ask it which icon to show. This may be all well and good if the CPL module actually is registered, so that its Control Panel items would show when browsing the Control Panel. The exploitation is simply that the target's CPL module is (still) not registered but is (instead) malware.

Chances are that you remember CVE-2010-2568 and its exploitation differently. After all, Microsoft had it that the vulnerability "exists because Windows incorrectly parses shortcuts" and is exploited by "a specially crafted shortcut." Some malware analysts went further and talked of a "malformed `.LNK` file."

But that's all rubbish! A syntactically valid `.LNK` file for the exploitation can be created using nothing but the ordinary user interface for creating a shortcut to a Control Panel item. Suppose an attacker has written malware in the form of a CPL module that hosts a Control Panel item whose icon is to be resolved dynamically. Then all the attacker *has* to do at the attacker's computer is as follows.

- First copy this CPL module to the USB drive;
- register this CPL module so that it will show in the Control Panel;
- open the Control Panel and find the Control Panel item; and,
- Ctrl-Shift drag this item to the USB drive to create a `.LNK` file.

Call the result a "specially crafted shortcut" if you want, but it looks to me like a very ordinary shortcut created by very ordinary steps. When the USB drive is browsed on the victim's computer,

[18] `http://www.geoffchappell.com/notes/security/stuxnet/ctrlfldr.htm`
[19] MS10-046 and CVE-2010-2568
[20] HP Enterprise, March 2015
[21] MS15-020, CVE-2015-0096

attacker's `.LNK` file on the USB drive is correctly parsed to discover that it's a shortcut to a Control Panel item that's hosted by the attacker's CPL module on the USB drive. Though this CPL module is not registered for execution as a CPL module on the victim's computer, it does get executed. The cause of this unwanted execution is entirely that the Control Panel is credulous that what is *said* to be a Control Panel item actually *is* one. What the Control Panel was vulnerable to was not a parsing error but a spoof.[22]

Microsoft certainly understood this at the time, for even though the words Control Panel do not appear in Microsoft's description of the vulnerability (except in boilerplate directions for such things as applying patches and workarounds), the essence of the first fix was the addition to `shell32.dll` of a routine that symbol files tell us is named `CControlPanelFolder::_IsRegisteredCPLApplet`.

Control Panel Icons

This `CControlPanelFolder` class is the shell's implementation of the COM class that is creatable from the Control Panel's well-known CLSID. Asking which icon to show for a Control Panel item starts with a call to this class' `GetUIObjectOf` method to get an `IExtractIcon` interface to a temporary object that represents the given item. Calling this interface's `GetIconLocation` method then gets directions for where to load the icon from.

The input to `GetUIObjectOf` is a binary packaging of the item's basic characteristics, which I'll refer to collectively as the *item ID*. The important ones for our purposes are: a pathname to the CPL module that hosts the item; an index for the item's icon among the module's resources; and a display name for the item. The case of interest is that when the icon index is zero, the icon is not cached from any prior execution of the CPL module, but is to be resolved dynamically, i.e., by asking the CPL module. Proceeding to `GetIconLocation` causes the CPL module to be loaded, called and unloaded.

This is all by design. It's a design with more moving parts than some would like, especially for just this one objective. But it fits the generality of shell folders so that highly abstracted and widely varying shell folders can present a broadly consistent user interface, while meeting a particular goal for the Control Panel. It's what lets a Control Panel item, or a shortcut to one, change its icon according to the current state of whatever the item exists to control.

I stress this because more than a few commentators blame the vulnerability on what they say was a bad design decision decades ago to load icons from DLLs, as if this of itself risks getting the DLL to execute. What happens is instead much more specific. Though CPL modules are DLLs and do have icons among their resources, the reason a CPL module may get executed for its icon is not to get the

[22] *Although parser bugs have a special place in Pastor's heart, it's good to be reminded occasionally that not every bug is a parser bug, and that there are other buggy things besides parsers!—PML*

icon but to ask explicitly which icon to get.

Note that I have not tied down who calls `GetUIObjectOf` or where the item ID comes from. The usual caller is SHELL32 itself, as a consequence of opening the Control Panel, e.g., in the Windows Explorer, to browse it for items to show. Each item ID is in this case being fed back to the class, having been produced by other methods while enumerating the items. In Stuxnet's exploit the caller is again SHELL32, but in response to browsing a shortcut to one Control Panel item. The item ID is in this case parsed from a shortcut (`.LNK`) file. Another way the call can come from within SHELL32 is automatically when starting the shell if a Control Panel item has been pinned to the Start Menu. The item ID is in this case parsed from registry data. More generally, the call can come from just about anywhere, and the item ID can come from just about anywhere, too.

One thing is common to all these cases, however, because the binary format of this item ID is documented only as being opaque to everyone but the Control Panel. If everyone plays by the rules, any item ID that the Control Panel's `GetUIObjectOf` ever receives can only have been obtained from some earlier interaction with the Control Panel. (Though not necessarily the *same* Control Panel!)

Input Validation

As security researchers, we've all seen this movie before—in multiple re-runs, even. Among the lax practices that were common once but which we now regard as hopelessly naive is that a program trusts what it reads from a file or a registry value, etc., on the grounds that the storage was private to the program or anyway won't have gotten messed with. Not very long ago, programs routinely didn't even check that such input was syntactically valid. Nowadays, we expect programs to check not just the syntax of their input but the meaning, so that they are not tricked into actions for which the present provider is not authorised (or ought to not even know how to ask).

For the Control Panel, the risk is that even if the item ID has the correct syntax what actually gets parsed from it may be stale. The specified CPL module was perhaps registered for execution some time ago but isn't now. Or, perhaps, it is still registered, but only for some other user or on some other computer. And this is just what can go wrong even though all the software that's involved plays by the rules. As hackers, we know very well that not all software does play by the rules, and that some deliberately makes mischief. That the format of the item ID is not documented will not stop a sufficiently skilled reverse engineer from figuring it out, which opens up the extra risk that an item ID may be *confected*. (Stick with me on this, because we'll do it ourselves later.)

Asking which icon to show for a Control Panel item gives an object-lesson in how messy the progress towards what we now think of as minimally prudent validation can be. Not until Windows 2000 did the Control Panel implementation make even the briefest check that an item ID it received was syntactically plausible. Worse, even though Windows NT 4.0 had introduced a second format, to support Unicode, it differentiated the two without questioning whether it had been given either. When the check for syntax did come, it was only that the item ID was not too small, and that the icon index was within a supported range.

Checking that the module's pathname and the item's display name, if present, were actually null-terminated strings that lay fully within the received data wasn't even *attempted* until Windows 7. I say attempted because this first attempt at coding it was defective. A malformed item ID could induce SHELL32 to read a byte from outside the item ID—only as far as 10 bytes beyond, and thus unlikely to access an invalid address, but outside nonetheless. Even a small bug in code for input validation is surely not welcome, but what I want to draw attention to is that this bug conspicuously was not addressed by the fix of CVE-2010-2568. A serious check of the supposed strings in the item ID came soon, but not, as far as I know, until later in 2010 for Windows 7 SP1.

Please take this in for a moment. While Microsoft worked to close off the spoof by having `GetUIObjectOf` check that the CPL module as named in the item ID is one that can be allowed to execute, Microsoft described the vulnerability as a parsing error—yet did nothing about errors in pre-existing code that checked the item ID for syntax! Wouldn't you think that if you're telling the world that the problem is a parsing error, then you'd want to look hard into everything nearby that involves any sort of parsing?

The suggestion is strong that Microsoft's talk of

[23] *I wonder what would happen if programmers got in the habit of taking the right approach—pitchforks applied to the protocol*

a parsing error was only ever a sleight of hand. As programmers, we've all written code with parsing errors. So many edge cases![23] To have such an error in your otherwise well-written code is only inevitable. Software is hand-crafted, after all. To talk of a parsing error is to appeal to the critics' recognition of fallibility. A parsing error can be the sort of an easy slip-up that gets you a 99 instead of a 100 on a test.

Falling for a spoof, however, seems more like a conceptual design failure. It's only natural that Microsoft directed attention to one rather than the other. My only question for Microsoft is how deliberate was the misdirection. Why so many security researchers went along with it, I won't ever know. This, too, is a conceptual failure—and not just mine.

First Fix

Still, it's a plus that fixing CVE-2010-2568 meant not only getting the item ID checked ever so slightly better for syntax, but also checking it for its meaning, too. Checking, however, is only the start. What do you do about a check that fails?

Were it up to me, thinking just of what I'd like for my own use of my own computer, I'd have all `CControlPanelFolder` methods that take an item ID as input return an error if given any item ID that specifies a CPL module that is not currently registered. My view would be that even if the item ID is only stale rather than confected (keep reading!), then wherever or whenever the specified CPL module is or was registered, it's not registered *now* for my use on this computer—and so it shouldn't show if I browsed the Control Panel. I'd rather not accept it for any purpose at all, let alone run the risk that it gets executed.

Microsoft's view, whether for a good reason or bad, was nothing like this firm. First, it regarded the problem case as more narrow, not just that the specified CPL module is not currently registered (so that the item ID is at least stale, if not actually faked), but also that the specified icon index is zero (this being, we hope, the only route to unwanted execution) and anyway only for `GetUIObjectOf` when queried for an `IExtractIcon` interface. Second, the fix didn't reject but *sanitised*.[24] It let the problem case through, but as if the icon index were given as -1 instead of 0.

Perhaps this relaxed attitude was motivated just by a general (and understandable) desire for the least possible change. Perhaps there was a known case that had to be supported for backwards compatibility. I can't know either way, but what I hope you've already woken to is the following contrast between rejection and sanitisation. To reject suspect input may be more brutal than you need, but it has the merit of *certainty*. The suspect input goes no further, and any innocent caller should at least have anticipated that you return an error. To "sanitise" suspect input and proceed as if all will now be fine is to depend on the deeper implementation—which, as you already know, had not checked this input for itself!

What Lies Beneath

By deeper implementation I mean to remind you that `GetUIObjectOf` is just the entry point for asking which icon to show. There is still a long, long way to go: first for the temporary object that supplies the `GetIconLocation` method for the given item; and then, though apparently only if the preceding stage has zero for the icon index, to the more general support for loading and calling CPL modules. Moreover, this long, long way goes through old, old code, with all the problems that can come from that. To depend on any of it for fixing a bug, especially one that you know real-world attackers are probing for edge cases, seems—at best—foolhardy.

To sense how foolhardy, let's have some demonstrations of where this deeper implementation can go wrong. An attacker whose one goal is to see if the first fix can be worked around would most easily follow the execution from `GetUIObjectOf` down. Many security researchers would follow, too—perhaps mumbling that their lot is always to be reacting to the attackers and never getting ahead. One way to get ahead is to study in advance as much of the general as you can so that you're better prepared whenever you have to look into the specific. This is why, when I examine what might go wrong with trying to fix CVE-2010-2568 by letting sanitised input through to the deeper implementation, I work in what you may think is the reverse of the natural direction.

designers—to address the root cause of these edge cases. —PML.

[24] *When neighbors whose software you'd like to trust tell you proudly that they "sanitize" input and "fix" it, so that inputs coming in as invalid would still be used—run. You'll thank us later. —PML*

Loading and Calling

Where we look at first into the deeper implementation is therefore the general support for loading and calling of CPL modules, but particularly of a CPL module that hosts a Control Panel item whose icon is to be resolved dynamically. For my 2010 article, I presented such a simple example.[25]

Whenever this CPL module is loaded, the first call to its exported `CPlApplet` function produces a message box that asks "Did you want me?", and whose title shows the CPL module's pathname. That much is done so that we can see when the CPL module gets loaded. What makes this CPL module distinctively of the sort we want to understand is that when we call to `CPlApplet` for the `CPL_INQUIRE` message, the answer for the icon index is zero.

Install There are several ways to register a CPL module for execution, but the easiest is done through—wait for it—the registry. Save the CPL module as `test.cpl` in some directory whose *path*, for simplicity and definiteness, contains no spaces and is not ridiculously long. Then create the following registry value shown in Figure 9.

To test, open the Control Panel so that it shows a list of items, not categories, and confirm that you don't just see an item named Test, but also see its message box. Yes, our CPL module gets loaded *and* executed just for *browsing* the Control Panel. Indeed, it gets loaded and executed multiple times. (Watch out for extra message boxes lurking behind the Control Panel.) Though it's not necessary for our purposes, you might, for completeness, confirm that the Test item does launch. When satisfied with the CPL module in this configuration as a base state, close any message boxes that remain open, close the Control Panel, too, and then try a few quick demonstrations.

By the way—I say it as if it's incidental, even though I can't stress it enough—two of these demonstrations begin by varying the circumstances as even a novice mischief-maker might. Each depends on a little extra step or rearrangement that you might stumble onto, especially if your experimental technique is good, but which is very much easier to add if its relevance is predicted from theoretical analysis.

If you doubt me, don't read on right away, but instead take my cue about putting spaces in the pathname and see how easily you come up with suitably quirky behaviour. Of course, theoretical analysis takes hours of intensive work, and often comes to nothing. There's a trade-off, but for investigating possibly subtle interactions with complex software the predictive power of theoretical analysis surely pays off in the long run.

But enough of my pleas to the computer security industry for investing more in studying Windows! Let us get on with the demonstrations.

Default File Extension? First, remove the file extension from the registry data. Open the Control Panel and see that the Test item no longer shows. Close the Control Panel. Rename `test.cpl` to `test.dll`. Open the Control panel and see that there's still no Test item. Evidently, neither `.cpl` nor `.dll` is a default file extension for CPL modules. Close the Control Panel. Why did I have you try this? Create *path*\test itself as any file you like, even as a directory. Open the Control Panel. Oh, now it executes `test.dll`!

Yes, if the pathname in the registry does not have a file extension, the Control Panel will load and execute a CPL module that has `.dll` appended, as if `.dll` were a default file extension—but only if the extension-free name also exists as at least some sort of a file-system object. Isn't this weird?

Spaces For our second variation, start undoing the first. Close the Control Panel, remove the subdirectory, and rename the CPL module to `test.cpl`. Then, instead of restoring the registry data to "*path*`\test.cpl`" make it "*path*`\test.cpl rubbish`." Open the Control Panel. Of course, the Test item does not show. Close the Control Panel and make a copy of the CPL module as "`test.cpl rubbish`." Open the Control Panel. See first that the copy named "`test.cpl rubbish`" gets loaded and executed. This, of course, is just what we'd hope. The quirk starts with the next message box. It shows that `test.cpl` gets loaded and executed, too!

Yes, if the registry data contains a space, the CPL module as registered executes as expected but then there's a surprise execution of something else. The Control Panel finds a new name by truncating the registered filename—the whole of it, including the *path*—at the first space. And, yes, if the result of the truncation has no file extension, then `.dll` gets

[25] `unzip pocorgtfo14.pdf CPL/testcpl.zip`

appended. (Though, no, the extension-free name doesn't matter now.)

Please find another Zen-friendly moment for taking this in. This quirky Wonderland surprise execution surely counts as a parsing error of some sort. It means that to fix a case of surprise execution that Microsoft presented as a parsing error, Microsoft trusted old code in which a parsing error could cause surprise execution. So it goes.

Length Finally, play with lengthening the pathname to something like the usual limit of `MAX_PATH` characters. That's 260, but remember that it includes a terminating null. Close the Control Panel. Make a copy of `test.cpl` with some long name and edit the registry data to match the copy that has this long name. Open the Control Panel. Repeat until bored. Perhaps start with the 259 characters of

```
1 c:\temp\cpltest\1123456789abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcde.cpl
```

and work your way down—or start with

```
1 c:\temp\cpltest\test.cpl 9abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcdef012
```

if you want to stay with the curious configuration where *one* CPL module is registered but *two* get executed. (My naming convention is that after the 16 characters of my chosen path, the filename part has each character show its 0-based index into the pathname, modulo 16, except that where the index is a multiple of 16 the character shows how many multiples. The ellipses each hide 160 characters.) Either way, for any version of Windows from the last decade, the Test item does not show, and the CPL module does not get loaded and executed—until you bring the pathname down to 250 characters, not including the terminating null.

This limit is deliberate. Starting with Windows XP and its support for Side-By-Side (SxS) assemblies, the Control Panel anticipates loading CPL modules in activation contexts. There are various ways that a CPL module can affect the choice of activation context. For one, the Control Panel looks for a file that has the same name as the CPL module, but with ".manifest" appended. Though this manifest need not exist, the Control Panel has, since Windows XP SP2, rejected any CPL module whose pathname is already too long for the manifest's name to fit the usual `MAX_PATH` limit. (The early builds of Windows XP just append without checking. That they got away with it is a classic example of a buffer overflow that turns out to be harmless.)

Key:	`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Control Panel\CPLs`
Value:	anything, e.g., `Test`
Type:	`REG_SZ` or `REG_EXPAND_SZ`
Data:	*`path`*`\test.cpl`

Figure 9. CPL Module Registry Entry

The Exec Name

As we move toward the specifics of loading and calling a CPL module to ask which icon to show, it's as well to observe that this lower-level code for loading and calling CPL modules in general is not just quirky in some of its behaviors, but also in how it gets its inputs. Reasons for that go back to ancient times and persist, so that CPL modules can be loaded and executed via the `RUNDLL32.EXE` program, the lower-level code for loading and calling CPL modules that receives its specification of a Control Panel item as text—as if it were supplied on a command line. For this purpose, the text appears to be known in Microsoft's source code as the item's *exec name*. It is composed as the module's pathname between double-quotes, then a comma, and then the item's display name.

Perhaps this comes from wanting to reuse as much legacy code as possible. The loading and executing of a CPL module specifically to ask which icon to show for one of that module's Control Panel items—even though this task is no longer ever done on its own from any command line—is handled as a special case with a slightly modified exec name: the module's pathname, a comma, a (signed) decimal representation of the icon index, another comma, and the item's display name.[26]

The absence of double-quotes around the module's pathname in this modified exec name is much of the reason for the quirky behaviour demonstrated above when the pathname contains a space. It goes further than that, however.

I ask you again to take another Wonderland Zen moment of reflection. The `GetUIObjectOf` method receives the module's pathname, the item's icon index, and the item's display name—among other things—in a binary package. It parses them out of the package and then into this modified exec name, i.e., as text, which the deeper implementation will have to parse. What could go wrong with that?

The immediate answer is that the modified exec name is composed in a buffer that allows for 0x022A characters, but, until Microsoft's second fix, only `MAX_PATH` characters are allowed for the copy that's kept for the object that gets created to represent the Control Panel item for the purpose of providing an `IExtractIcon` interface. This mismatch of allowances is ancient. Worse, even though Windows Server 2003 (chronologically, but Windows XP SP2, by the version numbers) had seen Microsoft introduce the mostly welcome `StringCb` and `StringCch` families of helper routines for programmers to work with strings more securely, this particular copying of a string was *not* converted to these functions until Windows Vista—and even then the programmer could blow away much of its point by not checking it for failure.

If the CPL module's pathname is just long enough, the saved exec name gets truncated so that it keeps the comma but loses at least some of the icon index. When the `GetIconLocation` method parses the (truncated) exec name, it sees the comma and infers that an icon index is present. If enough of the icon index is retained such that digits are present, including after a negative sign, then the only consequence is that the inferred icon index is numerically wrong. If the CPL module's pathname is exactly the "right" length, meaning 257 or 258 characters (not including a terminating null), then the icon index looks to be empty or to be just a negative sign, and is interpreted as zero.[27]

It's time for another of those Wonderland moments. To defeat a spoof that Microsoft misrepresented as a parsing error, Microsoft dealt with a suspect zero by proceeding as if the zero had been -1, but then an actual parsing error in the deeper implementation could turn the -1 back to zero!

The practical trouble with this parsing error, which is perhaps the reason it wasn't noticed at the time, is that it kicks in only if the CPL module's pathname is longer than the 250-character maximum that we demonstrated earlier. An item ID that could trigger this parsing error isn't ever going to be created by the Control Panel. It can't, for instance, get fed to `GetUIObjectOf` from a shortcut file that we created simply by a Ctrl-Shift drag. If we want to demonstrate this parsing error without resorting to a Windows version that's so old that the Control Panel doesn't have the 250-character limit, the item ID would need to be faked. We need a specially crafted shortcut file after all.

Shortcut Crafting Making an uncrafted shortcut file is straightforward if you're already familiar with programming the Windows shell. The shell provides a creatable COM object for the job, with interfaces whose methods allow for specifying what the shortcut will be a shortcut to, and for saving

[26] *At this point, you might feel exactly how Alice felt in Wonderland. The Cheshire Cat would approve. —PML*

[27] *And now we don't even need to ask what the Caterpillar was smoking. —PML*

the shortcut as a `.LNK` file. The target, being an arbitrary item in the shell namespace, is specified as a sequence of shell item identifiers that generalise the pathname of a file-system object. To represent a Control Panel item, we just need to start with a shell item identifier for the Control Panel itself, and append the item ID such as we've been talking about all along. Where crafting comes into it is that we've donned hacker hats, so that the item ID we append for the Control Panel item is *confected*. But enough about the mechanism! You can read the source code.[28]

To build, use the Windows Driver Kit (WDK) for Windows 7. The 32-bit binary suffices for 64-bit Windows. You may as well build for the oldest supported version, which is Windows XP, but the program does nothing that shouldn't work even for Windows 95.

To test, open a Command Prompt in some directory, e.g., *path*, where you have a copy of `test.cpl` from the earlier demonstrations of general behaviour. Again, for simplicity and definiteness, start with a *path* that contains no spaces and is not ridiculously long. To craft a shortcut to what might be a Control Panel item named Test that's hosted by this `test.cpl`, run the command

```
1 linkcpl /module:path\test.cpl /icon:0 /name:
      Test test.lnk
```

With the Windows Explorer, browse to this same directory. If running on an earlier version than Windows 7 SP1 without Microsoft's first fix, you should see the CPL module's message box even without having registered `test.cpl` for execution. For any later Windows version or if the first fix is applied, browsing the folder executes the CPL module only if it's been registered.

For full confidence in this base state, re-craft the shortcut but specify any number other than zero for the icon index. Confirm that browsing does not cause any loading and executing unless the shortcut records that the CPL module is of the sort that always wants to be asked which icon to show.

Very Long Names The point to crafting the shortcut is that we can easily use it to deliver to `GetUIObjectOf` an item ID that we specify in detail. Do note, however, that the shortcut is only convenient, not necessary. We could instead have a program confect the item ID, feed it to `GetUIObjectOf` by calling directly, and then call `GetIconLocation` and report the result.

Either way, the details that we want to specify are the module's pathname and the icon index. We'll provide pathnames that are longer than the Control Panel accepts when enumerating Control Panel items, but which nonetheless result in the expected loading and execution when the icon index is zero. Then, we'll demonstrate that when the pathname is just the right length, as predicted above, the loading and execution happen even when the icon index is non-zero. The assumption throughout is that the Windows you try this on does not have Microsoft's second fix.

We know anyway not to bother with the very longest possible name (except as a control case), since the truncation loses the comma from the exec name such that it will seem to have no icon index at all. Instead make a copy of `test.cpl` that has a 258-character name such as

```
1 c:\temp\cpltest\1123456789abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcd.cpl
```

Craft a `/icon:0` shortcut that has this same long name for the module's pathname. If testing on a Windows that has the first fix, also edit this long name into the registry. Browse the directory that contains the shortcut—and perhaps be a little disappointed that the CPL module does not get loaded and executed.

But now remember that delicious quirk in which a space in the module's pathname, within the 250-character limit, induces the loading and executing of *two* CPL modules, first as given and then as truncated at the first space. Copy `test.cpl` as

```
1 c:\temp\cpltest\test.cpl 9abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcdef01
```

Re-craft the shortcut by giving this name to the `/module` switch in quotes. Update the registration if appropriate. Still, the copy with the long name doesn't get loaded and executed—but, as you might have suspected, the copy we've left as `test.cpl` does! Indeed, because the copy with the long name

[28] `unzip pocorgtfo14.pdf CPL/linkcplsrc.zip CPL/linkscplbin.zip`

doesn't *have* to execute for this purpose, and because its Control Panel item won't show in the Control Panel, it doesn't need to be a copy. Even an empty file suffices!

Edge Cases By repeating with ever shorter pathnames, but also trying non-zero values for the icon index, we can now demonstrate that CVE-2010-2568 has its own edge cases, as predicted from theoretical analysis. The general case has zero for the icon index. The edge cases are that if the pathname is very long but contains a space in the first 250 characters, then the icon index need not be zero. The following table summarises the behaviour on a Windows that does not have CVE-2010-2568 fixed.

The length does not include a terminating null. The icon index is assumed to be syntactically valid: negative means 0xFF000000 to 0xFFFFFFFF inclusive; positive means 0x00000001 to 0x00FFFFFF inclusive. Execution is of the CPL module that is named by truncating the very long pathname at its first space. (Also, if this has no file extension, appending `.dll` as a default.)

Length	Icon Index	Exec?	Remarks
259	Any	No	
258	Zero	Yes	
	Non-Zero	Yes	Edge Case
257	Zero	Yes	
	Negative	Yes	Edge Case
	Positive	No	
Less	Zero	Yes	If Registered[29]
	Non-Zero	No	

CVE-2015-0096

The point to Microsoft's first fix of CVE-2010-2568 was to avoid execution unless the pathname in the item ID was that of a registered CPL module. But the decision to test the registration only if the icon index in the item ID was zero meant that the two edge cases were completely unaffected. Worse, when the icon index in the item ID was zero, changing the zero to -1 would turn the suspect item ID not into something harmless but into an edge case. Either way, the pathnames had to be so long that the edge cases turned into surprise execution only because of a quirk even deeper into the code such that the CPL module executes needed not to be the one specified.

CVE-2015-0096 appeared to be the first public recognition of this, not that you would ever guess it from the formal description or from anything that I have yet found that Microsoft has published about it. From Dave Weinstein's explanation, it appears that the incompleteness of the first fix was found by following the mind of an attacker frustrated by the first fix and seeking a way around it.

The second fix plausibly does end the exploitability, at least for the purpose of using shortcuts to Control Panel items as a way to spread a worm. The edge cases exist only because of a parsing error caused by a buffer overflow. The second fix increases the size of the destination buffer so that it does not overflow when receiving its copy of the exec name. For good measure, it also tracks the icon index separately, so that it anyway does not get parsed from that copy.

But the CPL module's filename continues to be parsed from that copy. If it contains a space, then the Control Panel still can execute two CPL modules, one as given and one whose name is obtained by truncating at the first space. Only because of this were the edge cases ever exploitable. Yet even as late as the original release of Windows 10—which is as far as I have yet caught up to for my studies—it remains true that if you can register "*path*`\test.cpl rubbish`" or "*path*`\space test.cpl`" for execution as a CPL module, then you can get *path*`\test.cpl` or *path*`\space.dll` loaded and executed by surprise. Is anyone actually happy about that?

Many ways seem to lead into this Wonderland, but is there a way out?

[29] Since the first fix, this executes only if registered.