The case of the file that won't copy because of an Invalid Handle error message

devblogs.microsoft.com/oldnewthing/20141031-00

October 31, 2014

Raymond Chen

A customer reported that they had a file that was “haunted” on their machine: Explorer was unable to copy the file. If you did a copy/paste, the copy dialog displayed an error.

1 Interrupted Action

Invalid file handle

Contract Proposal
Size: 110 KB
Date modified: 10/31/2013 7:00 AM

Okay, time to roll up your sleeves and get to work. This investigation took several hours, but you’ll be able to read it in ten minutes because I’m deleting all the dead ends and red herrings, and because I’m skipping over a lot of horrible grunt work, like tracing a variable in memory backward in time to see where it came from.[1]

The *Invalid file handle* error was most likely coming from the error code `ERROR_INVALID_HANDLE`. Some tracing of handle operations showed that a call to `GetFileInformationByHandle` was being passed `INVALID_FILE_HANDLE` as the file handle, and as you might expect, that results in the invalid handle error code.

Okay, but why was Explorer’s file copying code getting confused and trying to get information from an invalid handle?

Code inspection showed that the handle in question is normally set to a valid handle during the file copying operation. So the new question is, “Why *wasn’t* this variable set to a valid handle?”

Debugging why something didn't happen is harder than debugging why it *did* happen, because you can't set a breakpoint of the form "Break when X doesn't happen." Instead you have to set a breakpoint in the code that you're pretty sure is being executed, then trace forward to see where execution strays from the intended path.

The heavy lifting of the file copy is done by the `CopyFile2` function. Explorer uses the `CopyFile2ProgressRoutine` callback to get information about the copy operation. In particular, it gets a handle to the destination file by making a duplicate of the `hDestinationFile` in the `COPYFILE2_MESSAGE` structure. The question is now, "Why wasn't Explorer told about the destination file that was the destination of the file copy?"

Tracing through the file copy operation showed that the file copy operation actually *failed* because the destination file already exists. The failure would normally be reported as `ERROR_FILE_EXISTS`, and the offending `GetFileInformationByHandle` would never have taken place. Somehow the file copy was being treated as having succeeded even though it failed. That's why we're using an invalid handle.

The `CopyFile2` function goes roughly like this:

```
HRESULT CopyFile2()
{
 BOOL fSuccess = FALSE;
 HANDLE hSource = OpenTheSourceFile(); // calls SetLastError() on failure
 if (hSource != INVALID_HANDLE_VALUE)
 {
  HANDLE hDest = CreateTheDestinationFile(); // calls SetLastError() on failure
  if (m_hDest != INVALID_HANDLE_VALUE)
  {
   if (CopyTheStuff(hSource, hDest)) // calls SetLastError() on failure
   {
    fSuccess = TRUE;
   }
   CloseHandle(hDest);
  }
  CloseHandle(hSource);
 }
 return fSuccess ? S_OK : HRESULT_FROM_WIN32(GetLastError());
}
```

Note: This is not the actual code, so don't go whining about the coding style or the inefficiencies. But it gets the point across for the purpose of this story.

The `CreateTheDestinationFile` function failed because the file already existed, and it called `SetLastError` to set the error code to `ERROR_FILE_EXISTS`, expecting the error code to be picked up when it returned to the `CopyFile2` function.

On the way out, the `CopyFile2` function makes two calls to `CloseHandle`. `CloseHandle` on a valid handle is not supposed to modify the thread error state, but somehow stepping over the `CloseHandle` call showed that the error code set by `CreateTheDestinationFile` was being reset back to `ERROR_SUCCESS`. (Mind you, this was a poor design on the part of the `CopyFile2` function to leave the error code lying around for an extended period, since the error code is highly volatile, and you would be best served to get it while it's still there.)

Closer inspection showed that the `CloseHandle` function *had been hooked by some random DLL that had been injected into Explorer*.

The hook function was somewhat complicated (more time spent trying to reverse-engineer the hook function), but in simplified form, it went something like this:

```
BOOL Hook_CloseHandle(HANDLE h)
{
 HookState *state = (HookState*)TlsGetValue(g_tlsHookState);
 if (!state || !state->someCrazyFlag) {
  return Original_CloseHandle(h);
 }
 ... crazy code that runs if the flag is set ...
}
```

Whatever that crazy flag was for, it wasn't set on the current thread, so the intent of the hook was to have no effect in that case.

But it *did* have an effect.

The `TlsGetValue` function modifies the thread error state, even on success. Specifically, if it successfully retrieves the thread local storage, it sets the thread error state to `ERROR_SUCCESS`.

Okay, now you can put the pieces together.

- The file copy failed because the destination already exists.
- The `CreateTheDestinationFile` function called `SetLast-Error(ERROR_FILE_EXISTS)`.
- The file copy function did some cleaning up before retrieving the error code.
- The cleanup functions are not expected to alter the thread error state.
- But the cleanup function had been patched by a rogue DLL, and the hook function *did* alter the thread error state.
- This alteration caused the file copy function to think that the file was successfully copied even though it wasn't.
- In particular, the caller of the file copy function expects to have received a handle to the copy during one of the copy callbacks, but the callback never occurred because the file was never copied.
- The variable that holds the handle therefore remains uninitialized.

- This generates an invalid handle error when the code tries to use that handle.
- This error is shown to the user.

An injected DLL that patched a system call resulted in Explorer looking like an idiot. (As Alex and Gaurav well know, Explorer is perfectly capable of looking like an idiot without any help.)

We were quite fortunate that the error manifested itself as a failure to copy the file. Imagine if Explorer didn't use `GetFileInformationByHandle` to get information about the file that was copied. The `CopyFile2` function returns `S_OK` *even though it actually failed and no file was copied*. Explorer would have happily reported, "Congratulations, your file was copied successfully!"

Stop and think about that for a second.

A rogue DLL injected into Explorer patches a system call incorrectly and ends up causing all calls to `CopyFile2` to report success even if they failed. The user then deletes the original, thinking that the file was safely at the destination, then later discovers that, oops, looks like the file was not copied after all. Sorry, it looks like that rogue DLL (which I'm sure had the best of intentions) had a subtle bug that caused you to *lose all your data*.

This is why, as a general rule, Windows considers DLL injection and API hooking to be unsupported. If you hook an API, you not only have to emulate all the documented behavior, you also have to emulate all the *undocumented* behavior that applications unwittingly rely on.

(Yes, we contacted the vendor of the rogue DLL. Ideally, they would get rid of their crazy DLL injection and API hooking because, y'know, *unsupported*. But my guess is that they are going to stick with it. At least we can try to get them to fix their bug.)

[1] To do this, you identify the variable and set a breakpoint when that variable is allocated. (This can be tricky if the variable belongs to a class with hundreds of instances; you have to set the breakpoint on the correct instance!) When that breakpoint is hit, you set a write breakpoint on the variable, then resume execution. Then you hope that the breakpoint gets hit. When it does, you can see who set the value. "Oh, the value was copied from that other variable." Now you repeat the exercise with that other variable, and so on. This is very time-consuming but largely uninteresting so I've skipped over it.

Raymond Chen

Follow