Sending a window a WM_DESTROY message is like prank calling somebody pretending to be the police

devblogs.microsoft.com/oldnewthing/20110926-00

September 26, 2011

Raymond Chen

A customer was trying to track down a memory leak in their program. Their leak tracking tool produced the stacks which allocated memory that was never freed, and they all seemed to come from `uxtheme.dll`, which is a DLL that comes with Windows. The customer naturally contacted Microsoft to report what appeared to be a memory leak in Windows.

I was one of the people who investigated this case, and the customer was able to narrow down the scenario which was triggering the leak. Eventually, I tracked it down. First, here's the thread that caused the leak:

```
DWORD CALLBACK ThreadProc(void *lpParameter)
{
 ...
 // This CreateWindow caused uxtheme to allocate some memory
 HWND hwnd = CreateWindow(...);
 RememberWorkerWindow(hwnd);
 MSG msg;
 while (GetMessage(&msg, NULL, 0, 0)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
 return 0;
}
```

This thread creates an invisible window whose job is to do *something* until it is destroyed, at which point the thread is no longer needed. The window procedure for the window looks like this:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
 ...
 switch (uMsg) {
 ... business logic deleted ...
 case WM_DESTROY:
  ForgetWorkerWindow(hwnd);
  PostQuitMessage(0);
  break;
 ...
 }
 return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Sinec this is the main window on the thread, its destruction posts a quit message to signal the message loop to exit.

There's nothing obviously wrong here that would cause `uxtheme` to leak memory. And yet it does. The memory is allocated when the window is created, and it's supposed to be freed when the window is destroyed. And the only time we exit the message loop is when the window is destroyed. So how is it that this thread manages to exit without destroying the window?

The key is how the program signals this window that it should go away.

```
void MakeWorkerGoAway()
{
 // Find the worker window if it is registered
 HWND hwnd = GetWorkerWindow();
 // If we have one, destroy it
 if (hwnd) {
  // DestroyWindow doesn't work for windows that belong
  // to other threads.
  // DestroyWindow(hwnd);
  SendMessage(hwnd, WM_DESTROY, 0, 0);
 }
}
```

The authors of this code first tried destroying the window with `DestroyWindow` but ran into the problem that you cannot destroy a window that belongs to a different thread. "But aha, since the `DestroyWindow` function sends the `WM_DESTROY` message, we can just cut out the middle man and send the message directly."

Well, yeah, you can do that, but that doesn't actually destroy the window. It just *pretends* to destroy the window by prank-calling the window procedure and saying "Ahem, um, yeah, this is the, um, window manager? (stifled laughter) And, um, like, we're just calling you to tell you, um, you're being destroyed. (giggle) So, um, you should like pack up your bags and (snort) sell all your furniture! (raucous laughter)"

The window manager sends the `WM_DESTROY` message to a window as part of the window destruction process. If you send the message yourself, then you're making the window *think* that it's being destroyed, even though it isn't. (Because it's `DestroyWindow` that destroys windows.)

The victim window procedure goes through its "Oh dear, I'm being destroyed, I guess I'd better clean up my stuff" logic, and in this case, it unregisters the worker window and posts a quit message to the message loop. The message loop picks up the `WM_QUIT` and exits the thread.

And that's the memory leak: The thread exited before all its windows were destroyed. That worker window is still there, because it never got `DestroyWindow`'d. Since the window wasn't actually destroyed, the internal memory used to keep track of the window didn't get freed, and there you have your leak.

"You just got punk'd!"

The correct solution is for the `MakeWorkerGoAway` function to send a message to the worker window to tell it, "Hey, I'd like you to go away. Please call `DestroyWindow` on yourself." You can invent a private message for this, or you can take advantage of the fact that the default behavior of the `WM_CLOSE` message is to destroy the window. Since our window procedure doesn't override `WM_CLOSE`, the message will fall through to `DefWindowProc` which will convert the `WM_CLOSE` into a `DestroyWindow`.

Now that you understand the difference between destroying a window and prank-calling a window telling it is being destroyed, you might be able to help Arno with his problem.

Raymond Chen

Follow