



DPI is the New DNS: the Tale of the Wandering Cursor

Devolutions

.....
**THE BEAST WE ARE DISCUSSING TODAY
IS CALLED A HIGGS-BUGSON**
.....

In the world of software, developers tend to build a special relationship with the nastiest bugs they encounter in their career, as if they somehow had a life of their own. Throughout years of chasing red herrings, professionals have come up with a taxonomy to classify them. A popular one is the [heisenbug](#), defined as a software bug that seems to disappear or alter its behavior when studied. If this makes no sense, imagine trying to fix a bug that you can easily reproduce, yet disappears while using a debugger, effectively preventing you from investigating it.

The beast we are discussing today is called a higgs-bugson, a type of bug characterized as difficult or impossible to reproduce in a controlled development environment. Unlike the heisenbug, this means the type of sporadic issue reported by multiple users with no obvious cause that we fail to reproduce in a consistent manner. A common culprit for higgs-bugsons is the domain name system (DNS), which is notorious for causing all sorts of inconsistent failures affecting all types of software, including Wayk Now.

This is why people often makes jokes about DNS: even if it defies all logic, the weird bug you can't explain is somehow caused by a DNS issue anyway. With the advent of high density displays, issues related to DPI scaling have now become the norm, to the point where we ask the following question: is DPI the new DNS? Behind the high quality text rendering and crisp, detailed images lies a world of pain for most developers. After reading the tale of the wandering cursor, you may never look at your high DPI display the same way.

A Wild Bug Appears

The story begins with sporadic reports of an offset mouse cursor inside a Wayk Now remote session. That is to say, the position of the mouse on the remote machine is not where the Wayk Now client thinks it is, leading to a frustrating experience where mouse clicks are directed to the wrong place. If you tried clicking the 'X' button on a window, you would see the cursor in the right place, but the actual click would happen elsewhere. Users were telling us on the forum and by email that the problem would come and go seemingly at random. Needless to say, analysis of problem reports and log files lead us nowhere and we were not able to reproduce the issue in our lab.

Start With Code

We moved on to analyzing the code - how does it work, and what might have changed to cause this? The code path involved in sending the mouse position to the remote session is simple, and we quickly ruled out memory corruption as a cause. What we did see is that at several points on the way, mouse coordinates need to be scaled to account for the display settings on both the local and remote computers. So what does that mean?

DPI, or dots-per-inch, is a measure of pixel density. High DPI displays have become mainstream in recent years: for example, a 4K monitor has around 8 million pixels. At 23", that monitor will have a DPI of 185 - about twice the standard 96 DPI desktop resolution. If we draw our application on the 4K screen without accounting for the higher pixel density, everything will appear tiny: a pixel is a pixel, but on the higher resolution screen it only takes 1/2 the physical space - we must scale it by 200% to appear the proper size. Windows lets you choose that scaling factor and decide the balance of size and crispness that looks best to you.

Windows has a long tail of backward compatibility, and by default applications are “DPI virtualized”. This means that whatever the scaling factor, Windows “pretends” to the application it’s running at a vanilla 96 DPI and secretly scales the graphics to the proper size. A neat trick for sure, but one that leads to blurry text and fuzzy images. The Wayk Now team is proud of our clients native high DPI support: we (and many other modern applications) achieve this by opting out of DPI virtualization and declaring to Windows “we understand high DPI, tell us the truth about the display and we pinky-promise to render properly!”. It’s hard to get right, but the results speak for themselves when the application renders crisply and clearly, even when moving between displays with different DPIs.

Back to the Mouse

But how does that relate to the mouse? On the client side, we receive the mouse coordinates relative to the session window. We need to map those coordinates into the desktop image - depending on the scale factor of the user’s screen, the dimensions of the remote display, and the zoom level and scroll location of the window. The mouse coordinates are then passed over the wire to the remote system, and forwarded to the capture process **NowSession.exe**. The capturer then has to do its own work: scaling the coordinates again, then mapping them into the “virtual screen” - Windows expects injected mouse coordinates to be normalized in a space covering all monitors and represented between 0 and 65535. You can easily see how a bug with scaling somewhere along the road could cause an offset cursor, but the fact it worked nearly all the time left us scratching our heads. A simple calculation bug would surely be easy to reproduce, not come and go seemingly at random?

Analysis of the code showed one suspicious area - to normalize the mouse coordinates into the virtual screen (the combined desktop of all attached monitors), we have to discover its dimensions and we do that with the Win32 function **GetSystemMetrics**. For example:

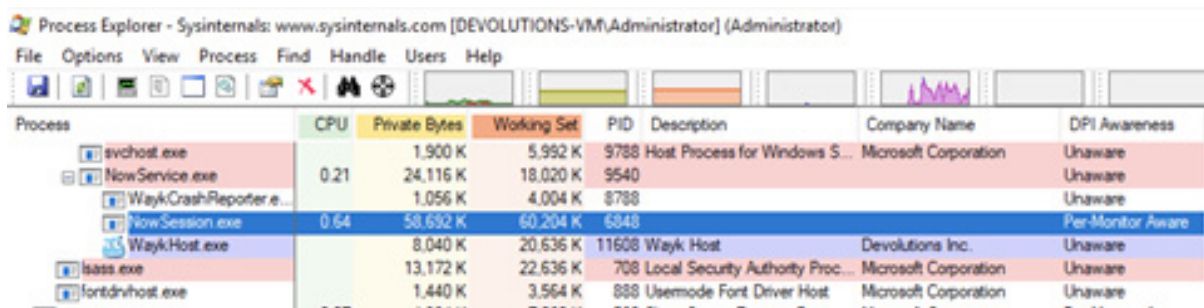
```
int virtualDeskWidth = GetSystemMetrics(SM_CXVIRTUALSCREEN);  
int virtualDeskHeight = GetSystemMetrics(SM_CYVIRTUALSCREEN);
```

Seems pretty normal, but we are calling these functions on every mouse event - if the result were to change mid-session, could that cause this effect? The desktop layout isn’t changing - that would be obvious to the user, so something must be affecting the return value of this function. Let’s take a look at the [documentation](#):

This API is not DPI aware, and should not be used if the calling thread is per-monitor DPI aware

High DPI, Low Times

Remember how I said the Wayk Now client opts-out of DPI virtualization? There are two ways to do that; by embedding a manifest (a specially crafted XML file detailing the executable capabilities), or calling one of a handful of Win32 functions. On Windows 10 you can see this for yourself: open Task Manager, switch to the details tab and add the column “DPI Awareness”. You’ll be able to see which applications are aware of high DPI and what level of support they have. And that’s exactly what we did. But wait a moment - you remember up above I mentioned our capture process, “NowSession.exe”? It shows “Per Monitor” DPI aware, but how? It’s a [rust](#) binary and, without being built by the relevant Microsoft tooling, contains no manifest; nor does it wrap any of the relevant function calls. It’s an oversight but in practice shouldn’t matter - that process doesn’t display any UI to appear blurry, as long as inputs and outputs are scaled consistently it should work fine, DPI virtualization or not. But how is it opting out?



The screenshot shows Process Explorer with a custom column for 'DPI Awareness'. The table below represents the data shown in the application:

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	DPI Awareness
svchost.exe		1,900 K	5,992 K	9788	Host Process for Windows S...	Microsoft Corporation	Unaware
NowService.exe	0.21	24,116 K	18,020 K	9540			Unaware
WaykCrashReporter.e...		1,056 K	4,004 K	8788			Unaware
NowSession.exe	0.64	58,652 K	60,204 K	6848			Per-Monitor Aware
Wayk.Host.exe		8,040 K	20,636 K	11608	Wayk Host	Devolutions Inc.	Unaware
lsass.exe		13,172 K	22,636 K	708	Local Security Authority Proc...	Microsoft Corporation	Unaware
fontdrvhost.exe		1,440 K	3,564 K	888	Usermode Font Driver Host	Microsoft Corporation	Unaware

The Plot Thickens

[Process Explorer](#) is like Task Manager on steroids. It gives a good insight into the state of running applications on your machine, and examining the “Environment” tab gives us a good clue:

`__COMPAT_LAYER=HighDpiAware`

Um, excuse me? Where did that come from? Following the hunch, we went searching in the registry and found this:



The screenshot shows the Registry Editor window with the following path selected: `Computer\HKEY_USERS\S-1-5-18\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers`. The table below represents the registry data shown:

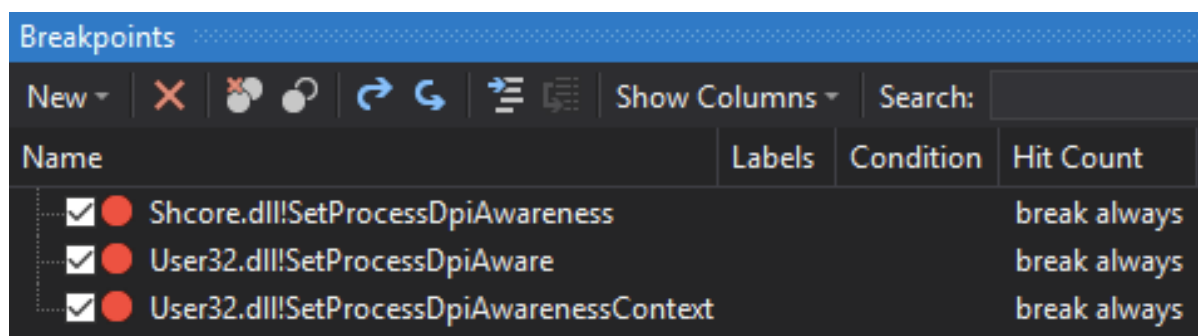
Name	Type	Data
[Default]	REG_SZ	(value not set)
C:\Program Files\Devolutions\Wayk Now\NowSession.exe	REG_SZ	HIGHDPIAWARE

(S-1-5-18 is the SID of the SYSTEM account - NowSession.exe is a privileged process and runs as SYSTEM)

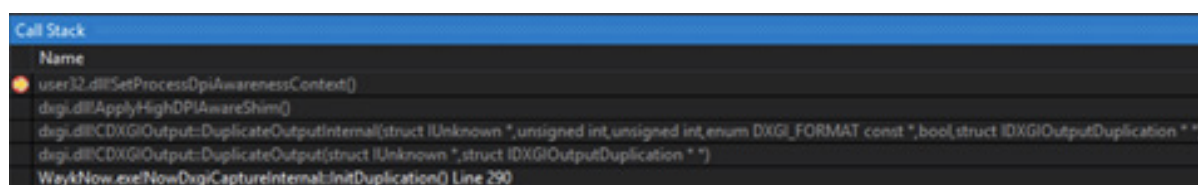
Some experimentation reveals that this registry key is an undocumented way for a user to opt-out individual applications from DPI virtualization (something that's also available in the "Compatibility" options of the executable properties in Windows Explorer). But we didn't create it, so who did? Let's delete it and see what happens...

Gone Fishin'

Visual Studio allows setting "function" breakpoints, which let us break on a named function even if it's not in our code. For good measure, we set a function breakpoint on the various functions that enable DPI awareness in a process:



Open Wayk Now and start a session, and straight away we caught a fish:



DXGI is the Windows API that we use to provide hardware accelerated screen capture. At this point we're about to start capturing the display - but we've already initialized the internal capture state, and since the process is currently DPI virtualized, everything is based on 96 DPI. Stepping forward in the debugger, our process becomes DPI aware (and the secret compatibility registry entry magically appears). Remember that **GetSystemMetrics** function call that happens on every mouse event? It will now return metrics based on the actual scale factor of the screen, throwing off our mouse calculations and offsetting the cursor. Subsequent sessions won't suffer this effect - the compatibility registry key has been created, and our capture process will be DPI aware from the get-go.

The Fix

The fix in this case turns out to be straightforward: ensure our capture process opts-out of DPI virtualization upfront, using one of the sanctioned techniques I mentioned earlier. The takeaways from this “higgs-bugson” are a little more interesting; Jeff Atwood famously wrote [“It’s always your fault”](#) - the tendency to blame your system or tools for errors over your own code. Well, in this case at least part of the blame lies in the undocumented **dxgi.dll!ApplyHighDpiAwareShim**; duplicating a high DPI display from a DPI virtualized application not only causes DXGI to adjust your processes’ DPI awareness (Microsoft itself says “[While specifying] via API is supported, it is not recommended”, with even more stark warnings in the [documentation](#)), but to set a hidden compatibility flag changing the behaviour on all future launches. What made this bug extra fun is the intricacy of scaling - depending on the scale factors of the respective displays in the Wayk session and even how the mouse is used, the issue may be more or less apparent - becoming more obvious the further your mouse moves from the top-left corner! And of course, a reconnection (or even display or desktop change) reinitializes the capturer with consistent scaling settings and the issue is gone forever - at least, on that machine...

I’m happy to report that this particular “higgs-bugson” is gone for good in the next release of Wayk Now! And we have a new inside joke: sometimes, it turns out not to be DNS related...