A closer look at the stack guard page

devblogs.microsoft.com/oldnewthing/20220203-00

February 3, 2022

Raymond Chen

In a discussion of why `IsBadXxxPtr` should really be called `CrashProgramRandomly`, I gave a brief description of the stack guard page:

> The dynamic growth of the stack is performed via guard pages: Just past the last valid page on the stack is a guard page. When the stack grows into the guard page, a guard page exception is raised, which the default exception handler handles by committing a new stack page and setting the *next* page to be a guard page.

Let's break this down a bit more.

Here's a thread's stack after the thread has been running for a little while. As is customary in memory diagrams, higher addresses are at the top, which means that the stack grows downward (toward lower addresses).

valid stack	committed	
	committed	
	committed	← Stack pointer
	committed	
	guard page	
	reserved	
	reserved	
	reserved	

The regular committed pages encompass all of the stack memory that the program has used so far. It may not be using all of it right now: Any memory beyond the red zone is off limits to the application. When the stack pointer recedes from its high water mark, the pages left behind are not decommitted.

The page just past the stack pointer's high water mark is a special type of committed page known as a guard page. A guard page is a page which raises a `STATUS_GUARD_PAGE_VIOLATION` exception the first time it is accessed.

Suppose that the stack pointer moves into the guard page, indicating that the thread has increased its stack requirements by one additional page.

valid stack	committed	
	committed	
	committed	
	committed	
	guard page	← Stack pointer
	reserved	
	reserved	
	reserved	

The moment the thread accesses memory from the guard page, the system converts it to a regular committed page (removing the `PAGE_GUARD` flag) and raises a `STATUS_GUARD_PAGE_VIOLATION` exception. The default exception handler deals with the exception by looking to see if the address lies in the current stack's guard page region. If so, then it upgrades the next reserved page to a guard page, and then resumes execution:

	Before		During		After	
valid stack	committed		committed		committed	valid stack
	committed		committed		committed	
	committed		committed		committed	
	committed		committed		committed	
	guard page	← Stack pointer →	committed	← Stack pointer →	committed	
	reserved		reserved		guard page	
	reserved		reserved		reserved	

reserved	reserved	reserved

Clearing the `PAGE_GUARD` flag on an access to a guard page means that once you access it, it stops being a guard page. This means that guard pages raise the guard page exception only on first access. If you fail to take action on a guard page exception, the system ignores it, and you lost your one chance to do something.

This is why our code to detect stack overflows makes sure to call `_resetstkoflw()` if it decides to recover. Resetting the stack overflow state consists of turning the `PAGE_GUARD` flag back on for the guard page, restoring the page to its former glory as a guard page so it can do its job of detecting stack growth.

This is how things go when everything is working right. But things don't always work right.

If one thread accesses another thread's guard page, perhaps due to a buffer overflow, or just an uninitialized pointer variable that happens to point there, that too will trigger the guard page exception. That exception is raised by the thread that did the accessing, which is not the thread that owns the stack. The default exception handler sees that the guard page exception is not for the current thread's stack, so it ignores it.[1]

Congratulations, your stack is now corrupted, because the guard page is gone:

valid stack	committed	
	committed	
	committed	← Stack pointer
	committed	
	committed	(oops)
	reserved	
	reserved	
	reserved	

Things proceed normally for a while, until the thread's stack needs to grow into what used to be the guard page.

valid stack	committed
	committed

	committed	
	committed	
	committed	← Stack pointer (oops)
	reserved	
	reserved	
	reserved	

Normally, this would trigger a guard page exception, and the system would do the usual thing of upgrading the next reserved page to a new guard page. However, that page is no longer a guard page, so execution just continues normally with no action taken.

Things still proceed as if everything were perfectly normal, but the consequences of your misdeeds finally catch up to you when the stack pointer crosses into a *second* new page, the first *reserved* page.

valid stack	committed	
	committed	
	committed	
	committed	
	committed	(oops)
	reserved	← Stack pointer (double oops)
	reserved	
	reserved	

This is also not a guard page, so no special stack expansion kicks in. You just get a stack overflow exception and die.

Such is the sad life of invalid memory access. You can corrupt your own process in a subtle way that doesn't show up until much, much later.

Next time, we'll investigate a stack overflow problem and learn how to detect whether this guard page corruption has occurred.

[1] In theory, the default exception handler could search through all the threads in the process and see if the address resides in a guard page of *any* thread, but it doesn't. One reason is that this would require cross-thread coordination with the thread whose guard page you accidentally accessed, as well as any other thread that also might be accessing that guard page at the same time. But the bigger reason is probably that the entire situation is a bug in the program anyway, and there's no point going out of your way to slow down the system in order to deal with things that programs shouldn't be doing anyway.

Raymond Chen

Follow