# Why do we even need to define a red zone? Can't I just use my stack for anything?

devblogs.microsoft.com/oldnewthing/20190111-00

January 11, 2019

Raymond Chen

On Windows, the stack grows downward from high addresses to low. This is sometimes architecturally defined, and sometimes it is merely convention. The value pointed-to by the stack pointer register is the value at the top of the stack, and values deeper on the stack reside at higher addresses. But what's up with the data at addresses *less than* the stack pointer?

| | | |
|---|---|---|
| | ⋮ | |
| top of stack | valid stack data<br>valid stack data<br>valid stack data | ←stack pointer |
| | below the stack<br>below the stack<br>below the stack | land of mystery |
| | ⋮ | |

The platform conventions for some but not all architectures define a *red zone*, which is a region of the stack below the stack pointer that is still valid for applications to use.

| | | |
|---|---|---|
| | ⋮ | |
| top of stack | valid stack data<br>valid stack data<br>valid stack data | ←stack pointer |
| | still valid<br>still valid | red zone |
| | here<br>be<br>dragons | off limits |

⋮

For Windows, the size of the red zone varies by architecture, and is often zero.

| Architecture | Red zone size |
|---|---|
| x86 | 0 bytes |
| x64 | 0 bytes |
| Itanium | 16 bytes* |
| Alpha AXP | 0 bytes |
| MIPS32 | 0 bytes |
| PowerPC | 232 bytes |
| ARM32 | 8 bytes |
| ARM64 | 16 bytes |

* The Itanium is unusual in that the red zone is placed *above* the stack pointer, rather than below it.

In the case of the PowerPC, the red zone is a side effect of the calling convention.

Any memory below the stack beyond the red zone is considered volatile and may be modified by the operating system at any time.

But seriously, why does the operating system even care what I do with my stack? I mean, it's *my* stack! The operating system doesn't tell me what to do with memory I allocate via `VirtualAlloc`. What makes the stack any different from any other memory?

Consider the following sequence on x86:

```
MOV     [esp-4], eax        ; save eax below the stack pointer
MOV     ecx, [esp-4]        ; read it into ecx
CMP     ecx, eax            ; are they the same?
JNZ     panic               ; N: something crazy happened[1]
```

Can the jump be taken?

Since there is no red zone on x86, the memory at negative offsets relative to the stack pointer may be overwritten at any time. Therefore, the above sequence is permitted to jump to `panic`.

A debugger may use the memory beyond the red zone as a convenient place to store some data. For example, if you use the `.call` command, the debugger will perform the nested call on the same stack, and likely use some of that stack space to preserve registers so that they can be restored after the `.call`ed function returns. Any data stored beyond the red zone will therefore be destroyed.

Even during normal operation, it's possible for the operating system to overwrite data beyond the red zone at any time. Here's one scenario where it can happen:

Suppose your thread gets pre-empted immediately after you store the data beyond the red zone. While your thread is waiting for a chance to resume execution, the memory manager pages out the code. Eventually, your thread resumes execution, and the memory manager tries to page it back in. Oh no, there's an I/O error during the page-in! The operating system pushes an exception frame onto the stack for the `STATUS_IN_PAGE_ERROR`, clobbering the data you had been hiding beyond the red zone.

The operating system then dispatches the exception. It goes to a vectored exception handler, which some other part of your program had installed specifically to handle this possibility, because your program might be run directly off a CD-ROM or unreliable network. The program displays a prompt to ask the user to reinsert the CD-ROM and offers an opportunity to retry. If the user says to retry, then the vectored exception handler returns `EXCEPTION_CONTINUE_EXECUTION`, and the operating system will restart the failed instruction.

This time, the restart succeeds because the CD-ROM is present and the code can be paged back in. The next instruction runs, the one that loads the beyond-the-red-zone value into the *ecx* register, but it doesn't load the value stored by the previous instruction because the `STATUS_IN_PAGE_ERROR` exception overwrote it. The comparison fails, and we jump to the label `panic`.

If you want to store data on the stack, push it properly: Decrement the stack pointer first, and then store the value onto the valid portion of the stack. Don't hide it beyond the red zone. That memory is volatile and may vanish out from under you.

[1] The coding convention for assembly language[2] says that comments for jump instructions should describe the result if the jump is taken. In the example above, the `CMP` instruction asks the question, "Are they the same?", and the `JNZ` instruction jumps if they are not equal. The comment therefore begins with "N:" indicating that the jump is taken if the answer to the previous question is *No*, and the rest of the comment describe what it means when the jump is taken.

[2] Yes, we have a coding convention for assembly language.

Raymond Chen

## Follow