How can a program survive a corrupted stack?

devblogs.microsoft.com/oldnewthing/20040116-00

January 16, 2004

Raymond Chen

Continuing from yesterday:

The x86 architecture traditionally uses the EBP register to establish a stack frame. A typical function prologue goes like this:

```
push ebp       ; save old ebp
mov  ebp, esp  ; establish new ebp
sub  esp, nn*4 ; local variables
push ebx       ; must be preserved for caller
push esi       ; must be preserved for caller
push edi       ; must be preserved for caller
```

This establishes a stack frame that looks like this, for, say, a __stdcall function that takes two parameters.

.. rest of stack ..	
param2	
param1	
return address	
saved EBP	<- EBP
local1	
local2	
…	
local-nn	
saved EBX	
saved ESI	

saved EDI	<- ESP

Parameters can be accessed with positive offsets from EBP; for example, param1 is [ebp+8]. Local variables have negative offsets from EBP; for example, local2 is [ebp-8].

Now suppose that a calling convention or function declaration mismatch occurs and extra garbage is left on the stack:

.. rest of stack ..	
param2	
param1	
return address	
saved EBP	<- EBP
local1	
local2	
…	
local-nn	
saved EBX	
saved ESI	
saved EDI	
garbage	
garbage	<- ESP

The function doesn't really feel any damage yet. The parameters are still accessible at the same positive offsets and the local variables are still accessible at the same negative offsets.

The real damage doesn't occur until it's time to clean up. Look at the function epilogue:

```
pop  edi       ; restore for caller
pop  esi       ; restore for caller
pop  ebx       ; restore for caller
mov  esp, ebp  ; discard locals
pop  ebp       ; restore for caller
retd 8         ; return and clean stack
```

In a normal stack, the three “pop” instructions match with the actual values on the stack and nobody gets hurt. But on the garbage stack, the “pop edi” actually loads garbage into the EDI register, as does the “pop esi”. And the “pop ebx” – which thinks it’s restoring the original value of EBX – actually loads the original value of the EDI register into EBX. But then the “mov esp, ebp” instruction fixes the stack back up, so the “pop ebp” and “retd” are executed with a repaired stack.

What happened here? Things sort of got put back on their feet. Well, except that the ESI, EDI, and EBX registers got corrupted. If you’re lucky, the values in ESI, EDI and EBX weren’t important and could have survived corruption. Or all that was important was whether the value was zero or not, and you were lucky and replaced one nonzero value with another. For whatever reason, the corruption of those three registers is not immediately apparent, and you end up never realizing what you did wrong.

Maybe the corruption has a subtle effect (say, you changed a value from zero to nonzero, causing the caller to go down the wrong codepath), but it’s subtle enough that you don’t notice, so you ship it, throw a party, and start the next project.

But then a new compiler comes along, say one that does FPO optimization.

FPO stands for “frame pointer omission”; the function dispenses with the EBP register as a frame register and instead just uses it like any other register. On the x86, which has comparatively few registers, an extra arithmetic register goes a long way.

With FPO, the function prologue goes like this:

```
sub  esp, nn*4 ; local variables
push ebp       ; must be preserved for caller
push ebx       ; must be preserved for caller
push esi       ; must be preserved for caller
push edi       ; must be preserved for caller
```

The resulting stack frame looks like this:

.. rest of stack ..
param2
param1
return address
local1
local2

…
local-nn
saved EBP
saved EBX
saved ESI
saved EDI <- ESP

Everything is now accessed relative to the ESP register. For example, local-nn is [esp+0x10].

Under these conditions, garbage on the stack is much more fatal. The function epilogue goes like this:

```
pop  edi       ; restore for caller
pop  esi       ; restore for caller
pop  ebx       ; restore for caller
pop  ebp       ; restore for caller
add  esp, nn*4 ; discard locals
retd 8         ; return and clean stack
```

If there is garbage on the stack, the four “pop” instructions will restore the wrong values, as before, but this time, the cleanup of local variables won’t fix anything. The “add esp, nn*4” will adjust the stack by what the function believes to be the correct amount, but since there was garbage on the stack, the stack pointer will be off.

.. rest of stack ..
param2
param1
return address
local1
local2 <- ESP (oops)

The “retd 8” instruction now attempts to return to the caller, but instead it returns to whatever is in local2, which is probably not valid code.

So this is an example of where optimizing your code reveals other people’s bugs.

Monday, I’ll give a much more subtle example of something that can go wrong if you use the wrong function signature for a callback.

Raymond Chen

Follow