What happened in real-mode Windows when somebody did a longjmp into a discardable segment?

devblogs.microsoft.com/oldnewthing/20140103-00

January 3, 2014

Raymond Chen

During the discussion of how real-mode Windows handled return addresses into discarded segments, Gabe wondered, "What happens when somebody does a `longjmp` into a discardable segment?" I'm going to assume that everybody knows how `longjmp` traditionally works so I can go straight to the analysis. The reason `longjmp` is tricky is that it has to jump to a return address that isn't on the stack. (The return address was captured in the `jmp_buf`.) If that segment got relocated or discarded, then the jump target is no longer valid. It would have gotten patched to a return thunk if it were on the stack, but since it's in a `jmp_buf`, the stack walker didn't see it, and the result is a return address that is no longer valid. (There is a similar problem if the data segment or stack segment got relocated. Exercise: Why don't you have to worry about the data segment or stack segment being discarded?) Recall that when a segment got discarded, all return addresses which pointed into that segment were replaced with return thunks. I didn't mention it explicitly in the original discussion, but there are three properties of return thunks which will help us here:

- It is safe to invoke a return thunk even if the associated code segment is in memory. All that happens is that the "ensure the segment is present" step is a nop, and the return thunk simply continues with its work of recovering the original state.
- It is safe to abandon a return thunk without needing to do any special cleanup. All the state used by the return thunk is stored in the patched stack itself, so if you want to abandon a return thunk, all you need to do is free the stack space.
- It is safe to reuse a return thunk. Since they are statically allocated, you can use them over and over as long as the associated code segment has not been freed.

The first property (idempotence of the return thunk) is no accident. It's required behavior in order for return thunks to work at all! After all, if the segment was loaded (say by a direct call or some *other* return thunk), then the return thunk needs to say, "Well, I guess that was easy," and simply skip the "load the target segment" step. (It still needs to do the rest of the work, of course.) The second property (abandonment) is also no accident. An application might decide to exit without returning all the way to `WinMain` (the equivalent of calling `ExitProcess` instead of returning from `WinMain`). This would abandon all the stack

frames between the exit point and the `WinMain`. The third property (reuse) is a happy accident. (Well, it was probably designed in for the purpose we're about to put it to right here.) Okay, now let's look at the jump buffer again. If you've been following along so far, you may have guessed the solution: Pre-patch the return address as if it had already been discarded. If it turns out that the segment was discarded, then the return thunk will restore it. If the segment is present (either because it was never discarded, or because it was discarded and reloaded, possibly at a new address), the return thunk will figure out where the code is and jump to it. Actually, since the state is being recorded in a `jmp_buf`, the tight space constraints of stack patching do not apply here. If it turns out you need 20 bytes of memory to record this information, then go ahead and make your `jmp_buf` 20 bytes. You don't have to try to make it all fit inside an existing stack frame. The `jmp_buf` therefore doesn't have to try to play the crazy air-squeezing games that stack patching did. It can record the return thunk, the handles to the data and stack segments, and the return IP without any encoding at all. And in fact, the `longjmp` function doesn't need to invoke the return thunk directly. It can just extract the segment number after the initial `INT 3Fh` and pass that directly to the segment loader. (There is a little hitch if the address being returned to is fixed; in that case, there is no return thunk. But that just makes things easier: The lack of a return thunk means that the return address cannot be relocated, so there is no patching needed at all!)

This magic with return thunks and segment reloading is internal to the operating system, so the core `setjmp` and `longjmp` functionality was provided by the kernel rather than the C runtime library in a pair of functions called `Catch` and `Throw`. The C runtime's `setjmp` and `longjmp` functions merely forwarded to the kernel versions.

Raymond Chen

Follow