"That and Only That"

Michael Barwise

The other day I attended the 2nd International Secure Systems Development Conference in London. For sheer concentration of brain power in one place at one time it's rarely been equalled. The discussions were scintillating and highly entertaining. But much of the substance centred around high level development life cycle models and post-coding testing regimes, with the tacit (and in a couple of cases explicit) assumption that coding securely ab initio is an over-expensive alternative to "conventional" coding - something that might be taught to a small elite who will be tasked with programming for "the critical infrastructure" but impractical for the mainstream development community. That seems to me a recipe for the perpetuation of the morass we're currently foundering in - frequent breaches, constant patching and no assurance that tomorrow won't see our own business doing a gold plated online digital prat fall. The reality is - it's *all* critical infrastructure now.

So I suggested we need to completely revise the way we educate programmers. They must never be taught anything other than "secure coding" - it's the only kind worth having. The responses to this at round table time ranged from "yes, we have started to engage government in a programme to work towards this" to "it's not really practical in the commercial world". But it occurs to me that you don't need a government programme and it should indeed be practical to achieve.

At the simplest level, there's just one rule in secure coding - the rule that forms the title of this blog *"That and Only That"*. You make use of it twice. First, every piece of code from the smallest module right up to the complete program must do exactly what the specification states it should *and nothing else*. Second, all data interfaces, external (facing the outside world) and internal (facing other code modules or programs) must accept only the data they require and expect in order to function as specified *and no other*.

These elementary but essential requirements have to be fulfilled at coding time. No development life cycle methodology can supervise in enough low-level detail to enforce them. It can only mandate checks for compliance at testing points, and those checks can by definition *never be exhaustive*. So the prime responsibility for fulfilling these requirements rests squarely with those at the coding desk. That means the coders have to be educated to think that way, and currently they're not. Mostly, they're taught a language and a development system, given a few in-house rules and left to get on with it.

A classic example of the result is a Windows icon vulnerability that seems ultimately to have found its way into Stuxnet. Apparently, rendering an icon with a negative value in the colour count field allowed malicious code execution via the Windows shell. Now I'm not a physicist so I may have missed something here - although I have done a lot of image processing - but I've never encountered an image with a negative number of colours. Not in the real visual world either. Ever. So it's utterly improper to use a signed integer to store that parameter. The reality of course is that the default integer type in the development environment was signed, and it was big enough to hold largest of the half-dozen or so alternative colour counts used by icons, so what the heck? Use it. Indeed the coder probably never even thought about it - "it's an int", end of discussion. But almost certainly the value in question is used as a pointer offset of some kind - maybe to create an array - so a negative value could be (and indeed was) fatal. But that never crossed the mind of the coder because all they could see was what the code should do. Not what it *could* do but *shouldn't*.

While development houses employ programmers who code with so little attention and forethought - relying almost entirely on post-facto error checking - their customers are doomed to eternal torment. Modern software is so voluminous and branches so deeply that it's impossible to test exhaustively. That's been proved mathematically. You'll never find *all* the bugs. Currently we try to alleviate the problem by devolving the low-level code generation more and more from the human to the machine - witness Java, which conceals the inescapable machine-level use of pointers from the coder. But although this approach reduces in the short term the incidence of specific errors that we've already identified, it encourages a less attentive approach to coding, which means the next generation development tools have to devolve even more from the human to the machine. But by then a whole new set of errors will have emerged. It's ultimately a zero outcome game for all players.

In order to win, we must change the rules of the game. We must employ coders who make fewer mistakes in the first place. That means we need to start the education of prospective coders, not from fancy new development systems that allow them to pay less attention, but instead from both the

obligation and ways to pay *more* attention, to exercise more forethought, and rigorously to use the rule that heads this blog. Only once they have fully and permanently embraced these essentials can they be allowed the luxury of using tools operating at high levels of abstraction for productivity's sake.

Originally appeared on the Infosecurity Network, May 2011