The Proof is in the Testing:

After writing BackTalk columns for over 15 years, I'm out of ideas. No clues. Nothing to write about. Zip. Zilch.

I usually have a great stash of ideas – and I am SO used to grabbing my iPhone, and telling Siri to remind me "Write up story of the non-working GPS" or "Maybe hibernating skunks for next column?" Not this time. Not a single idea involving Testing and Diagnostics.

Of course, I have a great excuse. I am teaching a course on high-integrity programming this semester here at the university. As any BackTalk reader for the last 15 years might know, I was (and still am) an Ada advocate. It's a really good language for teaching high-integrity coding practices. We have decided to cover more "secure programming" in our curriculum, and I wanted to test-drive a course as "Special Topics" before we make it a permanent part of the curriculum. It's keeping me pretty busy.

Nice thing about Ada – the language thinks about security for you. It gets real picky about mathematical conversions, and cheerfully lets the user know if there is a overflow or underflow in the conversion. In fact, you have to make a conscious decision to turn off error checking. And, why would you ever do that

Imagine a very simple Java program that inputs an integer from the keyboard, and adds one to it. In Java, if the number is equal to the largest possible integer, then adding one converts is to the smallest integer. It "wraps around." And vice versa – subtracting one from the smallest negative integer will give the largest possible positive integer. Note that Java is one of the most used languages used in introductory computer science – currently it's #2 in popularity. You might think that C++ would be #1, but #1 is Python. However, Java is more of a "software engineering" language than Python, and many educators expect it to move back to #1 next year. All I am saying is that if you can't trust adding 1 to a number – what can you trust? It makes teaching "safe and secure" programming a challenge.

What about other common languages? In Python, the integer is converted to a "long", which has unlimited length (obviously, implemented in software, not hardware). In C and C++, the behavior is undefined. In C#, it is possible to automatically "catch" this – but only if you are in a checked context.

What I am saying is that once you've had the thought, "Gee – I am adding to an integer here – should I worry about overflow?" you can't unthink it. It is no longer an error. It's a condition you considered, and then you made the decision to not worry about it.

And what if the software later fails because of an overflow or underflow? Of course, that is highly unlikely, right? Heard of the Ariane 5 (https://www.ima.umn.edu/~arnold/disasters/ariane.html)?

Brief summary – back in 1996 the European Space Agency launched the Ariane 5 rocket, designed to launch two satellites into orbit. The Ariane 5 was the result of over $7 billion in development, and the rocket and cargo itself cost over $500 million. On its first launch, the guidance system catastrophically failed. How was this possible? After all, the guidance software was based on the well-tested guidance software from the previous rocket version, the Ariane 4.

> *Let me quote from James Gleick (http://www.around.com/ariane.html): "…the guidance system's own computer tried to convert one piece of data -- the sideways velocity of the rocket -- from a 64-bit format to a 16-bit format. The number was too big, and an overflow error resulted. ….. in this case, the programmers had decided that this particular velocity figure would never be large enough to cause trouble. After all, it never had been before. Unluckily, Ariane 5 was a faster rocket than Ariane 4. One extra absurdity: the calculation containing the bug, which shut down the guidance system, which confused the on-board computer, which forced the rocket off course, actually served no purpose once the rocket was in the air. Its only function was to align the system before launch. So it should have been turned off. But engineers chose long ago, in an earlier version of the Ariane, to leave this function running for the first 40 seconds of flight."*

Years of work. $7 billion in development. And "the programmers had decided…" If only they had used Ada, right? As a matter of fact, the Ariane 4 and 5 software was written in Ada, which automatically checks for overflow and underflow, should have triggered a exception that could have been safely detected, handled, and recovered from. However, to quote from (http://en.wikipedia.org/wiki/Ariane_5):

The software was originally written for the Ariane 4 where efficiency considerations (the computer running the software had an 80% maximum workload requirement) led to four variables being protected with a handler while three others, including the horizontal bias variable, were left unprotected because it was thought that they were "physically limited or that there was a large margin of error."

In other words – due to hardware considerations (the Ariane 4 CPU was overloaded) – consciously ignoring automatic overflow and underflow checking saved a few machine cycles. Don't get me wrong – the Ariane 4 software was perfect for the given requirements (hardware and software). The processor in the Ariane 4 was heavily loaded – and the decision to turn off checking was probably well analyzed. It was only much, much later, when the code was reused for a different set of requirements, that the decision should have been revisited. But the code was "rock(et) solid" and well-tested – so why test it again? Copy and Paste. Control-C, Control-V, and go. Nothing to worry about, right?

It's even sadder that there was an exact backup copy of the guidance software running on the Ariane 5 – an exact copy. So when the primary guidance software failed, control was immediately transferred to the backup copy. Running the same code, and encountering the exact same unhandled error.

Software is like that. As Marin David Condic pointed out in his excellent write up found at (http://www.adapower.com/index.php?Command=Class&ClassID=FAQ&CID=328), it would be like reusing Corvette tires on a large 18 wheeler. Just because they are "tires" – the requirements and assumptions made initial development are no longer valid. What's "good" for Corvette tires might not be "good" for a fully loaded semi. Likewise, code needs to be analyzed and tested, even if it's been trusted for years. It's just not safe to reuse otherwise.

But then, we already know that testing and diagnostics are important, right?

David A. Cook
Stephen F. Austin State University

P.S. Come to think of it, maybe I do have an idea for this BackTalk than might work.