The Bower of Tabel

One of my serious research interests as a computer scientist is the vulnerabilities in programming languages. Programming languages are of course the essence of how we turn ideas into computation. In the very early days, computers were programmed right down at the 0s and 1s level and we made lots of mistakes. Gradually we developed more and more expressive languages such as COBOL for data processing applications, Fortran for numerical computation, C for systems programming, (the Linux kernel and many open source projects are written in this) and more recently C++. In the aerospace industry, Ada is often used.

In the last 40 years, we have tried to standardise these languages so that their behaviour is well-defined in every aspect so that they work the same way on all implementations and are free of surprises. Fortran was first standardised in 1966 and nowadays most languages have been standardised and often re-standardised, (C was first standardised in 1990 and again in 1999).

You might think that when we manage to standardise a language that all the i's are dotted and the t's crossed. In other words, when we turn our great new spreadsheet, database application, web service or whatever into a computer program, that the language we used is free from surprise and does exactly what we expect as defined by the standard. Unfortunately, this may be a very long way from the truth. Modern programming languages are highly complex and are standardised by committees and in spite of everybody's best efforts, the resulting language will inevitably contain compromise, inconsistency, incompleteness and some bits which simply do not work as expected for one reason or another. We call these vulnerabilities. When programmers use them, the program that contains them can and will fail in unexpected ways. Some of these such as the infamous gets() function in C have also been exploited in many security compromises.

You might also like to think that language committees and compiler implementors provide for this and warn the programmer if they inadvertently rely on such a feature. Unfortunately, this is usually not the case and as a result, such failures occur with monotonous regularity and with a similar frequency to 20 years ago. All languages contain such problems and some are worse than others. It is perfectly possible to avoid them but as can be seen in a number of experiments old and new, programmers often do not, usually through ignorance, which is of course no excuse.

I was reminded of this because of a delicious irony which occurred recently. The language vulnerability working group of which I am a member was sent a copy of a labyrinthine set of rules for C++ which without any measurement support, purported to make it safer for use in aerospace use. Now C++ is arguably the most complex programming language ever standardised and is notorious for unexpected behaviour. The resulting set of rules was distributed as a Word file. Unfortunately, this file crashed a number of the copies of Word in use by my colleagues when opened, (Word 2000 and some but not all service packs of Word 2003). Word is far more ubiquitously tested than any aerospace software is likely to be, so for a special prize, guess what language Word is written in ...

l.hatton@kingston.ac.uk