Document WG 14/N1667
Date: 2012-12-05

Secure C coding rules, introductory words

These rules may have three uses, as:

- guidance to programmers developing new code
- a benchmark for reviewers of existing code
- a specification for static tool developers

The rules are designed to provide a check against a set of programming flaws that are known from practical experience to have led to security issues. Whilst rule checking can be performed manually, with increasing program complexity, this rapidly becomes infeasible, hence the use of static analysis tools are recommended.

However, it should be recognised that, in general, these SecureC rules are computationally undecideable, as indeed, are most coding rules. This is a consequence of the 'halting' theorem of Computer Science [*ref*]. This theorem states that it is not, in general, possible to statically determine the exact control flow of a program. This means that any property dependant on control flow cannot always be determined. A consequence of this undecideable is that any tool may be unable to determine whether or not a given rule is satisfied in specific circumstances. In addition, the widespread presence of infeasible code in most projects may also lead to unexpected results from an analysis tool.

However checking is performed the analysis may generate:

- false negatives: failure to report a real flaw in the code. This is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security. As such most tools err on the side of caution, which tends to generate false positives. However, there may be cases where it is deemed better to report some 'high risk' flaws and miss others, rather than overwhelm the user with false positives.
- false positive: the tool reports a flaw when actually there isn't one. This may occur because the code becomes sufficiently complex that the tool cannot perform a complete analysis. The use of features such as function pointers and libraries may make false positives more likely
- inappropriate true positive: the tool correctly reports a flaw in the program (as defined by the coding rules). However, there is either:
    - a good reason why in these specific circumstances the rule should be ignored. For example, rule [boolasgn] largely prevents the use of assignment in conditional contexts: `if (x = y)...` being reported as an error as `if (x == y)...` may have been intended. The rule allows three exceptions, where the programmer can be expected to know and require that the behaviour is assignment rather than comparison, e.g. `if ((x = y) != 0)...` However, there may be other circumstances where the programmer knows the program will have the correct behaviour, but which are not covered by the exceptions.
    - the user is aware of information that the analysis tool is not, that makes the concern of the rule inappropriate. For example, many of the rules involve the use of

> tainted values, i.e. those outside the control of the program. It may be that the user knows that a 'tainted' value is being read from a file inside a firewall, which cannot be manipulated by an attacker, so the value can be regarded as trusted rather than tainted.

As it is unrealistic to expect any static analysis to be 100% accurate or appropriate, it is recommended that users of these rules put into place a 'deviation' mechanism, to handle false positives and inappropriate true positives. However, it is not the role of this standard to define such a management mechanism.

One approach to a 'deviation' mechanism would be to ensure that when a positive error report is received that is believed to be false or inappropriate, then a justification is recorded to document why, in these specific circumstances, the report should be ignored. The justification should be expressed as though there were an expectation that it may be reviewed by an independent (and possibly critical) third party, who will need to find the argument presented compelling.

Blanket deviations, such as 'we're ignoring all reports against rule XYZ', are inappropriate.

There is also the possible scenario whereby a tool reports a flaw but, due to the complexity and undecidability of the analysis, it is not possible for the users to determine either the accuracy of the diagnosis or what steps are needed to remove the alleged flaw. In such circumstances, the use of a deviation mechanism may be the only alternative to a substantial code rewrite. This problem is one argument for the production of simple code.