Write Safe AND Secure Application Code with MISRA C:2012

Introduction

When examined with a critical eye, the commonly held belief that security and safety critical code are hugely different to each other is a conundrum. Why would that be?

Within the safety domain, the aim for software developers is to produce code that performs as required, whilst ensuring that erroneous behaviour does not result in an accident.

Within the security domain, their aim is to produce software that performs as required whilst ensuring that manipulation of input data does not result in denial of service or the leaking of sensitive data.

Best practise for the development of either safety or security critical code is to apply a formalised software development process, starting with a set of requirements and tracing those requirements through to executable code. Undefined, unspecified and implementation-defined behaviours within the C language can lead to safety or security failures. And data handling errors such as invalid values, domain violations, tainted data, and leaking of confidential information can prevent both safety and security objectives from being realised.

With so much commonality between perceived optimal practices for safety and security critical code, it is a puzzle as to why there is a common misconception that MISRA[i] is just for safety-related not for security-related projects. In response to that misconception, in April 2016, MISRA released "MISRA C:2012 – Addendum 2[ii]" which highlights which of the 46 C Secure[iii] rules are covered by the MISRA C:2012[iv] guidelines.

Even though MISRA C:2012 Amendment 1[v] was written to further ensure complete coverage of the C Secure rules in the MISRA C:2012 standard, to a large extent it does so by enhancing the language of existing checks. For the most part, these enhancements explain why those checks are important from a security perspective with reference to the ISO C Secure Guidelines, particularly with regards to the use of "untrustworthy data."

In other words, the original MISRA C:2012 document has always targeted concerns such as buffer overruns and memory errors, and they have always been important for both safety and security. It has always promoted the detection of inconsistent data use, pertinent for all critical code. More generally, it has always aimed to ensure that defects are not introduced into the code, rather than adopting a set of checks to try and identify them after the fact.

The Importance of Process Standards and Guidelines

Safety-critical industries such as aerospace, automotive, rail, and medical, use process standards that address the rigour in which activities need to be performed during the development life cycle stages with respect to the functional safety of the system being developed. Coding standards and guidelines, such as MISRA C, are a critical part of this process. MISRA C defines a subset of the C language suitable for developing any application with high-integrity or high-reliability requirements. Although MISRA guidelines were originally designed to promote the use of the C language in safety-critical

embedded applications within the motor industry, they have gained widespread acceptance in many other industries as well.

The illustration in Figure 1 is a typical example of a table from ISO 26262-6:2011[vi], which mirrors similar tables both in IEC 61508[vii], and in other derivatives such as IEC 62304[viii] (used in the development of medical devices). It shows the coding and modelling guidelines to be enforced during implementation, superimposed with an indication of where compliance can be confirmed using automated tools.

These guidelines combine to make the resulting code more reliable, less prone to error, easier to test, and/or easier to maintain.

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity	++✓	++✓	++✓	++✓
1b	Use of language subsets	++✓	++✓	++✓	++✓
1c	Enforcement of strong typing	++✓	++✓	++✓	++✓
1d	Use of defensive implementation techniques	O✓	+✓	++✓	++✓
1e	Use of established design principles	+✓	+✓	+✓	++✓
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+✓	++✓	++✓	++✓
1h	Use of naming conventions	++✓	++✓	++✓	++✓

"++" The method is highly recommended for this ASIL. "+"
The method is recommended for this ASIL.
"o" The method has no recommendation for or against its usage for this ASIL.
✓ Supported by the LDRA tool suite

Figure 1 - Mapping the capabilities of the LDRA tool suite to "Table 6: Methods for the verification of the software architectural design" specified by ISO 26262-6:2011

The Safe and Secure System

The enterprise computing community has traditionally taken a "fail-first and patch-later" approach to secure system development. This development life-cycle consists of a largely laissez-faire attitude to development, and the subsequent application of penetration tests, fuzz tests and fault injection to expose and correct any unwanted behaviour. Such a reactive approach is not adequate when safety critical applications are involved, where functional safety standards already demand a much more proactive development approach (Figure 2) – and that proactive attitude is equally essential where a connected system must be dependable, trustworthy and resilient in order to protect critical data.

Developers of functionally safe systems in accordance with such as DO-178, ISO 26262 and IEC 61508 are required to perform a functional safety risk assessment as part of the development lifecycle. Not only does it make sense to mirror that approach to perform a functional security risk assessment, but it is obligatory if those security risks represent a potential safety risk too. The identification of security risks involved in developing and deploying the product should be assessed and mitigation activities reflected in the security requirements. The design and coding stages can then also reflect the aspects of security requirements along with functional and non-functional requirements.

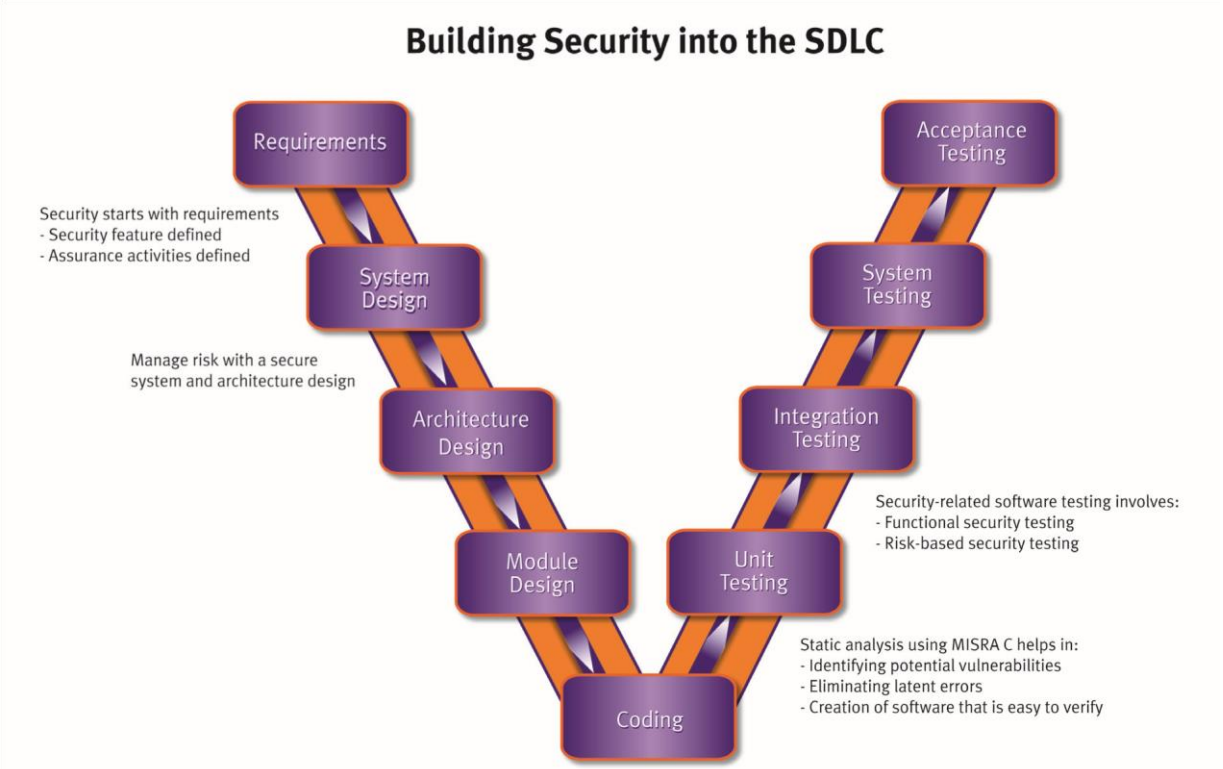


Figure 2 - The traditional V software development life cycle model incorporates security activities from the early stages

MISRA C Security Amendments

After the publication of MISRA C:2012, the WG14[ix] committee responsible for maintaining the C standard published the ISO/IEC 17961:2013 C Language Security Guidelines[x], designed to limit the use of the C language to a subset excluding the more vulnerable features of the language. The intention was for all rules to be enforceable using static analysis such that their detection could be automated without generating excessive false positives.

It was in response to ISO/IEC 17961 that the MISRA committee developed “MISRA C:2012 – Addendum 2”, highlighting which of the 46 C Secure rules are covered by the original MISRA C: 2012 guidelines. MISRA C:2012 Amendment 1 was written to further ensure complete coverage of the C Secure rules. The amendment is an extension MISRA C:2012.

It establishes 14 new guidelines for secure C coding to improve the coverage of the concerns highlighted by the ISO C Secure Guidelines including, for example, issues pertaining to the use of “untrustworthy” data—a well-known security vulnerability. By following the additional guidelines, developers can more thoroughly analyse their code and can assure regulatory authorities that they have adopted best practice. This is becoming critical in many fields of endeavour including the automotive industry, the Industrial Internet of Things (IIoT), and the medical device sector – in short, wherever security threats have led to OEM demands for developers to prove that their software meets the highest standards for security as well as safety.

Insecure Coding Examples and Related rules

To put the amendment into context, it is useful to review examples of where the additional rules apply.

Example 1: Rule 12.5

Rule 12.5 states, "The sizeof operator shall not have an operand which is a function parameter declared as "array of type"

Many developers use the "sizeof" operator to calculate the size of an array. In a normal scenario that works fine. But when that approach is used on an array passed as a function parameter, that parameter is passed as a "pointer to type." Consequently the attempt to calculate the number of elements usually returns an incorrect value, as illustrated in Figure 2 – and in this case, results in an array bound being exceeded.

array.c

```
1   #include <stddef.h>
2   #include <stdint.h>
3
4   static int32_t f ( int32_t A[ 2 ] )
5   {
6     size_t size = sizeof ( A ) / sizeof ( A[ 0 ] );
7
8     return A[ size - 2 ];
9   }
10
11
12  int main ( void )
13  {
14    int32_t A[] = { 1, 2 };
15
16    return f( A );
17  }
18
```

Figure 2: Source code example

A static analysis tool can be used to check for the use of such syntax (Figure 3).

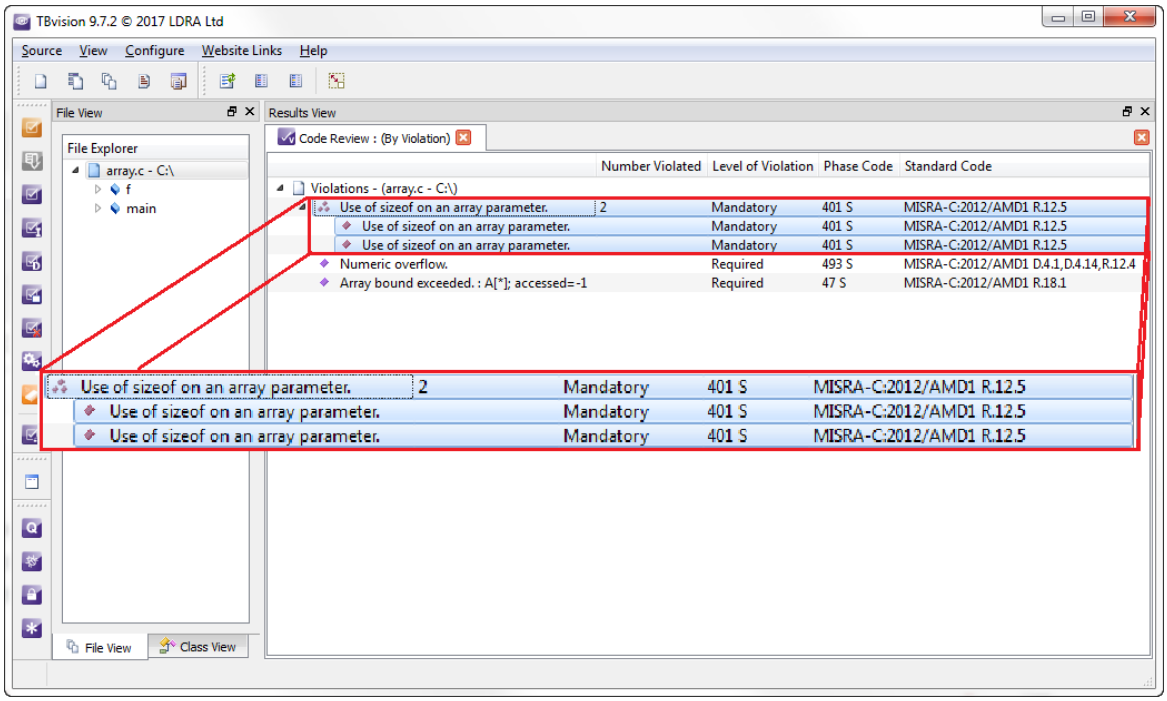


Figure 3 - The LDRA TBvision tool detects the MISRA C:2012 rule violation for the "sizeof" operator example

Example 2: Rule 22.7

Rule 22.7 states "The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF."

An "EOF" (End Of File) return value from standard library functions is used to indicate that the relevant stream has either reached the end of the file, or an error has occurred in reading from or writing to that file. The macro "EOF" is defined as an "int" with a negative value.

If the "EOF" value is captured in a variable of incorrect type, then it may become indistinguishable from a valid character code. It is therefore important to use an "int" to store the return code from such functions such as "getchar()" or "fgetc()", and to avoid because the common practice of storing the result in a char.

```
void f1 (void)
{
   char ch;
   ch = (char)getchar();
   if (EOF != (int) ch)
   /* Non-compliant - getchar returns an int which is cast to a narrower type */
     {
     }
}
```

Automatic Detection of Rule Violation at an Early Stage

Peer reviews represent a traditional approach to enforcing adherence to such guidelines, and whilst they still have an important part to play, automating the more tedious checks using tools is far more efficient, less prone to error, repeatable, and demonstrable (Figure 4).

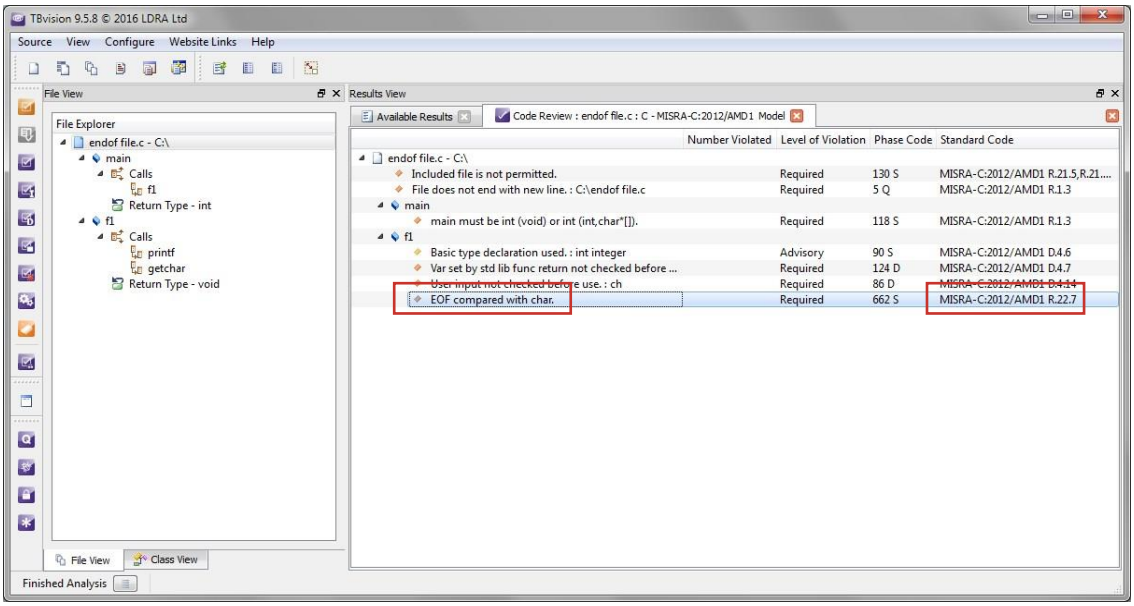


Figure 4 - LDRA TBvision reports the Rule 22.7 (EOF comparison with char) violation in the source code

Choosing a language subset

Although there are several language subsets (or less formally, “coding standards”) to choose from, these have traditionally been focused primarily on safety, rather than security. More lately with the advent of such as the Industrial Internet of Things, connected cars, and connected heart pacemakers, that focus has shifted towards security to reflect the fact that systems such as these, once naturally secure through isolation, are now increasingly accessible to aggressors.

There are, however, subtle differences between the differing subsets which is perhaps a reflection of the development dichotomy between designing for security, and appending some measure of security to a developed system. To illustrate this, it is useful to compare and contrast the approaches taken by the authors of MISRA C and CERT C with respect to security.

Retrospective adoption

MISRA C:2012 categorically states that “MISRA C should be adopted from the outset of a project. If a project is building on existing code that has a proven track record then the benefits of compliance with MISRA C may be outweighed by the risks of introducing a defect when making the code compliant.”

This contrasts in emphasis with the assertion of the CERT C[xi] authors that although “the priority of this standard is to support new code development…. A close-second priority is supporting remediation of old code”

Of course, as with the system as a whole, the level of risk involved with the compromise of the system will reflect on the approaches to be adopted. Certainly, the retrospective application of any language subset is better than nothing, but late adoption does not represent best practice.

Relevance to safety, high integrity and high reliability systems

MISRA C:2012 “define[s] a subset of the C language in which the opportunity to make mistakes is either removed or reduced. Many standards for the development of safety-related software require, or recommend, the use of a language subset, and this can also be used to develop any application with high integrity or high reliability requirements”. The accurate implication of that statement is that MISRA C was always appropriate for security critical applications even before the security enhancements introduced by MISRA C:2012 Amendment 1.

CERT C attempts to be more all-encompassing, as reflected in its introductory suggestion that “safety-critical systems typically have stricter requirements than are imposed by this standard … However, the application of this coding standard will result in high-quality systems that are reliable, robust, and resistant to attack”.

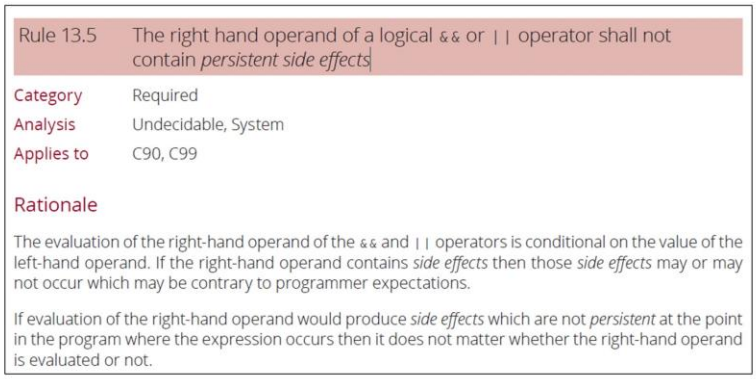
Decidability

The primary purpose of a requirements-driven software development process as exemplified by ISO 26262 is to control the development process as tightly as possible to minimize the possibility of error or inconsistency of any kind. Although that is theoretically possible by manual means, it will generally be far more effective if software tools are used to automate the process as appropriate.

In the case of static analysis tools, that requires that the rules can be checked algorithmically. Compare, for example, the excerpts shown in Figure 5, both of which address the same issue. The approach taken by MISRA is to prevent the issue by disallowing the inclusion of the pertinent construct. CERT C instead asserts that the developer should “be aware” of it.

Of course, there are advantages in each case. The CERT C approach is clearly more flexible; something of particular value if rules are applied retrospectively. MISRA C:2012 is more draconian, and yet by avoiding the side effects altogether the resulting code is certain to be more portable, and it can be automatically checked by a static analysis tool. It is simply not possible for a tool to check whether a developer is “aware” of side effects – and less possible still to ascertain whether “awareness” equates to “understanding”.

The net effect is that a static analysis tool can make the same checks but the detection of an issue has different implications. For MISRA – "You have a violation that either needs to be removed or a deviation introduced". For CERT – "Did you mean to do this this"? The former is clearly easier to police.



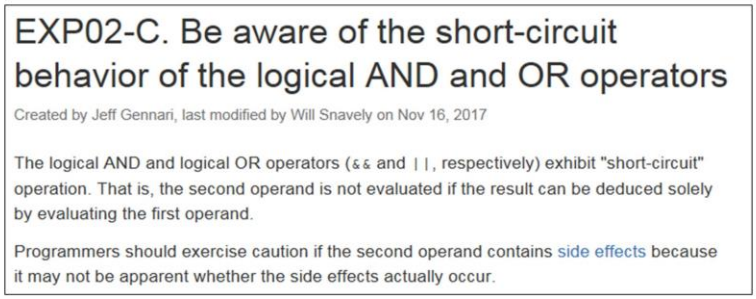
Rule 13.5	The right hand operand of a logical && or || operator shall not contain *persistent side effects*
Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

The evaluation of the right-hand operand of the && and || operators is conditional on the value of the left-hand operand. If the right-hand operand contains *side effects* then those *side effects* may or may not occur which may be contrary to programmer expectations.

If evaluation of the right-hand operand would produce *side effects* which are not *persistent* at the point in the program where the expression occurs then it does not matter whether the right-hand operand is evaluated or not.

Excerpt from MISRA C:2012



EXP02-C. Be aware of the short-circuit behavior of the logical AND and OR operators

Created by Jeff Gennari, last modified by Will Snavely on Nov 16, 2017

The logical AND and logical OR operators (&& and ||, respectively) exhibit "short-circuit" operation. That is, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand.

Programmers should exercise caution if the second operand contains side effects because it may not be apparent whether the side effects actually occur.

Excerpt from CERT C

Figure 5: Contrasting approaches to the definition of coding rules

Precision of rule definitions

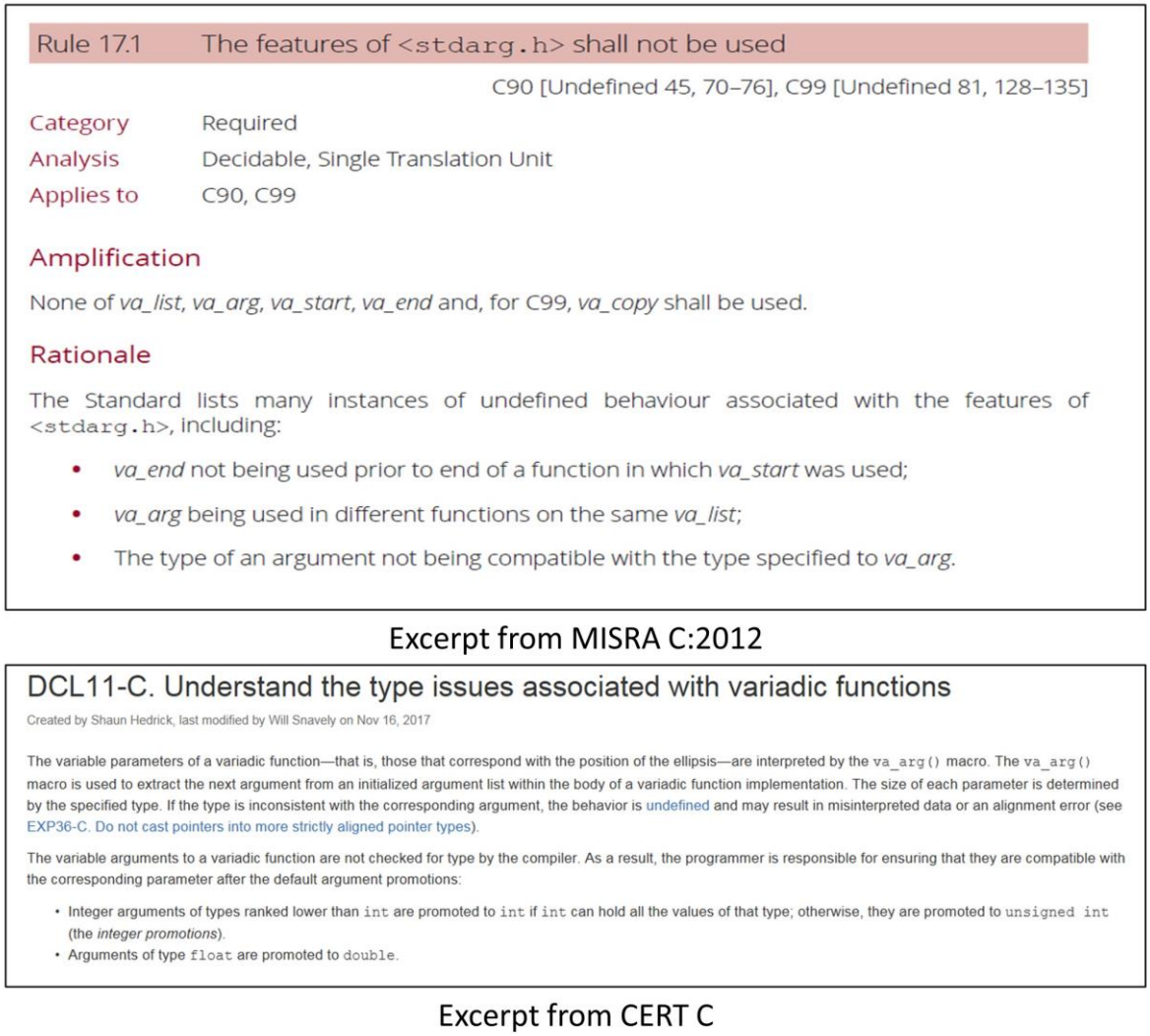
The stricter, more precisely defined approach of MISRA does not only lend itself to a standard more suitable for automated checking. It also addresses the issue of language misunderstanding more convincingly than CERT C.

Evidence suggests that there are particular characteristics of the C language which are responsible for most of the defects found in C source code[xii], such that around 80% of software defects are caused by the incorrect usage of about 20% of the available C or C++ language constructs. By restricting the use of the language to avoid the parts that are known to be problematic, it becomes possible to avoid writing associated defects into the code and as a result, the software quality greatly increases.

This approach also addresses a more subtle issue surrounding the personalities and capabilities of individual developers. Simple statistics tell us that of all the C developers in the world, 50% of them have below average capabilities – and yet it is very rare indeed to find a development team manager who would acknowledge that they recruit any such individuals. More than that, in any software development team, there will some who are more able than others and it is human nature for people not to highlight the fact if there are things they don't understand. More than that, it is common for less experienced programmers to be writing code, especially in large teams; typically the most experienced members will be involved in management and requirements definition with the new intake being used to code from the decomposed requirements.

Figure 6 uses the handling of variadic functions to illustrate how this approach differs from that of CERT C. CERT C calls for developers to "understand" the associated type issues, but doesn't suggest how a situation might be handled where a developer is, despite the best of intentions, harbouring a misunderstanding.

A counter argument might be that there will be developers who are very aware of the type issues associated with variadic functions, who make very good use of them, and who may feel affronted by the tighter restrictions on their use. However, for highly safety or security critical systems, MISRA would assert that because the “opportunity to make mistakes is either removed or reduced”, that is a price well worth paying.



Rule 17.1 The features of <stdarg.h> shall not be used

C90 [Undefined 45, 70–76], C99 [Undefined 81, 128–135]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

None of *va_list*, *va_arg*, *va_start*, *va_end* and, for C99, *va_copy* shall be used.

Rationale

The Standard lists many instances of undefined behaviour associated with the features of <stdarg.h>, including:

- *va_end* not being used prior to end of a function in which *va_start* was used;
- *va_arg* being used in different functions on the same *va_list*;
- The type of an argument not being compatible with the type specified to *va_arg*.

Excerpt from MISRA C:2012

DCL11-C. Understand the type issues associated with variadic functions

Created by Shaun Hedrick, last modified by Will Snavely on Nov 16, 2017

The variable parameters of a variadic function—that is, those that correspond with the position of the ellipsis—are interpreted by the `va_arg()` macro. The `va_arg()` macro is used to extract the next argument from an initialized argument list within the body of a variadic function implementation. The size of each parameter is determined by the specified type. If the type is inconsistent with the corresponding argument, the behavior is undefined and may result in misinterpreted data or an alignment error (see EXP36-C. Do not cast pointers into more strictly aligned pointer types).

The variable arguments to a variadic function are not checked for type by the compiler. As a result, the programmer is responsible for ensuring that they are compatible with the corresponding parameter after the default argument promotions:

- Integer arguments of types ranked lower than `int` are promoted to `int` if `int` can hold all the values of that type; otherwise, they are promoted to `unsigned int` (the *integer promotions*).
- Arguments of type `float` are promoted to `double`.

Excerpt from CERT C

Figure 6: Comparing differing precision of rule definition

Conclusions

Best practise for the development of either safety or security critical code is to apply a formalised software development process, starting with a set of requirements and tracing those requirements through to executable code. Even so, undefined, unspecified and implementation-defined behaviours within the C language can lead to safety or security failures in the resulting code base. And data handling errors such as invalid values, domain violations, tainted data, and leaking of confidential information can prevent both safety and security objectives from being realised.

MISRA C:2012 is not the only coding standard option for those with a need to develop secure code. For example, The correct application of either CERT C or MISRA C:2012 will certainly result in more secure code than if neither were to be applied. However, for safety or security critical applications, MISRA C is considerably less error prone both because it is specifically designed for such systems and as a result of its stricter, more decidable rules. Conversely, there is an argument for using the CERT C standard if the application is not critical but is to be connected to the internet for the first time. The retrospective application of CERT C might then be a pragmatic choice to make, though it would likely be accompanied by a list of issues where confirmation of intent is required .

Speaker

Company Details

LDRA
Portside
Monks Ferry
Wirral
CH41 5LH

Tel: 0151 649 9300
Fax: 0151 649 9666
E-mail: info@ldra.com

Contact Details

---

[i] MISRA – The Motor Industry Software Reliability Association https://www.misra.org.uk/Publications/tabid/57/Default.aspx
[ii] MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure", ISBN 978-906400-15-6 (PDF), April 2016.
[iii] ISO/IEC TS 17961:2013 Information technology – Programming languages, their environments and system software interfaces – C secure coding rules
[iv] MISRA C:2012 - Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013
[v] MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012, ISBN 978-906400-16-3 (PDF), April 2016.
[vi] ISO 26262-6:2011 Road vehicles – Functional safety – Part 6: Product development at the software level
[vii] IEC 61508-1:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements
[viii] IEC 62304 International Standard Medical device software – Software life cycle processes Consolidated Version Edition 1.1 2015-06
[ix] International standardization working group for the programming language C JTC1/SC22/WG14 http://www.open-std.org/jtc1/sc22/wg14/
[x] ISO/IEC TS 17961:2013 Information technology – Programming languages, their environments and system software interfaces – C secure coding rules
[xi] SEI CERT C Coding Standard https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
[xii] Applying the 80:20 Rule in Software Development. Jim Bird. Nov 15,2013. https://dzone.com/articles/applying-8020-rule-software