Addressing your insecurities with CERT C

Eliminating insecure coding practices and undefined behaviours that can lead to exploitable vulnerabilities and unreliable applications

www.ldra.com

Introduction

The CERT C Coding Standard consists of a set of guidelines designed to assist in the development of safe, reliable, and secure systems. It was developed by the Computer Emergency Response Team (CERT) division of the Software Engineering Institute (SEI), with reviews and contributions from many industry experts[1]. SEI is a federally funded pioneer of programming practices, based at Carnegie-Mellon University and supported by the US Department of Defense.

SEI's CERT division was formed in response to the Morris worm incident of 1988, and focuses on improving the security and resilience of computer systems and networks. CERT is well respected for its expertise and works closely with organisations from both private and public sectors including Cisco, Oracle, and the US Department of Homeland Security. The idea of a secure coding standard for the C language was born in 2003 of CERT's discussions with the ISO/IEC JTC1/SC22/WG 14 committee (WG14) responsible for the C language standard.

This paper discusses how LDRA tools can be used to check and help ensure compliance to CERT C. One of the key differentiators between software quality analysis tools is that some perform dynamic analysis, while others perform static analysis. Static analysis is performed without actually executing the software, whereas dynamic analysis involves the execution of all or some of that software. Tools used to check coding rules are concerned with the source code rather than its execution, and so use static analysis techniques.

CERT C

Although it has been twice published in book form, the CERT C coding standard continuously evolves to incorporate the comments and suggestions of C language experts and so the primary reference is CERT's secure programming wiki.

The primary aim of this evolution is to ensure that the CERT C coding standard promotes current best practices at all times with due regard to safety, reliability and security. These best practices are determined through a risk assessment which measures:

- The likelihood of a violation of the rule causing a defect
- The severity of the defect
- The cost of fixing the defect

Some standards include coding style guidelines to aid readability and maintainability, providing rules such as prescribed tab indentation and bracket alignment. These considerations are specifically outside the scope of CERT C which only touches on coding style where it can lead to defects. For instance, the lowercase letter 'l' is easy to mistake for the number 1 in many fonts, so the CERT C rule *DCL16-C. "Use 'L,' not 'l,' to indicate a long value*" is designed to deal with that.

The CERT C coding standard subdivides its guidelines into "rules" and "recommendations". For a guideline to be defined as a rule, it is required to meet three conditions.

- Violation of the guideline is likely to result in a defect that may adversely affect the safety, reliability, or security of a system, for example, by introducing a security flaw that may result in an exploitable vulnerability.
- The guideline does not rely on source code annotations or assumptions.
- Conformance to the guideline can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Guidelines are defined as recommendations when they fail to meet one or more of these conditions but are still likely to improve the safety, reliability, or security of software systems.

In addition to the distinction between rules and recommendations, CERT C is divided into 18 categories. These range in scope from guidelines relating to the preprocessor, through language characteristics such as integers and floating points, to API, POSIX and Windows considerations. Clearly, the CERT C guidelines are intended to be used selectively - rules concerning POSIX or Windows operating systems have no importance when dealing with a bare-metal application, for example.

[1] https://www.securecoding.cert.org/confluence/display/c/Acknowledgments

It is easy to spot the authors' sources of inspiration amongst guidelines. For example, although the C language standards[2] are extensive in defining how compiled C language should behave, they are by no means exhaustive. One of the challenges inherent in developing secure C code surrounds the issue of the remaining "undefined" or "unspecified" behaviour, and there is a group of guidelines in the CERT C standard to deal with it.

Another group of guidelines was derived from the work of the MITRE Corporation (MITRE) who focus on a number of related fields including cybersecurity[3]. CERT draws on MITRE's expertise by citing and cross referencing MITRE CVE and CWE entries in the CERT wiki.

Verifying Compliance with the CERT C Secure Coding Standard

LDRA's static analysis tools hold a superset of the rules and guidelines associated with CERT C and other standards, and combine syntactical analysis with data and control flow models. Such an approach permits two levels of abstraction, ensuring that checks are made with reference to not only the dangerous details of the C language, but also in the context of the program as a whole.

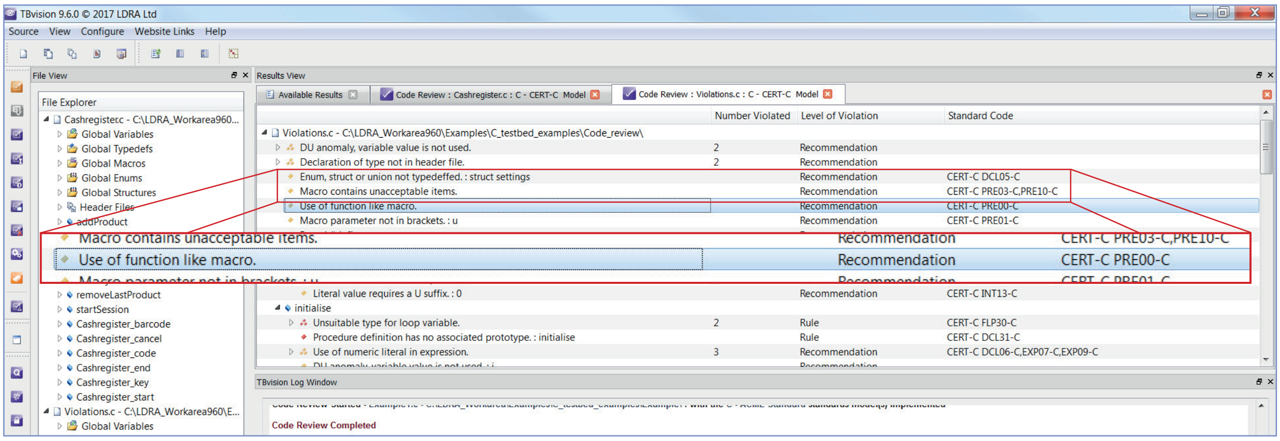
The tools implement a number of analysis phases to apply a catalogue of different checks, a subset of which is applied to confirm compliance with CERT C guidelines.

Main Static Analysis Phase

During the Main Static Analysis Phase, the code is parsed to enable the tool to check for problems with variable declaration, macro usage, function call usage, and local control flow and data flow.

Expression-level data

CERT C Rules concerned with the use of expression-level data are verified during this phase. Examples include *ARR37-C "Do not add or subtract an integer to a pointer to a non-array object"* (Figure 1) and EXP45-C "*Do not perform assignments in selection statements*".



	Number Violated	Level of Violation	Standard Code
Violations.c - C:\LDRA_Workarea960\Examples\C_testbed_examples\Code_review\			
DU anomaly, variable value is not used.	2	Recommendation	
Declaration of type not in header file.	2	Recommendation	
Enum, struct or union not typedeffed. : struct settings		Recommendation	CERT-C DCL05-C
Macro contains unacceptable items.		Recommendation	CERT-C PRE03-C,PRE10-C
Use of function like macro.		Recommendation	CERT-C PRE00-C
Macro parameter not in brackets. : u		Recommendation	CERT-C PRE01-C
Literal value requires a U suffix. : 0		Recommendation	CERT-C INT13-C
initialise			
Unsuitable type for loop variable.	2	Rule	CERT-C FLP30-C
Procedure definition has no associated prototype. : initialise		Rule	CERT-C DCL31-C
Use of numeric literal in expression.	3	Recommendation	CERT-C DCL06-C,EXP07-C,EXP09-C

Code Review Completed

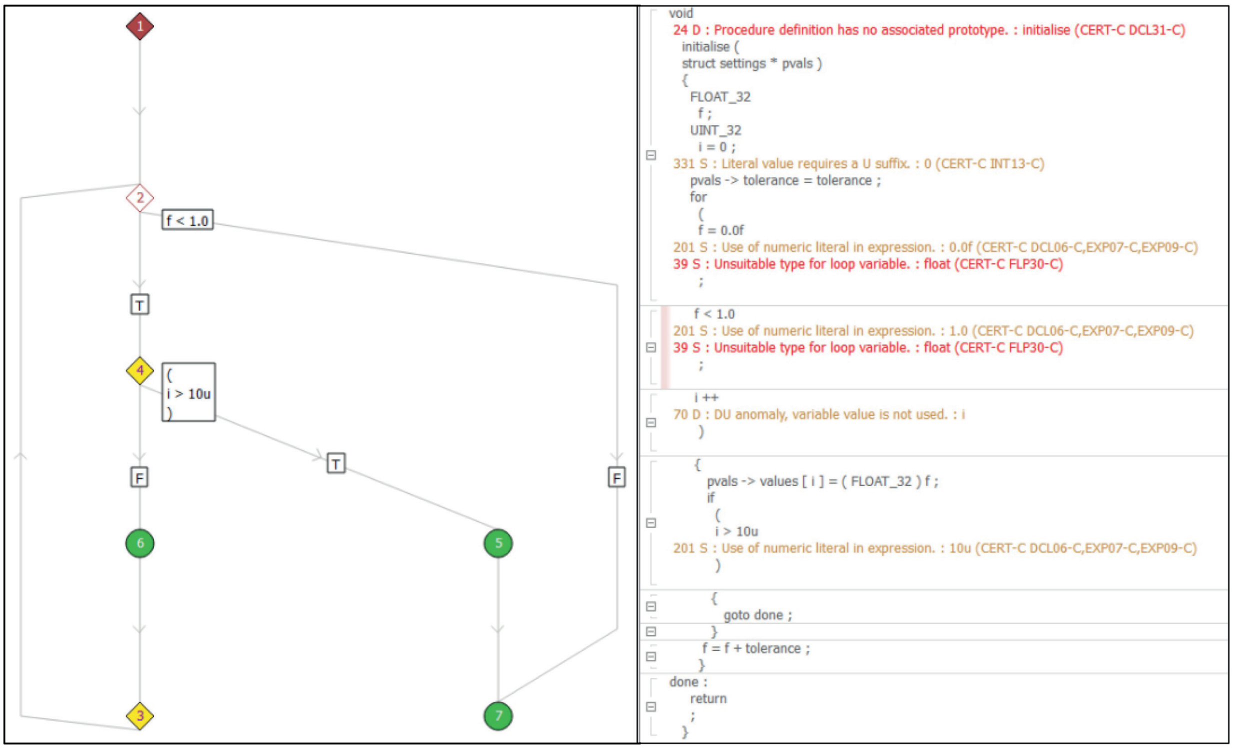
Figure 1: Reporting a violation of CERT C ARR37-C – "Do not add or subtract an integer to a pointer to a non-array object"

Control flow

In addition to reporting on these syntactical issues, the Main Static Analysis phase also establishes an understanding of the control flow paths within the code. This provides the information for the tool not only to present those flow paths in graphical form, but also to generate a reformatted version of the code such that there is only one instruction per line.

[2] http://www.open-std.org/jtc1/sc22/wg14/www/standards

[3] https://www.mitre.org/capabilities/cybersecurity/overview?category=all



```
void
 24 D : Procedure definition has no associated prototype. : initialise (CERT-C DCL31-C)
  initialise (
  struct settings * pvals )
  {
   FLOAT_32
    f ;
   UINT_32
    i = 0 ;
 331 S : Literal value requires a U suffix. : 0 (CERT-C INT13-C)
   pvals -> tolerance = tolerance ;
   for
    (
    f = 0.0f
 201 S : Use of numeric literal in expression. : 0.0f (CERT-C DCL06-C,EXP07-C,EXP09-C)
 39 S : Unsuitable type for loop variable. : float (CERT-C FLP30-C)
    ;
    f < 1.0
 201 S : Use of numeric literal in expression. : 1.0 (CERT-C DCL06-C,EXP07-C,EXP09-C)
 39 S : Unsuitable type for loop variable. : float (CERT-C FLP30-C)
    ;
    i ++
 70 D : DU anomaly, variable value is not used. : i
    )
    {
     pvals -> values [ i ] = ( FLOAT_32 ) f ;
     if
      (
      i > 10u
 201 S : Use of numeric literal in expression. : 10u (CERT-C DCL06-C,EXP07-C,EXP09-C)
      )
      {
       goto done ;
      }
     f = f + tolerance ;
    }
done :
   return
   ;
  }
```

Figure 2: Presenting the control flow in graphical form, alongside reformatted source code

It also provides the underlying information for additional CERT C guideline checks. One such example is the recommendation *MEM05-C "Avoid large stack allocations*." MEM05-C includes the suggestion that recursion should be avoided because neither the control flow nor the resource use associated with such a mechanism is easily understood.

Data Flow Analysis Phase

This second static analysis phase collates information relating to the data flow, complementing the reformatted source code and the control flow to complete a 3 dimensional model. The combination of these elements provides the basis of a mechanism to check variable use in complex control flow scenarios, and in the context of CERT C, to check for the violation of rules such as EXP30-C "*Do not depend on order of evaluation between sequence points*".

It is interesting to reflect on how that is approached automatically, and hence to consider the effort that would be required to verify compliance through manual code review.

A sequence point is a point in program execution at which all side effects are evaluated before going on to the next step. They are often mentioned in reference to C and C++, because they are a core concept for determining the validity and, if valid, the possible results of expressions.

As a practical example, consider two functions *f()* and *g()* in C or C++.

Scenario 1: Expression *f()+g()*

The plus operator is not associated with a sequence point, and therefore in the expression it is possible that either f() or g() will be executed first.

Scenario 2: Expression *f(),g()*

The comma operator is associated with a sequence point, and therefore in the code f(),g() the order of evaluation is defined: first f() is called, and then g() is called[4].

[4] http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html

The point of the rule is to ensure that the outcome of an expression such as that in Scenario 1 does not depend on what happens before it is executed. In order to be certain of that, it is necessary to consider the different potential orders of evaluation between sequence points.

To show that manually would be time consuming, demand a very high level of expertise and concentration, and even then would be subject to human error. Automating the process is a far more pragmatic approach, because LDRA static analysis is able to recognise these situations by analysing not only what operators and variables are being used, but also the context of the expression structure and how variables in that expression might be affected by prior use.

Cross Reference Analysis Phase

Once the data flow picture is complete, it is possible to build on that information by cross- referencing the symbols used throughout the system. This analysis exposes information about where, how and how often variables are used, and also provides a foundation for the assessment of potential mismatches. For example, differences between how a symbol is defined and declared can lead to undefined behaviour, and is another example of a check relevant to CERT C recommendation *MSC15-C "Do not rely upon undefined behaviour*".

This cross referencing also permits the extension of more perfunctory checks completed in earlier phases, such as extending the analysis of potential array usage violations to instances spanning more than one procedure, pertinent to the CERT C Rule *ARR30-C "Do not form or use out-of-bounds pointers or array subscripts*". This multi-level approach is common throughout the LDRA static analysis, and is significant because it is the more subtle violations spanning several functions which are likely to be the most difficult to detect – and, unfortunately, often the most likely to be present.

Comparing Static Analysis Tools

There is a plethora of different static analysis tools on the market, and it is easy to draw up a list of requirements. In this context, it is useful to consider a summary of the information associated with Automated Detection tools available on the CERT website[5].

The analysis shows a total of 306 CERT C guidelines. Figure 3 and Figure 4 represent data collated on the basis that where there is an indication that a particular tool is able to detect a violation to any extent whatsoever, the related guideline can be considered to be implemented.

Implementation Tables

Tool	Known	Unknown	Total
Co. A	7	0	7
Co. B	96	0	96
Co. C	79	43	122
Co. D	50	0	50
Co. E	11	0	11
Co. F	63	0	63
Co. G	0	3	3
Co. H	36	12	48
Co. I	20	1	21
Co. J	135	0	135
LDRA tool suite	200	1	201
Co. L	10	0	10
Co. M	2	24	26

Figure 3: Summary showing the number of guidelines addressed by each tool, out of a total of 306 in the standard.

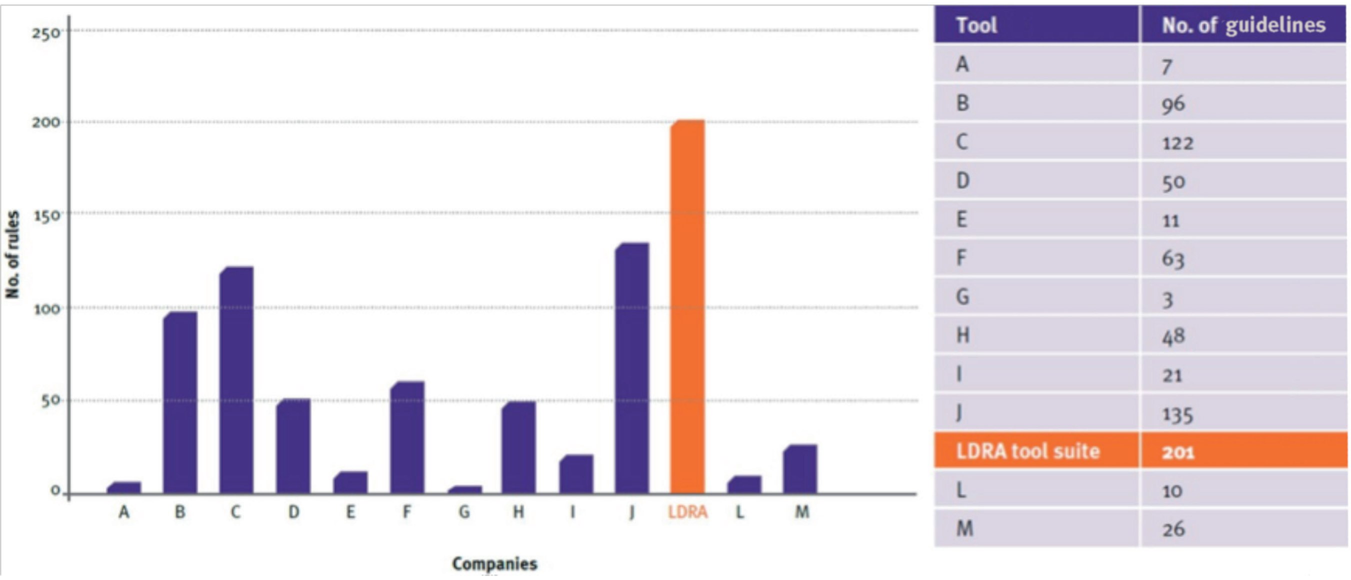
[5] http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html

In Figure 3, the “Known” and “Unknown” categories relate to the CERT website descriptions of both the CERT C guidelines and the static analysis tools.

- The “Known” column shows the number of guidelines for which there is either a confirmation of coverage, a description of coverage, or both.
- The “Unknown” column shows the number of guidelines for which there is neither confirmation of coverage, nor a description of it.

It is estimated that at least 42 of the CERT C guidelines are not checkable by static analysis, perhaps because they are not precisely defined, or because knowledge of user intent may be required for implementation. One example of the latter case is the CERT C recommendation *ARR00-C “Understand how arrays work”*. No tool can guarantee that a programmer fully understands every aspect of array functionality applicable to his program, but a static analysis tool can identify particular coding practices that suggest otherwise.

Figure 4 shows a direct comparison between the analysis products listed on the CERT website.



Tool	No. of guidelines
A	7
B	96
C	122
D	50
E	11
F	63
G	3
H	48
I	21
J	135
LDRA tool suite	201
L	10
M	26

Figure 4: Tool vendor comparison for CERT C compliance

Conclusion

It is possible to implement the Carnegie Mellon Software Engineering Institute (SEI) CERT C Secure Coding Standard manually, and to verify that implementation by means of code review. However, such a task is onerous and prone to error, and for most of the CERT C guidelines there are established static analysis tools available which promise a far more efficient and effective approach.

In comparing these tools, it is important to remember that although the number of CERT C guidelines implemented may provide an obvious comparison statistic, generally it is the easiest ones to include which are common to all of the offerings. Unhappily, the violation checks which are difficult to implement for tool vendors tend to be those which are also tricky to analyse through inspection!

It is also useful to compare the completeness of detection for each of the rules. Many of the guidelines encompass several different possible violations. Are they all considered? And how extensively is each one dealt with?

Finally, consider that the tools will not be used in isolation but rather as part of a complete development toolchain. Seamless integration into that toolchain will yield rewards in the form of efficient and effective development.

www.ldra.com

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Inc.
2540 King Arthur Blvd, 3rd Floor, 12th Main Lewisville Texas 75056
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit B-3, Third floor Tower B, Golden Enclave
HAL Airport Road Bengaluru 560017
Tel: +91 80 4080 8707
e-mail: india@ldra.com

Addressing your insecurities with CERT C v2.0 06/17