How this Coding Standard is Organized

- Identifiers
- Noncompliant Code Examples and Compliant Solutions
- Coding Conventions
- Exceptions
- Risk Assessment
- Automated Detection
- Related Vulnerabilities
- Related Guidelines
- CERT-CWE Mapping Notes
- Bibliography

This coding standard is organized into 15 chapters containing rules in specific topic areas followed by four appendices. Appendix A contains the bibliography. Appendix B lists the definitions of terms used throughout the standard. Appendix C lists the undefined behaviors from the C Standard, Annex J, J.2 [ISO/IEC 9899:2011], numbered and classified for easy reference. These numbered undefined behaviors are referenced frequently from the rules. Appendix D lists unspecified behaviors from the C Standard, Annex J, J.2 [ISO/IEC 9899:2011]. These unspecified behaviors are occasionally referenced from the rules as well.

Most rules have a consistent structure. Each rule in this standard has a unique identifier, which is included in the title. The title and the introductory paragraphs define the rule and are typically followed by one or more pairs of noncompliant code examples and compliant solutions. Each rule also includes a risk assessment, related guidelines, and a bibliography (where applicable). Rules may also include a table of related vulnerabilities. The recommendations in this wiki are organized in a similar fashion.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

- A three-letter mnemonic representing the section of the standard
- A two-digit numeric value in the range of 00 to 99
- A suffix that represents the associated language or platform.
  - "-C" for the SEI CERT C Coding Standard
  - "-CPP" for the SEI CERT C++ Coding Standard
  - "-J" for the SEI CERT Oracle Coding Standard for Java
  - "-PL" for the SEI CERT Perl Coding Standard

The three-letter mnemonic can be used to group similar coding practices and to indicate which category a coding practice belongs to.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00 to 29 are reserved for recommendations, and values in the range of 30 to 99 are reserved for rules. (The values used for the are different.) Rules and recommendations are frequently referenced from the guidelines in this standard by their identifier and title.

Here are some example identifiers with an explanation of each:

- INT50-CPP Do not cast to an out-of-range enumeration value
  - This identifier indicates a rule
  - "INT" stands for the Integer category
  - "50" is the unique identifier
  - "-CPP" stands for the C++ language
- EXP00-J Do not ignore values returned by methods
  - This identifier indicates a rule
  - "EXP" stands for the Expressions category
  - "00" is the unique identifier
  - "-J" stands for the Java language
- FLP00-C. Understand the limitations of floating-point numbers
  - This identifier indicates a recommendation
  - "FLP" stands for the Floating Point category
  - "00" is the unique identifier
  - "-C" stands for the C programming language

Noncompliant Code Examples and Compliant Solutions

Noncompliant code examples illustrate code that violates the guideline under discussion. It is important to note that these are only examples, and eliminating all occurrences of the example does not necessarily mean that the code being analyzed is now compliant with the guideline.

Noncompliant code examples are typically followed by compliant solutions, which show how the noncompliant code example can be recoded in a secure, compliant manner. Except where noted, noncompliant code examples should contain violations only of the guideline under discussion. Compliant solutions should comply with all of the secure coding rules but may on occasion fail to comply with a recommendation.

Coding Conventions

Unless otherwise specified, all code should compile on a reasonably modern compiler, when it is following compliance with the standard. For example, you can require GCC to conform to the C11 standard with the parameter --std=c11.
Code that is only expected to run on a particular subset of platforms should have those platforms mentioned in the code’s section header, e.g.: Compliant Solution (POSIX). Likewise, code that is only expected to run on more modern versions of C should indicate the oldest standard that supports them, e.g.: Compliant Solution (C99).

In order to compile the code, you will need to include appropriate header files. For example, if the code invokes malloc(), you may need to include the stdlib.h header.

Many coding examples must lead with some variables initialized to valid values. If any code example (compliant or non-compliant) requires a local variable to be initialized with an unspecified but valid value, then that code should be wrapped inside a function that takes the variable as a function argument. Consequently, unless otherwise noted, function arguments should be assumed to point to valid values. For example, character pointers should not be NULL, but rather point to valid null-terminated byte strings. Checking that function arguments are valid is important, and is described in detail in recommendation API00-C. Functions should validate their parameters. However, the code involved with checking argument validity would be redundant across all code examples, and so function arguments in these code examples are not validated.

Many code examples will contain ellipsis in comments. This indicates that the comment may be replaced by arbitrary code that satisfies the comment. A comment with only ellipsis suggests that the code may do anything.

Proper error handling is a controversial subject, and many applications and libraries provide their own idiosyncratic error handling mechanisms. See Rule 12. Error Handling (ERR) and Rec. 12. Error Handling (ERR) for our guidelines on handling errors. When our code detects that an error condition might have occurred, and handling that error condition is not endemic to the guideline itself, we will use the comment: /* Handle Error */. This comment implies that the error is somehow addressed, so that the code does not fall through. The code may abort, or fix the error somehow. For example:

```c
char *str = malloc(10);
if (str == NULL) {
    /* Handle Error */
}
/* ... str can not be NULL here. Work with str... */
```

Exceptions

Any rule or recommendation may specify a small set of exceptions detailing the circumstances under which the guideline is not necessary to ensure the safety, reliability, or security of software. Exceptions are informative only and are not required to be followed.

Risk Assessment

Each guideline in the CERT C Coding Standard contains a risk assessment section that attempts to provide software developers with an indication of the potential consequences of not addressing a particular rule or recommendation in their code (along with some indication of expected remediation costs). This information may be used to prioritize the repair of rule violations by a development team. The metric is designed primarily for remediation projects. It is generally assumed that new code will be developed to be compliant with the entire coding standard and applicable recommendations.

Each rule and recommendation has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 60812]. Three values are assigned for each rule on a scale of 1 to 3 for severity, likelihood, and remediation cost.

Severity—How serious are the consequences of the rule being ignored?

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Examples of Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low</td>
<td>Denial-of-service attack, abnormal termination</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
<td>Data integrity violation, unintentional information disclosure</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
<td>Run arbitrary code</td>
</tr>
</tbody>
</table>

Likelihood—How likely is it that a flaw introduced by violating the rule can lead to an exploitable vulnerability?

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unlikely</td>
</tr>
<tr>
<td>2</td>
<td>Probable</td>
</tr>
<tr>
<td>3</td>
<td>Likely</td>
</tr>
</tbody>
</table>

Remediation Cost—How expensive is it to comply with the rule?

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Detection</th>
<th>Correction</th>
</tr>
</thead>
</table>

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. The products range from 1 to 27, although only the following 10 distinct values are possible: 1, 2, 3, 4, 6, 8, 9, 12, 18, and 27. Rules and recommendations with a priority in the range of 1 to 4 are Level 3 rules, 6 to 9 are Level 2, and 12 to 27 are Level 1. The following are possible interpretations of the priorities and levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Priorities</th>
<th>Possible Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>12, 18, 27</td>
<td>High severity, likely, inexpensive to repair</td>
</tr>
<tr>
<td>L2</td>
<td>6, 8, 9</td>
<td>Medium severity, probable, medium cost to repair</td>
</tr>
<tr>
<td>L3</td>
<td>1, 2, 3, 4</td>
<td>Low severity, unlikely, expensive to repair</td>
</tr>
</tbody>
</table>

Specific projects may begin remediation by implementing all rules at a particular level before proceeding to the lower priority rules, as shown in the following illustration:

Recommendations are not compulsory and are provided for information purposes only.

Automated Detection

Both rules and recommendations frequently have sections that describe automated detection. These sections provide additional information on analyzers that can automatically diagnose violations of coding guidelines. Most automated analyses for the C programming language are neither sound nor complete, so the inclusion of a tool in this section typically means that the tool can diagnose some violations of this particular rule. The Secure Coding Validation Suite can be used to test the ability of analyzers to diagnose violations of rules from ISO/IEC TS 17961:2013, which is related to the rules in this standard.

The information in the automated detection sections on this wiki may be

- provided by the vendors
- determined by CERT by informally evaluating the analyzer
- determined by CERT by reviewing the vendor documentation

Where possible, we try to reference the exact version of the tool for which the results were obtained. Because these tools evolve continuously, this information can rapidly become dated and obsolete.
Related Vulnerabilities

The risk assessment sections on the wiki also contain a link to search for related vulnerabilities on the CERT website. Whenever possible, CERT Vulnerability Notes are tagged with a keyword corresponding to the unique ID of the coding guideline. This search provides you with an up-to-date list of real-world vulnerabilities that have been determined to be at least partially caused by a violation of this specific guideline. These vulnerabilities are labeled as such only when the vulnerability analysis team at the CERT/CC is able to evaluate the source code and precisely determine the cause of the vulnerability. Because many vulnerability notes refer to vulnerabilities in closed-source software systems, it is not always possible to provide this additional analysis. Consequently, the related vulnerabilities field tends to be somewhat sparsely populated.

Related vulnerability sections are included only for specific rules in this standard, when the information is both relevant and interesting.

Related Guidelines

For each entry in a Related Guidelines table, CERT has determined that there is some code flaw for which there is both a violation of some condition of the CERT guideline and a condition of the external-to-CERT guideline, where that condition is violated or that condition is described as a flaw.

Related Guidelines Table headings definitions

- **Taxonomy:** A named set of coding rules, weaknesses, standards, or guidelines such as Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules [ISO/IEC TS 17961:2013]; Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use [ISO/IEC TR 24772:2013]; MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems [MISRA C:2012]; and CWE IDs in MITRE’s Common Weakness Enumeration (CWE) [MITRE 2010].

- **Taxonomy item:** A single named (and/or numbered) item in a taxonomy

- **Relationship:** For each entry in a Related Guidelines table, CERT has determined that there is some code flaw for which there is both a violation of some condition of the CERT guideline and a condition of the external-to-CERT guideline, where that condition is violated or that condition is described as a flaw.

These relationships may be defined in a precise or imprecise way. For Common Weakness Enumeration (CWE), CERT has made precise mappings between CERT C rules and CWEs, as described below. For other taxonomies of coding flaws or secure coding (such as MISRA or ISO/IEC TR 24772:2013), so far, CERT has made only imprecise (“Unspecified Relationship”) mappings. An “Unspecified Relationship” label indicates there is some overlapping code flaw condition, but the extent is unspecified.

If the mapping was made using an automated process developed by CERT, and not yet verified manually, the mapping is marked at the end with “(A)”. Precise relationships explain more about the extent to which conditions of the CERT guideline and external guideline match.

In the simplest case, the guidelines are exactly equal (the relationship is labelled “Exact”). CERT’s “partial mapping” terms (“Partial overlap”, “Guideline subset of <EXTERNAL_GUIDELINE>”, “<EXTERNAL_GUIDELINE > subset of rule”) describe relationships between the guideline items using the language of sets, where the guideline item (a CERT guideline or an <EXTERNAL_GUIDELINE> entry) is a set that holds one or more conditions. By subset we mean a proper subset, that “A  B” means every element (meaning, every condition) in A is also in B, but there exists at least one element in B that is not in A. If a condition of a program violates a CERT rule “R” and also exhibits an <EXTERNAL_GUIDELINE> “E”, that condition is in the overlap between “R” and “E”.

For each CWE that has a partial mapping to a CERT rule, we have documented the nature of what the rule and CWE have in common, what is exclusive to the rule, and what is exclusive to the CWE, in a section titled “CERT-CWE Mapping Notes”.

The 10 main precise relationship labels CERT uses are mostly the same as the 10 CWE Mapping Fit relationship labels, with 3 different labels.

<table>
<thead>
<tr>
<th>Different but related terms:</th>
<th>CERT term</th>
<th>MITRE term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule subset of CWE</td>
<td>CWE_More_Abstract</td>
<td></td>
</tr>
<tr>
<td>CWE subset of rule</td>
<td>CWE_More_Specific</td>
<td></td>
</tr>
<tr>
<td>Partial overlap</td>
<td>Imprecise</td>
<td></td>
</tr>
</tbody>
</table>

An 11th label “None” is specified in cases where previous mappings existed but it has been determined that there is no overlap of conditions.

Table column formats:

- **Taxonomy:** Taxonomy name (e.g., “CWE”) followed by version name that was mapped, if that is known (e.g., “CWE 2.11”, “CERT 2016”, or “MISRA”).
- **Taxonomy item:** A single named (and/or numbered) item in a taxonomy, sometimes with the full title text of the item and sometimes with a hyperlink to the item.
- **Relationship:** A combined entry with fields for date mapped, organization that did the mapping, and relationship (all in same cell for one mapping, separated by a colon, with one entry per line): `<(Optional “Prior to ”)YYYY-MM-DD: ORGANIZATION: RELATIONSHIP(Optional “(A)” )>`. Where specified by mapping day, precise mappings done by CERT use the latest published edition of a non-CERT taxonomy along with the latest published edition of the CERT standard plus changes on the CERT wiki. Precise mappings done by an external organization use the latest published edition of the CERT standard and their own latest published edition. Examples below:
The related guidelines sections contain links to guidelines in related standards, technical specifications, and guideline collections such as Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules [ISO/IEC TS 17961:2013]; Information Technology—Programming Languages —Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use [ISO/IEC TR 24772:2013]; MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems [MISRA C:2012]; and CWE IDs in MITRE’s Common Weakness Enumeration (CWE) [MITRE 2010].

You can create a unique URL to get more information on CWEs by appending the relevant ID to the end of a fixed string. For example, to find more information about CWE-192, Integer Coercion Error,” you can append 192.html to http://cwe.mitre.org/data/definitions/ and enter the resulting URL in your browser: http://cwe.mitre.org/data/definitions/192.html.

The other referenced technical specifications, technical reports, and guidelines are commercially available.

CERT-CWE Mapping Notes

CERT’s “partial mapping” terms {Partial overlap, Rule subset of CWE, CWE subset of rule} describe relationships between the taxonomy items using the language of sets, where the taxonomy item (a CERT rule or a CWE weakness) is a set that holds one or more conditions. If a condition of a program violates a CERT rule “R” and also exhibits a CWE weakness “W”, that condition is in the overlap between the rule and weakness.

For each CWE that has a partial mapping to a CERT rule, in this section we document the nature of what the rule and CWE have in common, what is exclusive to the rule, and what is exclusive to the CWE. Sometimes what is exclusive or shared is simply described by set equations using taxonomy items, but other times documentation of what is shared may include function names or data types, or some prose description of shared characteristics.

Notation: “Intersection(A, B) =” the right side of the equals sign defines if there is an overlap between A and B, meaning if a condition of a program exists with an overlapping area between A and B.

Notation: “A B” means every element in A is also in B, but there exists at least one element in B that is not in A. (It means A is a “proper subset” of B.)

Notation: “A B” means A is not a proper subset of B.

Notation: “Intersection(A, B) =” means no element in A is also in B. (It means the intersection is the empty set.)

Notation: “A - B =” the right side of the equals sign defines what element(s), if any, exist in A that are not also in B.

No CERT C rule or recommendation is identical to any other CERT C rule or recommendation.

In this section, Independent() means all the rules listed within are independent: that is, every pair of two rules listed they have an empty intersection. Most CERT rules are designed to be independent, that is, they have no overlap. (This applies only to rules, not recommendations).

For CWE that have been identified as having at least a partial overlap with another CERT rule R1, for current mapping to CERT rule R2: In the Mapping Notes section we consider C’s possible overlap or exclusion of the CWE overlap area with R1. We also consider the relationship of R1 and R2. If any. (By defining the relationship between the CERT rules that separately have at least some overlap with the CWE of interest, the mapping notes further define the conditions of overlap and/or non-overlap between the primary CWE-to-CERT-rule mapping of interest.)

Regarding partial overlap, we try to find segments of code as examples that are inseparable and exhibit both code flaws. An example of separable code:

```c
INT_MAX + 1; FLT_MAX + 1.0;
static char x[3];
char* foo() {
    int x_int = (int) x; // x_int = 999 eg
    return x_int + 5; // returns 1004, violates CWE 466
}
...
int y_int = foo(); // violates CWE-466
char* y = (char*) y_int; // well-defined but y may be invalid, violates INT36-C
char c = *y; // indeterminate value, out-of-bounds read, violates CWE-119
```
Bibliography

Most guidelines have a small bibliography section that lists documents and sections in those documents that provide information relevant to the guideline.