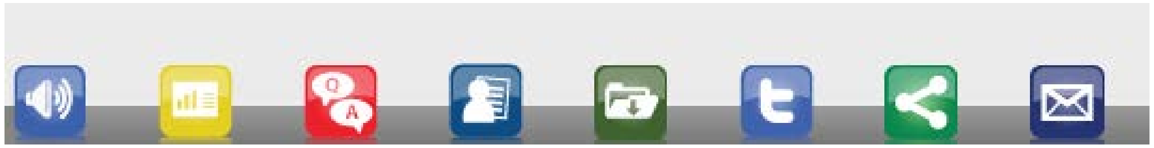
CERT

Source Code Analysis Laboratory (SCALe)

Robert C. Seacord

How to interact with Us



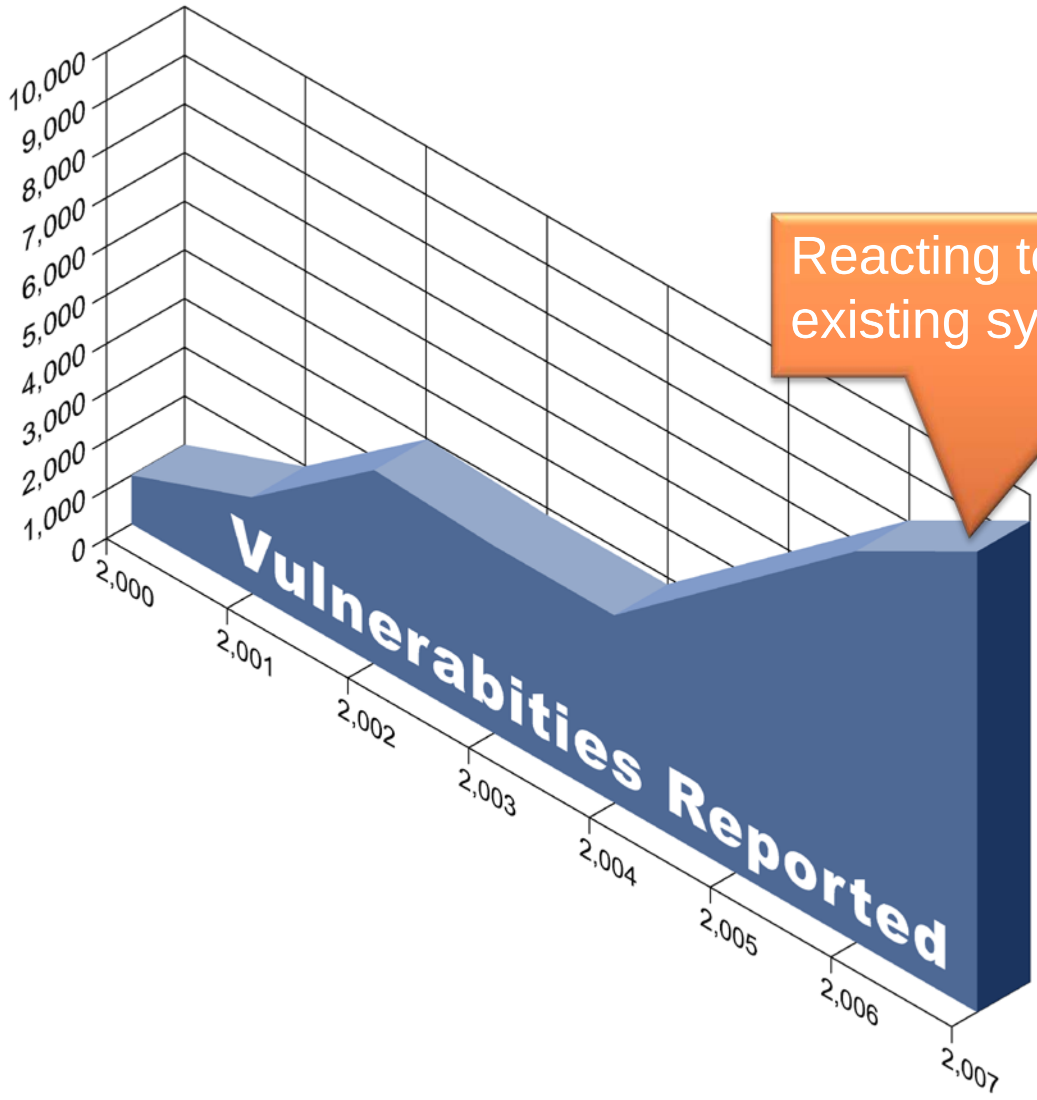
Today’s Speaker

Robert C. Seacord is a computer security specialist and writer. He is the author of books on computer security, legacy system modernization, and component-based software engineering.

Robert manages the Secure Coding Initiative at CERT, located in Carnegie Mellon's Software Engineering Institute in Pittsburgh, Pennsylvania. CERT, among other security-related activities, regularly analyzes software vulnerability reports and assesses the risk to the Internet and other critical infrastructure. Robert is an adjunct professor in the Carnegie Mellon University School of Computer Science and in the Information Networking Institute.

Robert started programming professionally for IBM in 1982, working in communications and operating system software, processor development, and software engineering. Robert also has worked at the X Consortium, where he developed and maintained code for the Common Desktop Environment and the X Window System. Robert has a bachelor's degree in computer science from Rensselaer Polytechnic Institute.

Increasing Vulnerabilities



Reacting to vulnerabilities in existing systems is not working

Application Security

Problem Description

Increasingly, compiler writers are taking advantage of undefined behaviors in the C and C++ programming languages to improve optimizations.

Frequently, these optimizations are interfering with the ability of developers to perform cause-effect analysis on their source code, that is, analyzing the dependence of downstream results on prior results.

Consequently, these optimizations are eliminating causality in software and are increasing the probability of software faults, defects, and vulnerabilities.

Undefined Behaviors

Behaviors are classified as “undefined” by the standards committees to:

- give the implementer license not to catch certain program errors that are difficult to diagnose;
- avoid defining obscure corner cases which would favor one implementation strategy over another;
- identify areas of possible conforming language extension: the implementer may augment the language by providing a definition of the officially undefined behavior.

Implementations may

- ignore undefined behavior completely with unpredictable results
- behave in a documented manner characteristic of the environment (with or without issuing a diagnostic)
- terminate a translation or execution (with issuing a diagnostic).

Compiler Optimizations

The basic design of an optimizer for a C compiler is largely the same as an optimizer for any other procedural programming language.

The fundamental principle of optimization is to replace a computation with a more efficient method that computes the same result.

However, some optimizations change behavior

- Eliminate undefined behaviors (good)
- Introduce vulnerabilities (bad)

“As If” Rule 1

The ANSI C standard specifies the *results* of computations as if on an *abstract machine*, but the *methods* used by the compiler are not specified.

In the abstract machine, all expressions are evaluated as specified by the semantics.

An actual implementation need not evaluate part of an expression if it can deduce that

- its value is not used
- that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

The compiler’s optimizer is free to choose any method that produces the correct result.

“As If” Rule 2

This clause gives compilers the leeway to remove code deemed unused or unneeded when building a program.

This is commonly called the “as if” rule, because the program must run *as if* it were executing on the abstract machine.

While this is usually beneficial, sometimes the compiler removes code that it thinks is not needed, even if the code has been added with security in mind.

Implementation Strategies

Hardware behavior

- Generate the corresponding assembler code, and let the hardware do whatever the hardware does.
- For many years, this was the nearly-universal policy, so several generations of C and C++ programmers have assumed that all compilers behave this way.

Super debug

- Provide an intensive debugging environment to trap (nearly) every undefined behavior.
- This policy severely degrades the application’s performance, so is seldom used for building applications.

Total license

- Treat any possible undefined behavior as a “can’t happen” condition.
- This permits aggressive optimizations.

Adding a Pointer and an Integer

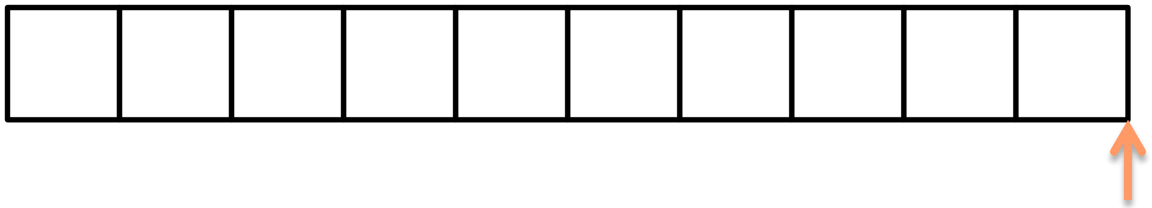
The C Standard states:

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand.

An expression like **`P[N]`** is translated into **`*(P+N)`**.

Adding a Pointer and an Integer

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.



If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.

Bounds Checking 1

A programmer might code a bounds-check such as

```
char *ptr; // ptr to start of array
char *max; // ptr to end of array
size_t len;
if (ptr + len > max)
  return EINVAL;
```

No matter what model is used, there is a bug.

If `len` is very large, it can cause `ptr + len` to overflow, which creates undefined behavior.

Under the hardware behavior model, the result would typically wrap-around—pointing to an address that is actually *lower* in memory than `ptr`.

Bounds Checking 2

In attempting to fix the bug, the experienced programmer (who has internalized the hardware behavior model of undefined behavior) might write a check like this:

```
  if (ptr + len < ptr || ptr + len > max)
    return EINVAL;
```

However, compilers that follow the total license model may optimize out the first part of the check leaving the whole bounds check defeated

This is allowed because

- if `ptr` plus (an unsigned) `len` compares less than `ptr`, then an undefined behavior occurred during calculation of `ptr + len`
- the compiler can assume that undefined behavior never happens
- consequently `ptr + len < ptr` is dead code and can be removed

Algebraic Simplification

Optimizations may be performed for comparisons between `P + V1` and `P + V2`, where `P` is the same pointer and `V1` and `V2` are variables of some integer type.

The total license model permits this to be reduced to a comparison between `V1` and `V2`.

However, if `V1` or `V2` are such that the sum with `P` overflows, then the comparison of `V1` and `V2` will not yield the same result as actually computing `P + V1` and `P + V2` and comparing the sums.

Because of possible overflows, computer arithmetic does not always obey the algebraic identities of mathematics.

Algebraic Simplification Applied

In our example:

```
if (ptr + len < ptr || ptr + len > max)
  return EINVAL;
```

this optimization translates as follows:

```
ptr + len < ptr
ptr + len < ptr + 0
len < 0 (impossible, len is unsigned)
```

Mitigation

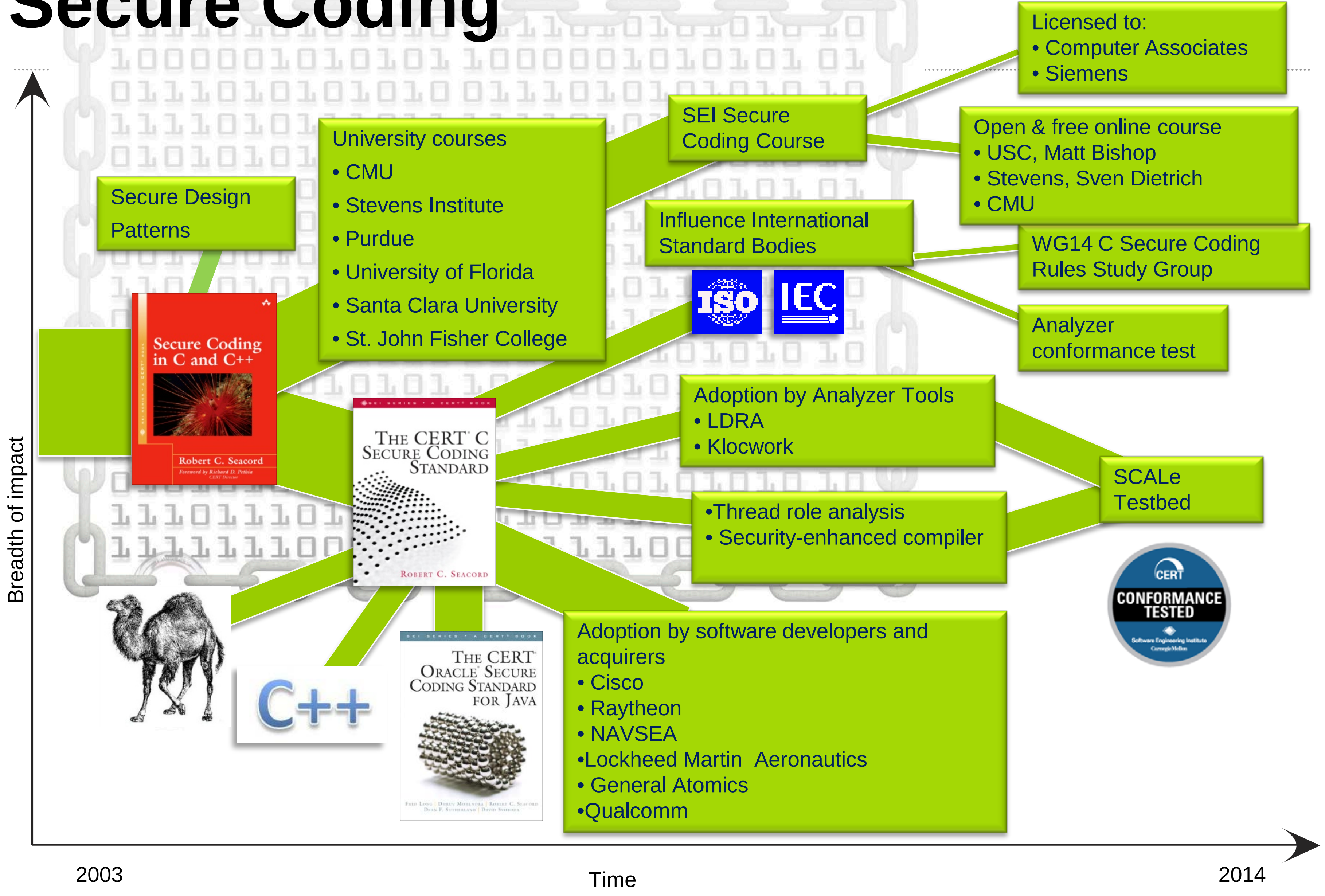
This problem is easy to remediate, once it is called to the attention of the programmer, such as by a diagnostic message when dead code is eliminated.

For example, if it is known that `ptr` is less-or-equal-to `max`, then the programmer could write:

```
if (len > max – ptr)
    return EINVAL;
```

This conditional expression eliminates the possibility of undefined behavior.

Secure Coding


Breadth of impact
Time
2003
2014
Secure Coding in C and C++
Robert C. Seacord
The CERT C Secure Coding Standard
Robert C. Seacord
The CERT Oracle Secure Coding Standard for Java
C++
ISO
IEC
CERT CONFORMANCE TESTED

Secure Design Patterns

University courses
- CMU
- Stevens Institute
- Purdue
- University of Florida
- Santa Clara University
- St. John Fisher College

SEI Secure Coding Course

Licensed to:
- Computer Associates
- Siemens

Open & free online course
- USC, Matt Bishop
- Stevens, Sven Dietrich
- CMU

Influence International Standard Bodies

WG14 C Secure Coding Rules Study Group

Analyzer conformance test

Adoption by Analyzer Tools
- LDRA
- Klocwork

SCALe Testbed

- Thread role analysis
- Security-enhanced compiler

Adoption by software developers and acquirers
- Cisco
- Raytheon
- NAVSEA
- Lockheed Martin Aeronautics
- General Atomics
- Qualcomm

CERT Secure Coding Standards

CERT C Secure Coding Standard

- Version 1.0 (C99) - published
- Version 2.0 (C11) - under development

CERT C++ Secure Coding Standard

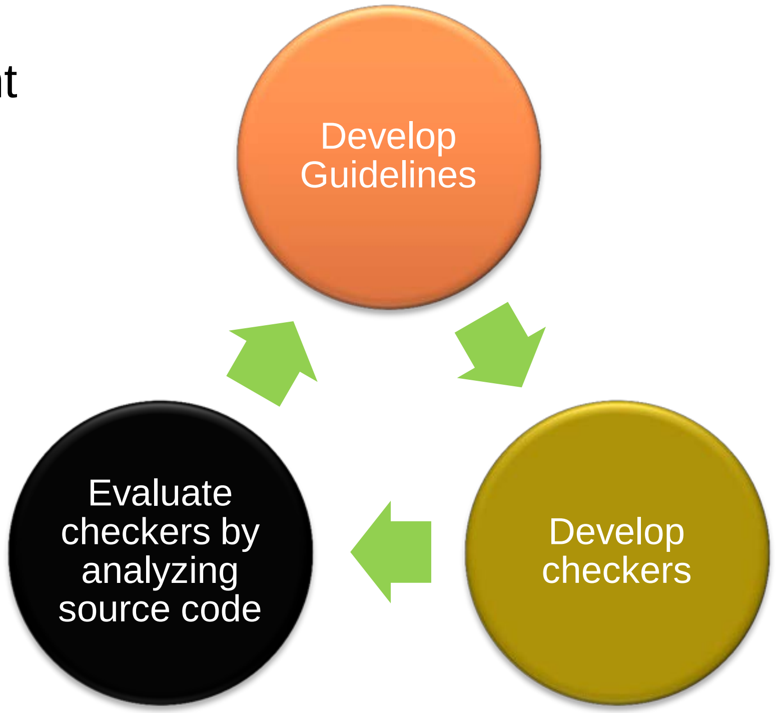
- Version 1.0 (C++ 11) under development

CERT Oracle Secure Coding Standard for Java

- Version 1.0 for Java SE 6 published
- Static analysis under development

The CERT Perl Secure Coding Standard

- Version 1.0 under development

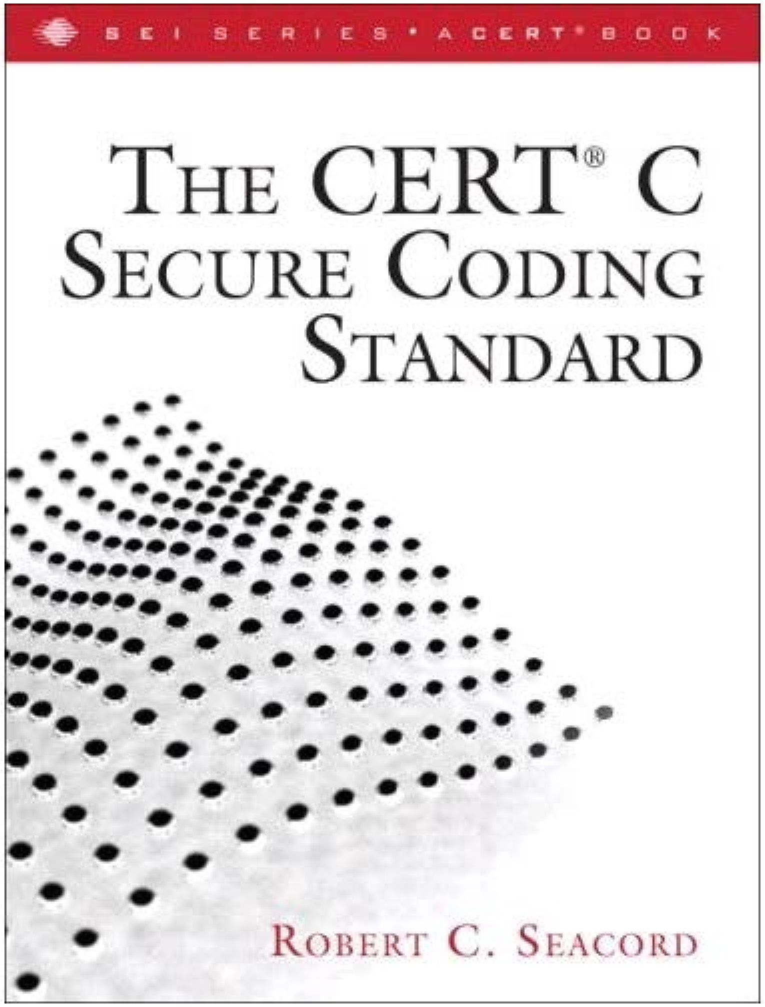


The CERT C Secure Coding Standard

Developed with community involvement, including over 500 registered participants on the wiki.

Version 1.0 published by Addison-Wesley in September, 2008.

- 134 Recommendations
- 89 Rules



Noncompliant Examples & Compliant Solutions

Noncompliant Code Example

In this noncompliant code example, the **`char`** pointer **p** is initialized to the address of a string literal. Attempting to modify the string literal results in undefined behavior.

```
char *p = "string literal"; p[0] = 'S';
```

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array **`a`**. The string stored in a can be safely modified.

```
char a[] = "string literal"; a[0] = 'S';
```

Risk Assessment

Risk assessment is performed using failure mode, effects, and criticality analysis

Severity – how serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	low	denial-of-service attack, abnormal termination
2	medium	data integrity violation, unintentional information disclosure
3	high	run arbitrary code

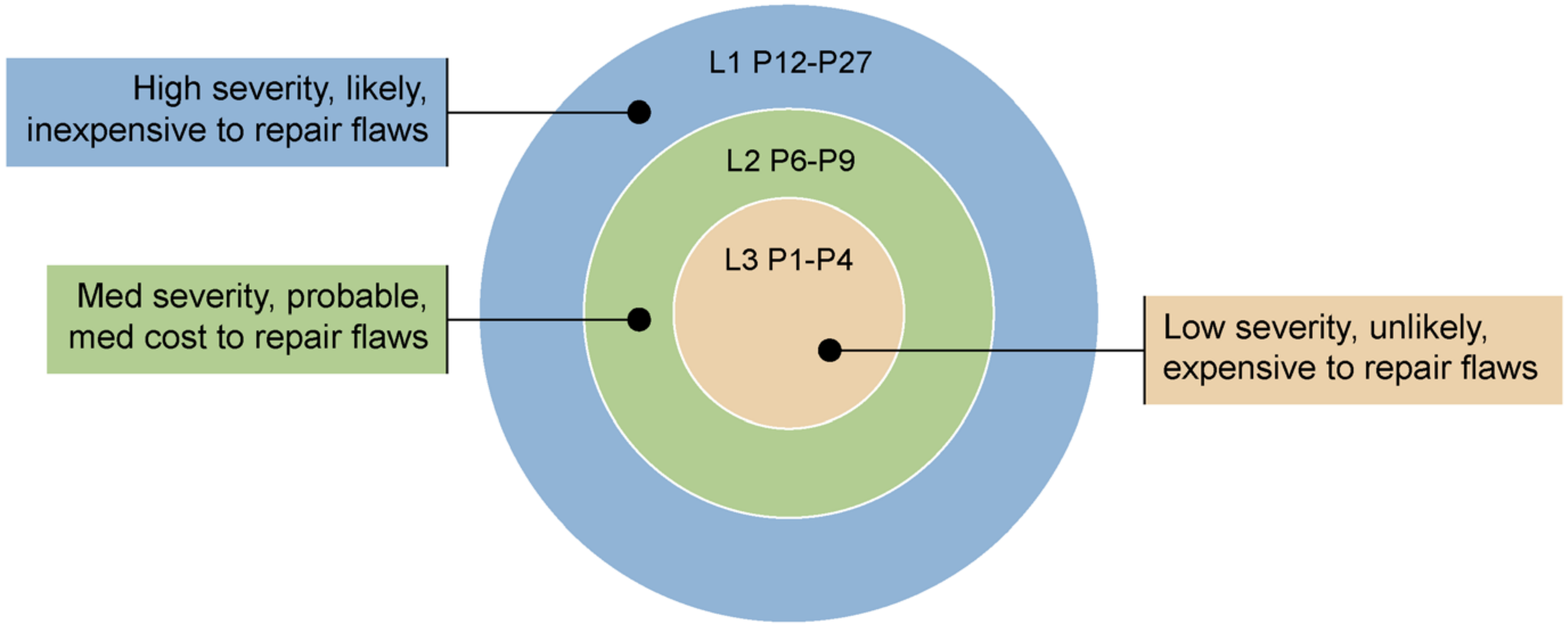
Likelihood – how likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	unlikely
2	probable
3	likely

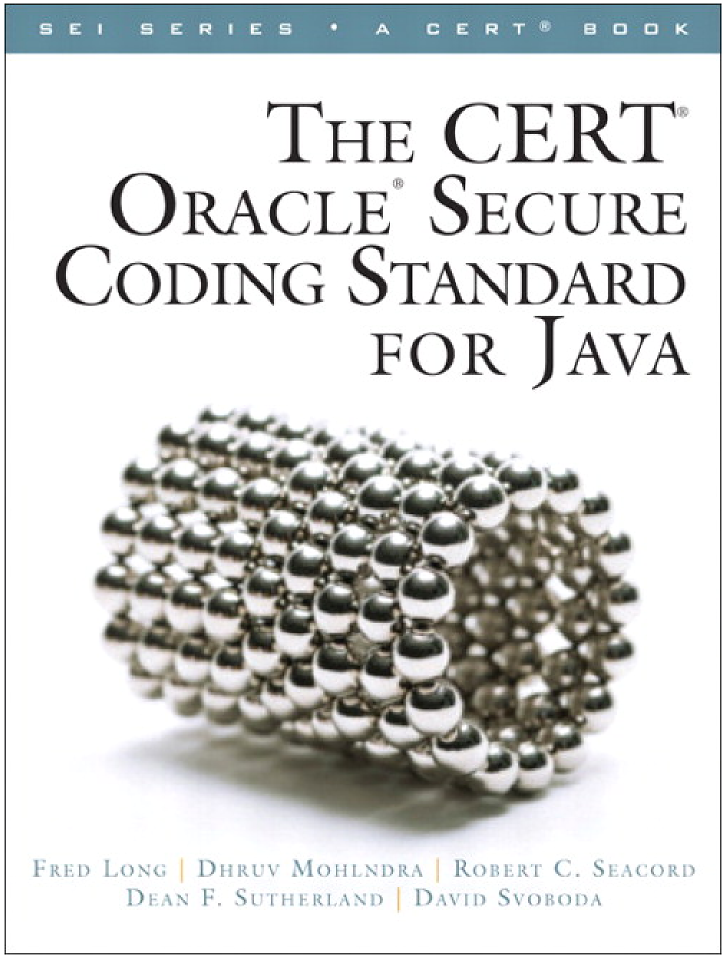
Cost – the cost of mitigating the vulnerability.

Value	Meaning	Detection	Correction
1	high	manual	manual
2	medium	automatic	manual
3	low	automatic	automatic

Priorities and Levels


L1 P12-P27
L2 P6-P9
L3 P1-P4
High severity, likely, inexpensive to repair flaws
Med severity, probable, med cost to repair flaws
Low severity, unlikely, expensive to repair flaws

Secure Coding Standard for Java


SEI SERIES • A CERT® BOOK
THE CERT® ORACLE® SECURE CODING STANDARD FOR JAVA
FRED LONG | DHRUV MOHLNDRA | ROBERT C. SEACORD
DEAN F. SUTHERLAND | DAVID SVOBODA

"In the Java world, security is not viewed as an add-on a feature. It is a pervasive way of thinking. Those who forget to think in a secure mindset end up in trouble. But just because the facilities are there doesn't mean that security is assured automatically. A set of standard practices has evolved over the years. ***The Secure® Coding® Standard for Java™*** is a compendium of these practices. These are not theoretical research papers or product marketing blurbs. This is all serious, mission-critical, battle-tested, enterprise-scale stuff."

–**James A. Gosling,** Father of the Java Programming Language

Scope

The CERT Oracle Secure Coding Standard for Java focuses on the Java Standard Edition 6 Platform (Java SE 6) environment and includes rules for secure coding using the Java programming language and libraries.

The Java Language Specification (3rd edition) [JLS 2005] prescribes the behavior of the Java programming language and served as the primary reference for the development of this standard.

This coding standard also addresses new features of the Java SE 7 Platform, primarily, as alternative compliant solutions to secure coding problems that exist in both the Java SE 6 and Java SE 7 platforms.

Source Code Analysis Laboratory

The CERT Source Code Analysis Laboratory (SCALe) is an operational capability for application conformance testing against one of CERT’s secure coding standards.

- A detailed report of findings is provided to the customer to repair
- After the developer has addressed these findings, the product version is certified as conforming to the standard
- The certification is published in a registry of certified systems

Industry Demand

Conformance with CERT Secure Coding Standards can represent a significant investment by a software developer, particularly when it is necessary to refactor or modernize existing software systems.

However, it is not always possible for a software developer to benefit from this investment, because it is not always easy to market code quality.

A goal of conformance testing is to provide an incentive for industry to invest in developing conforming systems.

- perform conformance testing against CERT secure coding standards
- verify that a software system conforms with a CERT secure coding standard
- use CERT "seal" when marketing products
- maintain a certificate registry with the certificates of conforming systems

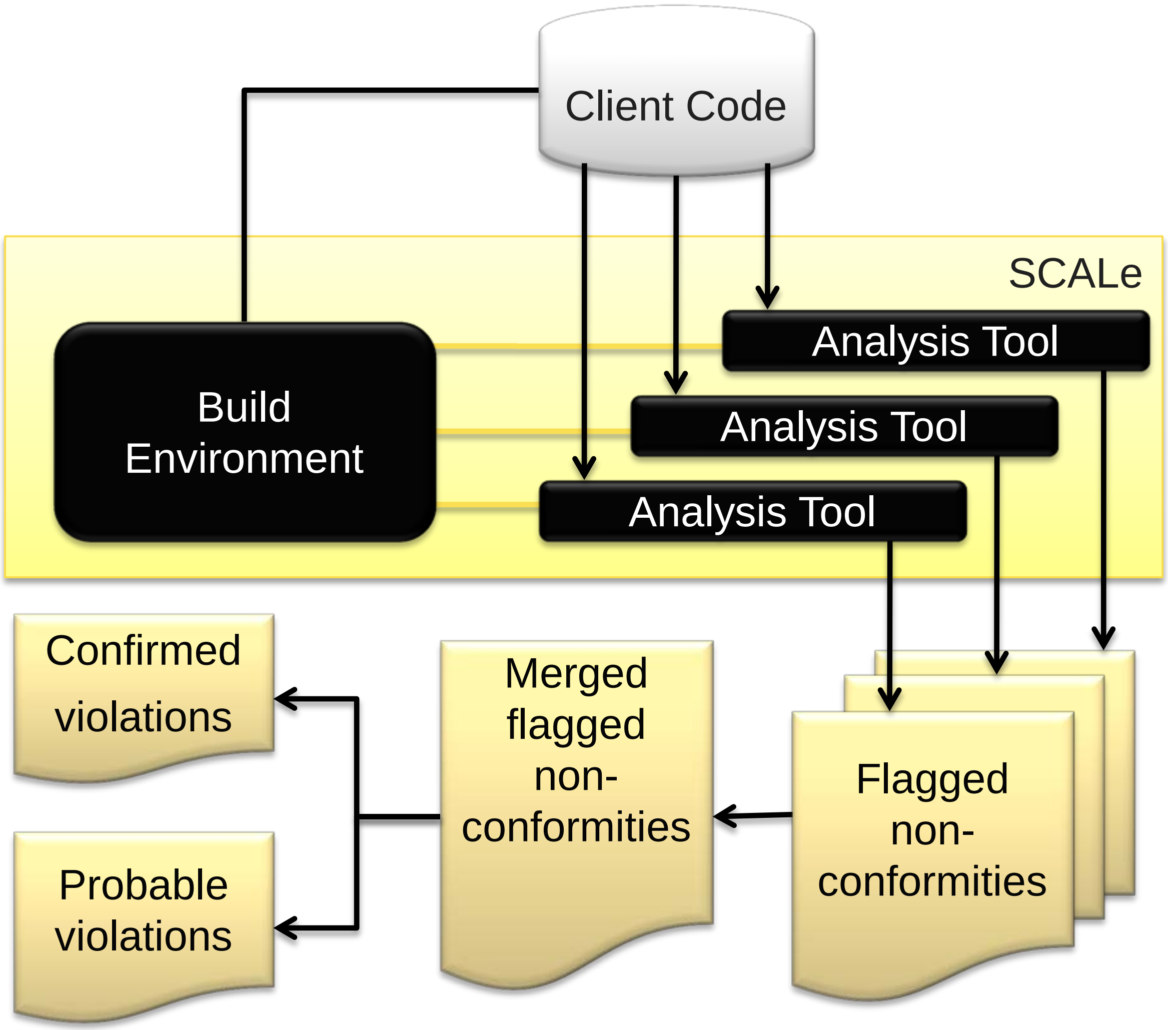
CERT SCALe Seal

Developers of software that has been determined by CERT to conform to a secure coding standard may use the to describe the conforming software on the developer’s website.

The seal must be specifically tied to the software passing conformance testing and not applied to untested products, the company, or the organization.

Use of the CERT SCALe seal is contingent upon the organization entering into a service agreement with Carnegie Mellon University and upon the software being designated by CERT as conforming.

Conformance Testing Process



Conformance Testing

The use of secure coding standards defines a proscriptive set of rules and recommendations to which the source code can be evaluated for compliance.

For each secure coding standard, the source code is certified as provably nonconforming, conforming, or provably conforming against each guideline in the standard:

Provably nonconforming	The code is provably nonconforming if one or more violations of a rule are discovered for which no deviation has been allowed.
Conforming	The code is conforming if no violations of a rule can be identified.
Provably conforming	Finally, the code is provably conforming if the code has been verified to adhere to the rule in all possible cases.

Evaluation violations of a particular rule ends when a “provably nonconforming” violation is discovered.

Deviation Procedure

Strict adherence to all rules is unlikely; consequently, deviations associated with specific rule violations are necessary.

Deviations can be used in cases where a true positive finding is uncontested as a rule violation but the code is nonetheless determined to be secure.

This may be the result of a design or architecture feature of the software or because the particular violation occurs for a valid reason that was unanticipated by the secure coding standard.

- In this respect, the deviation procedure allows for the possibility that secure coding rules are overly strict.

For More Information

Visit CERT® web sites:

http://www.cert.org/secure-coding/

https://www.securecoding.cert.org/

Contact Presenter

Robert C. Seacord

rcs@cert.org

(412) 268-7608

Contact CERT:

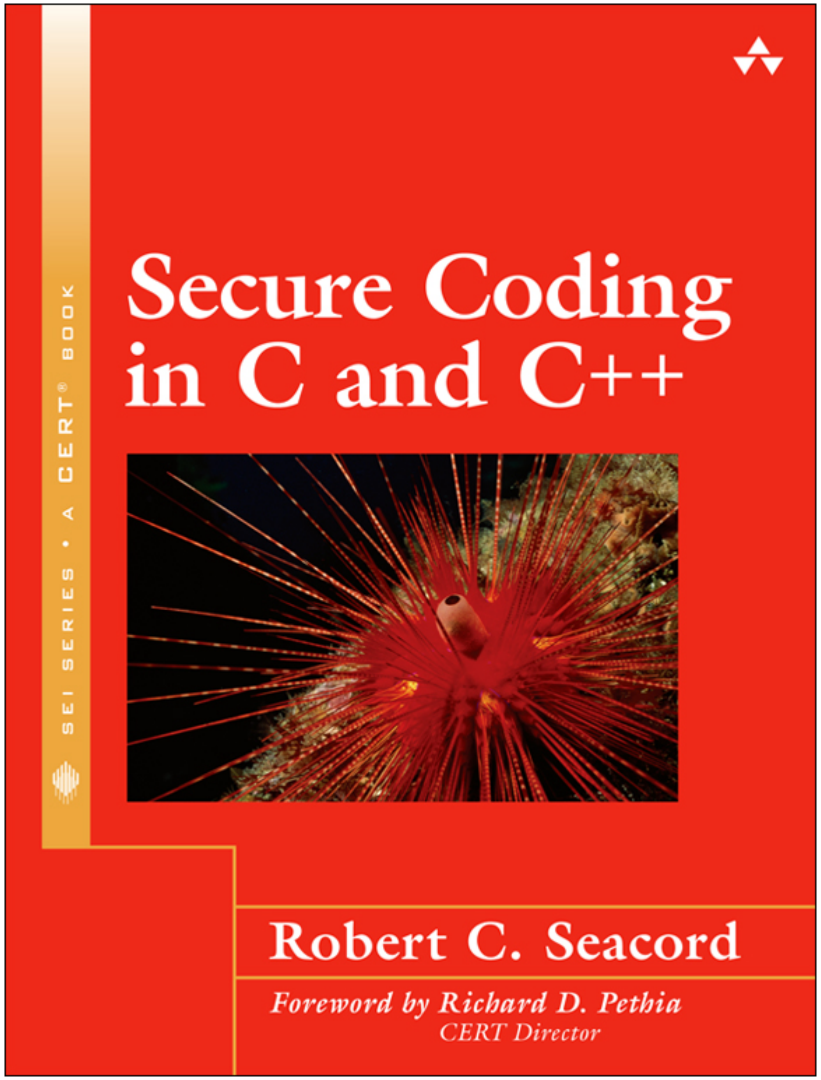
Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890

USA



SEI SERIES • A CERT® BOOK
Secure Coding in C and C++
Robert C. Seacord
Foreword by Richard D. Pethia
CERT Director