



Secure Coding Software Assurance Forum

Robert C. Seacord



Secure Coding Initiative

Initiative Goals

Work with [software developers](#) and [software development organizations](#) to eliminate vulnerabilities resulting from coding errors before they are deployed.

Overall Thrusts

Advance the [state of the practice](#) in secure coding

Identify common programming errors that lead to software vulnerabilities

Establish standard secure coding practices

Educate software developers

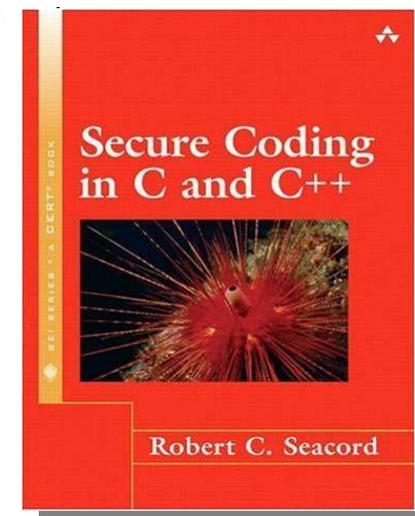
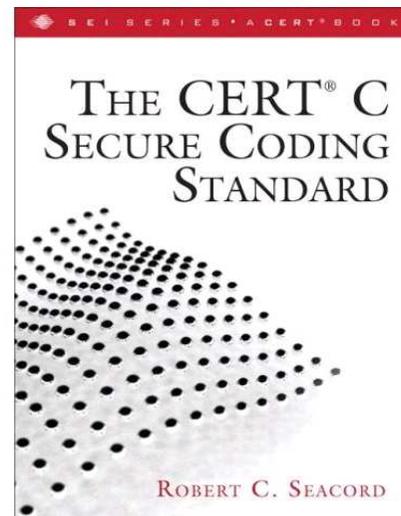
Current Capabilities

Secure coding standards
www.securecoding.cert.org

Source code analysis and conformance testing

Training courses

Involved in international standards development.

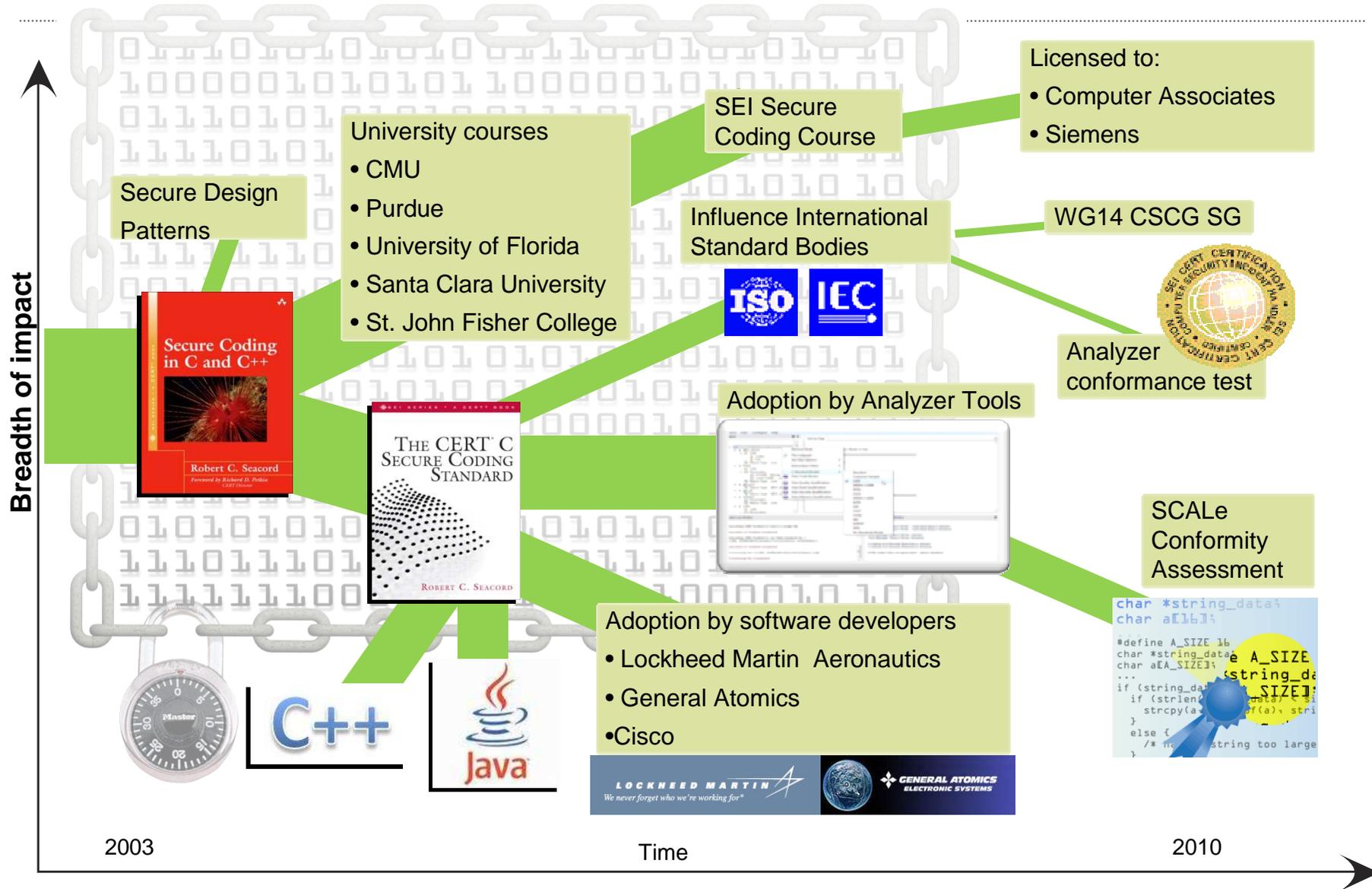


CERT Secure Coding Initiative

Reduce the number of vulnerabilities to a level where they can be handled by computer security incident response teams (CSIRTs)

Decrease remediation costs by eliminating vulnerabilities *before* software is deployed

Secure Coding Roadmap



Agenda

Secure Coding Standards

SCALe

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

CERT Secure Coding Standards

CERT C Secure Coding Standard

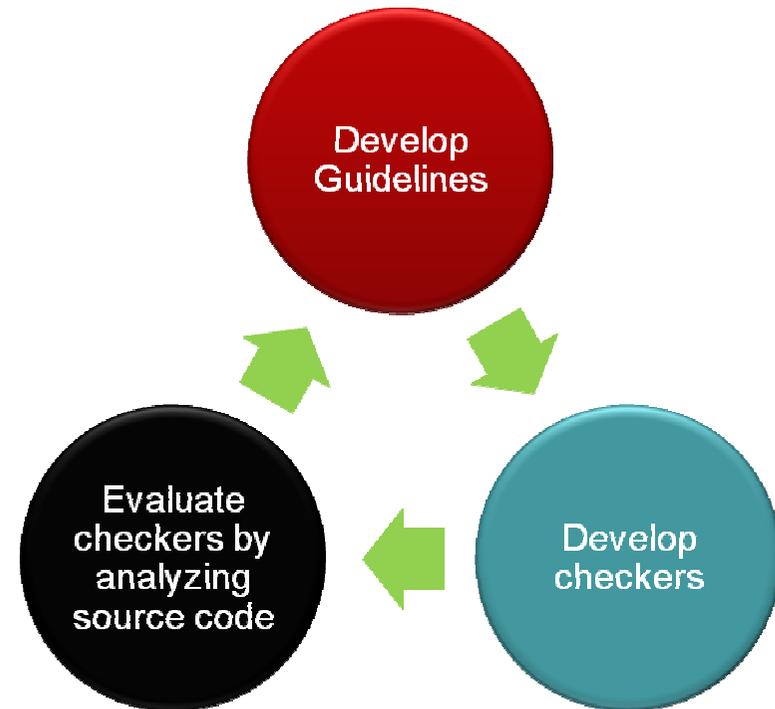
- Extend to C1X
- Analyzable C Secure Coding Guidelines Technical Report

CERT C++ Secure Coding Standard

- Completion of CERT C++ Secure Coding Standard
- Static analysis checkers

CERT Oracle Secure Coding Standard for Java

- Completion of Java Secure Coding Standard
- Static analysis checkers

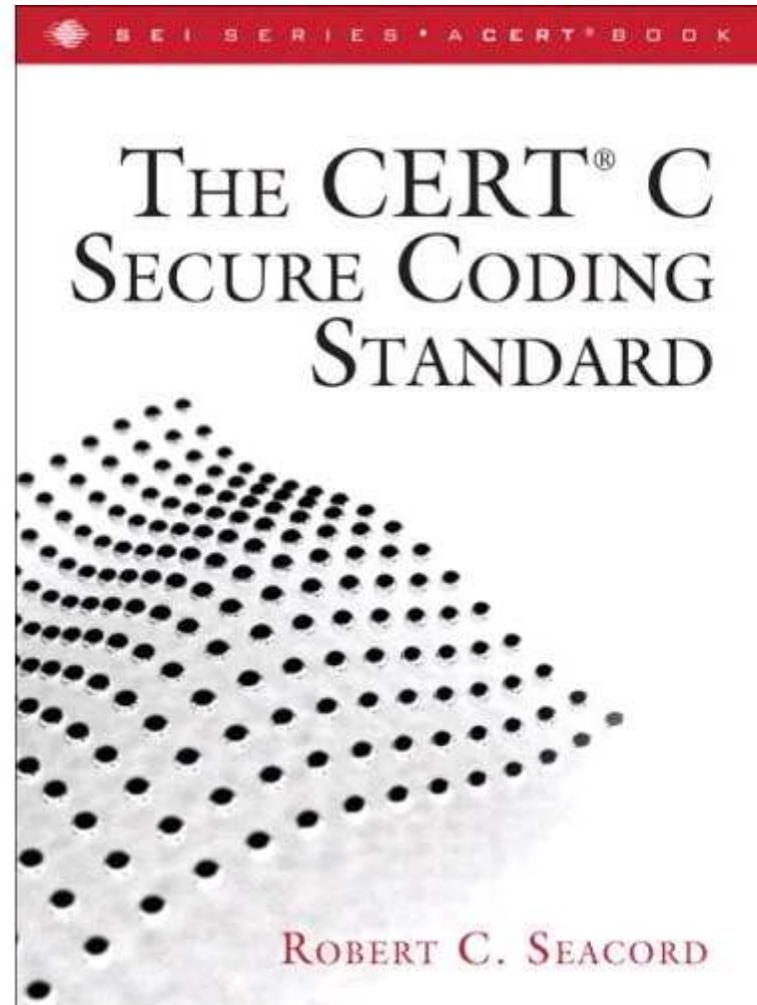


The CERT C Secure Coding Standard

Developed with community involvement, including over 320 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.

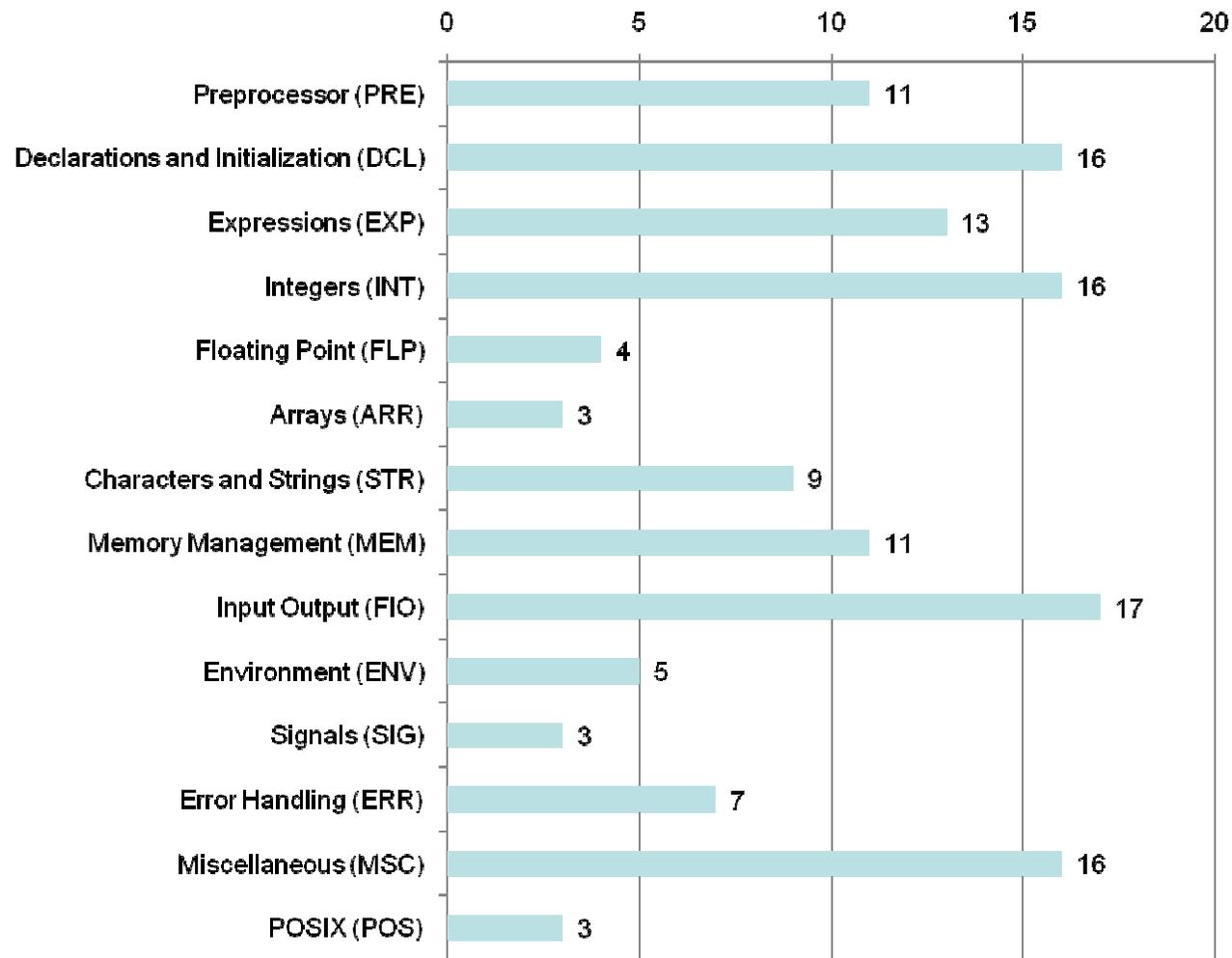
```
char *p = "string literal"; p[0] = 'S';
```

Compliant Solution

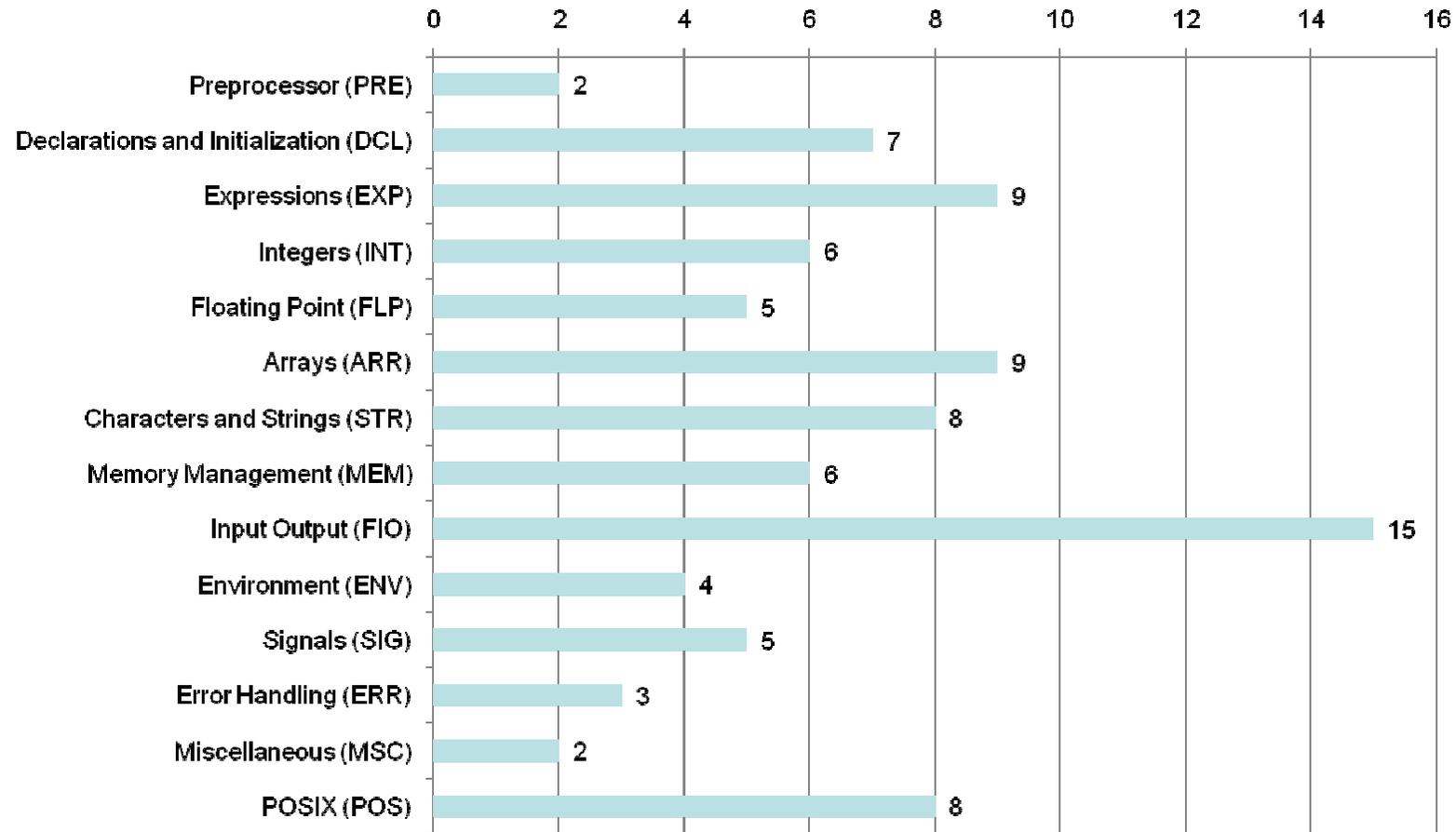
As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array `a`. The string stored in `a` can be safely modified.

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

Distribution of C Recommendations



Distribution of C Rules



Failure Mode, Effects, and Criticality Analysis

<p>Severity – how serious are the consequences of the rule being ignored?</p> <p>Likelihood – how likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?</p> <p>Cost – the cost of mitigating the vulnerability.</p>	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					
	Value			Meaning				
	1	unlikely						
	2	probable						
	3	likely						
	Value		Meaning		Detection		Correction	
	1	high	manual		manual		manual	
2	medium	automatic		automatic		manual		
3	low	automatic		automatic		automatic		

FIO30-C. Exclude user input from format strings

Risk Assessment

Failing to exclude user input from format specifiers may allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	3 (high)	3 (probable)	3 (low)	P27	L1

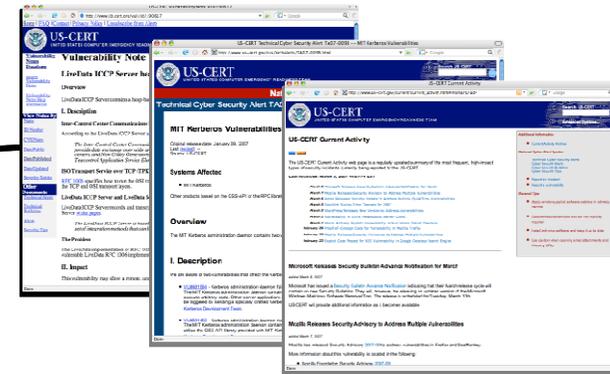
Two recent examples of format string vulnerabilities resulting from a violation of this rule include [Ettercap](#)^a and [Samba](#)^a. In Ettercap v.NG-0.7.2, the ncurses user interface suffers from a format string defect. The `curses_msg()` function in `ec_ncurses.c` calls `wdg_scroll_print()`, which takes a format string and its parameters and passes it to `vw_printw()`. The `curses_msg()` function uses one of its parameters as the format string. This input can include user-data, allowing for a format string vulnerability [[VU#286468](#)]. The Samba AFS ACL mapping VFS plug-in fails to properly sanitize user-controlled filenames that are used in a format specifier supplied to `snprintf()`. This security flaw becomes exploitable when a user is able to write to a share that uses Samba's `afsacl.so` library for setting Windows NT access control lists on files residing on an AFS file system.

Examples of vulnerabilities resulting from the violation of this rule can be found on the [CERT website](#)^a.

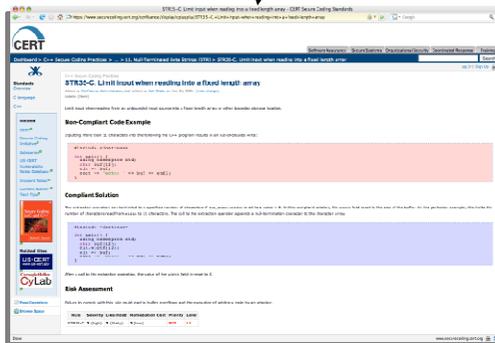
CERT Mitigation Information

Vulnerability Note VU#649732

This vulnerability occurred as a result of failing to comply with rule [FIO30-C](#) of the CERT C Programming Language Secure Coding Standard.



US CERT Technical Alerts



CERT Secure Coding Standard

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the [CERT website](#).

Secure Coding Standard Applications

Establish secure coding practices within an organization

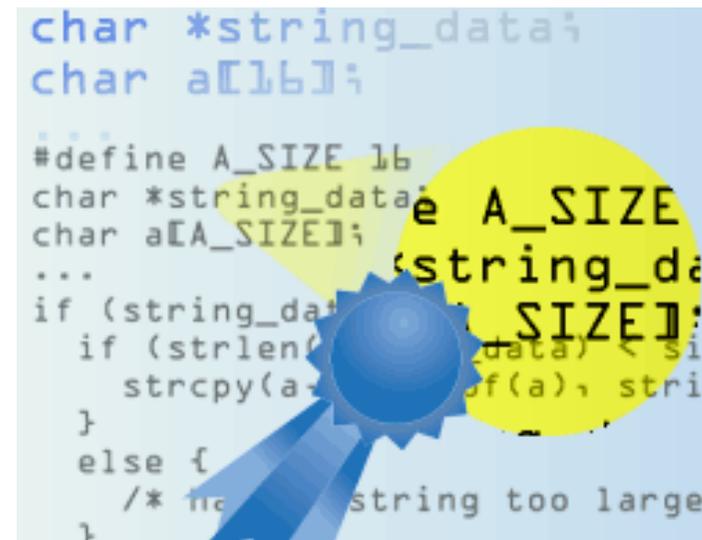
- may be extended with organization-specific rules
- cannot replace or remove existing rules

Train software professionals

Certify programmers in secure coding

Establish requirements for software analysis tools

Software Certification



```
char *string_data;  
char a[16];  
  
#define A_SIZE 16  
char *string_data; e A_SIZE  
char a[A_SIZE]; (string_da  
...  
if (string_da A_SIZE]:  
    if (strlen(data) < si  
        strcpy(a, f(a), stri  
    }  
else {  
    /* no string too large  
}
```

Organizational Adoption

Software developers organizations that have begun to require code to conform to the CERT C Secure Coding Standard:



Software tools that (partially) enforce the CERT C Secure Coding Standard:



True Positives vs. Flagged Nonconformities

Do not apply the `sizeof` operator to an expression of pointer type

(ARR01-C) Applying the `sizeof` operator to an expression of pointer type can result in under allocation, partial initialization, partial copying, or other logical incompleteness or inconsistency if, as is usually the case, the programmer means to determine the size of an actual object. If the mistake occurs in an allocation, subsequent operations on the under-allocated object may lead to buffer overflows.

Ratio of true positives (bugs) to flagged nonconformities:

Software System	TP/FNC	Ratio
Mozilla Firefox version 2.0	6/12	50%
Linux kernel version 2.6.15	10/126	8%
Wine version 0.9.55	37/126	29%
xc, version unknown	4/7	57%

Agenda

Secure Coding Standards

SCALe

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

Source Code Analysis Laboratory

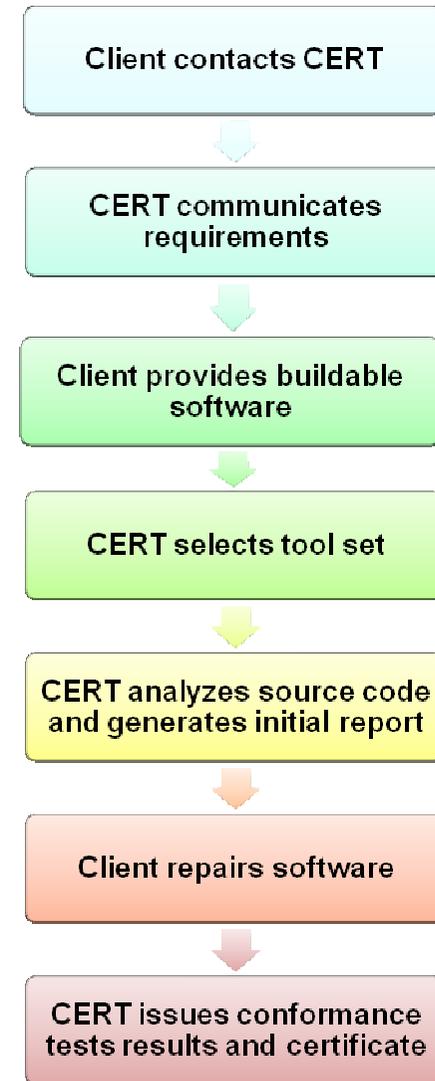
The CERT Source Code Analysis Laboratory (SCALE) is an operational capability for application conformance testing against one of CERT's secure coding standards.

- 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.

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.

INT30-C.	Provably nonconforming
INT31-C.	Documented deviation
INT32-C.	Conforming
INT33-C.	Provably Conforming



Government Demand

CERT secure coding initiative has performed source code assessments for various government agencies.

The Application Security and Development Security Technical Implementation Guide (STIG)

- is being specified in DoD acquisition programs' Request for Proposals (RFPs).
- provides security guidance for use throughout an application's development lifecycle.

Section 2.1.5, “Coding Standards” of the Application Security and Development STIG identifies the following requirement:

(APP2060.1: CAT II) The Program Manager will ensure the development team follows a set of coding standards."

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 otherwise 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
- maintain a certificate registry with the certificates of conforming systems

Conformance Certificates

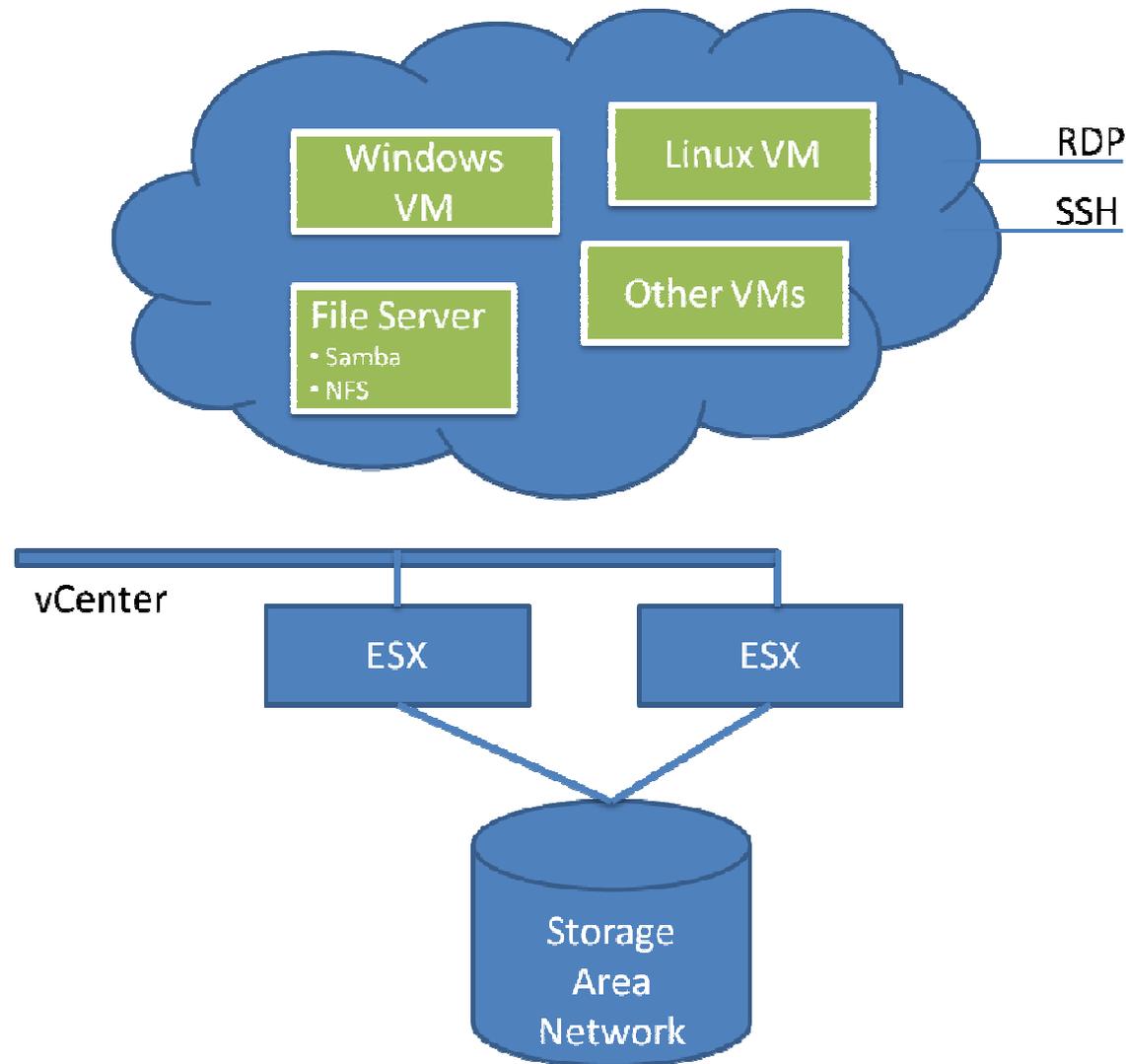
Certificates contain the name and version of the software system which passed the conformance test, and the results of the test.

Similar process followed by The Open Group (see <http://www.opengroup.org/collaboration-services/certification.html>)

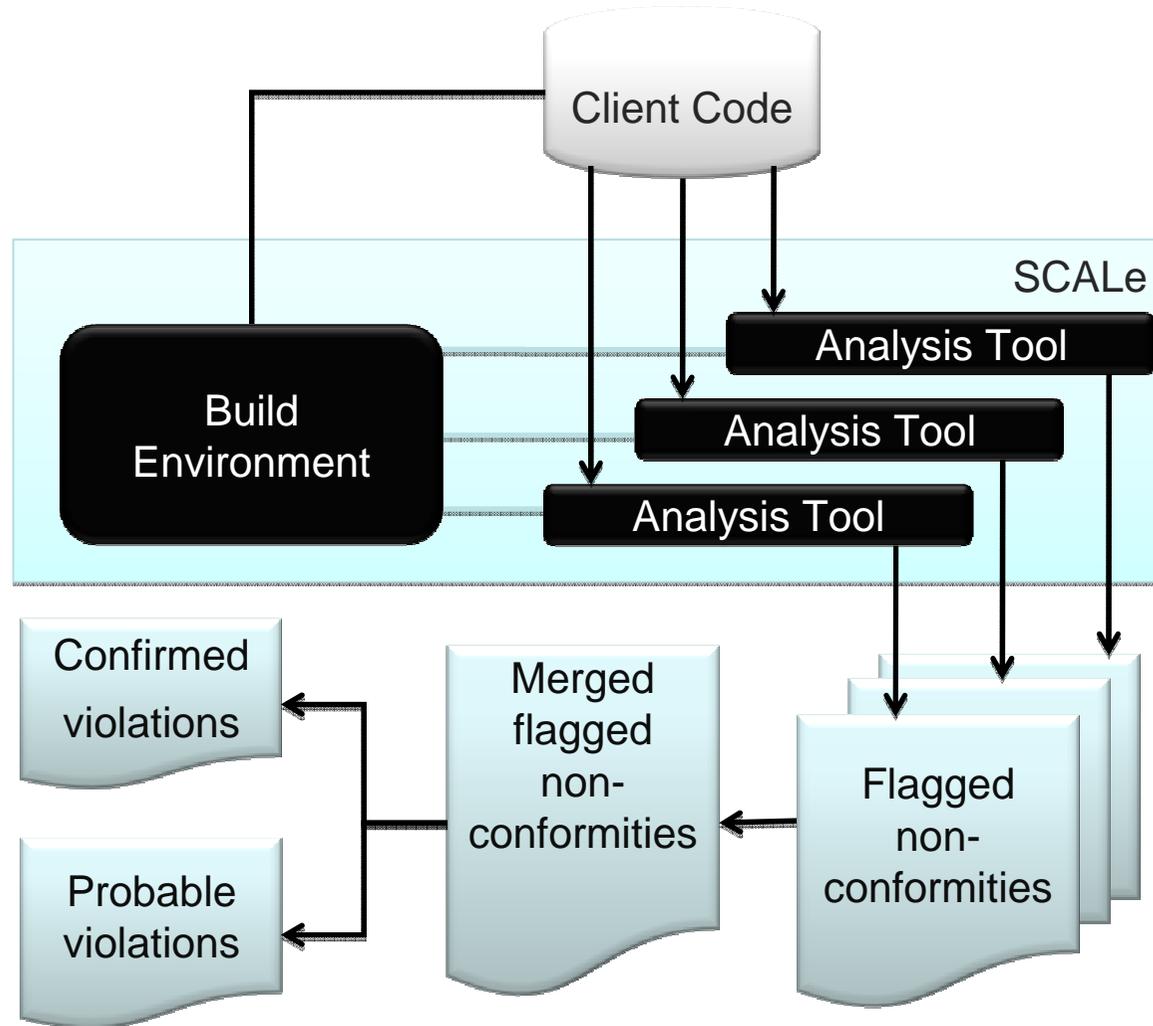
Initially, all assessments are performed by CERT

In the future, third-parties may be accredited to perform certifications.

Source Code Analysis Laboratory



Conformance Testing Process



Agenda

Secure Coding Standards

SCALe

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

As-If Infinitely Ranged Integers

The purpose of the AIR integer model is to either

- produce a value which is equivalent to a value that would have been obtained using infinitely ranged integers
- result in a runtime constraint violation.

This model is generally applied to both signed and unsigned integers but may be enabled or disabled per compilation unit.

AIR Integer Model

In the AIR integer model, when an **observation point** is reached, and before any **critical undefined behavior** occurs, if traps have not been disabled, and if no traps have been raised, then any integer value in the output is correctly represented (“as if infinitely ranged”).

An observation point occurs at an output, including a volatile object access.

Traps are implemented using either

- existing hardware traps (such as divide-by-zero)
- by invoking a runtime-constraint handler

Observation Points

AIR Integers do not **require** that an exception is raised every time there is an integer overflow or truncation error.

It is acceptable to delay catching an incorrectly represented value until an observation point is reached just before it either

- affects the output
- causes a **critical undefined behavior** (as defined by the C1X Analyzability Annex).

This model improves the ability of compilers to optimize, without sacrificing safety and security.

Availability

Requirements

- A patched version of GCC 4.5.0 to insert the overflow and truncation checks
- A patched `stdlib.h` file to include the runtime-constraint handler definitions from ISO/IEC TR 24731-1
- The `libconstraint` library, which defines the constraint handlers used by AIR Integers

Can all be downloaded from:

<http://www.cert.org/secure-coding/integralsecurity.html>

Testing vs. Runtime Protection

AIR integers can be used in both dynamic analysis and as a runtime protection scheme.

There is a well understood tradeoff between runtime overhead and development costs.

- Providing correctness “guarantees” requires extensive testing and excruciating attention to detail
- Development costs can be decreased by adding runtime protection mechanisms however this will
 - increase the size of the executable
 - Introduce runtime overhead
- Runtime protection mechanisms still require a viable recovery strategy
- It is reasonable to provide some level of assurance combined with runtime checks, but you don’t want to pay twice

As-if Infinitely Ranged (AIR) Integers

AIR integers is a model for automating the elimination of integer overflow and truncation in C and C++ code.

- integer operations either succeed or trap
- uses the runtime-constraint handling mechanisms defined by ISO/IEC TR 24731-1 and C1X Annex L “Analyzability”
- generates constraint violations for overflow, wrapping, and truncation

AIR integer model has been fully implemented in a proof-of-concept modification to the GCC compiler Version 4.5.0 for IA-32 processors
SPECINT2006 macro-benchmarks

Optimization Level	Control Ratio	Analyzable Ratio	% Slowdown
-O0	4.92	4.60	6.96
-O1	7.21	6.77	6.50
-O2	7.38	6.99	5.58

AIR Integer Efficacy Study

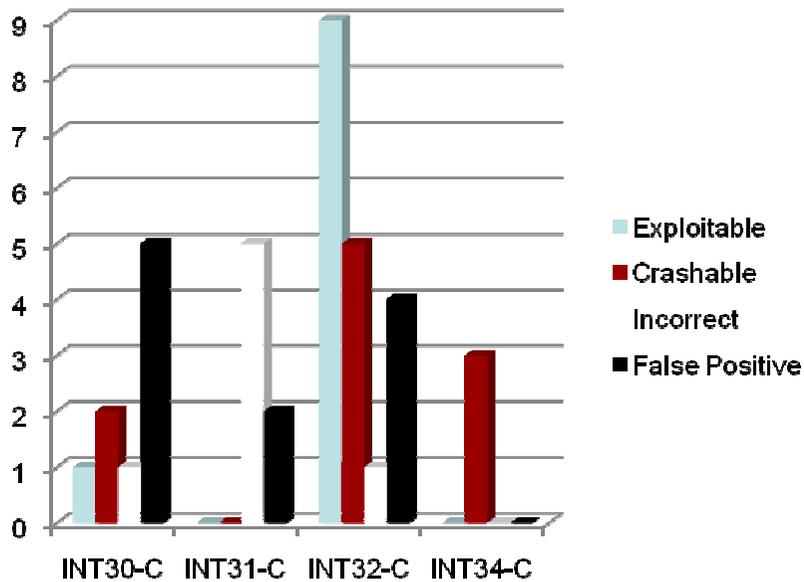
Jasper and FFmpeg libraries instrumented using the modified GCC compiler and fuzz tested.

Violations of the following CERT C Secure Coding Standard rules were discovered:

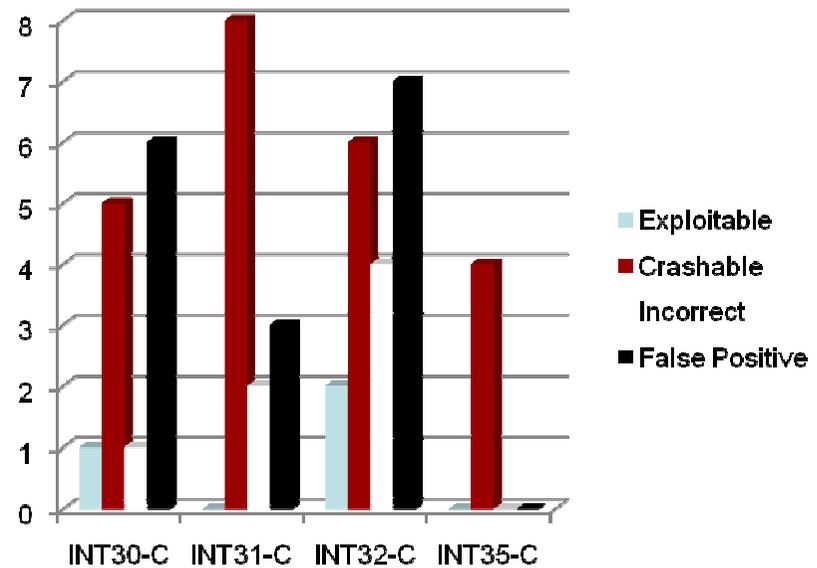
- INT30-C. Ensure that unsigned integer operations do not wrap
- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- INT32-C. Ensure that operations on signed integers do not result in overflow
- INT34-C. Do not shift a negative number of bits or more bits than exist in the operand
- INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

AIR Integer Efficacy Study

Defects discovered in JasPer image processing library



Defects discovered in FFmpeg audio/video processing library



Agenda

Secure Coding Standards

SCALE

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

Secure C Compiler

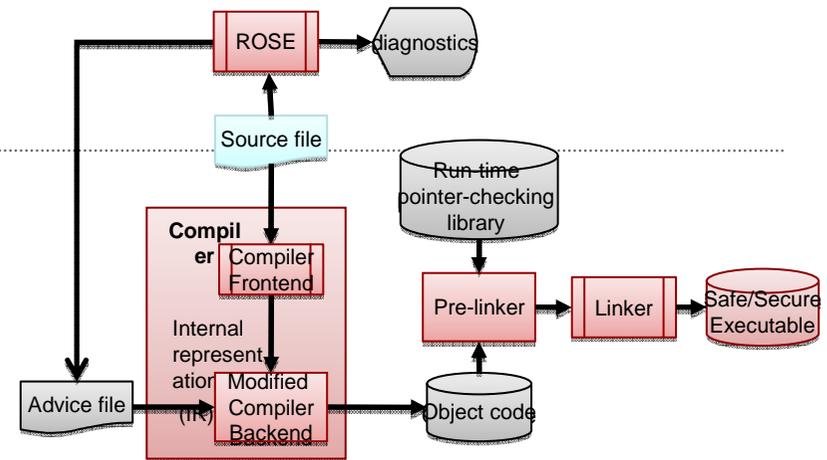
Develop a holistic solution to the problem that includes

- C1X analyzability annex
- As-if infinitely ranged (“AIR”) integers
- Plum Hall Safe Secure C/C++ methods (SSCC)
- C and C++ Secure Coding Guidelines

This solution eliminates the vulnerabilities:

- Writing outside the bounds of an object (e.g., buffer overflow)
- Reading outside the bounds of an object
- Arbitrary reads/writes (e.g., wild-pointer stores)
- Integer overflow and truncation

Proof-of-concept implementation of the AIR integer model built upon Clang/LLVM has been completed



Completed So Far

Chose Clang/LLVM for integer overflow portion.

Implemented checking for AIR Integer model.

- There were already checks for some signed operations (via `-ftrapv`), but they were incomplete and buggy. These were fixed and extended to cover the AIR Integer model.
- Checks were added for unsigned operations.

Strategy: Insert checks for all operations that might need them, then optimize away as many as possible.

- Avoids spreading optimizations out all over the compiler; concentrates them in the optimizer.
- All the work is done in the LLVM intermediate representation, not the machine-specific assembly code.
- Therefore, the AIR Integer checking should work for all of LLVM's target architectures (though this has only been tested for x86-64).

Future Plans

Choose a platform for buffer overflow prevention.

Over the next year: Add buffer overflow prevention capabilities to a compiler, for a single-threaded environment.

The following year: Extend to a multithreaded environment.

Platform Contenders

Clang/LLVM, possibly including the SAFECode project at the University of Illinois.

<http://sva.cs.illinois.edu>

EDG/ROSE, which offers a higher-level look at the program.

Cetus was considered but will probably not be used because the others are more mainstream and have clearer licenses.

Agenda

Secure Coding Standards

SCALE

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

ISO/IEC International Standards

C++ standards committees (WG21 & PL22.16)

- Hosted Pittsburgh meeting which resulted in Final Committee Draft of C++

C Standards Committees (WG14 & PL22.11)

- CERT Chairs INCITS PL22.11 Chair
- Hosting May 2011 meeting in Boulder, Colorado
- Developed multiple proposals for improved security which have been adopted by WG14 for C1X

C Secure Coding Rules Study Group (CSCR SG)

ISO/IEC JTC 1/SC 22/WG 23 Programming Language
Vulnerabilities

PL22 (programming languages)

CSCR SG History

The idea of C secure coding guidelines arose during the discussion of the managed strings proposal at the Berlin meeting of the ISO/IEC JTC 1/SC 22/WG 14 for standardization of the C language in March, 2006.

The closest existing product at the time, MISRA C, was generally viewed by the committee as inadequate because, among other reasons, it precluded all the language features which had been introduced by ISO/IEC 9899:1999.

CERT C Secure Coding Guidelines

In collaboration with the software assurance and C language development communities, CERT developed **The CERT C Secure Coding Standard** to provide secure coding guidance to developers.

Recent History

The CERT C Secure Coding guidelines were first reviewed by WG14 at the London (April 2007) meeting and again at the Kona meeting (August 2007)

In 2009, CERT developed a set of automatically-enforceable C Secure Coding Guidelines and contributed this document to ISO/IEC for use in the standardization process.

C Secure Coding Guidelines SG

Purpose: Study the problem of producing analyzable secure coding guidelines for C99 and C1x

First meeting held on October 27, 2009

Meetings will be held the first and third Wednesday of each month by teleconference

- David Keaton/CERT is the chair
- Martin Sebor/CISCO is the vice-chair
- Robert Seacord is the project editor

CSCR SG Wiki:

<https://www.securecoding.cert.org/confluence/display/CSCG/C+Secure+Coding+Guidelines>

Mailing list : wg14-cscg-l@cert.org

CSCG SG

The study group is studying:

1. where the work belongs,
2. what to use as a base document (if any), and
3. what is the appropriate deliverable

Test Suites

CERT agrees to sponsor and coordinate a test suite under BSD-type license (freely available for any use)

May make use of, or be integrated with, the NIST SAMATE Reference Dataset

Dangerous Optimizations and the Loss of Causality

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

Undefined behaviors are identified in the standard:

- If a “**shall**” or “**shall not**” requirement is violated, and that requirement appears outside of a constraint, the behavior is undefined.
- Undefined behavior is otherwise indicated in this International Standard by the words “**undefined behavior**”
- by the omission of any explicit definition of behavior.

There is no difference in emphasis among these three; they all describe “behavior that is undefined”.

C99 Annex J.2, “Undefined behavior,” contains a list of explicit undefined behaviors in C99.

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).

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

From C99 §6.5.6p8:

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 $\mathbf{P}[\mathbf{N}]$ is translated into $\mathbf{*}(\mathbf{P}+\mathbf{N})$.

Adding a Pointer and an Integer

C99 Section 6.5.6 says

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.

C1X Analyzability Annex

This annex specifies optional behavior that can aid in the analyzability of C programs.

An implementation that defines

`__STDC_ANALYZABLE__` shall conform to the specifications in this annex.

Definitions

out-of-bounds store: an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared volatile, fetch) one or more bytes that lie outside the bounds permitted by this Standard.

bounded undefined behavior: undefined behavior (3.4.3) that does not perform an out-of-bounds store.

NOTE 1 The behavior might perform a trap.

NOTE 2 Any values produced or stored might be indeterminate values.

critical undefined behavior: undefined behavior that is not bounded undefined behavior.

NOTE The behavior might perform an out-of-bounds store or perform a trap.

Requirements

If the program performs a trap (3.19.5), the implementation is permitted to invoke a runtime-constraint handler. Any such semantics are implementation-defined.

All undefined behavior shall be limited to bounded undefined behavior, except for the following which are permitted to result in critical undefined behavior.

Critical Undefined Behaviors

1. An object is referred to outside of its lifetime (6.2.4).
2. An lvalue does not designate an object when evaluated (6.3.2.1).
3. A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).
4. The operand of the unary * operator has an invalid value (6.5.3.2).
5. Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).
6. An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).
7. The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.21.3).
8. A string or wide string utility function is instructed to access an array beyond the end of an object (7.22.1, 7.27.4).

Agenda

Secure Coding Standards

SCALe

AIR Integers

Secure C Compiler

Standards

Secure Coding Education

Secure Coding in C/C++ Course

Four day course provides practical guidance on secure programming

- provides a detailed explanation of common programming errors
- describes how errors can lead to vulnerable code
- evaluates available mitigation strategies
- <http://www.sei.cmu.edu/products/courses/p63.html>

Useful to anyone involved in developing secure C and C++ programs regardless of the application

Licensed to Computer Associates and Siemens to train internal software developers

CMU Courses

Offered as an undergraduate elective in the School of Computer Science in S07, S08, S09, and S10

- More of a vocational course than an “enduring knowledge” course.
- Students are interested in taking a class that goes beyond “policy”

Secure Software Engineering graduate course offered at INI in F08, F09, FY10

Secure Coding Education

Developing online version of “Secure Coding in C and C++” course taught at CMU and by the SEI.

Working with the **Eberly Center for Teaching Excellence** and the **Open Learning Initiative** at CMU

“ [OLI is] an amazing and critical piece of work. . . The idea of these virtual labs and intelligent tutoring systems, I think, can really revolutionize education. And we need to revolutionize education. ”

Bill Gates

Co-chair and Trustee of the Bill & Melinda Gates Foundation

[Speaking at Carnegie Mellon](#)

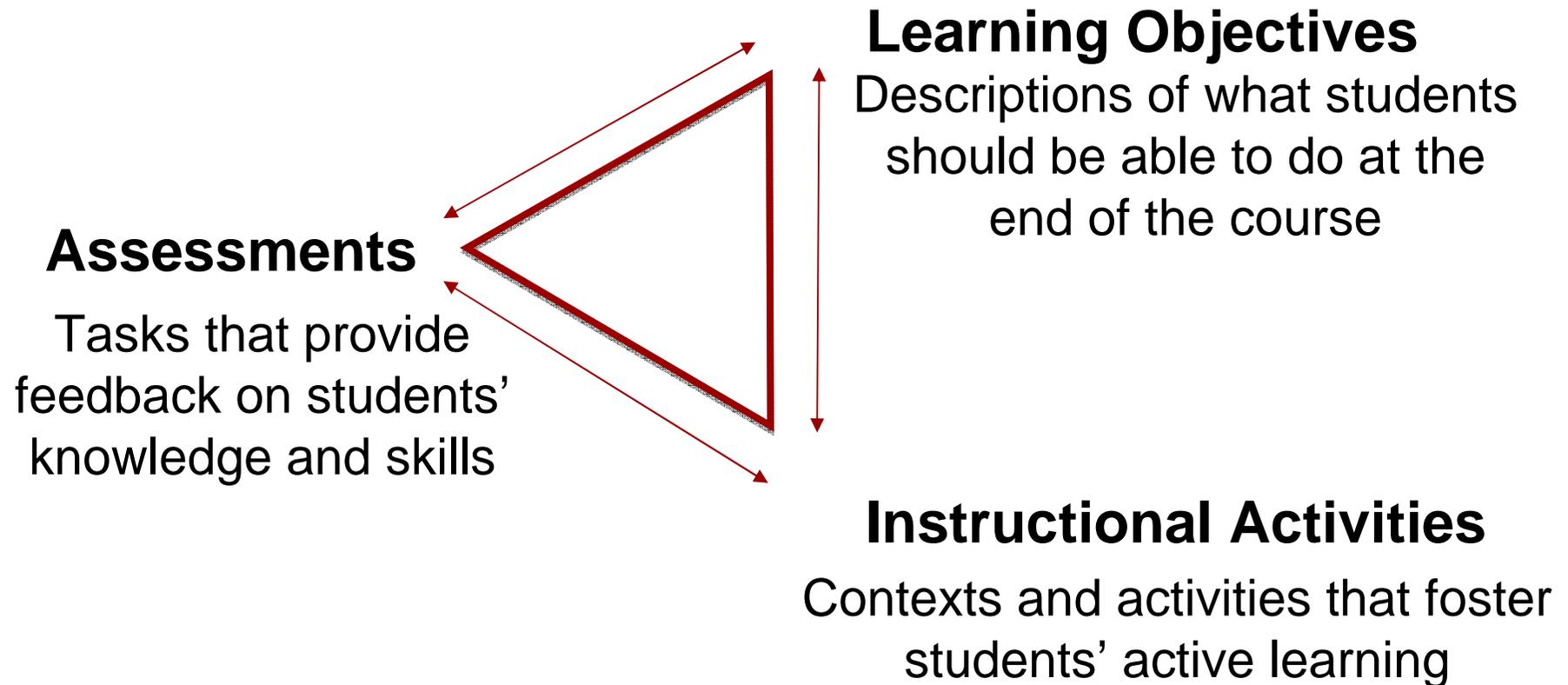
OLI Goals

Produce exemplars of scientifically based online courses and course materials that enact instruction and support instructors

Provide open access to these courses and materials

Develop a community of use, research & development that contributes to the evaluation, continuous improvement, and ongoing growth of the courses and materials.

The Course Design Triangle



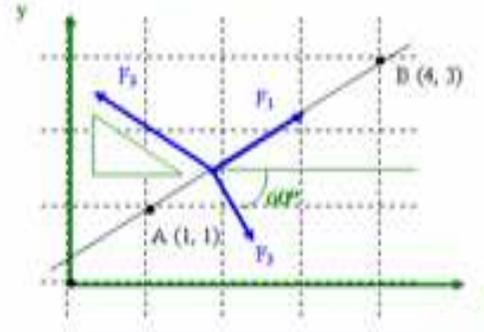
Learners receive support in the problem-solving context

Determine the sum of three concurrent forces:

Force F_1 has a magnitude of 9N, its line of action passes through points A (1, 1) and B (4, 3)

Force F_2 has a magnitude of 2N, its line of action is parallel to a 3-4-5 triangle

Force F_3 has a magnitude of 6N, its line of action is at 60 degrees to the horizontal



Hint

What is the magnitude of the sum?
 $R =$ N

What is the direction of the sum?
 $\theta =$ degrees

Recall

Step 1: Resolve each force into components:

$F_{1x} =$ <input type="text"/> N	$F_{1y} =$ <input type="text"/> N
$F_{2x} =$ <input type="text"/> N	$F_{2y} =$ <input type="text"/> N
$F_{3x} =$ <input type="text"/> N	$F_{3y} =$ <input type="text"/> N

Step 2: Find the components of the sum by summing components of the forces:
 $R_x = \Sigma F_x =$ N $R_y = \Sigma F_y =$ N

Step 3: Find the magnitude of the sum $R = \sqrt{R_x^2 + R_y^2}$
 $R =$ N

Step 4: Find the direction of the sum $\theta = \tan^{-1} \frac{R_y}{R_x}$
 $\theta =$ degrees

What is a Cognitive Tutor?

A computerized learning environment whose design is based on cognitive principles and whose interaction with students is based on that of a (human) tutor

- making comments when the student errs
- answering questions about what to do next
- maintaining a low profile when the student is performing well.

Accelerated Learning Results

OLI students completed course in half a semester, meeting half as often during that time

OLI students showed significantly greater learning gains (on the national standard “CAOS” test for statistics knowledge)

No significant difference between OLI and traditional students in follow-up measures of knowledge retention given a semester later

These results have been replicated with a larger sample

70

Other Class Results

Community College accelerated learning study:

- OLI: 33% more content covered
- OLI: 13% learning gain vs. 2% in traditional face-to-face class

Large State University:

- OLI: 99% completion rate
- Traditional face-to-face class: 41% completion rate

End of Course Student Survey for Accelerated Online

85% Definitely Recommend

15% Probably Recommend

0% Probably not Recommend

0% Definitely not Recommend

Quotes

Student Quote: "This is so much better than reading a textbook or listening to a lecture! My mind didn't wander, and I was not bored while doing the lessons. I actually learned something."

Instructor Quote: "The format [of the accelerated learning study] was among the best teaching experiences I've had in my 15 years of teaching statistics."

For More Information

Visit CERT® web sites:

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

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

Contact Presenter

Robert C. Seacord

rsc@cert.org

(412) 268-7608

Contact CERT:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890

USA

