Secure Coding – Avoiding Future Security Incidents

Robert Seacord
Secure Coding Team Lead

Seacord has over 25 years of software development experience in industry, defense, and research. Seacord's principal areas of expertise include software security, C, C++, and Java-programming languages, component-based development, graphical interface design, human factors. He has worked extensively with EJB, CORBA, JavaBeans, UNIX, Motif, the Common Desktop Environment (CDE), and other graphical user interface systems and technologies.

Seacord was a developer of Version 2.1 of CDE and Motif at the X Consortium. He was responsible for the addition of the printing-through-X capability and desktop integration for the Information Manager. Information Manager is a generalized SGML browser and new CDE 2.1 client. Seacord was also responsible for maintaining the overall quality and integrity of UIL, Mrm, Application Builder, and other CDE desktop libraries and clients. He was also responsible for the resolution of CDE 2.1 source code portability problems on the 6 CDE reference platforms: AIX, HP-UX, Solaris, Digital UNIX, UnixWare and UXP/DS.

NO WARRANTY

THIS MATERIAL OF CARNEGIE MELLON UNIVERSITY AND ITS SOFTWARE ENGINEERING INSTITUTE IS FURNISHED ON AN “AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this presentation is not intended in any way to infringe on the rights of the trademark holder.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

DM-00000-366

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

International Standards

Secure Coding Training

Secure Coding Research

Secure Software and Coding

Systems developed for and delivered to the DoD have security flaws and vulnerabilities that can be exploited by our enemies to neutralize our technological advantage on the battlefield.

http://en.wikipedia.org/wiki/File:Damaged_US_Army_AH-64_Apache,_Iraq.jpg

Relentless Adversaries and Vulnerabilities

Adversaries will likely continue to be present in our systems. Any portion of the cyber infrastructure may be susceptible to manipulation.

Deep reliance on commercial infrastructure, services, and products by the DoD is growing and is a double-edged sword.

Roadmap

Licensed to
- Computer Associates
- Siemens

SEI Secure Coding course

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

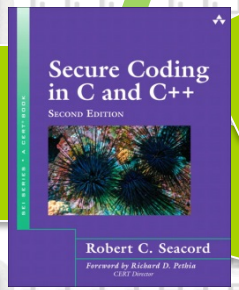
University courses
- CMU
- Stevens Institute
- Purdue
- University of Florida
- Santa Clara University
- St. John Fisher College

Secure design patterns

Influence international standards bodies

ISO/IEC TS 17961 C Secure Coding Rules

Analyzer conformance test



Adoption by analyzer tools:
- LDRA
- Klocwork

SCALe conformance testing

- Thread-role analysis
- Security-enhanced compiler
- Pointer ownership model

Adoption by software developers and acquirers:
- Cisco
- Oracle

CERT CONFORMANCE TESTED Software Engineering Institute Carnegie Mellon

Breadth of impact

2003

Time

2014

What Is Software Security?

Not the same as security software, such as

- Firewalls, intrusion detection, encryption
- Protection of the environment within which the software operates

Goal: Better, defect-free software that can function more robustly in its operational production environment

Application Security

Sources of Software Insecurity 1

Complexity, inadequacy, and change

Incorrect or changing assumptions (capabilities, inputs, outputs)

Flawed specifications and designs

Poor implementation of software interfaces (input validation, error and exception handling)

Inadequate knowledge of secure coding practices

Sources of Software Insecurity 2

Unintended, unexpected interactions

- with other components
- with the software's execution environment

Absent or minimal consideration of security during all lifecycle phases

Not thinking like an attacker

Most Vulnerabilities Are Caused by Programming Errors

64% of the vulnerabilities in the National Vulnerability Database in 2004 were due to programming errors

- 51% of those were due to classic errors like buffer overflows, cross-site scripting, injection flaws
- Heffley/Meunier (2004): Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?

Cross-site scripting, SQL injection at top of the statistics (CVE, Bugtraq) in 2006

"We wouldn't need so much network security if we didn't have such bad software security."

—Bruce Schneier

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

International Standards

Secure Coding Training

Secure Coding Research

CERT Secure Coding Standards

CERT C Secure Coding Standard

- Version 1.0 (C99) published in 2009
- Version 2.0 (C11) published in 2011
- ISO/IEC TS 17961 C Secure Coding Rules Technical Specification
- Conformance Test Suite

CERT C++ Secure Coding Standard

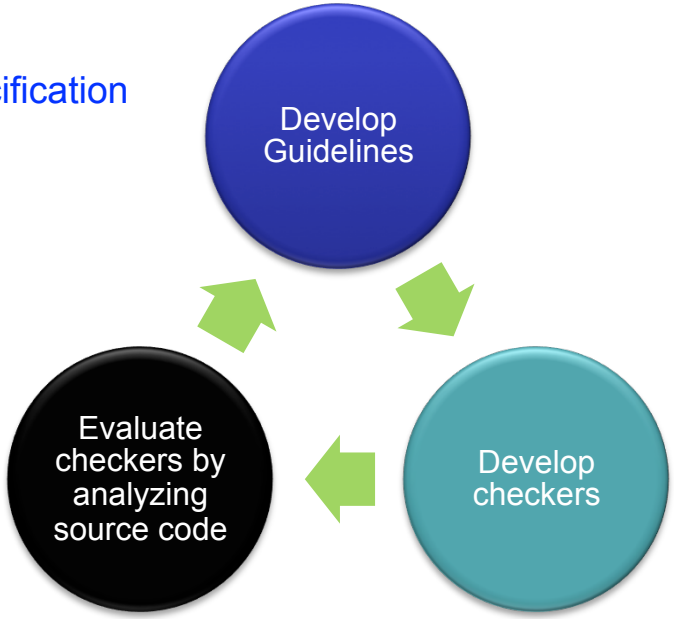
- Not completed/not funded

CERT Oracle Secure Coding Standard for Java

- Version 1.0 (Java 7) published in 2011
- Java Secure Coding Guidelines
- Identified Java rules applicable to Android development
- Planned: Android-specific version designed for the Android SDK

The CERT Perl Secure Coding Standard

- Version 1.0 under development



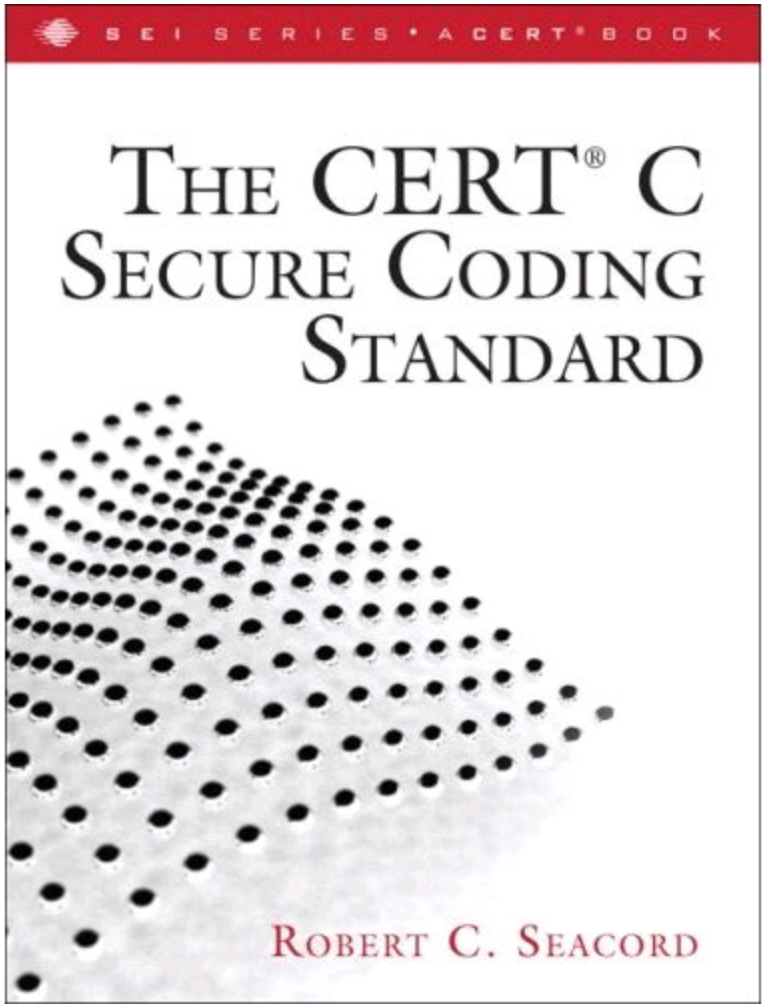
The CERT C Secure Coding Standard

Developed with community involvement

- 1,339 registered contributors on the wiki as of April 2013

Version 1.0 published by Addison-Wesley in September 2008

- 134 recommendations
- 89 rules



Noncompliant Examples and 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, an 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

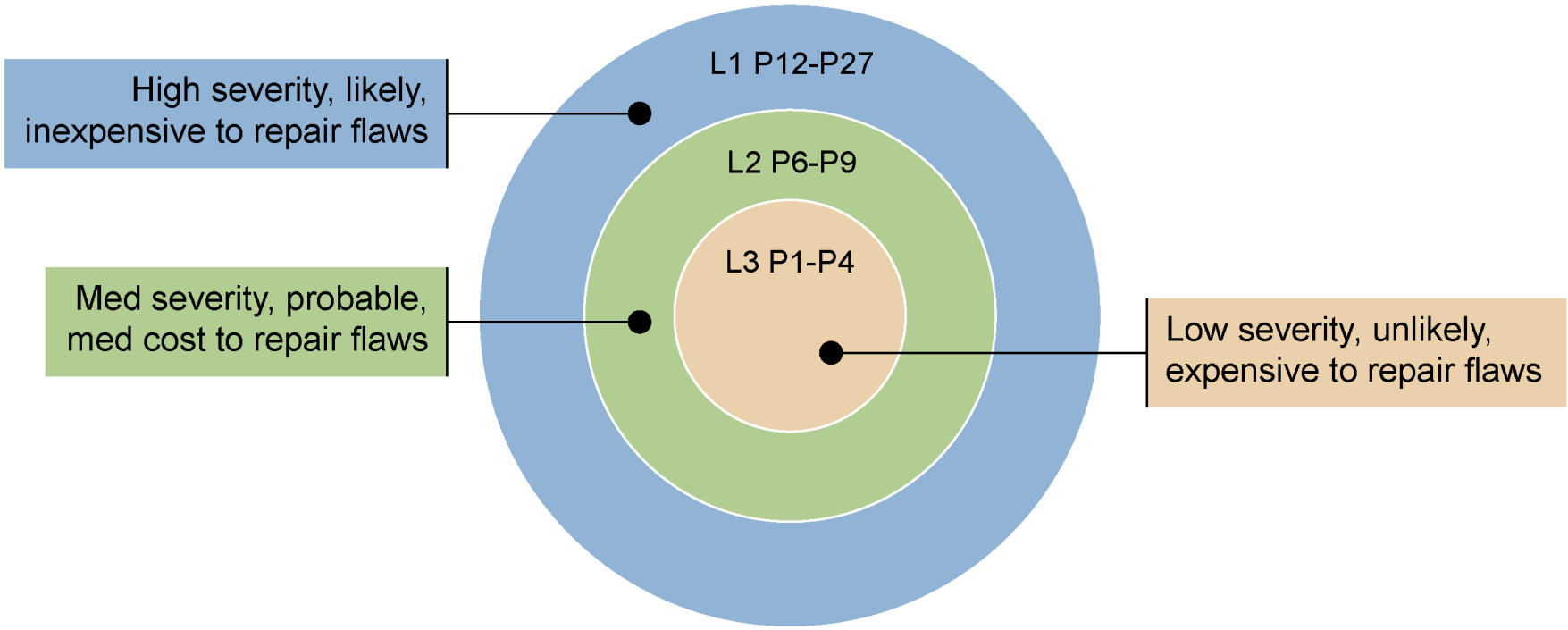
Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

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

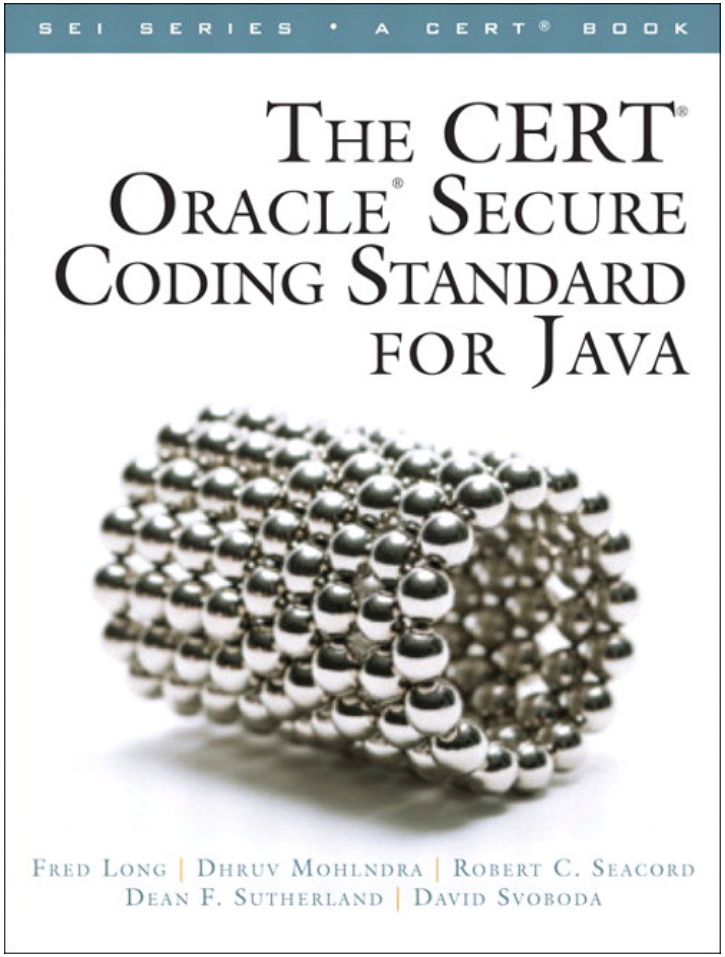


Related Guidelines (ENV04-C)

ENV04-C. Do not call `system()` if you do not need a command processor

CERT C++ Secure Coding Standard	ENV04-CPP. Do not call system() if you do not need a command processor
CERT Oracle Secure Coding Standard for Java	IDS07-J. Do not pass untrusted, unsanitized data to the Runtime.exec() method
ISO/IEC TR 24772:2013	Unquoted Search Path or Element [XZQ]
ISO/IEC TR 17961 (Draft)	Calling system [syscall]
MITRE CWE	CWE-78, Failure to sanitize data into an OS command (aka "OS command injection") CWE-88, Argument injection or modification

Secure Coding Standard for Java



"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 (Java SE 6) Platform environment and includes rules for secure coding using the Java programming language and libraries.

The Java Language Specification, third 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.

CERT Perl Secure Coding Standard

Provides a core of well-documented and enforceable coding rules and recommendations for Perl

Developed specifically for versions 5.12 and later of the Perl programming language

Contains just over 30 guidelines in eight sections:

- Input Validation and Data Sanitization
- Declarations and Initialization
- Expressions
- Integers
- Strings
- Object-Oriented Programming (OOP)
- File Input and Output
- Miscellaneous

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

International Standards

Secure Coding Training

Secure Coding Research

Source Code Analysis Laboratory

Source Code Analysis Laboratory (SCALe)

- Consists of commercial, open source, and experimental analysis
- Is used to analyze various code bases including those from the DoD, energy delivery systems, medical devices, and more
- Provides value to the customer but is also being instrumented to research the effectiveness of coding rules and analysis

SCALe customer-focused process:

1. Customer submits source code to CERT for analysis.
2. Source is analyzed in SCALe using various analyzers.
3. Results are analyzed, validated, and summarized.
4. Detailed report of findings is provided to guide repairs.
5. The developer addresses violations and resubmits repaired code.
6. The code is reassessed to ensure all violations have been properly mitigated.
7. The certification for the product version is published in a registry of certified systems.

Government Demand

SEC. 933 of the National Defense Authorization Act for Fiscal Year 2013 requires evidence that government software development and maintenance organizations and contractors are conforming in computer software coding to approved secure coding standards of the Department during software development, upgrade, and maintenance activities, including through the use of inspection and appraisals.

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

- is being specified in the 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 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 1

Developers of software that has been determined by CERT to conform to a secure coding standard may use the CERT SCALe seal 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.

CERT SCALe Seal 2

Except for patches that meet the following criteria, any modification of software after it is designated as conforming voids the conformance designation. Until such software is retested and determined to be conforming, the new software cannot be associated with the CERT SCALe seal.

Patches that meet all three of the following criteria do not void the conformance designation:

- The patch is necessary to fix a vulnerability in the code or is necessary for the maintenance of the software.
- The patch does not introduce new features or functionality.
- The patch does not introduce a violation of any of the rules in the secure coding standard to which the software has been determined to conform.

Conformance Certificates

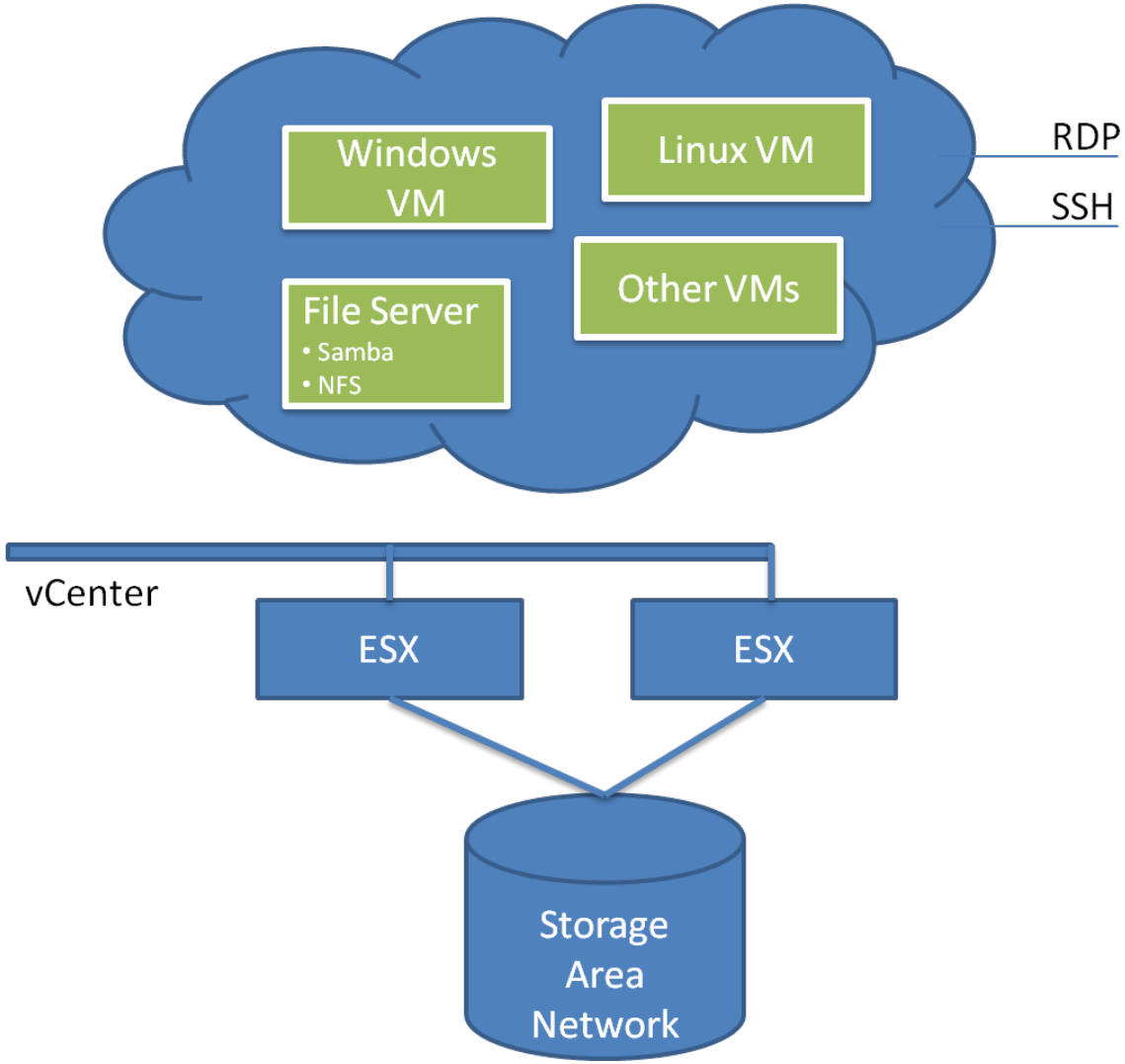
Certificates contain the name and version of the software system that passed the conformance test and the results of the test.

The process is similar to that followed by The Open Group (see http://www.opengroup.org/collaboration-services/certification.html).

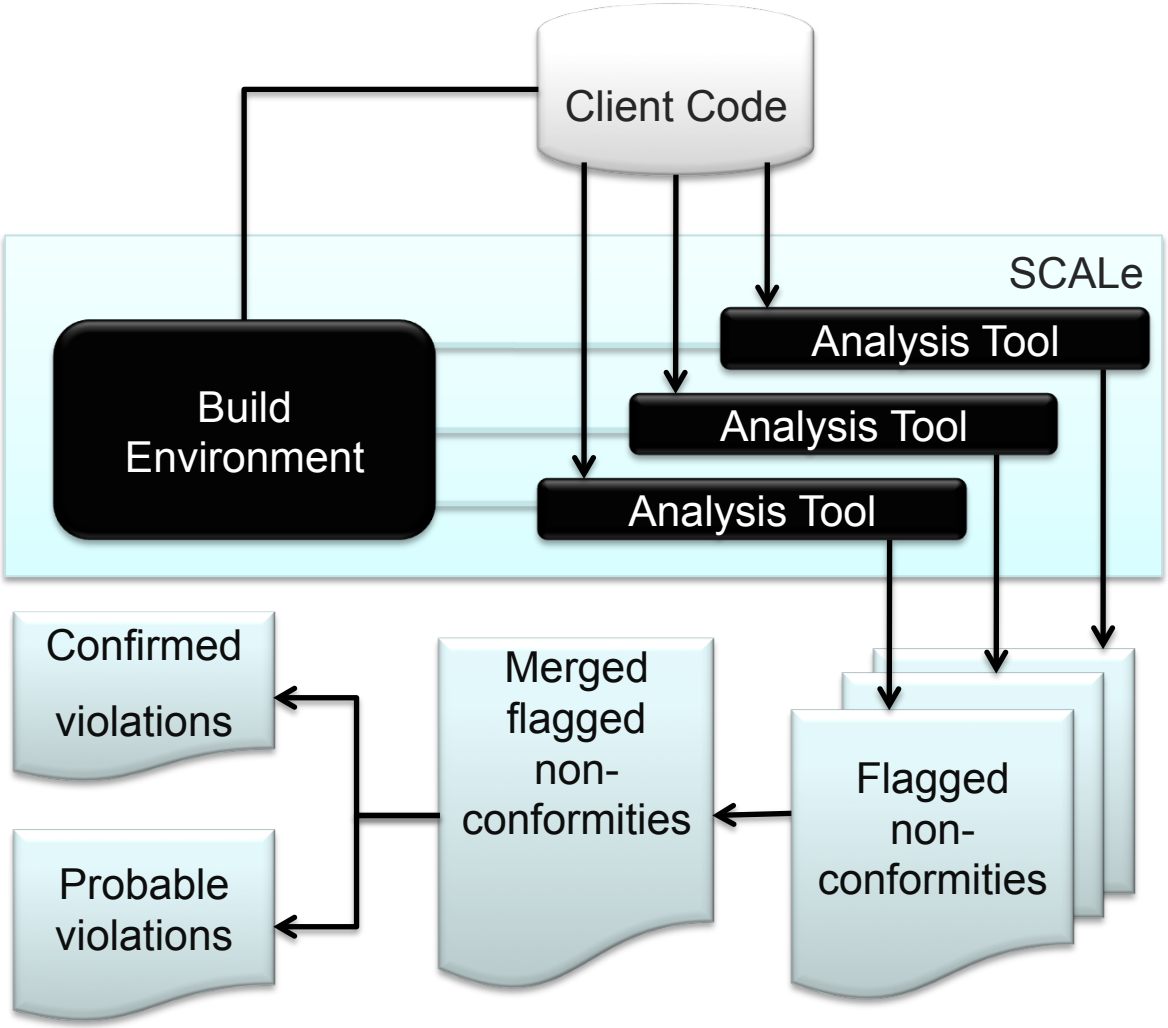
Initially, all assessments are performed by CERT.

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

Source Code Analysis Laboratory



Conformance Testing Process



Conformance Testing

The use of secure coding standards defines a proscriptive set of rules and recommendations by 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	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.

Static Analysis

Most SCALe analysis is performed by static analyzers.

- In general, determining conformance to coding rules is computationally undecidable.
- It may be impossible for *any* tool to determine statically whether a given rule is satisfied in specific circumstances.

Static Analysis Limitations

False negatives

- Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security.
- Most tools err on the side of caution and consequently generate false positives.

		False positives	
		Y	N
False negatives	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Unsound

- However, in some cases, it may be deemed better to report some high-risk flaws and miss others than to overwhelm the user with false positives.

False positives

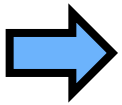
- The tool reports a flaw when one does not exist.
- False positives may occur because the code is sufficiently complex that the tool cannot perform a complete analysis.

Lot Tolerance Percent Defective

Lot Tolerance Percent Defective (LTPD) single sampling

Within a given bucket, there is 90% confidence that the bucket of flagged nonconformities for a given analyzer checker contains no more than 2% true positives, where 2% true positives is the previously determined Nominal Quality Level (LQ)

Bucket Size (# of flagged nonconformities for a given analyzer checker)	Sample Size for Nominal Limiting Quality in Percent (LQ) of 2%
16 to 25	100% sampled
25 to 50	100% sampled
51 to 90	50
91 to 150	80
151 to 280	95
281 to 500	105
501 to 1,200	125
1,201 to 3,200	200[a]
3,201 to 10,000	200[a]



False Positive Rate	Flagged Anomalies									
	1	2	3	4	5	6	7	8	9	10
0%	T	x	x	x	x	x	x	x	x	x
66%	F	F	T	x	x	x	x	x	x	x
87%	F	F	F	F	F	F	F	T	x	x

[a] at this LQ value and bucket size, the sampling plan would allow one observed true positive in the sample investigated, but the SCALe analyst would continue using the zero observed true positive rule to decide if the bucket is acceptable or not.

Deviation Procedure 1

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

Deviations can be used in cases in which 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.

Deviation Procedure 2

Deviations cannot be used for reasons of performance or usability or to achieve other nonsecurity attributes in the system.

A software system that successfully passes conformance testing must not present known vulnerabilities resulting from coding errors.

Deviation requests are evaluated by the lead assessor; if the developer can provide sufficient evidence that deviation does not introduce a vulnerability, the deviation request is accepted.

Deviations should be used infrequently because it is almost always easier to fix a coding error than to prove that the coding error does not result in a vulnerability.

Once the evaluation process is completed, a report detailing the conformance or nonconformance of the code to the corresponding rules in the secure coding standard is provided to the developer.

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

International Standards

Secure Coding Training

Secure Coding Research

Standard Development Organizations

ISO/IEC JTC1/SC22/WG14 is the international standardization working group for the programming language C.

INCITS Technical Committee PL22.11 is the

- U.S. organization responsible for the C programming language standard.
- U.S. TAG to ISO/IEC JTC 1 SC22/WG14 and provides recommendations on U.S. positions to the JTC 1 TAG.

ISO/IEC JTC1/SC22/WG21 is the international standardization working group for the programming language C++.

INCITS Technical Committee PL22.16 is the

- U.S. organization responsible for the C++ programming language standard.
- U.S. TAG to ISO/IEC JTC 1 SC22/WG21 and provides recommendations on U.S. positions to the JTC 1 TAG.

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/WG14 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 that had been introduced by ISO/IEC 9899:1999.

C Secure Coding Guidelines SG

WG14 established a study group to study the problem of producing analyzable secure coding guidelines for the C language.

- First meeting was held on October 27, 2009.
- Participants included analyzer vendors, security experts, language experts, and consumers.
- New work item approved March 2012; study group concluded.

WG14 produced ISO/IEC TS 17961 Draft. *Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules*, 2012.

ISO/IEC TS 17961

Applies to analyzers, including static analysis tools and C language compilers that wish to diagnose insecure code beyond the requirements of the language standard.

Enumerates secure coding rules and requires analysis engines to diagnose violations of these rules as a matter of conformance to this specification. These rules may be extended in an implementation-dependent manner, which provides a minimum coverage guarantee to customers of any and all conforming static analysis implementations.

The Preliminary Draft Technical Specification (PDTS) ballot reviewed at the Delft WG14 meeting, April 23–26, 2013.

- Ballot results
  - 12 National Bodies (NB) Approval as presented
  - 1 NB Approval with comments
  - 1 NB Disapproval of the draft
  - 5 NB Abstention
- Plan is for one more ballot round after a small editorial committee meets

Secure Coding Validation Suite

A set of tests to validate the rules defined in TS 17961,
these tests are based on the examples in this technical specification.

https://github.com/SEI-CERT/scvs

Distributed with a BSD-style license.

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

International Standards

Secure Coding Training

Secure Coding Research

SEI Secure Coding in C/C++ Training 1

The Secure Coding course is designed for C and C++ developers. It encourages programmers to adopt security best practices and develop a security mindset that can help protect software from tomorrow's attacks, not just today's.

Topics

- String management
- Dynamic memory management
- Integral security
- Formatted output
- File I/O

http://www.sei.cmu.edu/training/p63.cfm

SEI Secure Coding in C/C++ Training 2

Participants gain a working knowledge of common programming errors that lead to software vulnerabilities, how these errors can be exploited, and mitigation strategies to prevent their introduction.

Objectives

- Improve the overall security of any C or C++ application.
- Thwart buffer overflows and stack-smashing attacks that exploit insecure string manipulation logic.
- Avoid vulnerabilities and security flaws resulting from incorrect use of dynamic memory management functions.
- Eliminate integer-related problems: integer overflows, sign errors, and truncation errors.
- Correctly use formatted output functions without introducing format-string vulnerabilities.
- Avoid I/O vulnerabilities, including race conditions.

Online Secure Coding Course

Developed Integer Security course module "prototype" sponsored by the Department of Homeland Security

Completed a Strings module sponsored by Cisco

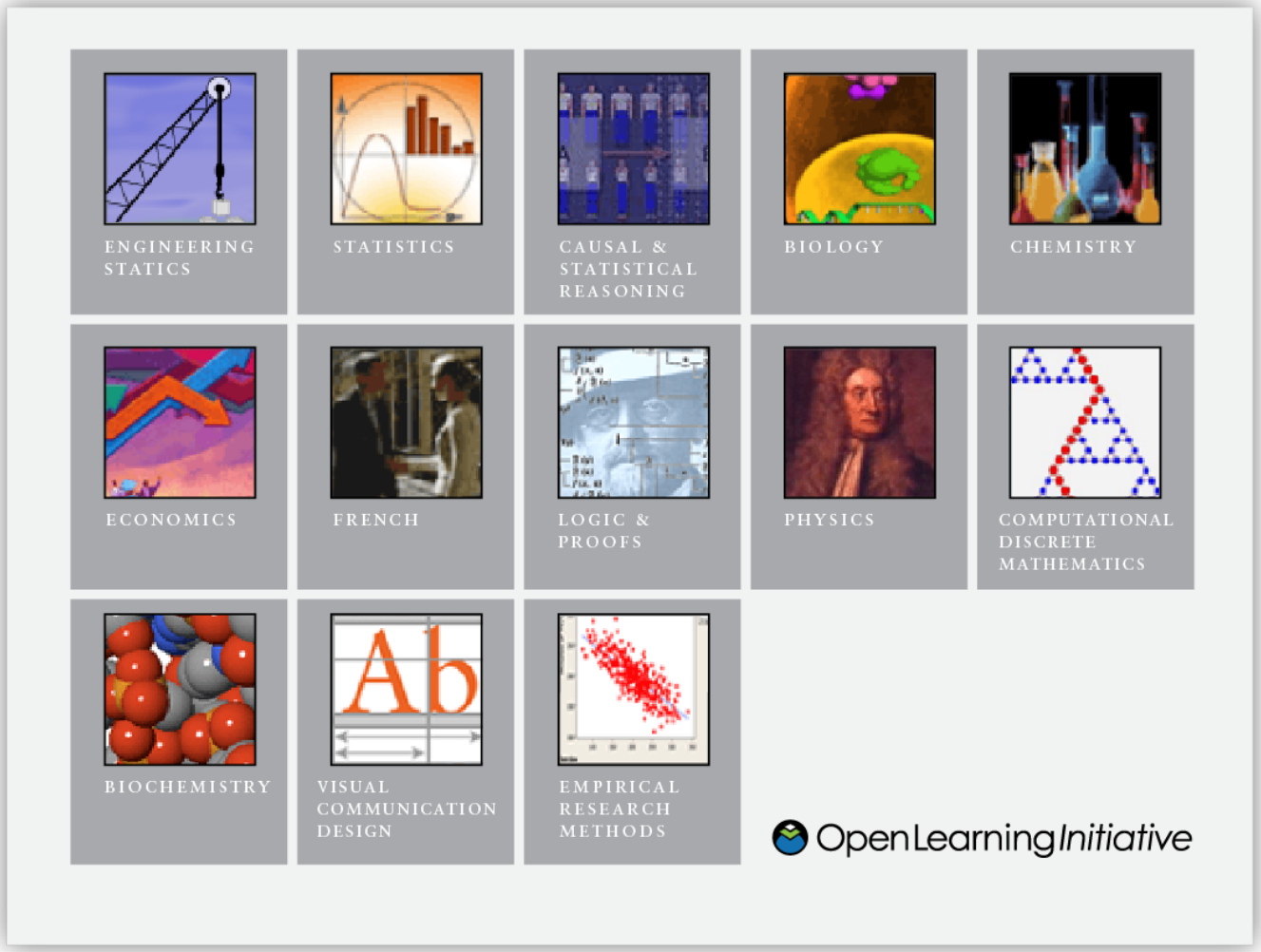
Dynamic Memory module developed with Siemens going into production

Concurrency module sponsored by the Department of Energy in development

Developed in collaboration with CMU's Open Learning Initiative

What Is CMU's Open Learning Initiative?

A grant-funded group offering innovative, scientifically based online learning environments designed to improve both quality and productivity in higher education



Secure Coding Course: Objectives 1

Strings

- Recognize the different string types in C and C++ language programs.
- Select the appropriate byte character types for a given purpose.
- Identify common string manipulation errors.
- Explain how vulnerabilities from common string manipulation errors can be exploited.
- Identify applicable mitigation strategies, evaluate candidate mitigation strategies, and select the most appropriate mitigation strategy (or strategies) for a given context.
- Apply mitigation strategies to reduce the introduction of errors into new code or repair security flaws in existing code.

Integer Security

- Explain and predict how integer values are represented for a given implementation.
- Predict how and when conversions are performed and describe their pitfalls.
- Select appropriate type for a given situation.
- Programmatically detect erroneous conditions for assignment, addition, subtraction, multiplication, division, and left and right shift.
- Recognize when implicit conversions and truncation occur as a result of assignment.
- Apply mitigation strategies to reduce introduction of errors into new code or repair security flaws in existing code.

Secure Coding Course: Objectives 2

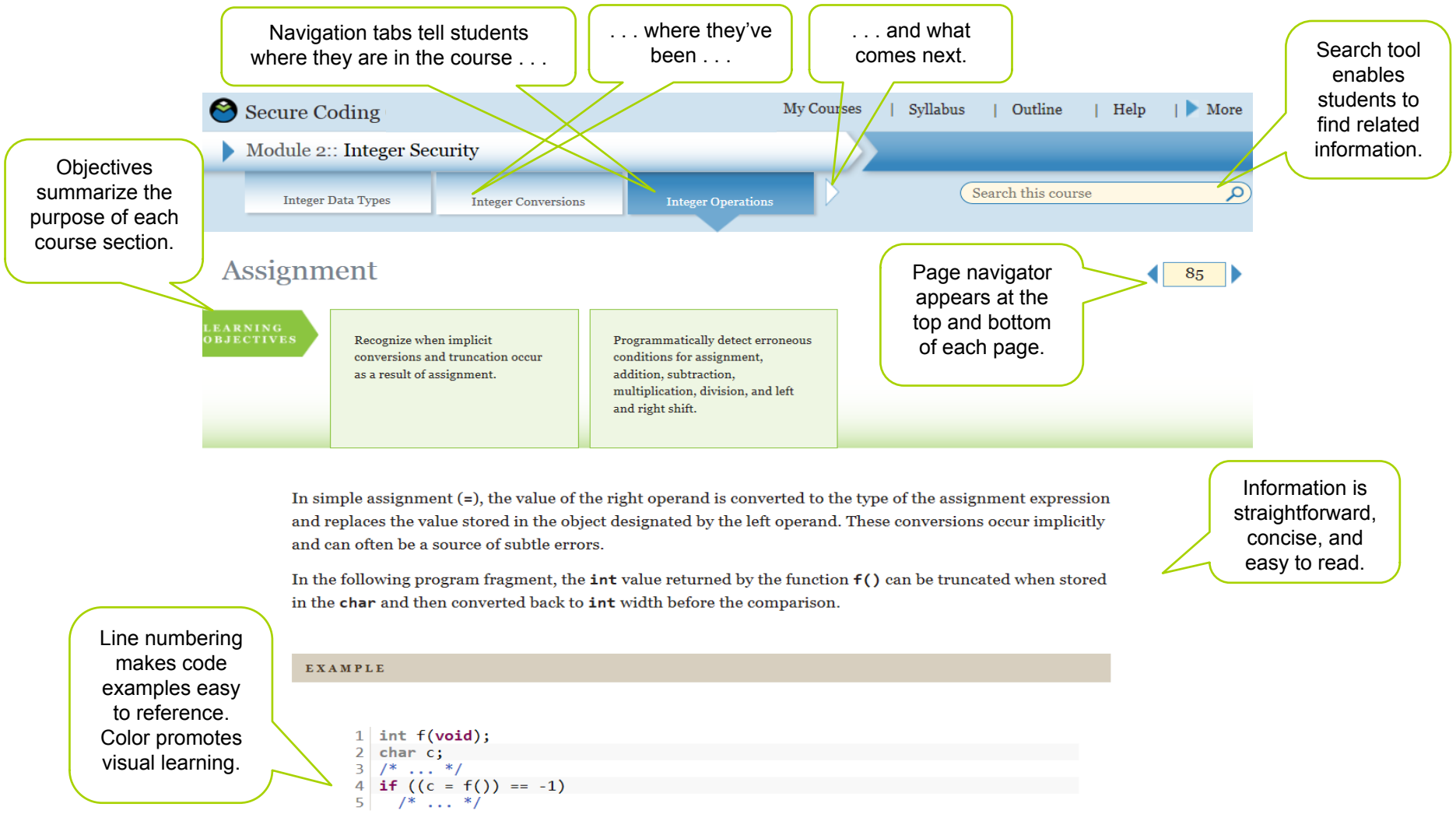
Dynamic Memory

- Use standard C memory management functions securely.
- Align memory suitably.
- Explain how vulnerabilities from common dynamic memory management errors can be exploited.
- Identify common dynamic memory management errors.
- Perform C++ memory management securely.
- Identify common C++ programming errors when performing dynamic memory allocation and deallocation.
- Identify common dynamic memory management errors.

Concurrency

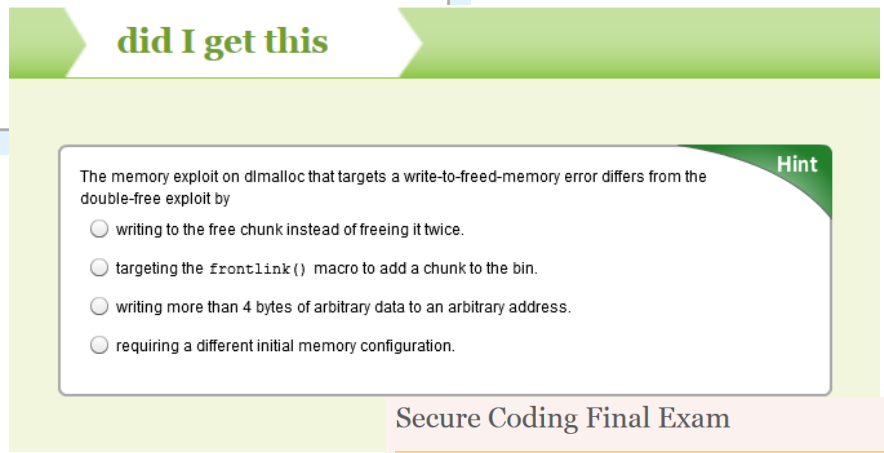
- Define concurrency and it's relationship with multithreading and parallelism.
- Calculate the potential performance benefits of parallelism in specific instances.
- Identify common errors in concurrency implementations.
- Identify common errors and attack vectors C++ concurrency programming.
- Apply common approaches for mitigating risks in C++ concurrency programming.
- Describe common vulnerabilities that occur from the incorrect use of

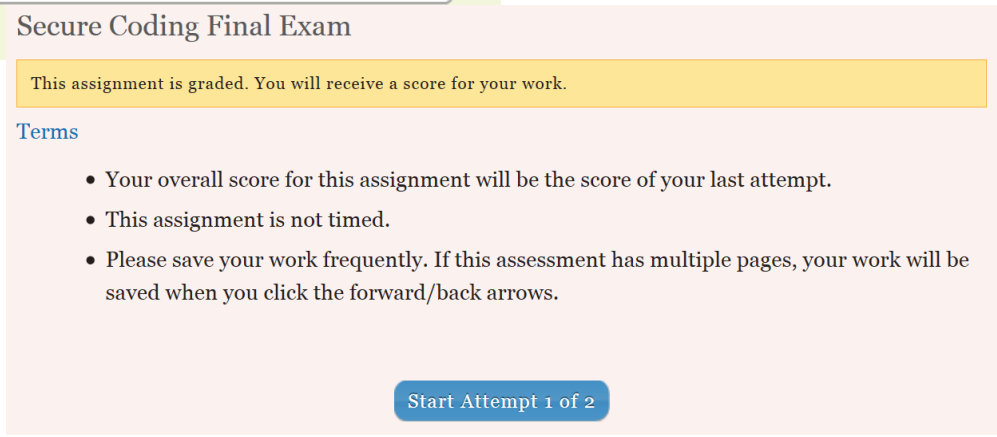
Secure Coding Course Interface



Secure Coding Online Assessments

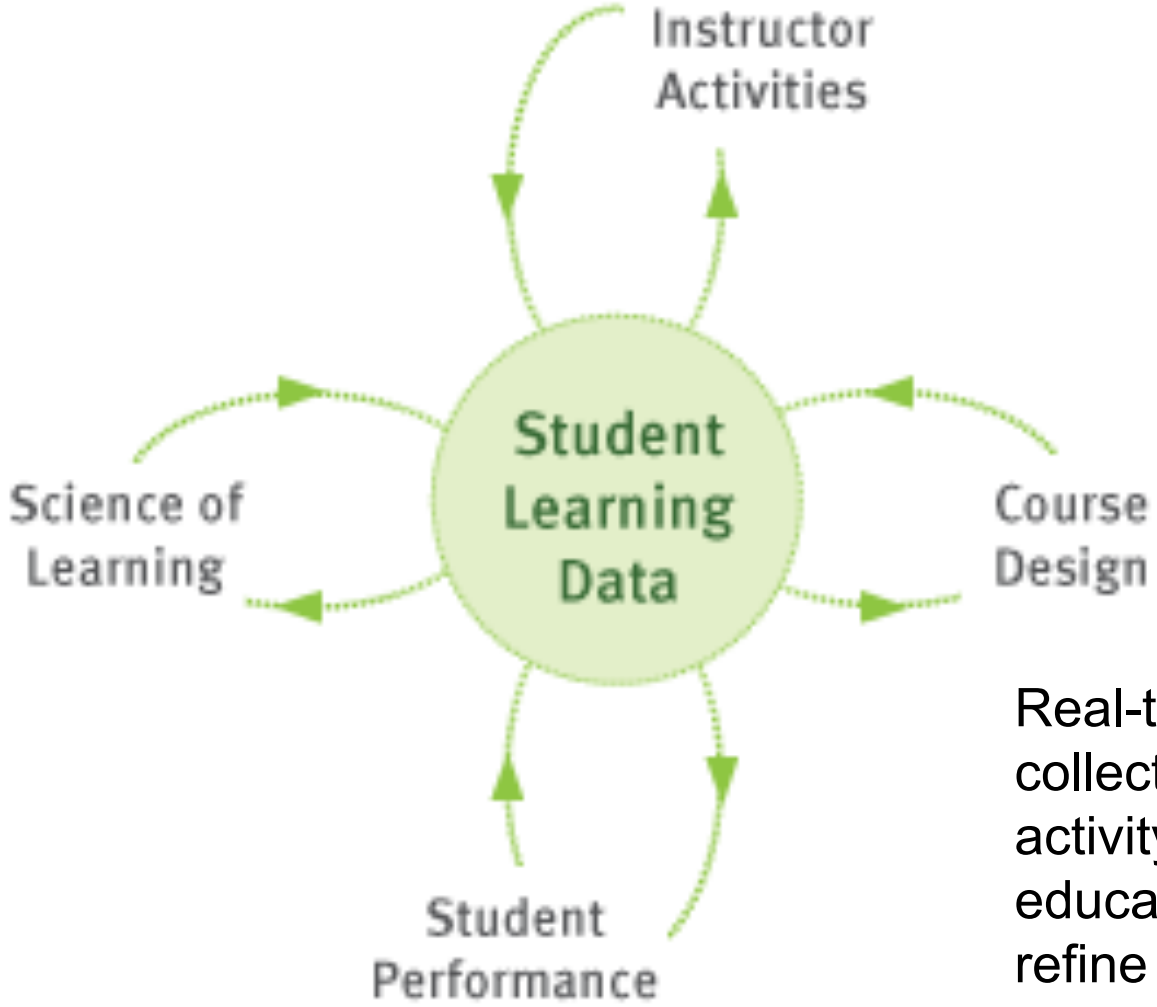
Learn by Doing and Did I Get This? activities reinforce information and help students check their progress.





Each module ends with a graded final exam.

Feedback Loops



Real-time data collection of student activity enables educators to iteratively refine their courses

Agenda

Software Security

CERT Secure Coding Standards

Conformance Testing

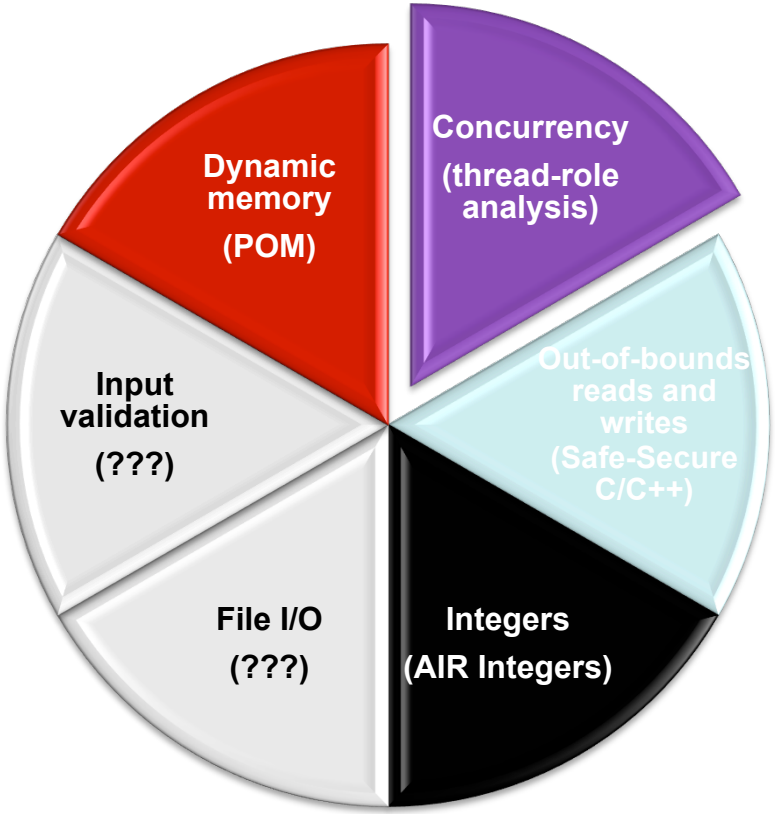
International Standards

Secure Coding Training

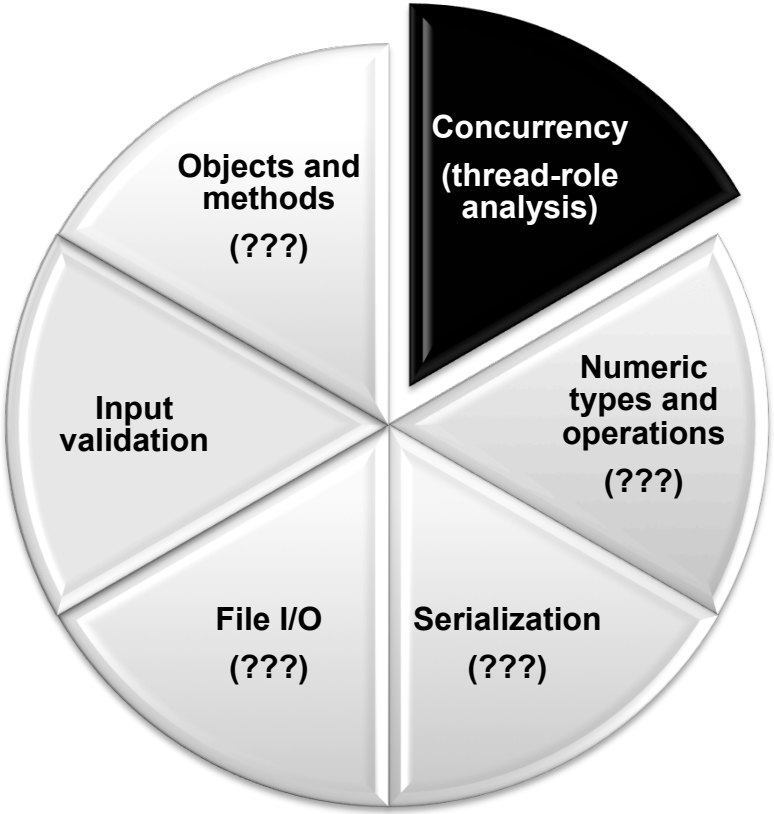
Secure Coding Research

Language Vulnerabilities

C Language (C11)

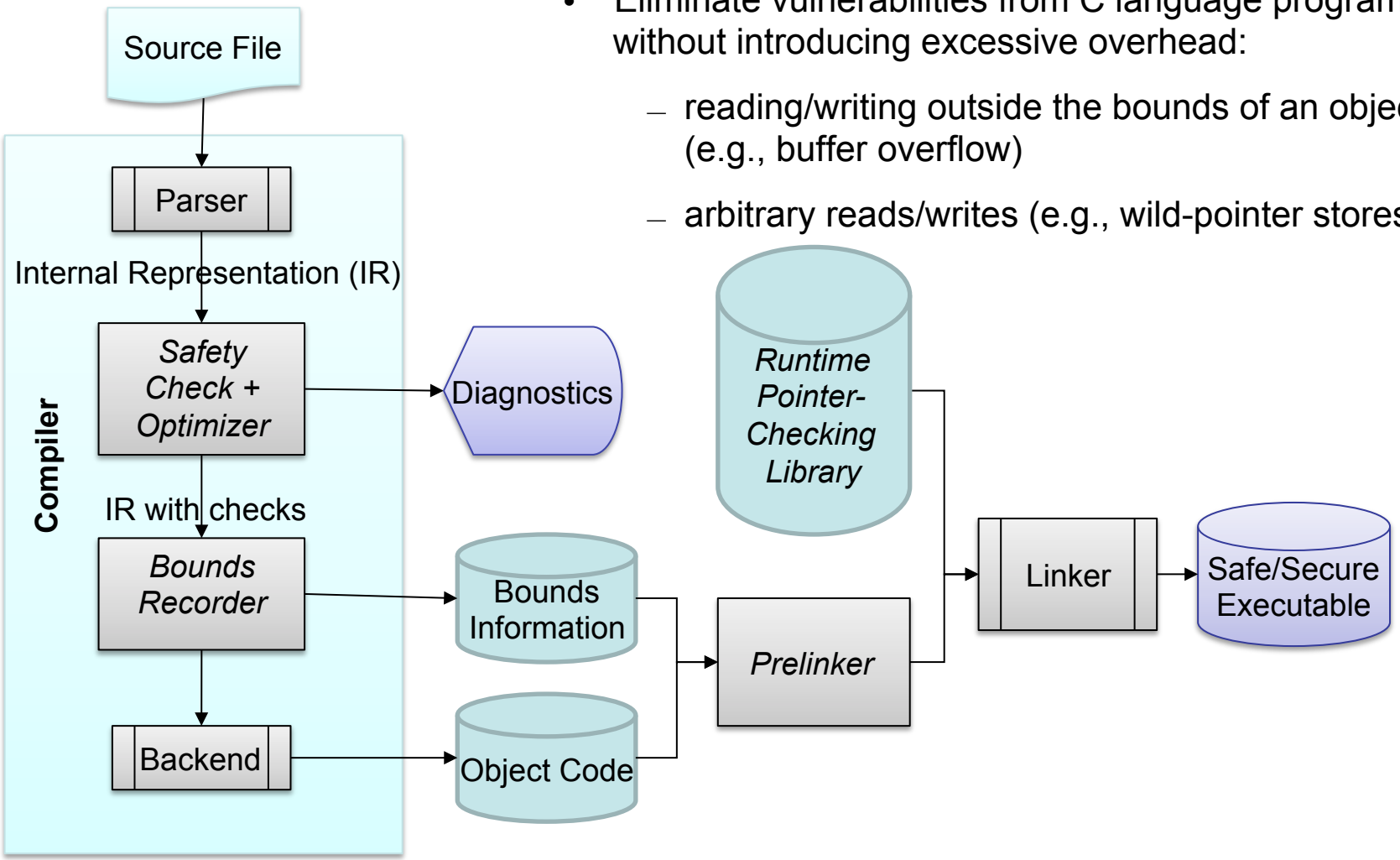


Java



Compiler-Enforced Buffer Overflow Elimination

- Eliminate vulnerabilities from C language programs without introducing excessive overhead:
  - reading/writing outside the bounds of an object (e.g., buffer overflow)
  - arbitrary reads/writes (e.g., wild-pointer stores)



Compiler-Enforced Buffer Overflow Elimination

```
for (size_t i = 0; i < 100; ++i)
    a[i] = i;  // possible BO
```

Eliminates 99/100 bounds checks

```
  %bitcast = bitcast i32* %a to i8*
  tail call void @__softboundcets_spatial_store_dereference_check(
    i8* %0, i8* %1, i8* %bitcast, i64 400) nounwind
  br label %for.body
```

Hoisted bounds check to before loop

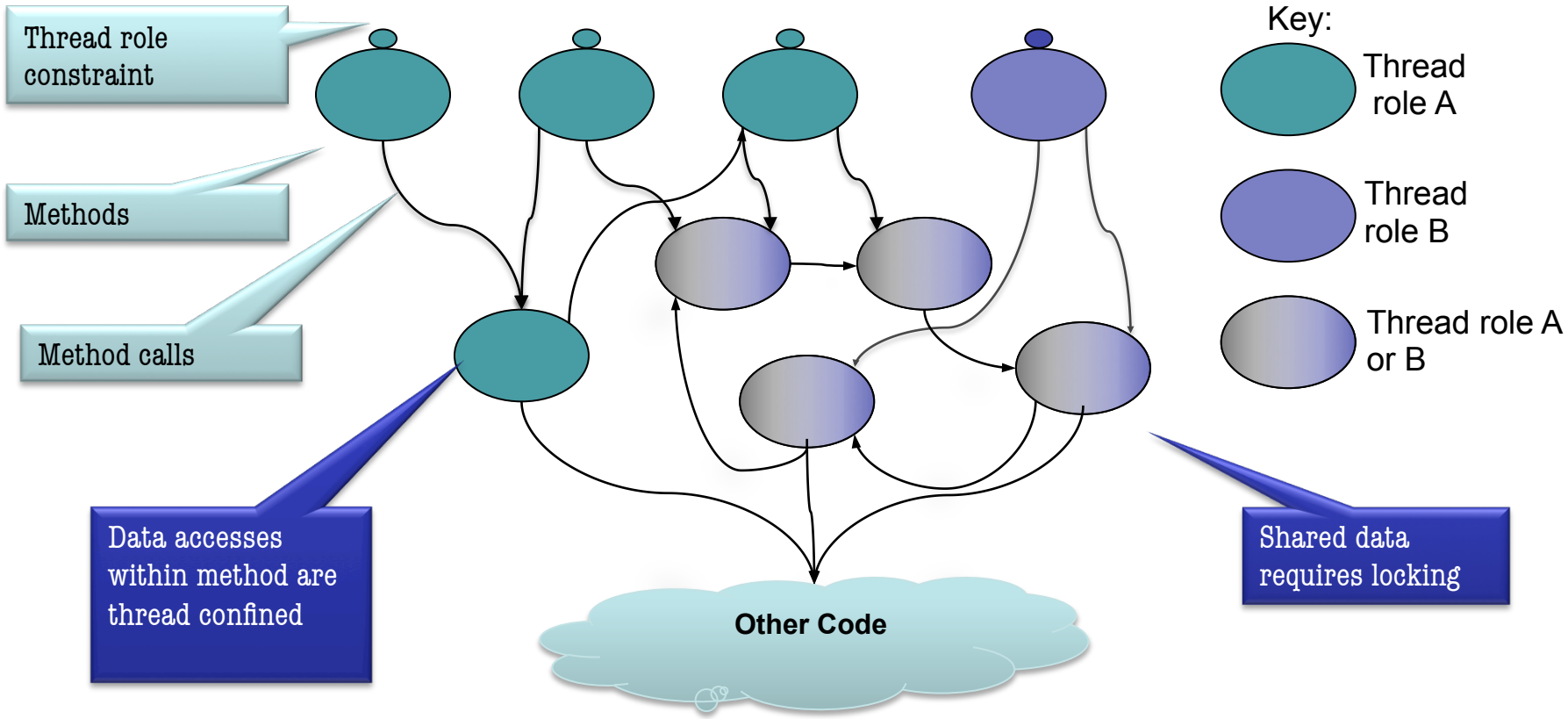
Start of for loop

```
for.body:                                         ; preds = %for.body, %entry
  %i.04 = phi i64 [ 0, %entry ], [ %inc, %for.body ] ; merge i from entry
points
  %conv = trunc i64 %i.04 to i32  ; convert i to 32 bits
  %arrayidx = getelementptr inbounds i32* %a, i64 %i.04  ; get pointer to a[i]
  tail call void @__softboundcets_spatial_store_dereference_check(
    i8* %0, i8* %1, i8* %bitcast, i64 4) nounwind
  store i32 %conv, i32* %arrayidx, align 4, !tbaa !0
  %inc = add i64 %i.04, 1  ; increment loop counter
  %exitcond = icmp eq i64 %inc, 100  ; check loop termination
   br i1 %exitcond, label %for.end, label %for.body
```

Provably inbounds write.

C11 Thread-Role Analysis

Thread-role analysis for C11. Create a proof-of-concept implementation of thread-role analysis for C11 to mitigate against vulnerabilities arising from concurrency errors such as state corruption and deadlock.



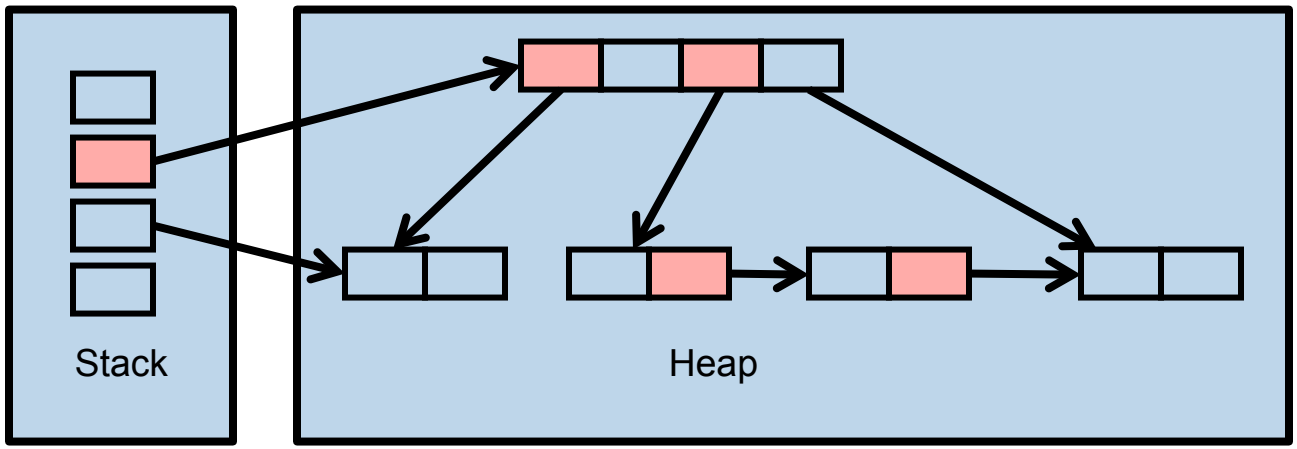
Pointer Ownership Model

A responsible pointer is a pointer that is responsible for freeing its pointed-to object. Only one pointer may be responsible for an object.

Responsible pointers form trees of heap objects with the tree roots living outside the heap.

Irresponsible pointers can point anywhere but can't free anything.

Ownership of an object can be transferred between responsible pointers, but one of the pointers must relinquish responsibility for the object.



Use of Responsible Pointers

```
void usage(char* msg) {
  fprintf(stderr, msg);
  free(msg);
}
int main(int argc, char** argv) {
  char* errmsg;
  if (argc > 2) {
    errmsg = malloc(100);
    if (errmsg != NULL) {
      snprintf(errmsg, 100, "Need more than %d arguments!", argc);
      usage(errmsg);
      free(errmsg);
      exit(1);
    }
  }
  // ...
}
```

`msg` is RESPONSIBLE

`usage()` consumes `msg`, *`msg` must be GOOD*

`errmsg` is RESPONSIBLE and UNINIT

`errmsg` becomes GOOD (or NULL)

`errmsg` can only be GOOD here.

`errmsg` consumed by `usage()`, becomes ZOMBIE

Oops, tried to consume a ZOMBIE!

For More Information

Visit CERT® websites:

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