Secure Coding Overview

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Copyright 2017 Carnegie Mellon University. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL 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.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM17-0546

Agenda

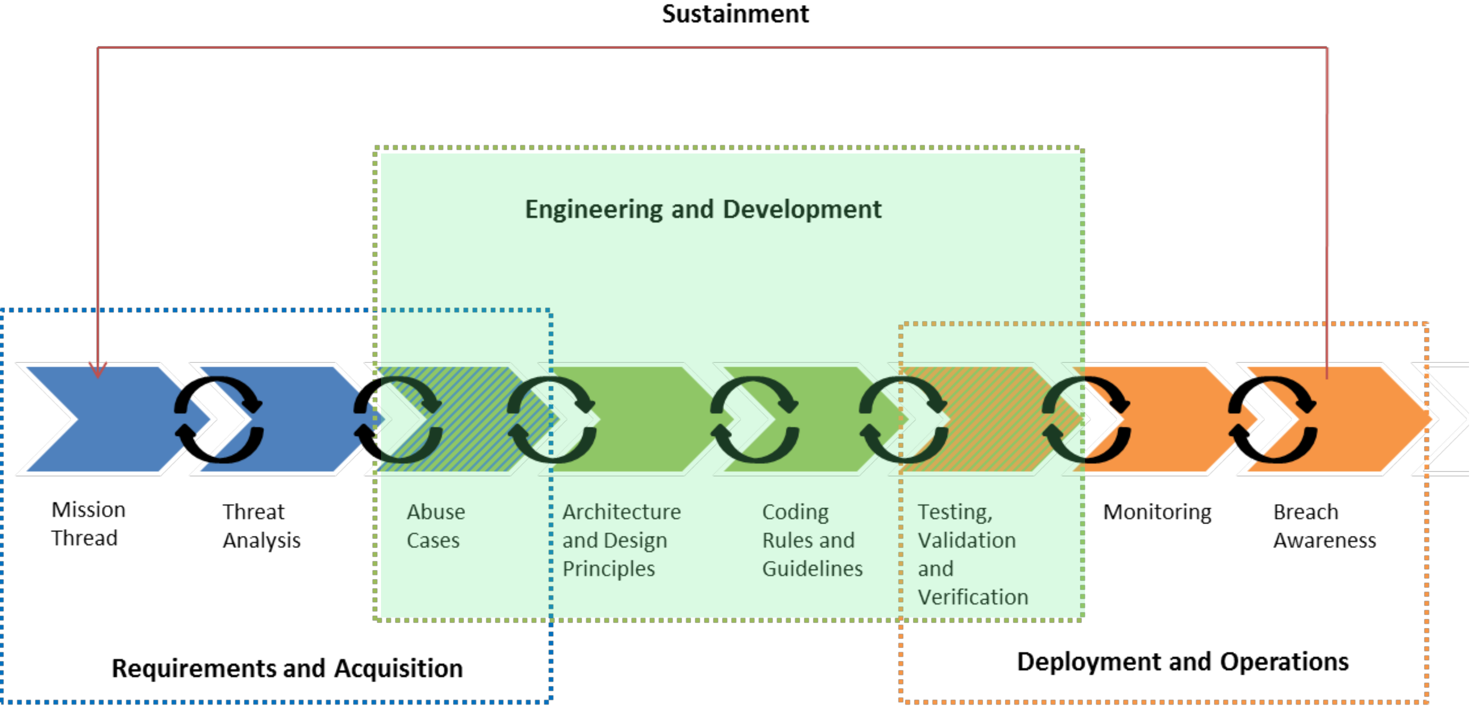
CERT Secure Coding Overview

- Secure Coding Guidelines
- SCALe Audits and Software
- Training
- International Standards
- Current Research

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Engineering and Development



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Most Vulnerabilities Are Caused by Programming Errors

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

- 51% of those were due to classic errors like buffer overflows, cross-site scripting, injection flaws

Top vulnerabilities include

- Integer overflow
- Buffer overflow
- Missing authentication
- Missing or incorrect authorization
- Reliance on untrusted inputs (aka tainted inputs)

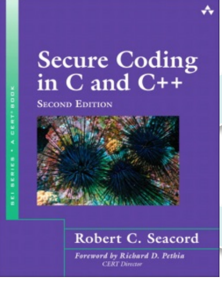
Sources: Heffley/Meunier: Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?
cwe.mitre.org/top25 Jan 6, 2015

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding History

Goal: Reduce number of code vulnerabilities before code gets to operational environments

Secure design patterns



University courses

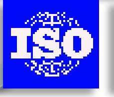
- CMU
- Stevens Institute
- Purdue
- University of Florida
- Santa Clara University
- St. John Fisher College

SEI Secure Coding courses

Online course & Professional Certificate

Instructor-led on-site course

Influence international standards bodies



ISO/IEC TS 17961 C Secure Coding Rules

Analyzer conformance test

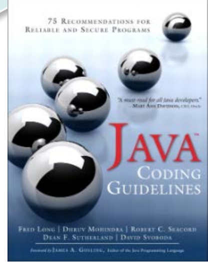
Used as input to MISRA C

Adoption by analyzer tools:

- LDRA
- Klocwork

SCALe conformance testing

- Interactive Development Environments
- Compiler-enforced Buffer Overflow Elimination
- C and C++ Thread Safety Analysis
- DidFail Android flow analysis



Adoption by software developers and acquirers:

- Cisco
- Oracle
- Various defense contractors

2003

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

CERT Secure Coding Standards

Collected wisdom from thousands of contributors on community wiki since Spring 2006

SEI CERT C Coding Standard

- Free PDF download: http://cert.org/secure-coding/products-services/secure-coding-download.cfm
- Basis for ISO TS 17961 C Secure Coding Rules

SEI CERT C++ Coding Standard

- Free PDF download (Released March 2017): http://cert.org/secure-coding/products-services/secure-coding-cpp-download-2016.cfm

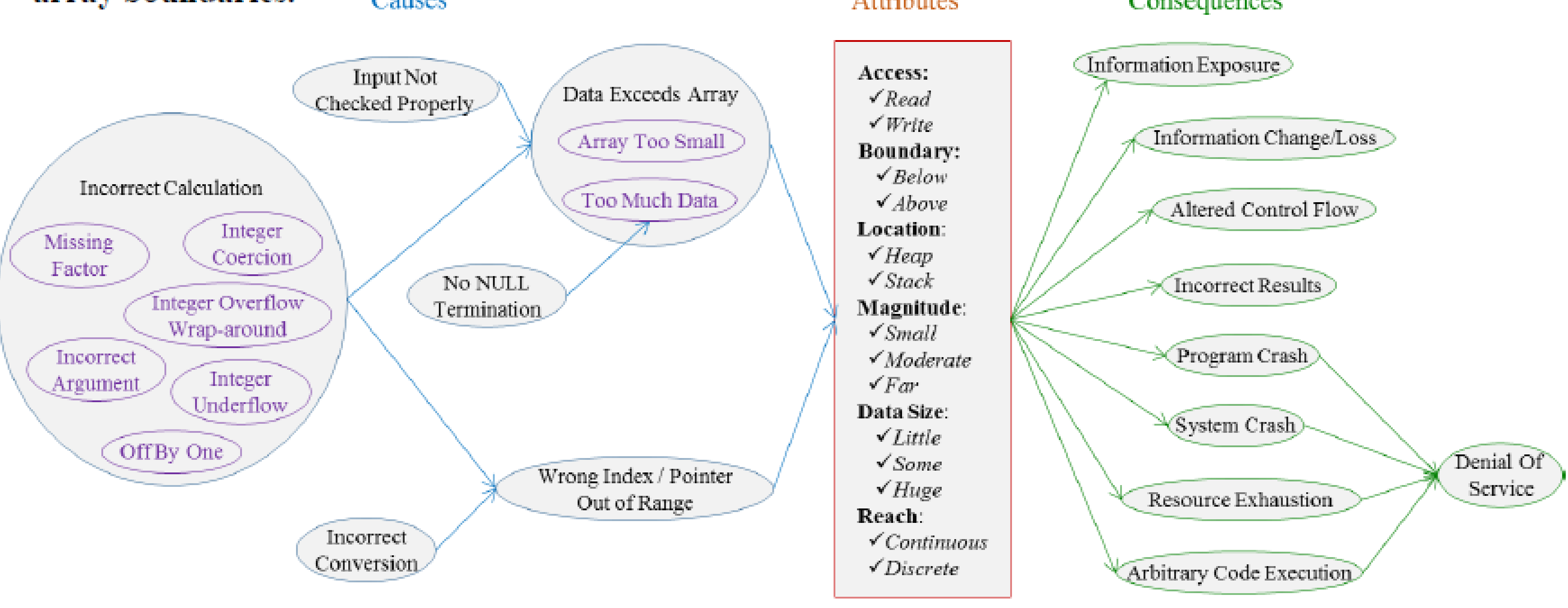
CERT Oracle Secure Coding Standard for Java

"Current" guidelines available on CERT Secure Coding wiki

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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Buffer overflow has many causes



Source: Bojanova, et al, "The Bugs Framework (BF): A Structured, Integrated Framework to Express Software Bugs",2016, http://www.mys5.org/Proceedings/2016/Posters/2016-S5-Posters_Wu.pdf

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rules and Recommendations

Rules and recommendations in the secure coding standards include

- Concise but not necessarily precise title
- Precise definition of the rule
- Noncompliant code examples or antipatterns in a pink frame—do not copy and paste into your code
- Compliant solutions in a blue frame that conform with all rules and can be reused in your code
- Risk Assessment

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rule Organization – Title & Definition

Pages / ... / Rec. 01. Declarations and Initialization (DCL)

DCL22-CPP. Functions declared with [[noreturn]] must return void

Title

Created by Aaron Ballman, last modified on Aug 24, 2016

As described in MSC55-CPP. Do not return from a function declared [[noreturn]], functions declared with the `[[noreturn]]` attribute must not return on any code path. If a function declared with the `[[noreturn]]` attribute has a non-`void` return value, it implies that the function returns a value to the caller even though it would result in undefined behavior. Therefore, functions declared with `[[noreturn]]` must also be declared as returning `void`.

Introduction & Normative Text

Concise but not necessarily precise title

Precise definition of the rule

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rule Organization – NCCE & CS

Noncompliant Code Example

In this noncompliant code example, the function declared with `[[noreturn]]` claims to return an `int`:

```
#include <cstdlib>

[[noreturn]] int f() {
  std::exit(0);
  return 0;
}
```

This example does not violate MSC55-CPP. Do not return from a function declared [[noreturn]] because `std::exit()` is declared `[[noreturn]]`, so the `return 0;` statement can never be executed.

Compliant Solution

Because the function is declared `[[noreturn]]`, and no code paths in the function allow for a return in order to comply with MSC55-CPP. Do not return from a function declared [[noreturn]], the compliant solution declares the function as returning `void` and elides the explicit `return` statement:

```
#include <cstdlib>

[[noreturn]] void f() {
  std::exit(0);
}
```

Noncompliant Code
Don't try this at home!

Noncompliant code examples or antipatterns in a pink frame—do not copy and paste into your code

Compliant Code
Fixes noncompliant code.

Compliant solutions in a blue frame that conform with all rules and can be reused in your code

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rule Organization – Risk Assessment & Detection

Risk Assessment

A function declared with a non-`void` return type and declared with the `[[noreturn]]` attribute is confusing to consumers of the function because the two declarations are conflicting. In turn, it can result in misuse of the API by the consumer or can indicate an implementation bug by the producer.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL22-CPP	Low	Unlikely	Low	**P3**	**L3**

Automated Detection

Tool	Version	Checker	Description
Clang	3.9	`-Winvalid-noreturn`	

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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?

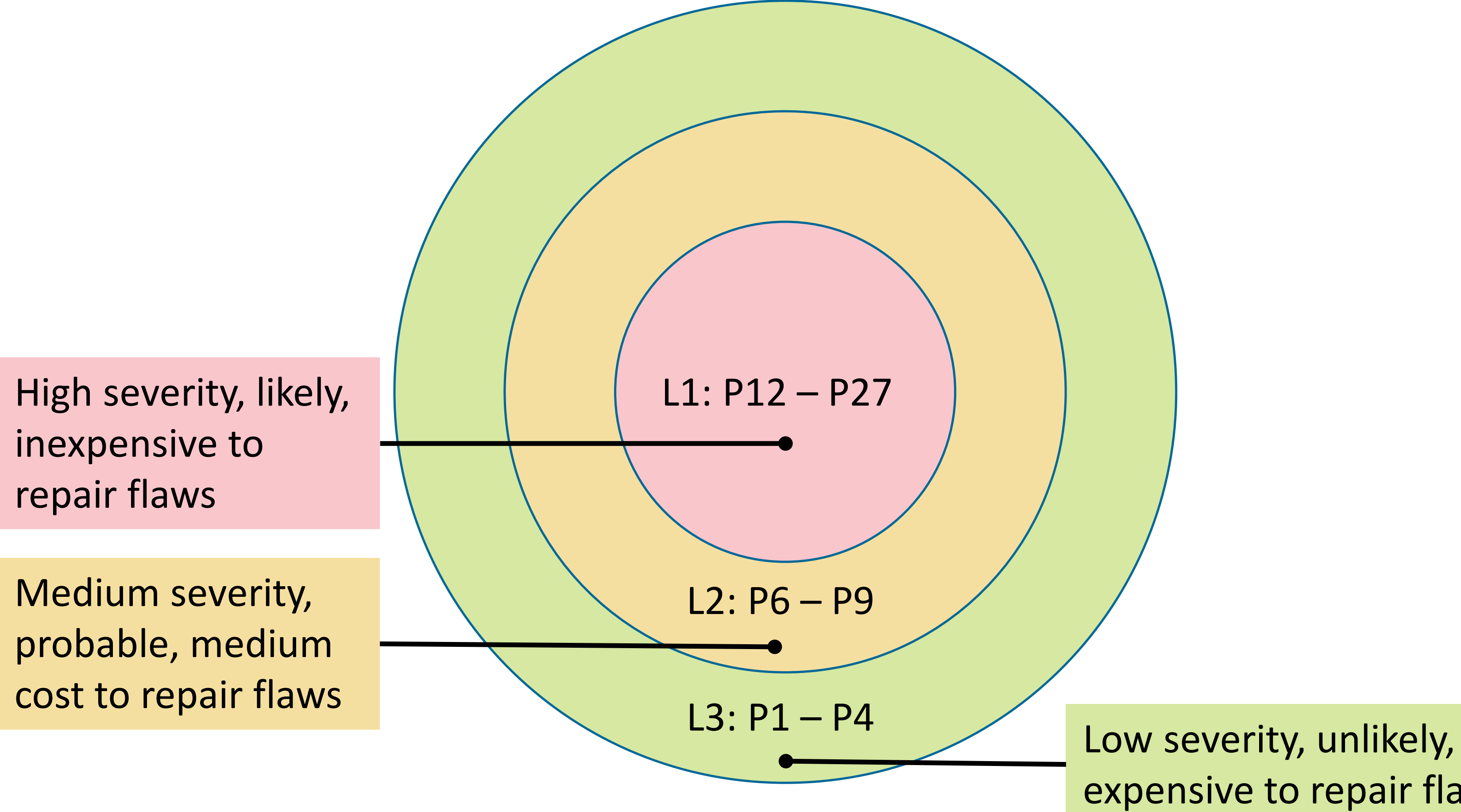
Value	Meaning
1	unlikely
2	probable
3	likely

Cost—The cost of mitigating the vulnerability.

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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Levels and Priorities



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rule Organization – Related Vulnerabilities, Guidelines & Bib

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	MSC54-CPP. Value-returning functions must return a value from all exit paths MSC55-CPP. Do not return from a function declared [[noreturn]]

Bibliography

[ISO/IEC 14882-2014]	Subclause 7.6.3, "Noreturn Attribute"

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

The C Coding Standard

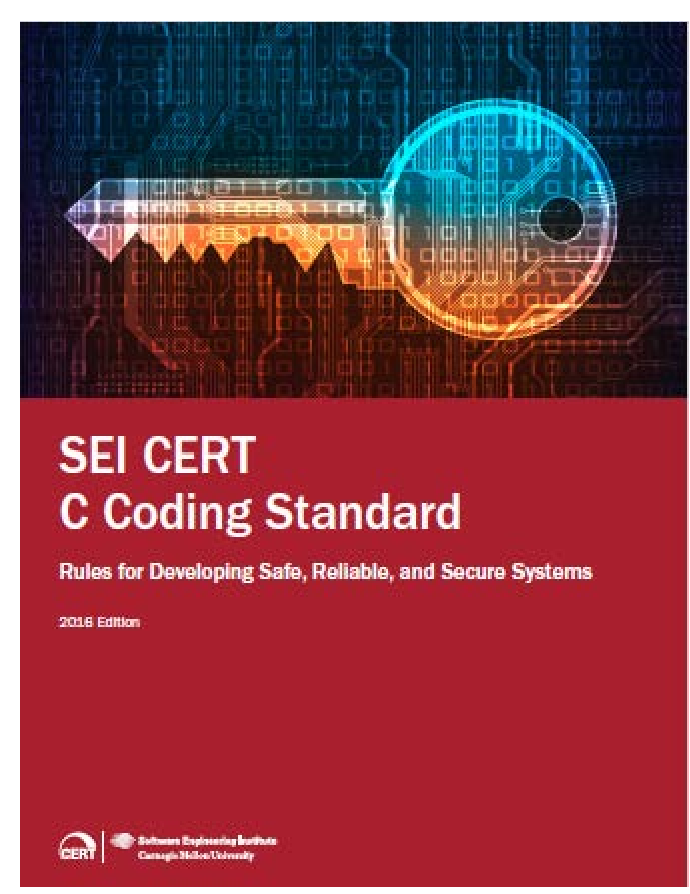
Developed with community involvement

- 1,568 registered experts on the wiki as of February 2014

Version 1.0 (C99) published by Addison-Wesley in September 2008

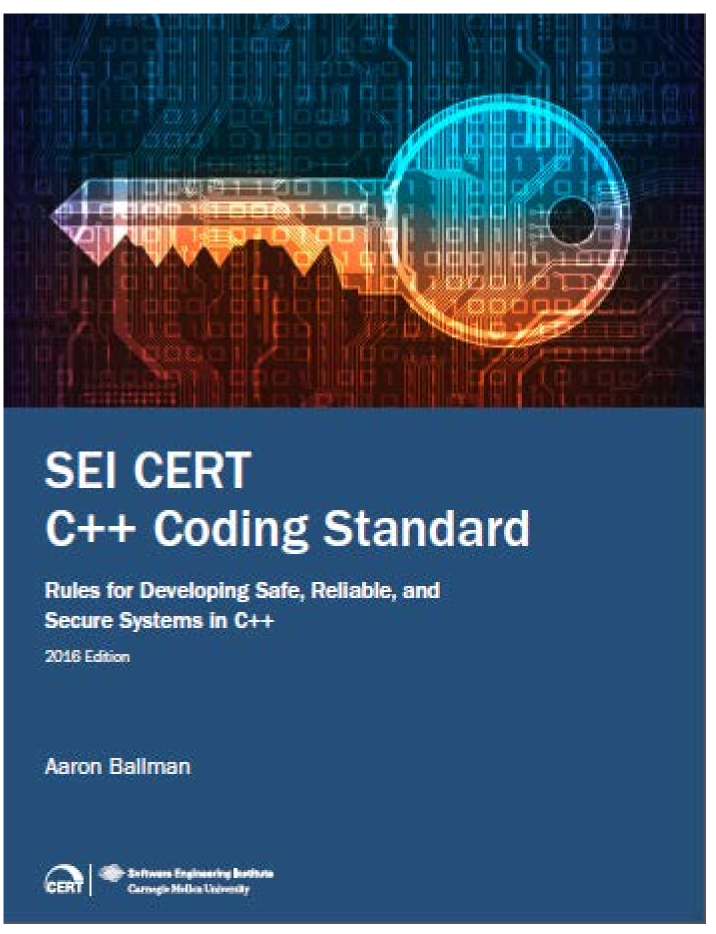
Version 2.0 was published in April 2014; extended for

- C11
- ISO/IEC TS 17961 Compatibility



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

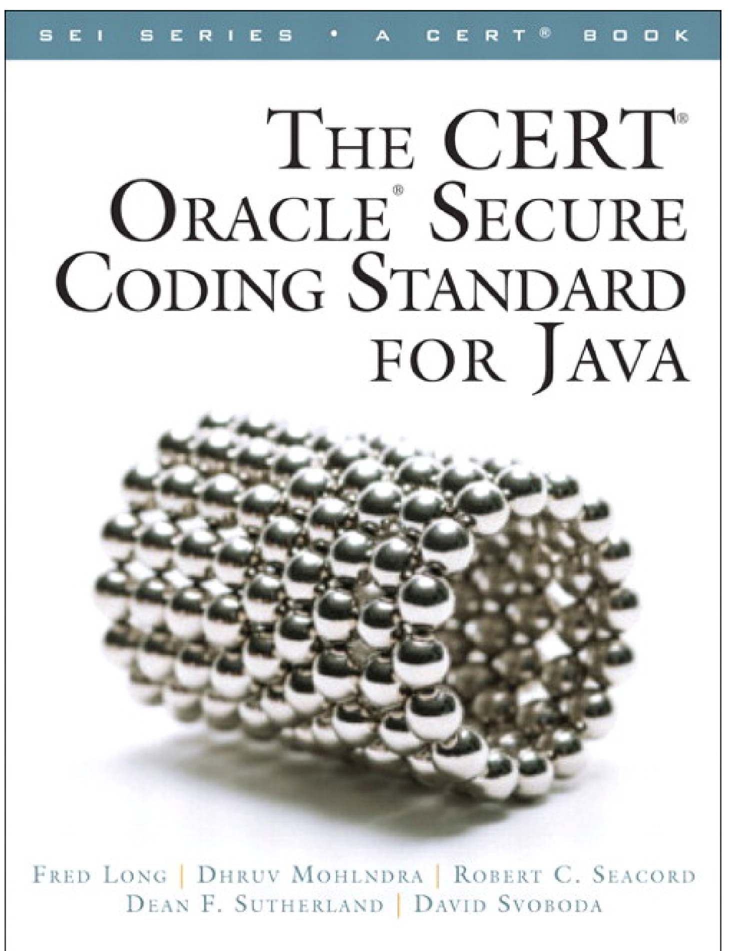
SEI CERT C++ Coding Standard



- Free PDF Download – 2016 Edition
- First published version – released March 2017
- Adds C++-unique features to the SEI CERT Coding Standards
  - 83 Rules
  - Object Oriented Programming
  - Containers
  - Most SEI CERT C Coding Standard rules also apply
- Adds features from C++14
  - Lambda objects
- Ongoing updates available at http://www.securecoding.cert.org wiki

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

The Java Coding Standard



"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 CERT Oracle™ 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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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, 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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

The Perl 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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Agenda

CERT Secure Coding Overview

- Secure Coding Guidelines
- SCALe Audits and Software
- Training
- International Standards
- Current Research

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Government Demand

- **CERT Secure Coding standards are under consideration as mandated standards in the** DoD IT Standards Registry (**DISR**). DISR mandated standards are essential for enabling interoperability of Net-Centric services across the *Global Information Grid* (GIG) enterprise. They are the minimum set of essential standards for the acquisition and development of all DoD information systems.
- 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 specified in the DoD acquisition programs' Request for Proposals (RFPs). 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."**

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

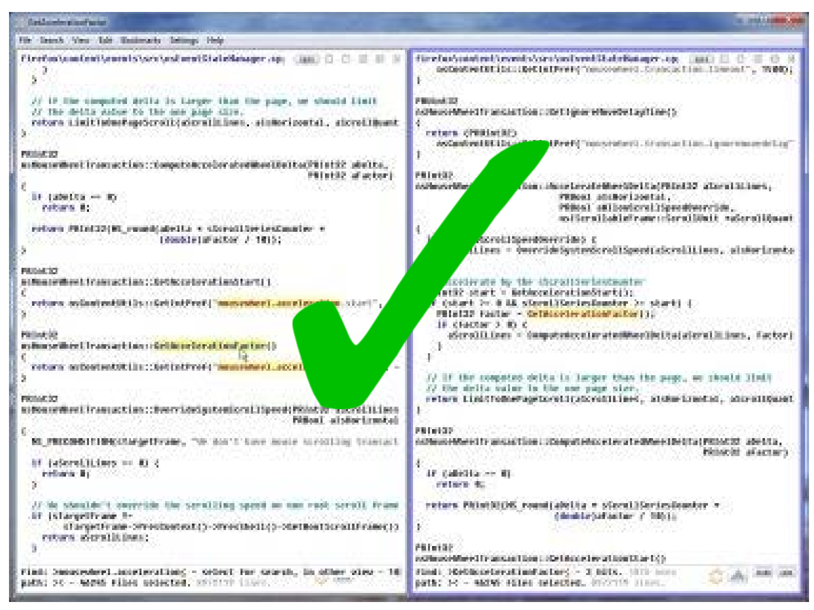
A goal of conformance testing is to provide an incentive for industry to invest in developing conforming systems:

- Perform conformance testing against CERT secure coding standards.
- Verify that a software system conforms with a CERT secure coding standard.
- Use the CERT SCALe seal when marketing products.
- Maintain a certificate registry with the certificates of conforming systems.

CERT
CONFORMANCE TESTED

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Use Static Testing and Source Code Analysis



Secure Code Analysis Laboratory (SCALe)

- C, C++, Java, PERL, Android rule conformance checking
- Information flows across applications
- Operating system call flows

Manual review against threat models

- Web services: SOAP, REST, XML-RPC
- Java applets
- Access control matrices
- Custom network or data protocols.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Source Code Analysis Laboratory

Source Code Analysis Laboratory (SCALe)

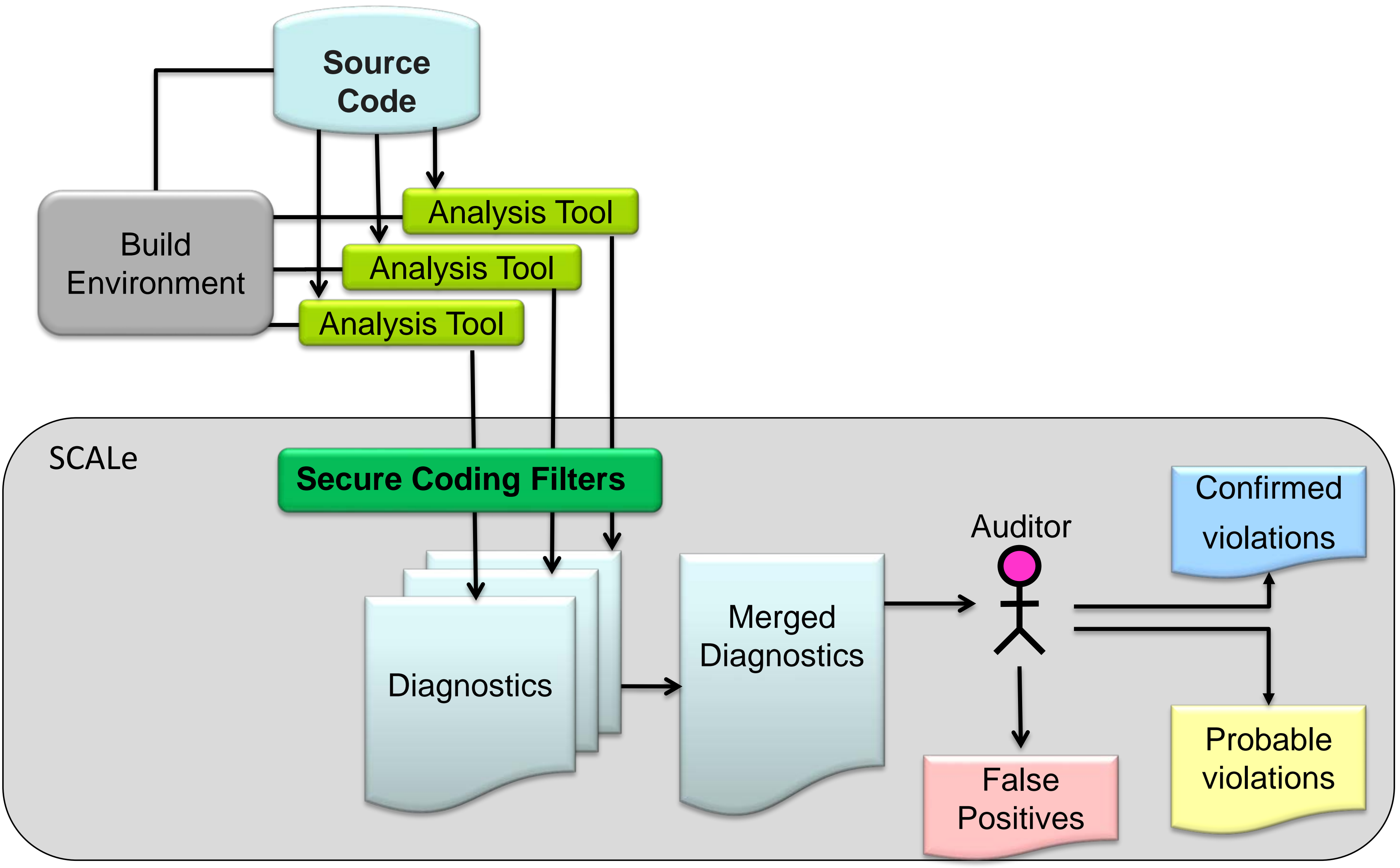
- Consists of commercial, open source, and experimental analysis
- Is used to analyze various code bases including those from the DoD, energy delivery systems, medical devices, and more
- Provides value to the customer but is also being instrumented to research the effectiveness of coding rules and analysis

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SCALe Secure Coding Conformance Process



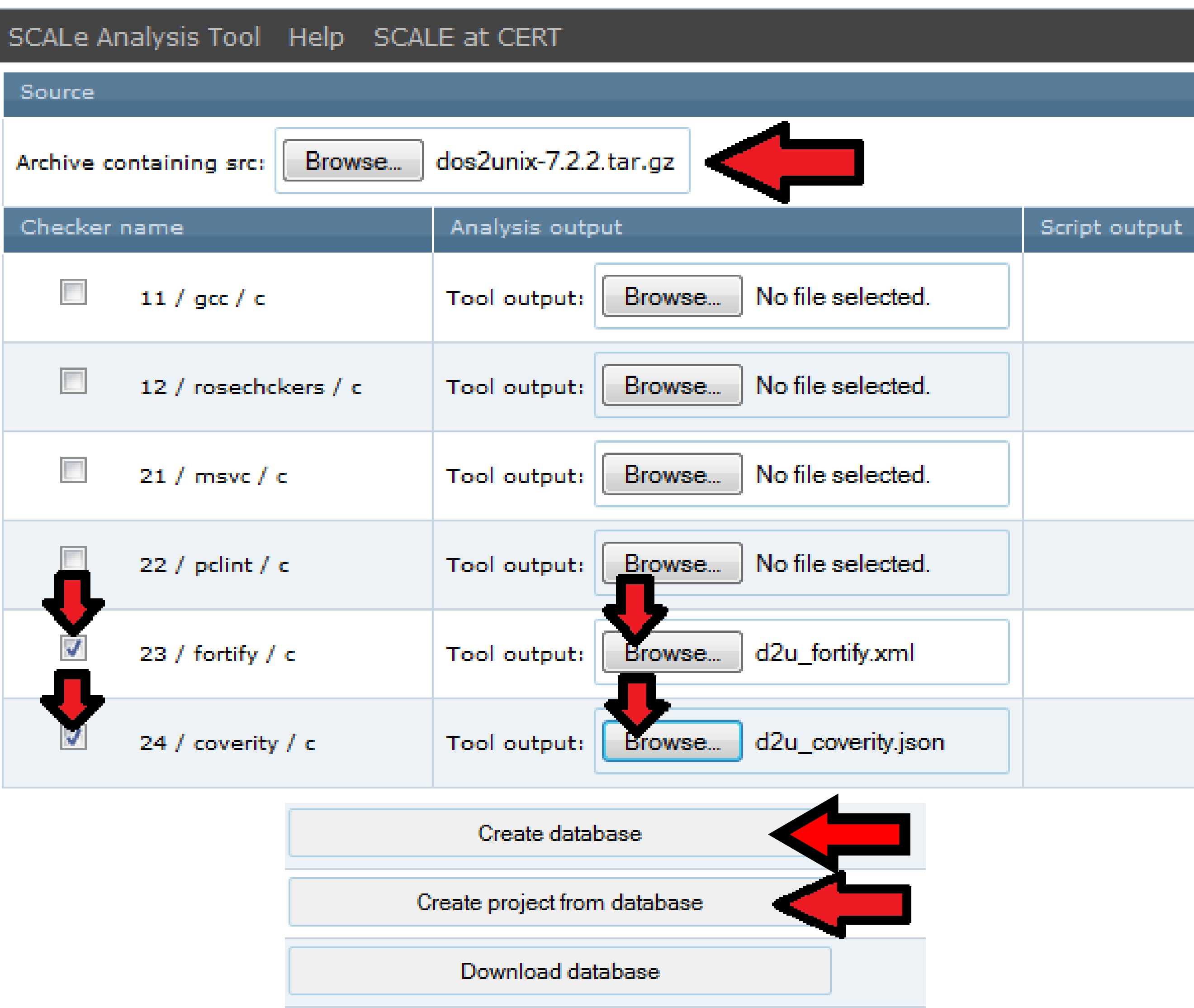
© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Select SCALe Assessments

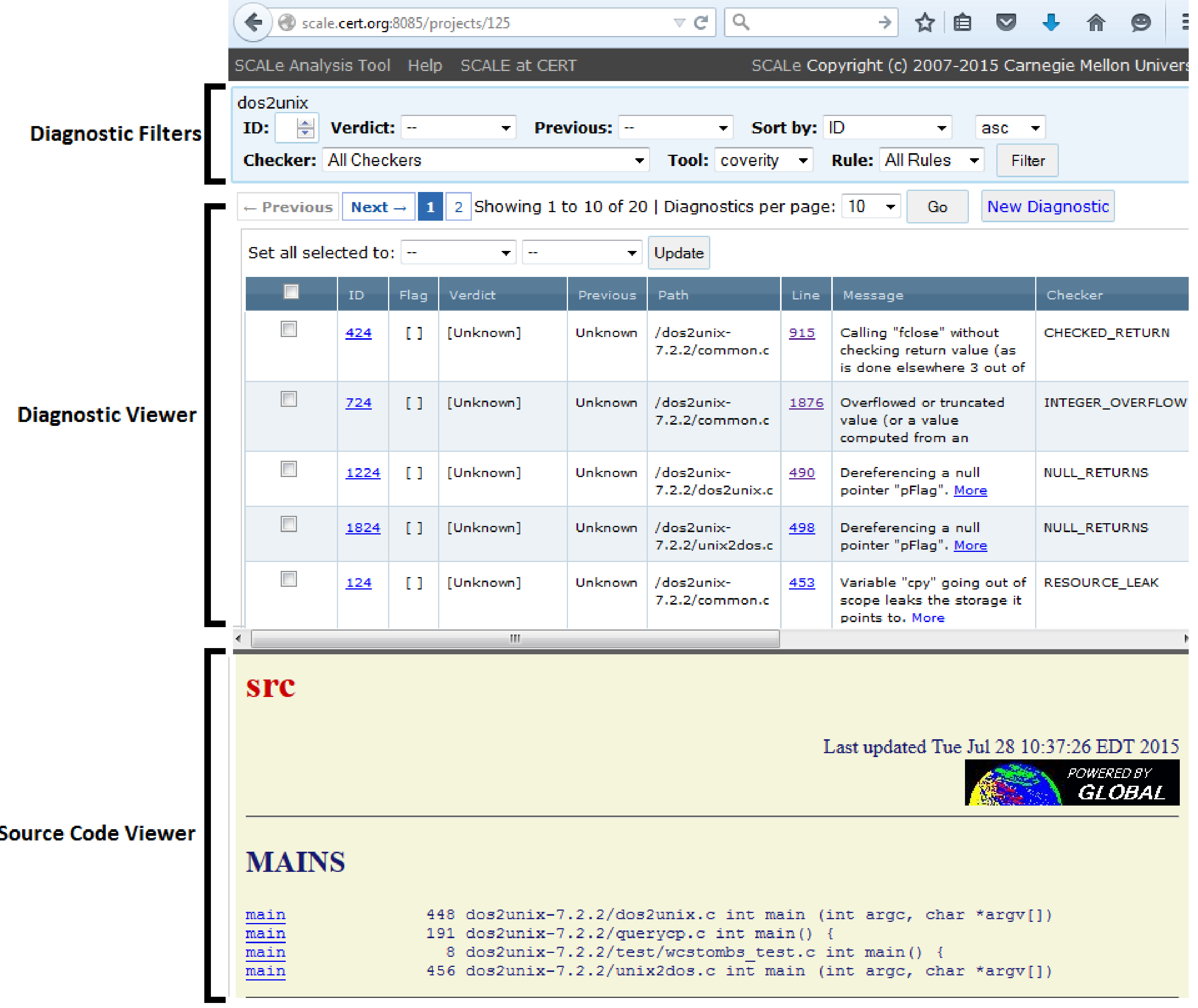
Codebase	Date	Customer	Lang	ksLOC	Rules	Diags	True	Suspect	Diag /KsLOC
A	6/12	Gov1	C++	38.8	12	1,071	52	1,019	27.6
B	3/13	Gov1	C	87.4	28	17,543	86	17,457	200.7
C	10/13	Gov2	C	9,585	18	289	159	130	0.03
D	6/12	Gov3	Java	4.27	18	345	117	228	80.8
E	9/12	Gov2	Java	61.2	33	538	288	250	8.8
F	11/13	Gov2	Java	17.6	21	414	341	73	23.5
G	2/14	Gov4	Java	653	29	8,526	64	8,462	13.1
H	3/14	Gov5	Java	1.51	8	53	53	0	35.1
I	5/14	Mil1	Java	403	27	3114	723	2,391	7.7
J	1/11	Gov3	Perl	93.6	36	6,925	357	6,568	74.0
K	5/14	Gov3	Perl	10.2	10	133	84	49	13.0

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Creating Project and Importing Diagnostics



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.



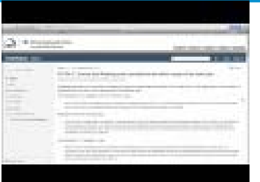
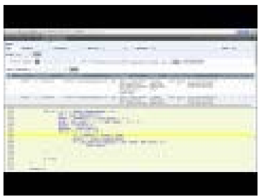
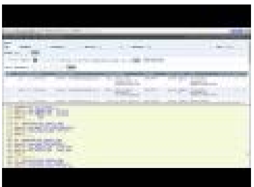
© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SCALe Web App Demos

Watch demonstration videos of SCALe on YouTube:

https://www.youtube.com/playlist?list=PLSNlEg26NNpwagA8kj9WMMr9jg8awKqJF

Select Videos:

	Source Code Analysis Laboratory (SCALe) Demo: Web UI Columns	8:04
	Source Code Analysis Laboratory (SCALe) Demo Web UI Heading	4:43
	Source Code Analysis Laboratory (SCALe) Demo: Web UI Code	3:01

For more about SCALe, see: http://www.cert.org/secure-coding/products-services/scale.cfm

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

CERT SCALe Seal 1

Software that has been determined by CERT to conform to a secure coding standard may be identified by the CERT SCALe seal on the software 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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Agenda

CERT Secure Coding Overview

- Secure Coding Guidelines
- SCALe Audits and Software
- Training
- International Standards
- Current Research

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding Professional Certificates

CERT Secure Coding Professional Certificates

Secure Coding Professional Certificates

Our certificate programs will help developers to increase security and reduce vulnerability within the programs they develop

Course, Exam, and Certificates for “C and C++” and “Java”
Online and Onsite course options available
Includes Secure Software Concepts and Secure Coding in specified languages

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SEI Secure Coding in C and 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.

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.

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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SEI Secure Coding in C and 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.

Topics

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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SEI Secure Coding in Java Training

The Secure Coding in Java course is designed to improve the secure use of Java. The course is useful to developers of Java SE, EE, and ME versions of the platform. Tailored to meet the needs of a development team, the course covers security aspects of

- Trust and Security Policies
- Validation and Sanitization
- The Java Security Model
- Declarations
- Expressions
- Object Orientation
- Methods
- Vulnerability Analysis Exercise
- Numerical Types in Java
- Exceptional Behavior
- Input/Output
- Serialization
- The Runtime Environment
- Introduction to Concurrency in Java
- Advanced Concurrency Issues

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding Course: Objectives 2

Dynamic Memory

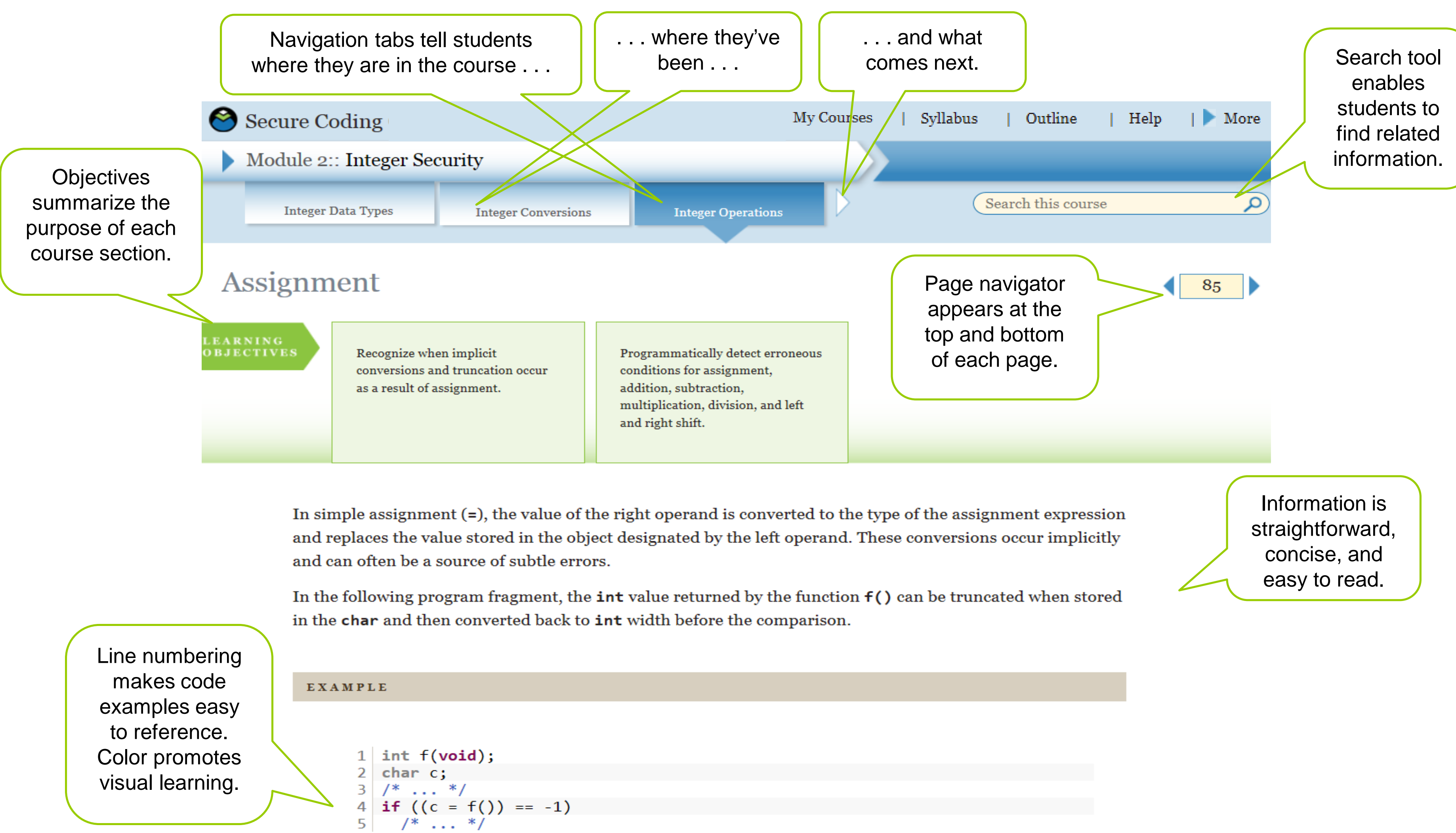
- Use standard C memory management functions securely.
- Align memory suitably.
- Explain how vulnerabilities from common dynamic memory management errors can be exploited.
- Identify common dynamic memory management errors.
- Perform C++ memory management securely.
- Identify common C++ programming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

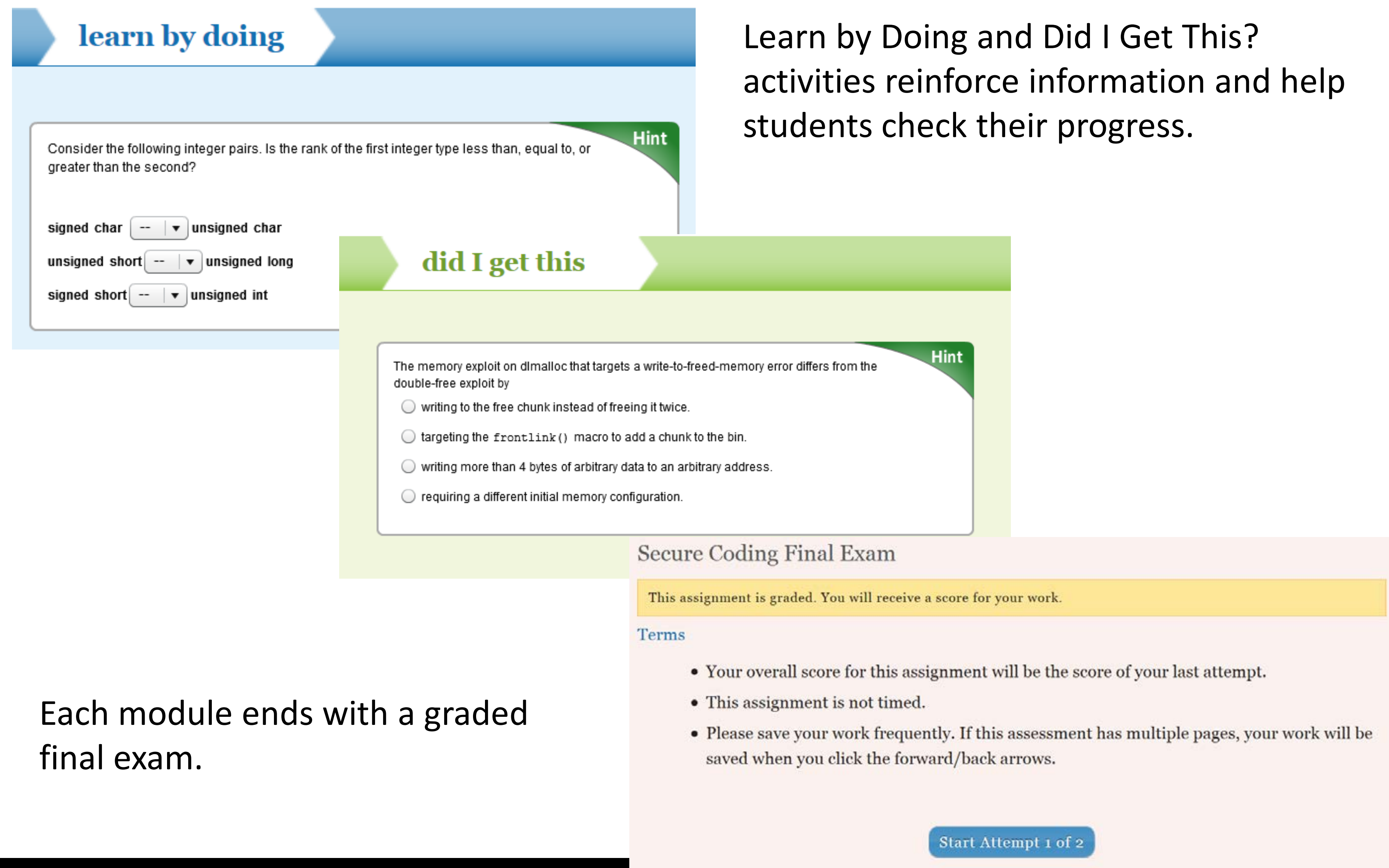
© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding Course Interface



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding Online Assessments

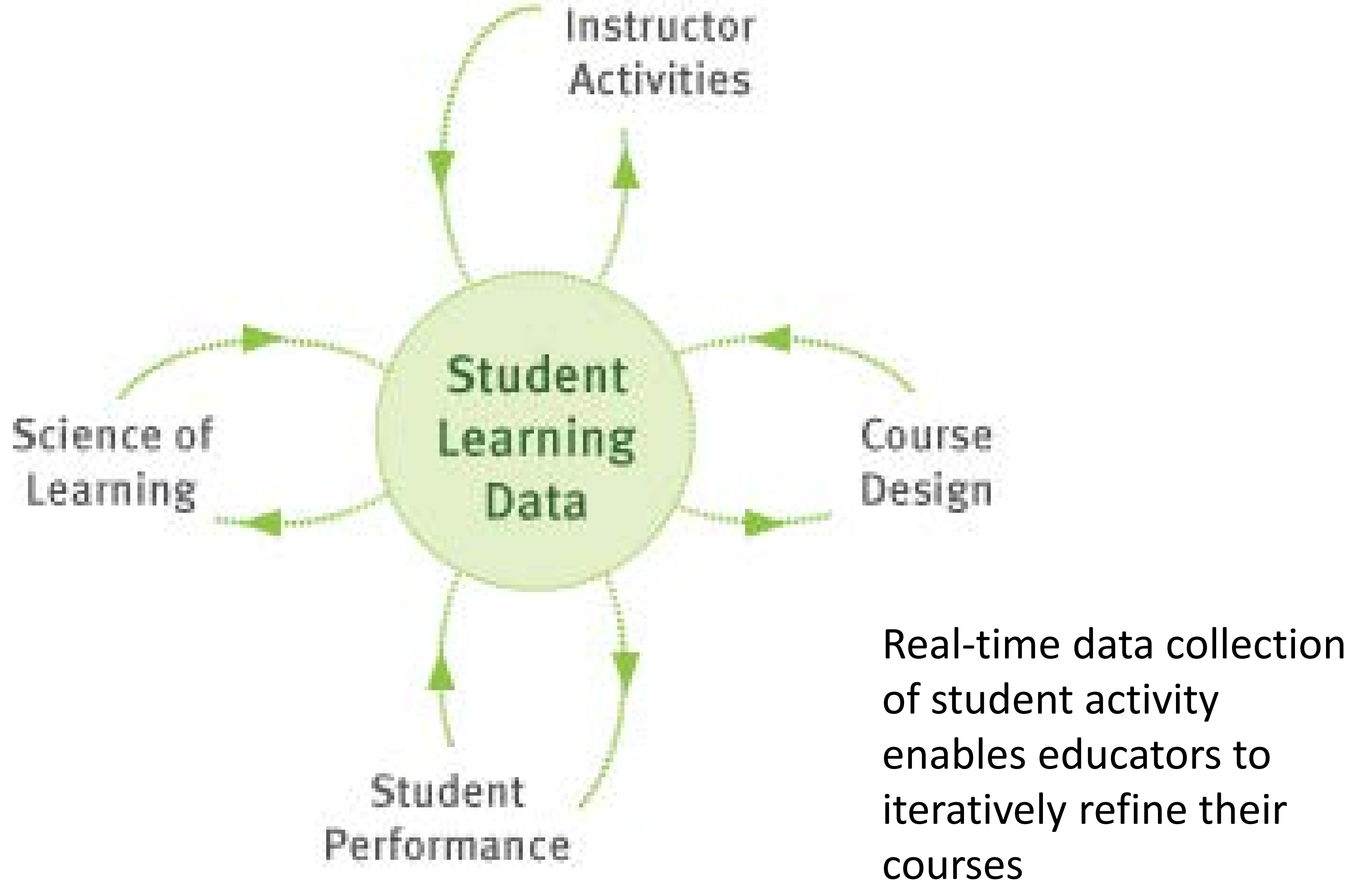


Learn by Doing and Did I Get This? activities reinforce information and help students check their progress.

Each module ends with a graded final exam.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Feedback Loops



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

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Agenda

CERT Secure Coding Overview

- Secure Coding Guidelines
- SCALe Audits and Software
- Training
- International Standards
- Current Research

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

ISO/IEC JTC1/SC22/WG23 is the international standardization working group for programming language vulnerabilities. WG23 is responsible for

- ISO/IEC TR 24772, *Guidance to avoiding vulnerabilities in programming languages*
- ISO/IEC 17960, *Code signing for source code*

INCITS Technical Committee PL22 is the

- U.S. umbrella organization responsible for programming language standards.
- U.S. TAG to ISO/IEC JTC 1 SC22 and SC22/WG23, and provides recommendations on U.S. positions to the JTC 1 TAG.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Standards Body Participation

ISO/IEC JTC1/SC22/WG14

- Daniel Plakosh, Secretariat
- David Svoboda

ISO/IEC JTC1/SC22/WG21

- David Svoboda

ISO/IEC JTC1/SC22/WG23

- David Svoboda

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

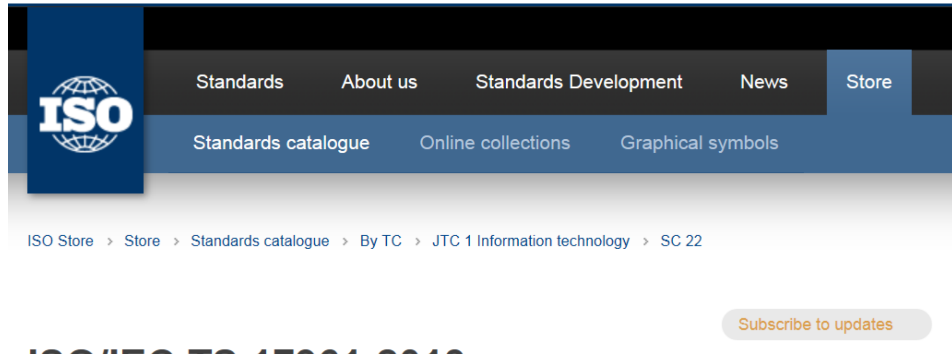
C Secure Coding Guidelines SG

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

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.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Agenda

CERT Secure Coding Overview

- Secure Coding Guidelines
- SCALe Audits and Software
- Training
- International Standards
- Current Research

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Secure Coding Research

Prioritizing Vulnerabilities using Classification Models

- Aggregating information from multiple analysis tools to make better predictions about whether a potential defect is true or not.

Automated Code Repair

- Fixing code based on anti-patterns and patterns for repair, rather than just alerting developers and testers to a potential defect.

Sensitive Dataflow Analysis among Android App Sets

- Detecting tainted data flows across multiple Android components

Integrating Secure Coding Rule analysis with Development Environments

- Moving secure coding analysis “to the left” to alert developers while coding, not just during a test phase after they are done.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Prioritizing Vulnerabilities

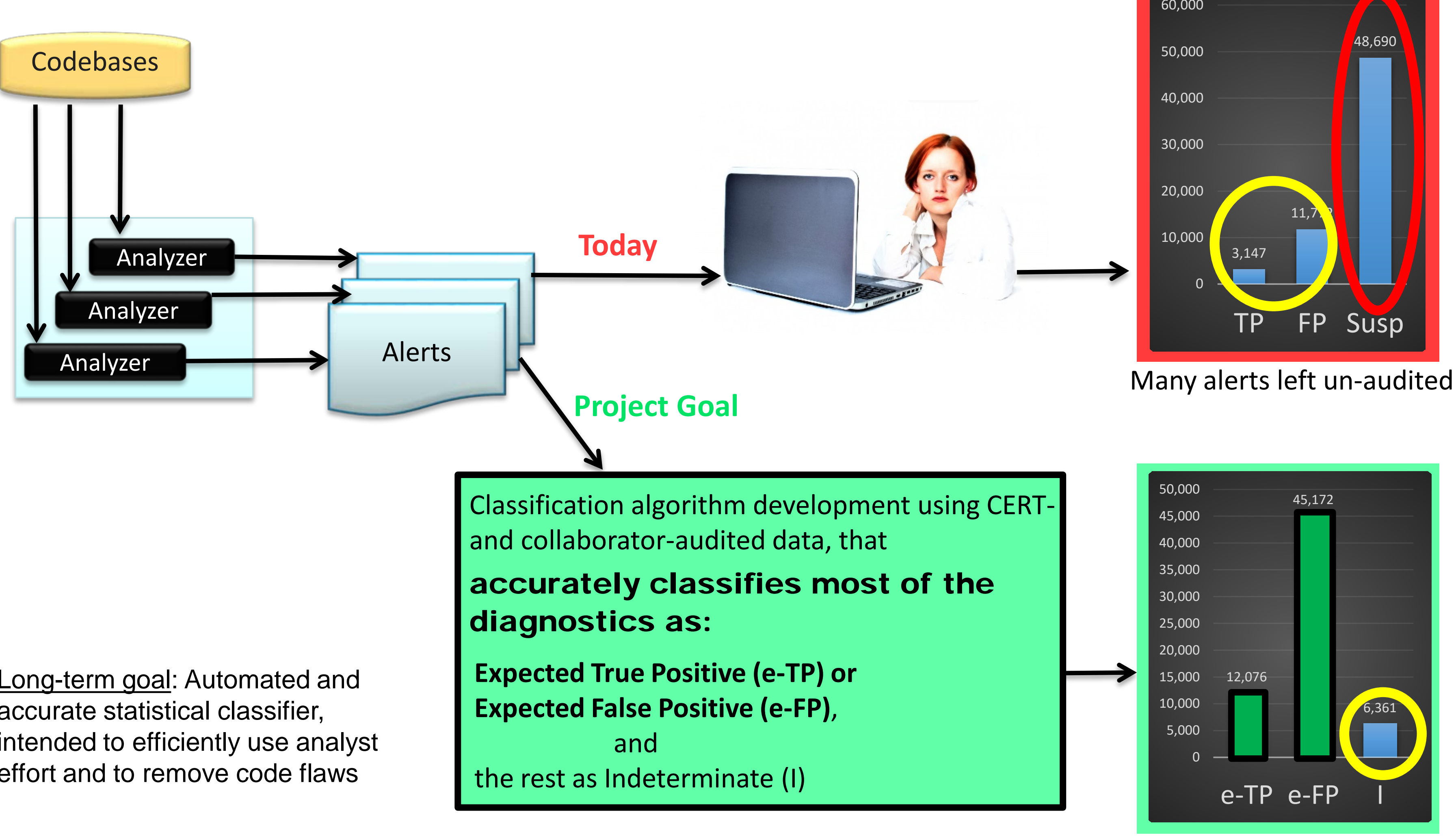


Image of woman and laptop from http://www.publicdomainpictures.net/view-image.php?image=47526&picture=woman-and-laptop “Woman And Laptop”

Prioritized, small number of alerts for manual audit

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Results with Transition Value

Software and paper: Classifier-development

- Code for developing classifiers in the R environment
- Paper: classifier development, analysis, & use [1]

Software: Enhanced-SCALe Tool (auditing framework)

- Added data collection
- Archive sanitizer
- Alert fusion
- Offline installs and virtual machine

Training to ensure high-quality data

- SEI CERT coding rules
- Auditing rules [2]
- Enhanced-SCALe use

Auditor quality test

- Test audit skill: mentor-expert designation

Conference/workshop papers:

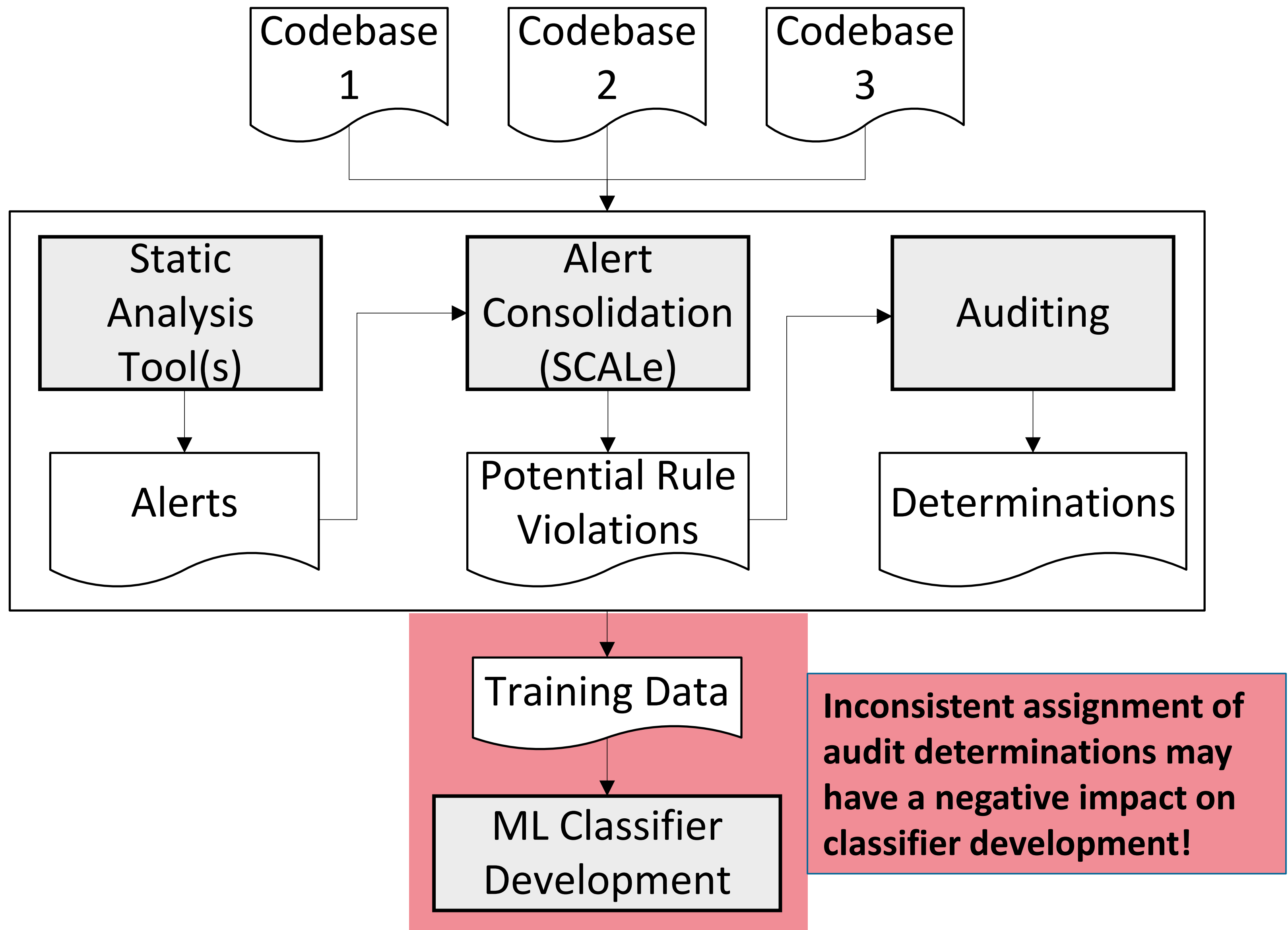
[1] Flynn, Snavely, Svoboda, Qin, Burns, VanHoudnos, Zubrow, Stoddard, and Marce-Santurio. “Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models”, work in progress.

[2] Svoboda, Flynn, and Snavely. “Static Analysis Alert Audits: Lexicon & Rules”, IEEE Cybersecurity Development (SecDev), November 2016.

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

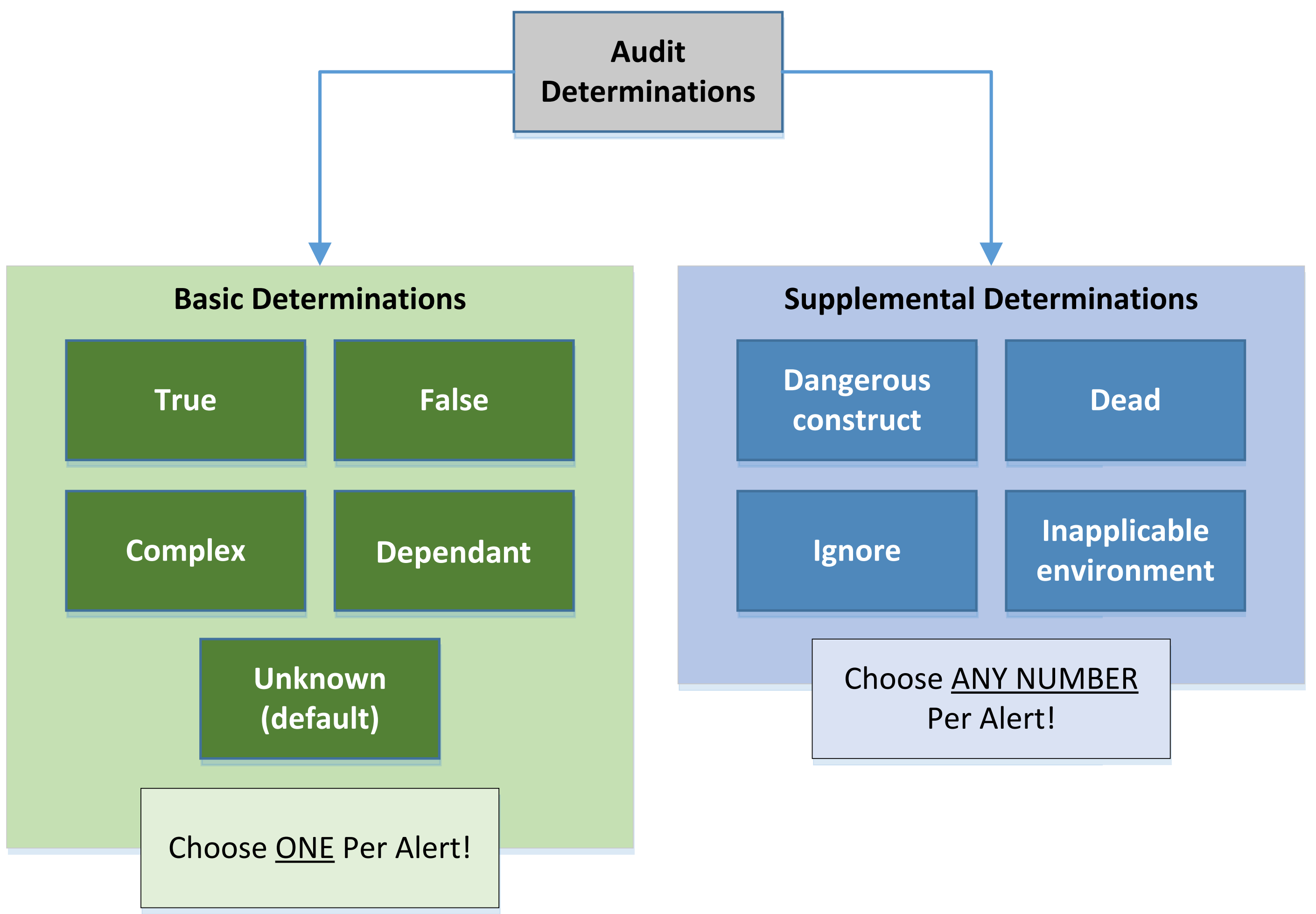
Background: Automatic Alert Classification



© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Lexicon: Audit Determinations



© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

SCALe Auditing Rules

1. Understand the language and the secure coding rule in question.
2. Some diagnostics are too complex to judge; they should be marked *suspicious*.
3. It is OK to mark a diagnostic true even if you think the code maintainers will protest.
4. Assume that external inputs to the program are malicious.
5. Unless instructed otherwise, assume that code must be portable.
6. When auditing a diagnostic, if you discover a second true violation, mark its diagnostic as *true*.
7. Do not arbitrarily extend the scope of a CERT rule.
8. Code that behaves as expected might still violate a CERT rule.
9. A diagnostic might indicate a true violation of the CERT coding rule, even if its message text is useless or incorrect.
10. Multiple messages help in understanding a diagnostic.
11. Assume no violations occur before the line in question.
12. Handle an alert in unreachable code depending on whether it is exportable.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Results with Transition Value: Sanitizer

New data sanitizer

- Anonymizes sensitive fields
- SHA-256 hash with salt
- Enables analysis of features correlated with alert confidence

SCALe project is in a SCALe database

- DB fields may contain sensitive information
- Sanitizing script anonymizes or discards fields
  - Diagnostic message
  - Path, including directories and filename
  - Function name
  - Class name
  - Namespace/package
  - Project filename

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Classifier Test Highlights

Classifiers made from all data, pooled:

All-rules (158) classifier accuracy:

- Lasso Logistic Regression: 88%
- Random Forest: 91%
- CART: 89%
- XGBoost: 91%

Single-rule classifier accuracy:

Rule ID	Lasso LR	Random Forest	CART	XGBoost
INT31-C	98%	97%	98%	97%
EXP01-J	74%	74%	81%	74%
OBJ03-J	73%	86%	86%	83%
FIO04-J*	80%	80%	90%	80%
EXP33-C*	83%	87%	83%	83%
EXP34-C*	67%	72%	79%	72%
DCL36-C*	100%	100%	100%	100%
ERR08-J*	99%	100%	100%	100%
IDS00-J*	96%	96%	96%	96%
ERR01-J*	100%	100%	100%	100%
ERR09-J*	100%	88%	88%	88%

* Small quantity of data, results suspect

General results (not true for every test)

- Classifier accuracy rankings for all-pooled test data:
  XGBoost ≈ RF > CART ≈ LR
- Classifier accuracy rankings for collaborator test data:
  LR ≈ RF > XGBoost > CART
- Per-rule classifiers generally not useful (lack data), but 3 rules (INT31-C best) are exceptions.
- With-tools-as-feature classifiers better than without.
- Accuracy of single language vs. all-languages data:
  C > all-combined > Java

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Rapid expansion of classification models to prioritize static analysis alerts for C

Problem: Security-related code flaws detected by static analysis require too much manual effort to triage, plus it takes too long to audit enough alerts to develop classifiers to automate the triage.

Solution: Rapid expansion of number of classification models by using "pre-audited" (equivalent to audited) code.

Approach:

1. Systematically map CERT C coding rules to named flaws in subsets of pre-audited code (published as true or false for the flaw)
2. Automated enhanced-SCALe analysis of pre-audited (not by SEI) codebases to gather sufficient code & alert feature info for classifiers
3. Use DoD collaborator data from auditing software they actually use as a validity check, and compare classifiers versus those based on pre-audited code (mostly small, uncomplicated tests).

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Automated Code Repair

Hypothesis: Many violations of rules follow a small number of anti-patterns with corresponding patterns for repair, and these can be feasibly recognized by static analysis.

- `printf(attacker_string)` → `printf("%s", attacker_string)`

We propose to create a tool to automatically repair defects in source code resulting from violations of the CERT Coding Standards.

Formalizable Constraints (to be formally verified):

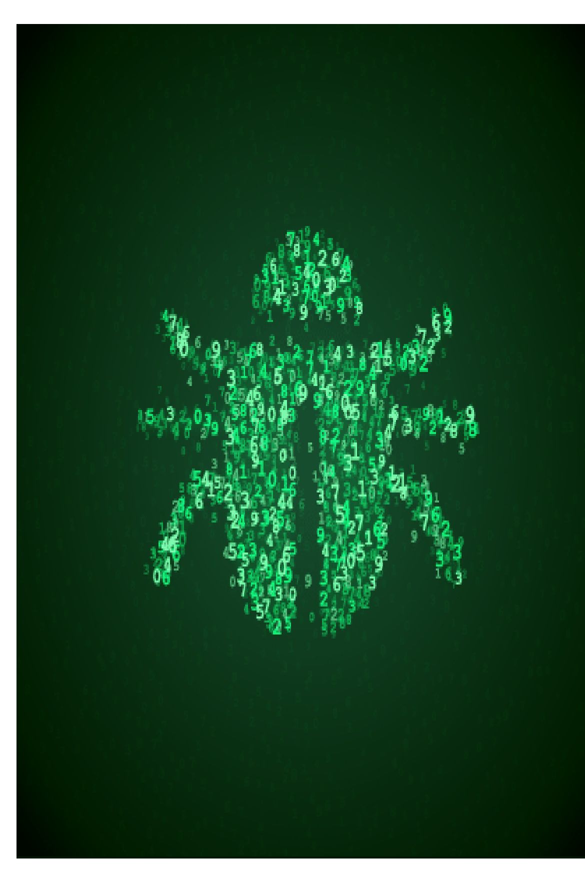
- The patched and unpatched program behave identically over the set of all traces that conform to the rules.
- No trace violates the rules.

Non-Formalizable Constraint:

- Repair in way that is plausibly acceptable to the developer.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Automated Code Repair – Motivation



Software vulnerabilities constitute a major threat

- A majority arise from common coding errors
- Shown by experience from source code analysis labs at CERT and DoD

Static analysis tools help, but:

- Typically are used late in the development process
- Produce an enormous number of warnings
- The volume of true positives often overwhelms the ability of the development team to fix the code

Huge amount of code in use by DoD

- Billions of lines of C code
- Unknown number of security vulnerabilities

Likely Code Candidates

- Large Code Base
- Dynamically Allocated Memory (Buffer Overflows)
- Variable-length Input

© 2017 Carnegie Mellon University

[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Integer Overflow

This past year (FY16), we developed techniques for automated repair of **integer overflows** that lead to **memory corruption**

Integers in C are represented by a fixed number of bits N (e.g., 32 or 64).

- Overflow occurs when the result cannot fit in N bits
- Modular arithmetic: Only the least significant N bits are kept

How does integer overflow lead to memory corruption?

1. Memory allocation: `malloc(·)`.
2. Bounds checks for an array

Example: Android Stagefright bugs (July 2015)

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Benefits

Eliminate security vulnerabilities at a **much lower cost** than manual repair

Integer overflows are a **very common** type of bug

- In CERT SCALe audits, about 80% of findings were related to fixed-width integers

Our technique:

- **Will not break working code**, provided *inferred specification* is correct (Next slide)
- Typically total slowdown < 5% (Based on theoretical model)
- False positives: Flagged operations that cannot actually overflow
  - Then our ‘repair’ just adds a little unnecessary overhead

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

wrappers.h

```
1. inline static size_t UADD(size_t lop, size_t rop) {
2.     size_t result;
3.     bool flag = __builtin_add_overflow(lop, rop, &result);
4.     if (flag) {result = SIZE_MAX;}
5.     return result;
6. }
```

- What if `dest_size` is `SIZE_MAX`?
- What if both sides of inequality overflow?
- What if overflow reaches a non-comparison sink?

Repair: **`UADD(start, n)`**

```
if (start + n <= dest_size) {
  memcpy(&dest[start], src, n);
} else {
  return -EINVAL;
}
```


© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Inference of Memory Bounds

Problem 1: Security vuls. Not just traditional buffer overflows.

Leakage of sensitive info (out-of-bounds reads):

- HeartBleed vulnerability, `BenignCertain` attack on Cisco PIX.
- Unaffected by mitigations such as ASLR and DEP.
- Re-usable buffer with stale data: bounded to valid portion of buffer.
- Affects even Java: e.g., Jetty leaked passwords (CVE-2015-2080).

Problem 2: Decompilation of binaries. We will reconstruct information of the form "bounds of pointer *p* is the interval [*n*, *m*]".

Solution & Approach: Static analysis to find & evaluate likely bounds.
(E.g., re-usable buffer: guess that upper bound for reading is the last position written.)

For decompilation: Report these bounds, use when naming variables.

For repair: Test with dynamic analysis – tentatively implement all bounds checks (even those subsumed by stricter bounds checks) as 'soft-fail' (just log a warning, don't abort). Can also repair to *Checked C* (David Tarditi).

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Android Information Leaks: Automated Detection

Problem: Exfiltration of sensitive data on mobile devices. Colluding apps, or combination of malicious app and leaky app, can use intents (messages sent to Android app components) to extract sensitive or private information from an Android phone.

Solution: Precisely detect (i.e., few false positives) malicious exfiltration of sensitive information from an Android phone (even across multiple components), in a practical time & memory bound.

Approach: Add context sensitivity to analysis, to reduce false positives, while retaining analytical speed by using DidFail's fast 2-phase static analysis method (that summarizes potential flows of sensitive data per-app and quickly analyzes per-app-set).

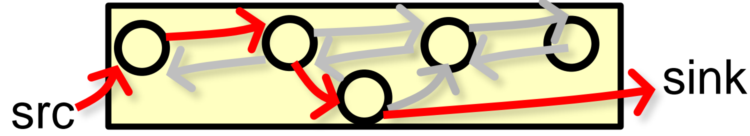
© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Android App Sets: Sensitive Dataflow

Problem: Colluding apps, or a combination of a malicious app and leaky app, can use intents (messages sent to Android app components) to extract sensitive or private information from an Android phone.

Goal: Precisely detect tainted flows **across multiple Android components** from sensitive information sources to restricted sinks.

- If such flows are discovered:
  - User might refuse to install app
  - App store might remove app



Achievements:

- First published static taint flow analysis for app sets (not just single apps)
- Fast user response: two-phase method uses phase-1 precomputation

Next: More precision using context sensitivity ⟹ fewer false alarms.

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Analysis of Android App Sets: Sensitive Dataflow

Goal: enforce confidentiality and integrity

Cutting-edge Android **app set** static dataflow analysis "DidFail" combines precise **single-component taint analysis** and **intent analysis**.

- Phase 1: Each app analyzed once, in isolation
  - Examine flow of tainted data from sources to sinks (including intents)
  - Examines intent properties to match senders and receivers
- Phase 2: For a particular set of apps
  - Generate taint flow equations
  - Iteratively solve equations
  - **Fast!**

Phase 2 fast because of Phase 1 pre-computation

Source code and binaries:
`http://www.cert.org/secure-coding/tools/didfail.cfm`

Next Work:
- More context sensitivity

Check(AppZ, List_MyApps)

"Flows possible are [POSSIBLE_FLOWS]. Do you want to install AppZ?"

App Store/Security System Provider

Stored Phase 1 analysis
App_1: $TaintFlowInfo_{A1}$, $IntentInfo_{A1}$
App_2: $TaintFlowInfo_{A2}$, $IntentInfo_{A2}$
...
App_X: $TaintFlowInfo_{AX}$, $IntentInfo_{AX}$

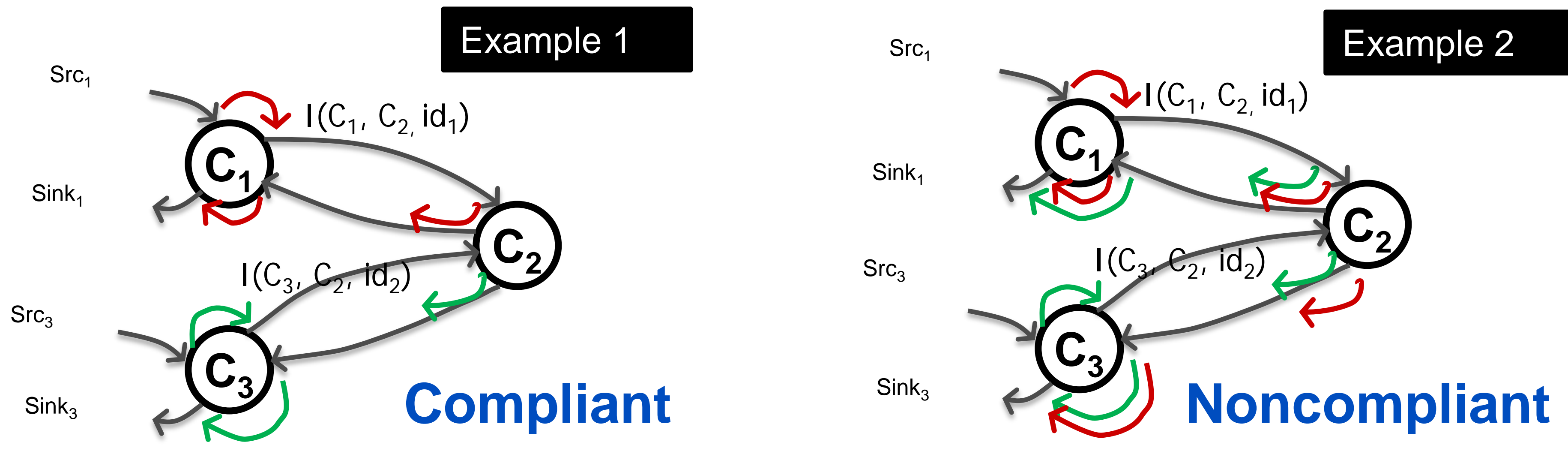
Phase 2 analysis
Output: potential tainted flows

Apps
App_1
App_2
App_3
App_4
App_5
...
App_X

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Usability: Policies to Determine Allowed Flows

Policy: Prohibit flow from Src_1 to $Sink_3$



Policies could come from:

- App store
- Security system provider
- Employer
- User options

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Select Publications

Books & Stds.

- *The SEI CERT C Coding Standard, 2016 Edition*
- *The SEI CERT C++ Coding Standard, 2016 Edition*
- *Java Coding Guidelines* (published 2013)
- *Secure Coding in C and C++, 2nd Edition* (published 2013)
- *ISO/IEC TS 17961 C Secure Coding Rules*

Papers & Presentations

- Prioritizing Alerts from Static Analysis with Classification Models (October 2016)
- Static Analysis Alert Audits: Lexicon & Rules (November 2016)
- Automated Code Repair (October 2016)
- Establishing Coding Requirements for Non-Safety-Critical C++ Systems (October 2016)
- *Beyond errno: Error Handling in C* (November 2016)
- Exploiting Java Serialization for Fun and Profit (September 2016)
- Improving the Automated Detection and Analysis of Secure Coding Violations (2014)
- Common Exploits and How to Prevent Them (August 2016)

Websites

- http://www.cert.org/secure-coding/
- http://www.cert.org/secure-coding/publications/
- http://www.cert.org/secure-coding/products-services/scale.cfm
- http://securecoding.cert.org/

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.

For More Information

David Svoboda
Software Security Engineer
Secure Coding Initiative
Phone: (412) 268-3965
Email: svoboda@cert.org

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Web
www.cert.org/secure-coding
www.securecoding.cert.org (wiki)

Subscribe to the CERT Secure Coding eNewsletter
mailto: info@sei.cmu.edu

© 2017 Carnegie Mellon University
[Distribution Statement A] This material has been approved for public release and unlimited distribution.