Secure Coding Overview

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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.

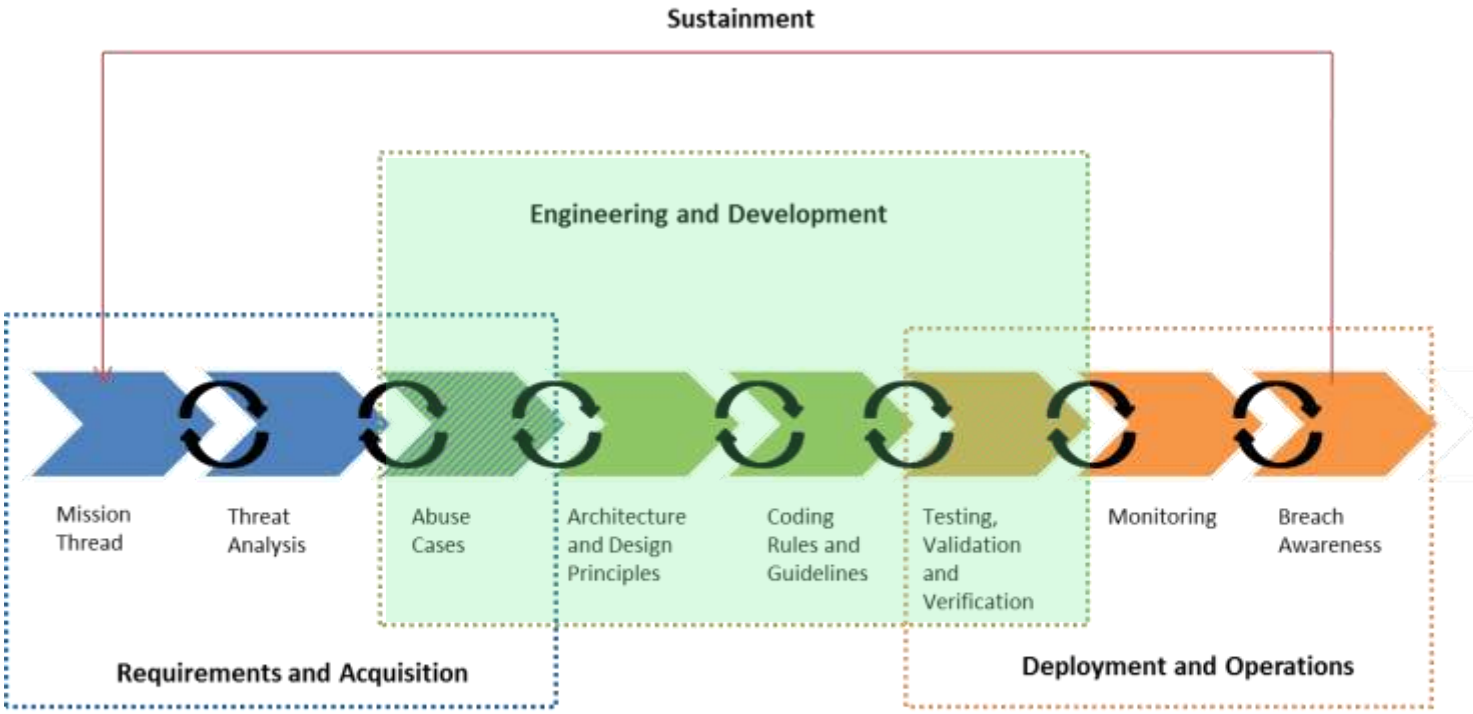
DM19-0412

Agenda

CERT Secure Coding Overview

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

Engineering and Development



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

Secure Coding History

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

Online course & Professional Certificate

SEI Secure Coding courses

Instructor-led on-site course

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

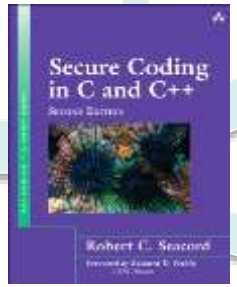
Secure design patterns

Influence international standards bodies

ISO IEC

ISO/IEC TS 17961 C Secure Coding Rules

Analyzer conformance test



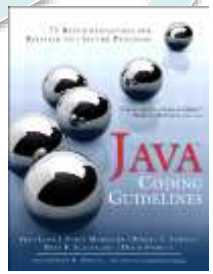
Used as input to MISRA C

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

CERT CONFORMANCE TESTED



Adoption by software developers and acquirers:
- Cisco
- Oracle
- Various defense contractors

2003

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):

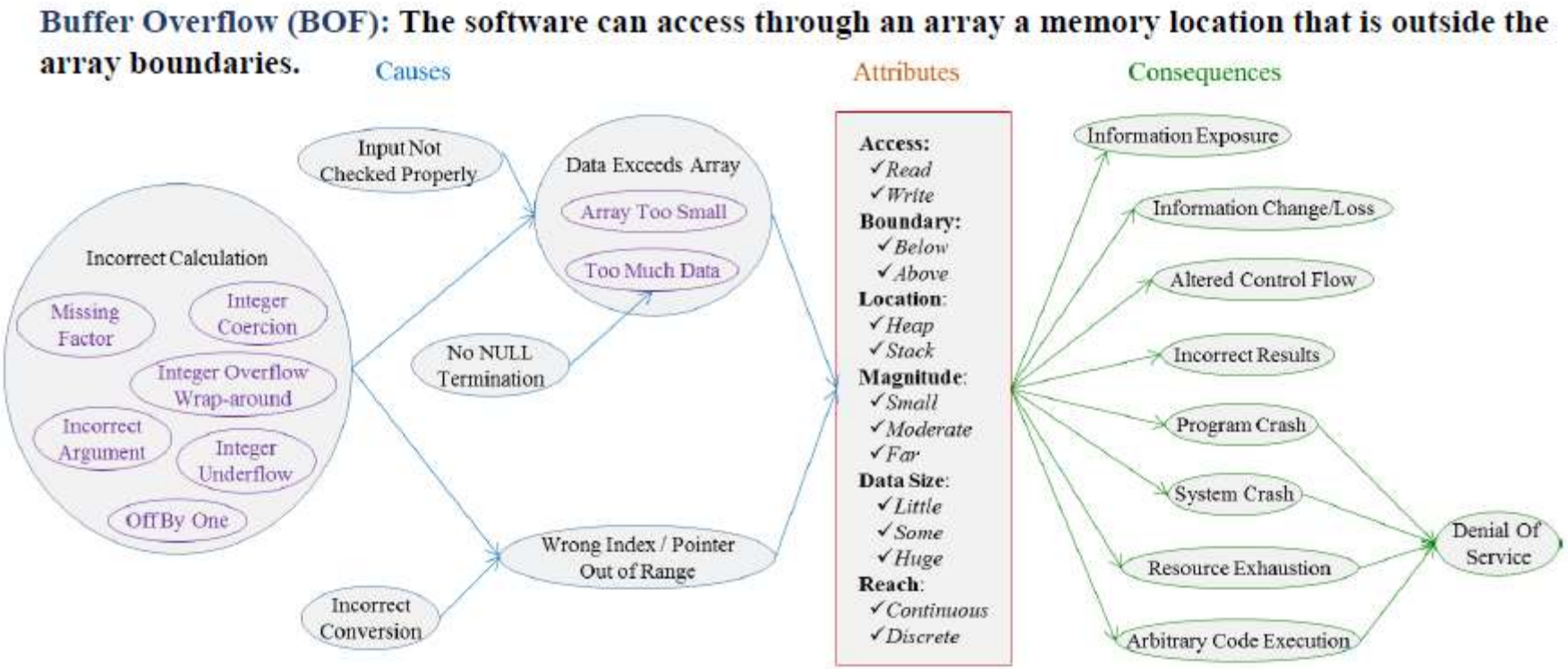
  http://cert.org/secure-coding/products-services/secure-coding-cpp-download-2016.cfm

CERT Oracle Secure Coding Standard for Java

“Current” guidelines available on CERT Secure Coding wiki

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

Buffer overflow has many causes



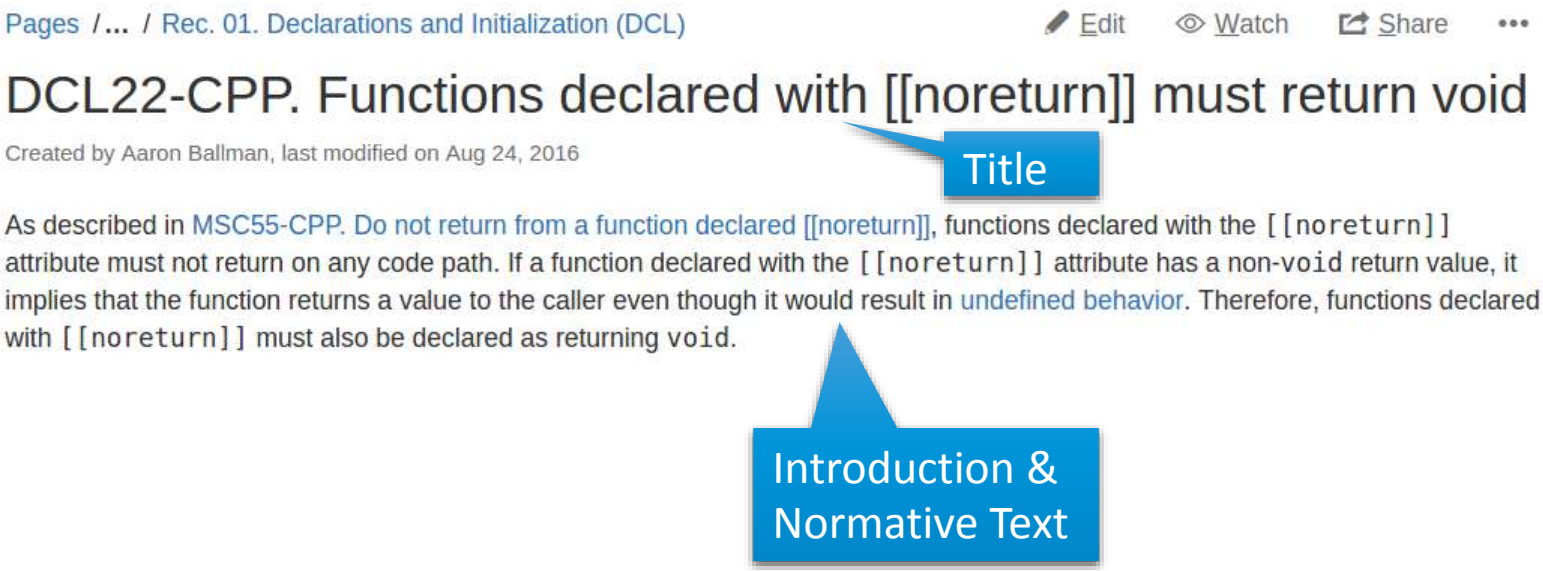
Source: Bojanova, et al, "The Bugs Framework (BF): A Structured, Integrated Framework to Express Software Bugs",2016, http://www.mys5.org/Proceedings/2016/Posters/2016-S5-Posters_Wu.pdf

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

Rule Organization – Title & Definition



Concise but not necessarily precise title

Precise definition of the rule

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

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`	

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

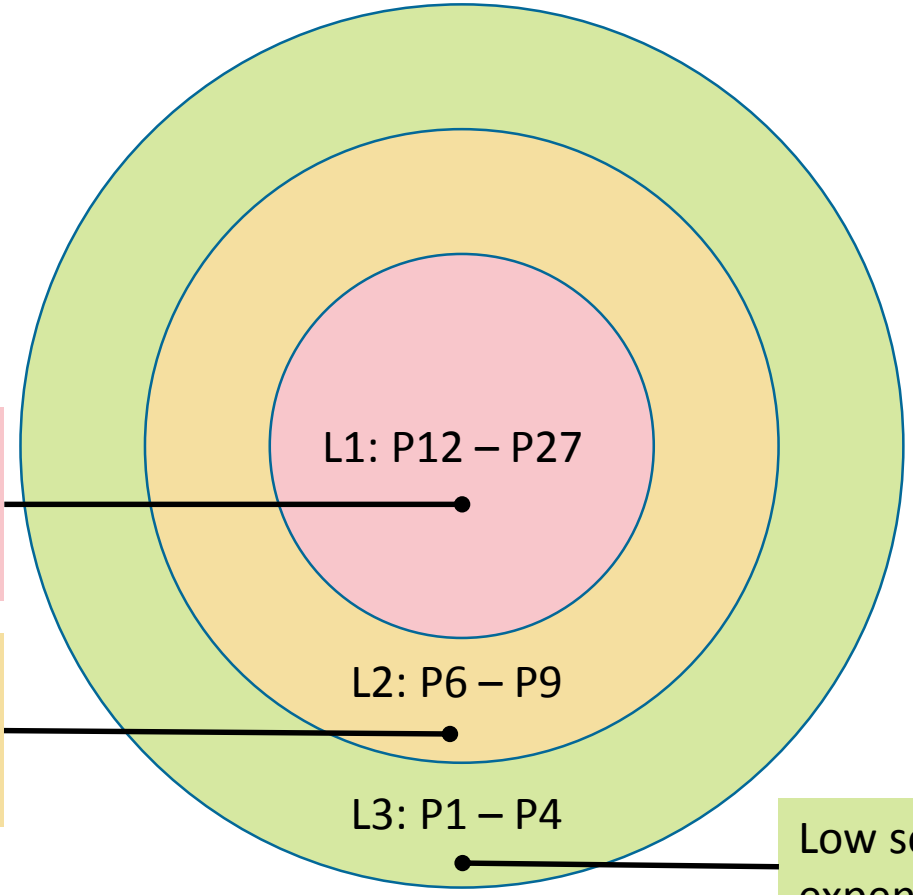
Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

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

Levels and Priorities



High severity, likely, inexpensive to repair flaws

Medium severity, probable, medium cost to repair flaws

Low severity, unlikely, expensive to repair flaws

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

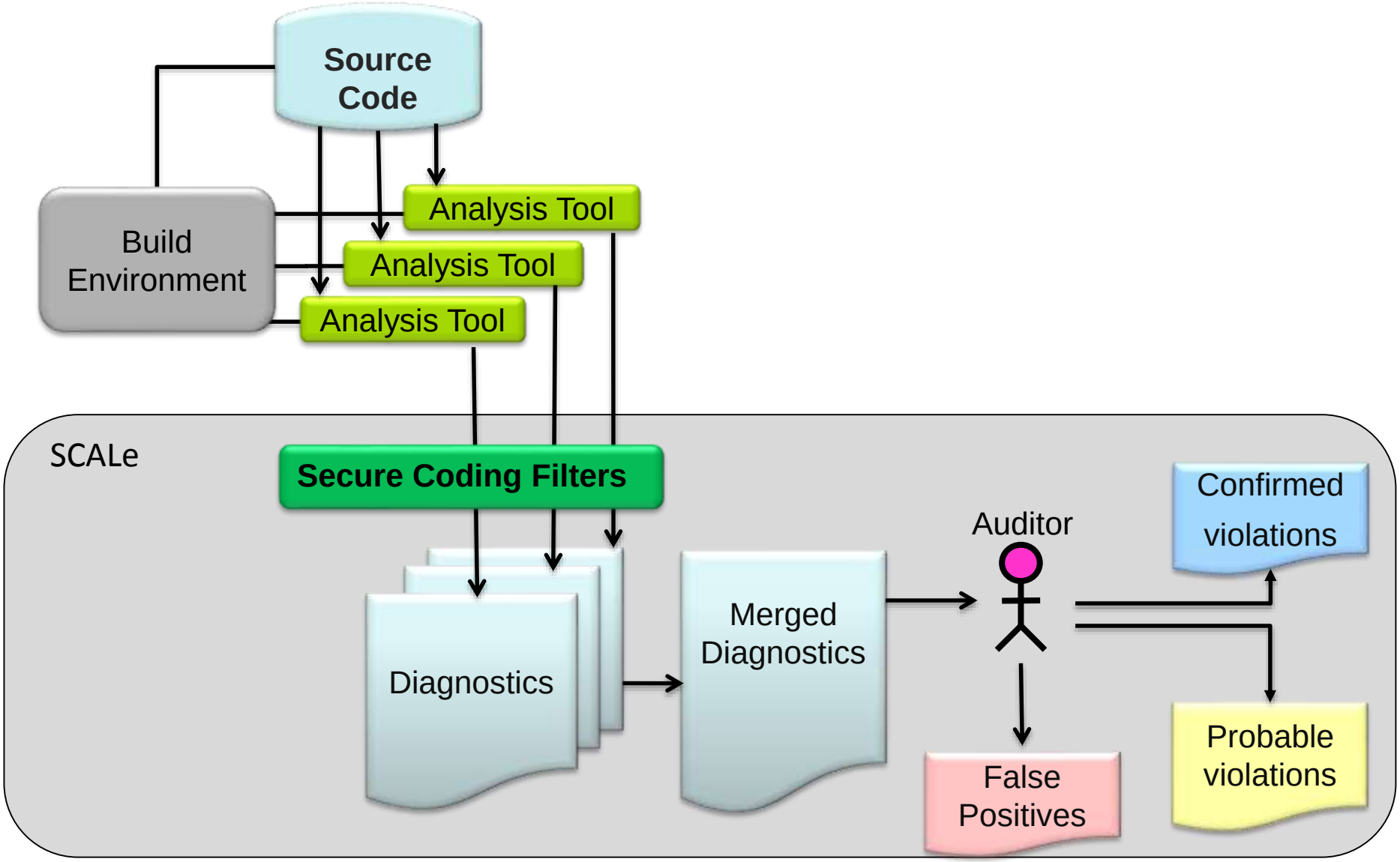
[ISO/IEC 14882-2014]	Subclause 7.6.3, "Noreturn Attribute"

Agenda

CERT Secure Coding Overview

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

SCALe Secure Coding Conformance Process



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.

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

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

Agenda

CERT Secure Coding Overview

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

Secure Coding Professional Certificates

CERT | Software Engineering Institute Carnegie Mellon University

CERT Secure Coding Professional Certificates

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

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

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

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

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 concurrency.

Agenda

CERT Secure Coding Overview

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

Standards Body Participation

ISO/IEC JTC1/SC22/WG14 – Programming Languages - C

- Daniel Plakosh
- David Svoboda

ISO/IEC JTC1/SC22/WG21 – Programming Languages – C++

- David Svoboda

ISO/IEC JTC1/SC22/WG23 – Programming Language Vulnerabilities

- David Svoboda

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.

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

CERT Secure Coding Overview

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

Secure Coding Research

Prioritizing Static Analysis Alerts using Classification Models

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

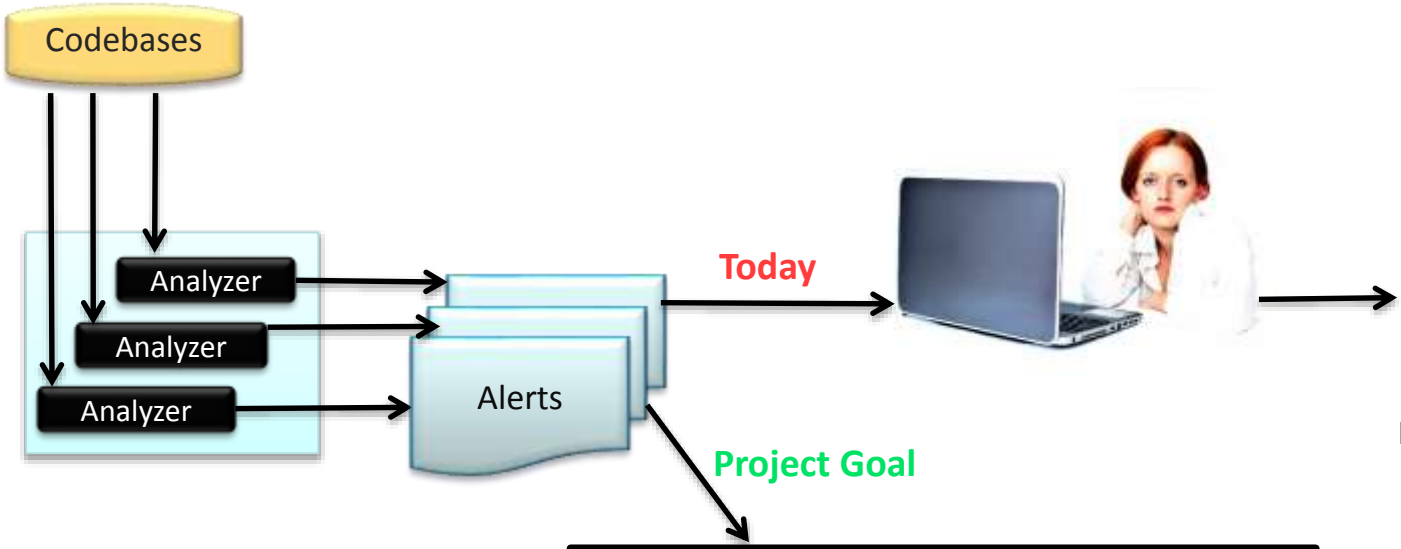
Automated Code Repair

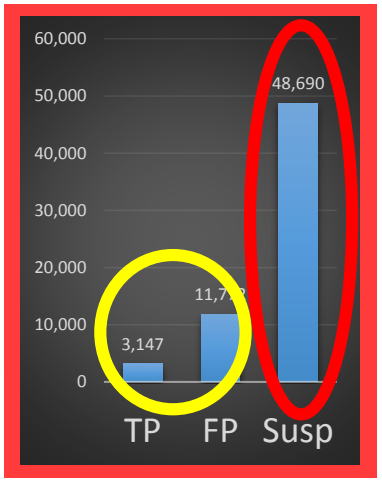
- Fixing code based on anti-patterns and patterns for repair, rather than just alerting developers and testers to a potential defect.

Predicting Security Flaws through Architectural Flaws

- Using tools that identify architectural flaws in source code to predict and find latent security flaws

Prioritizing Vulnerabilities



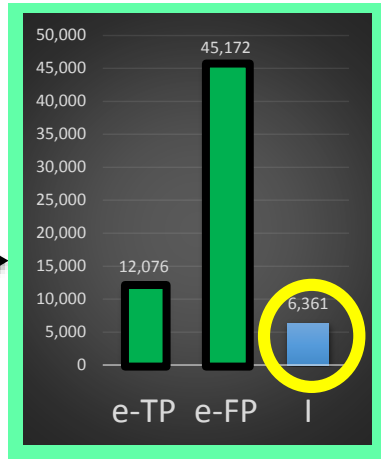


Many alerts left un-audited!

Classification algorithm development using CERT- and collaborator-audited data, that

accurately classifies most of the diagnostics as:

Expected True Positive (e-TP) or
Expected False Positive (e-FP),
and
the rest as Indeterminate (I)

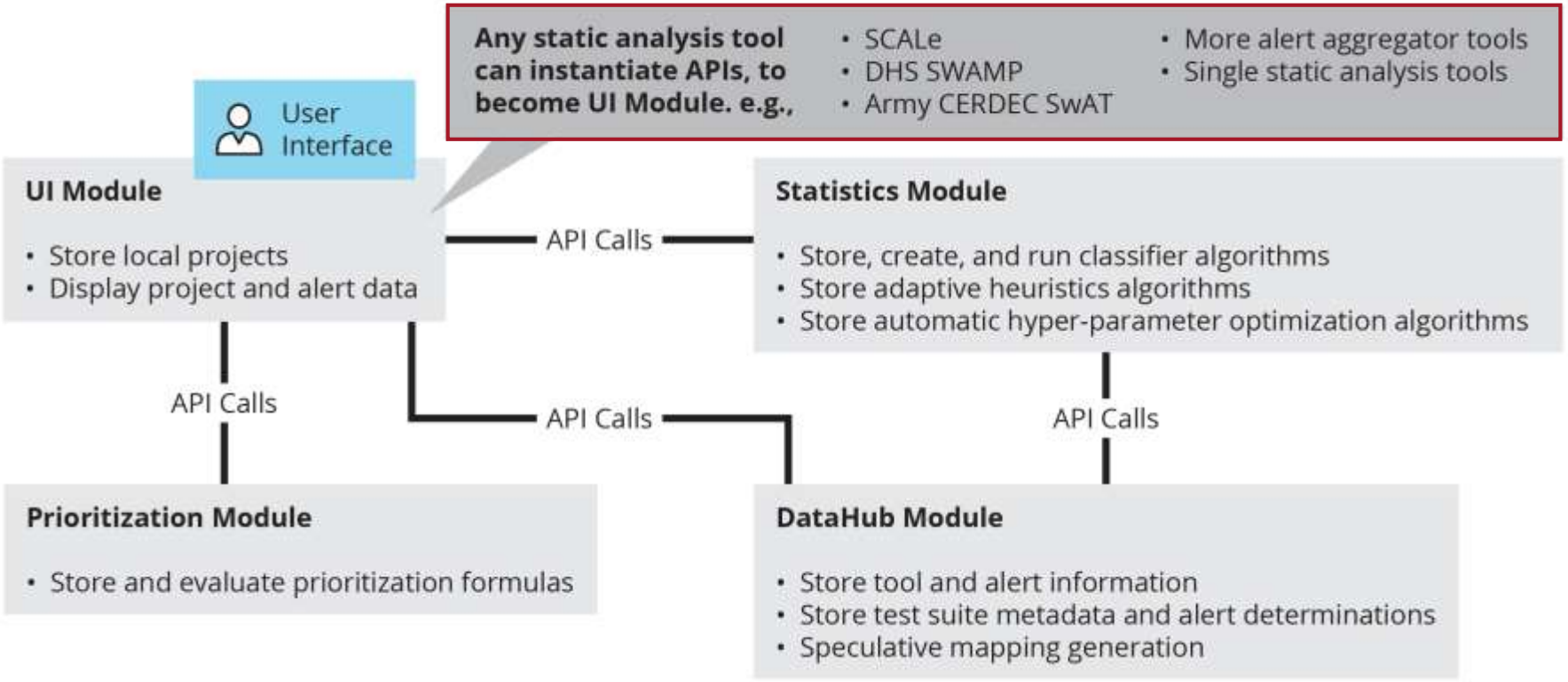


Prioritized, small number of alerts for manual audit

Long-term goal: Automated and accurate statistical classifier, intended to efficiently use analyst effort and to remove code flaws

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

SCAIFE Architecture



Publications and Presentations

Flynn and McNeil. “SCALe v.3: Automated Classification and Advanced Prioritization of Static Analysis Alerts”, SEI Blog, December 2018.

Flynn. “Improve Your Static Analysis audits Using CERT SCALe’s New Features”, Webcast, November 2018.

Flynn. “Automating Static Analysis Alert Handling with Machine Learning: 2016-2018”, presentation, September 2018.

Flynn, et al. “Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models”, SQUADE workshop at ICSE 2018, May 2018.

Flynn, et al. “Static Analysis Alert Test Suites as a Source of Training Data for Alert Classifiers”, SEI Blog, April 2018.

Flynn. “Prioritizing Security Alerts: A DoD Case Study”, SEI Blog, January 2017.

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

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.

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)

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

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;
}
```

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).

Predicting Security Flaws through Architectural Flaws

Problem Purpose: Use Architectural Analysis to predict and identify security flaws and estimate potential refactoring ROI

Approach: Retrospective analysis on open source projects that have issue data related to security flaws. Qualitative understanding of architectural and security flaw relationships, and quantitative analysis of correlations.

Results:

- Our analysis found weak correlation between architectural flaws and security defects at the distinct file level.
- Qualitative analysis and reproduction of previous study continues to suggest a relationship trend between architectural flaws and security defects.
- Collected lots of project data and developed tools for analysis

Problem

Software security defects → risk exposure and $$$.

Existing analysis methods have limitations:

- Some security flaws influenced by code structure and module relationships.
  - Not easily found or fixed locally.*

*"Analyzing Security Bugs from an Architectural Perspective," Kazman et al., 2017.

Approach – Today

- Source code
- Revision history
- Issue list

→

Tools
- Understand
- Titan
- DV8

[Generates: Design Structure Matrix (DSM)]

→

Architectural flaws and anti-patterns

→

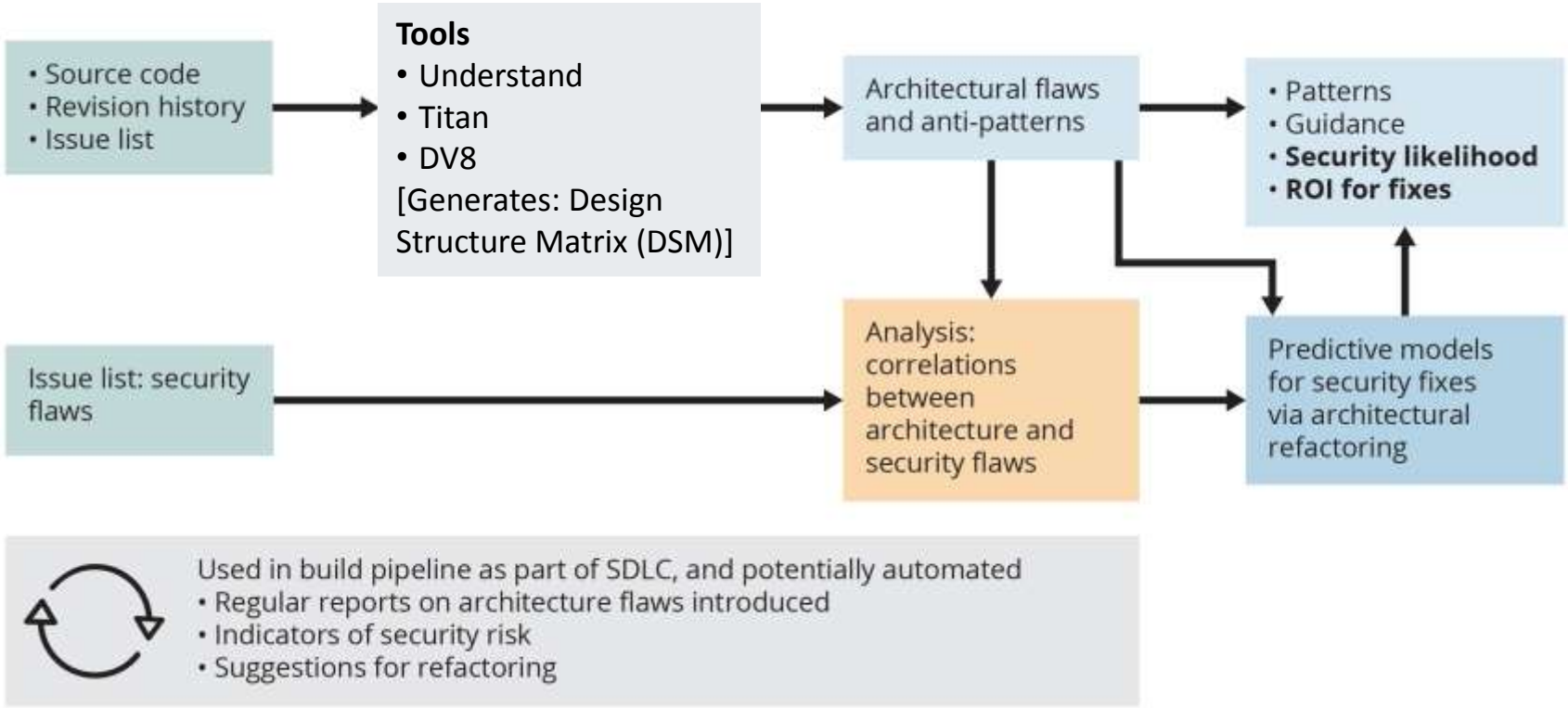
- Patterns
- Guidance

Improper Interface

		1	2	3	4	5	6	7	8	9	10
1	122654.content.browser.ssl.ssl_error_handler.h	(1)									
2	122654.content.browser.ssl.ssl_manager.h	I,6	(2)								
3	122654.content.browser.renderer_host.resource_dispatcher_host_impl.h	I,Pu,2	,2	(3)							
4	122654.content.browser.ssl.ssl_cert_error_handler.h	I,Pu,5	,12	,2	(4)						
5	122654.content.browser.ssl.ssl_error_handler.cc	U,I,6	,6	,2	C,I,4	(5)					
6	122654.content.browser.ssl.ssl_manager.cc	,4	C,I,11	I,2	C,I,10	C,4	(6)				
7	122654.content.browser.ssl.ssl_cert_error_handler.h	,4	C,U,I,23	I,2	I,10	,12	C,10	(7)			
8	122654.content.browser.rendererHost.socket_stream_dispatcher_host.h	I,Pu	,2		,2				(8)		
9	122654.content.browser.rendererHost.socket_stream_dispatcher_host.cc		I					C	U,I,9	(9)	
10	122654.content.browser.rendererHost.resource_dispatcher_host_impl.cc	,2	I,3	U,I,18	,2	,2	,2	C,3			(10)

C = Call; U = Use; I = Include; T = Type; S = Set; O = Override;
Pu = Public Inherit; ,# = # concurrent check-ins

Approach – Research and Vision



Data Collected and Analyzed

Chromium Browser

- ArchFlaw data
- Historical Issue type data (analyzed for “security” and “non-security” issues)
- Historical Commit data for all issues that were analyzed

Chromium Browser Issues x ArchFlaws	Has ArchFlaws	No ArchFlaws	Total
Files in Security Issues only	21	9	30
Files in Non-Security Issues only	14521	5038	19559
Files in Security & Non-Security Issues	798	41	839
Files in Neither Security nor Non-Security Issues	4591	3197	7788
Total	19931	8285	28216

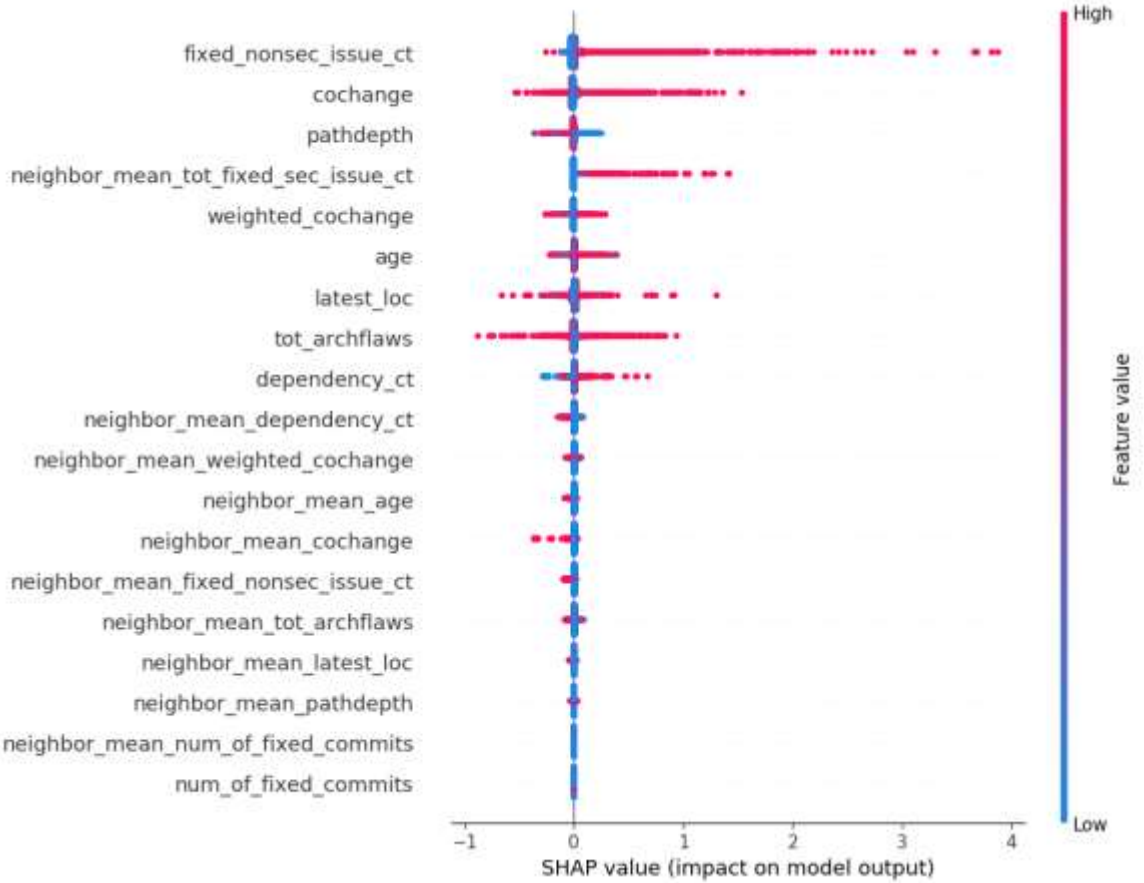
Quantitative Analysis

Chromium Browser
LightGBM

R-squared (all_features): 0.12

Top features:

- fixed_nonsec_issue_ct
- cochange
- pathdepth
- neighbor_mean_tot_fixed_sec_issue_ct
- weighted_cochange
- age
- latest_loc
- tot_archflaws



Future Recommendations

Analyze for additional features

- Per-file churn
- Time-split analysis of past security bug data
- Historical Structural Architectural Flaws
- Deep learning of source code (multiple Line-funded efforts)

Causal Analysis

- Can we identify strong correlations, and can we statistically show that features that are controllable can affect bugs and security bugs.

Architecture and Code Defect Relationship Dynamics

- Better modeling or measuring the dynamics of architecture and architectural flaws as it changes intentionally or incidentally with code changes, along with the introduction and removal of code defects.
- Control for distribution of number of commits and total churn

For More Information

Robert Schiela
Technical Manager
Secure Coding Initiative
Phone: (412) 268-3736
Email: rschiela@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