Secure Coding and Development Overview

Robert Schiela

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Document Markings

CERT Secure Coding Overview - Agenda

Vision and Purpose

Portfolio of Work

- Guidelines and Standards
- Tools
- Training
- Research
- Customers
- CERT & SEI Collaborations

Team

Financial Outlook

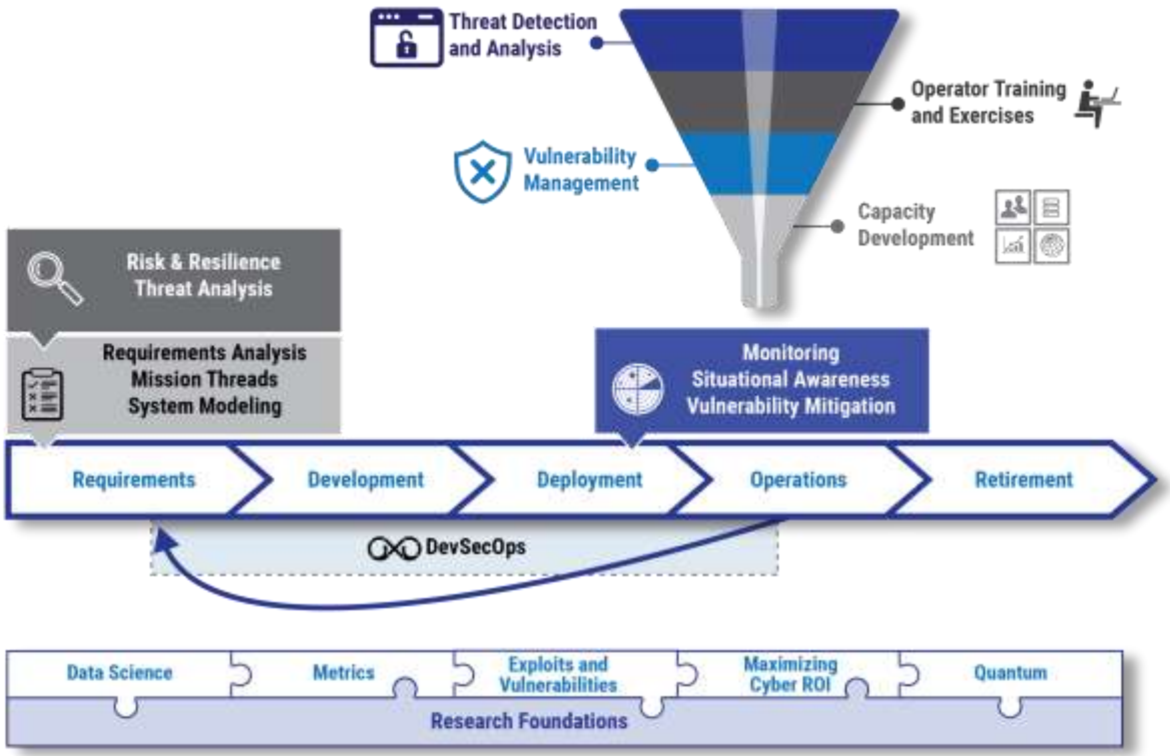
Future Directions

Secure Coding Vision and Purpose

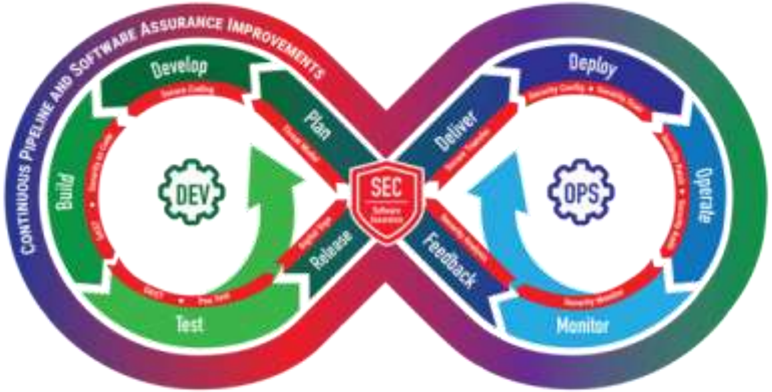
Vision: Software that is confidently free from security weaknesses caused by poor coding and development practices.

Purpose: To establish practical guidance, tools, and processes for assuring that organic and acquired software is developed without security weaknesses.

Improving the Full Lifecycle

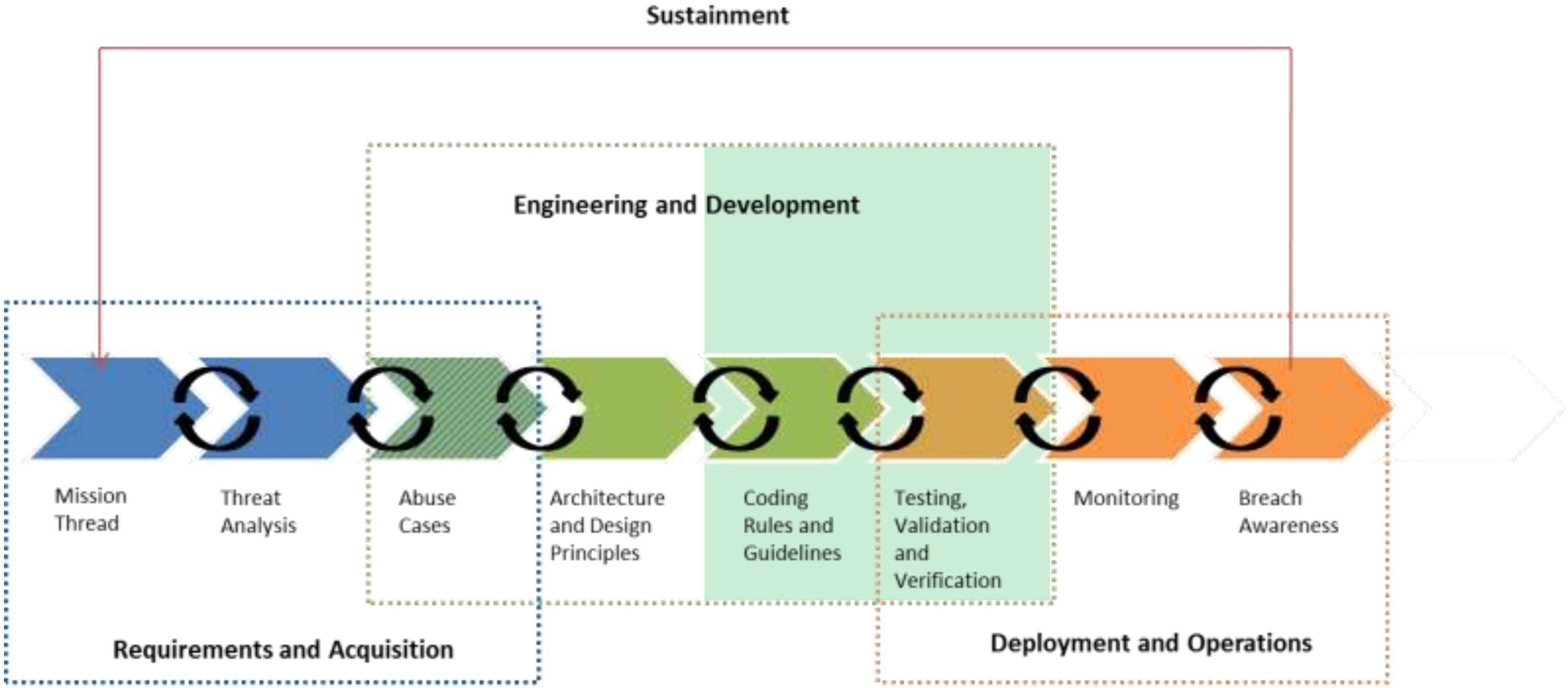


Engineering for Cyber Awareness



- Security Requirements Engineering
- System Modeling and Analysis
- Mission Threads
- Code security and cyber supply chain
  - Binary analysis
  - SBOM
- Architecture Analysis and Acquisition Guidance
  - Zero-trust
- DevSecOps improvements
  - Platform Independent Model – DevSecOps maturity
  - cATO
- Risk and resilience models and planning

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

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 Technical Specification 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

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 anti-patterns 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

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

Transition and Static Analysis Tools

Rosecheckers:

SEI developed tool that analyzes code for violations of CERT Secure Coding Standards that other tools didn’t cover.

Clang & Clang Static Analyzer:

Analyzers as part of the Clang/LLVM compiler suite. Checkers added to those tools by CERT and community to find violations to CERT Secure Coding Standards for C and C++.

Source Code Analysis Laboratory (SCALe):

Tool and process for verification and validation of secure coding practices and secure source code.

Source Code Analysis Laboratory

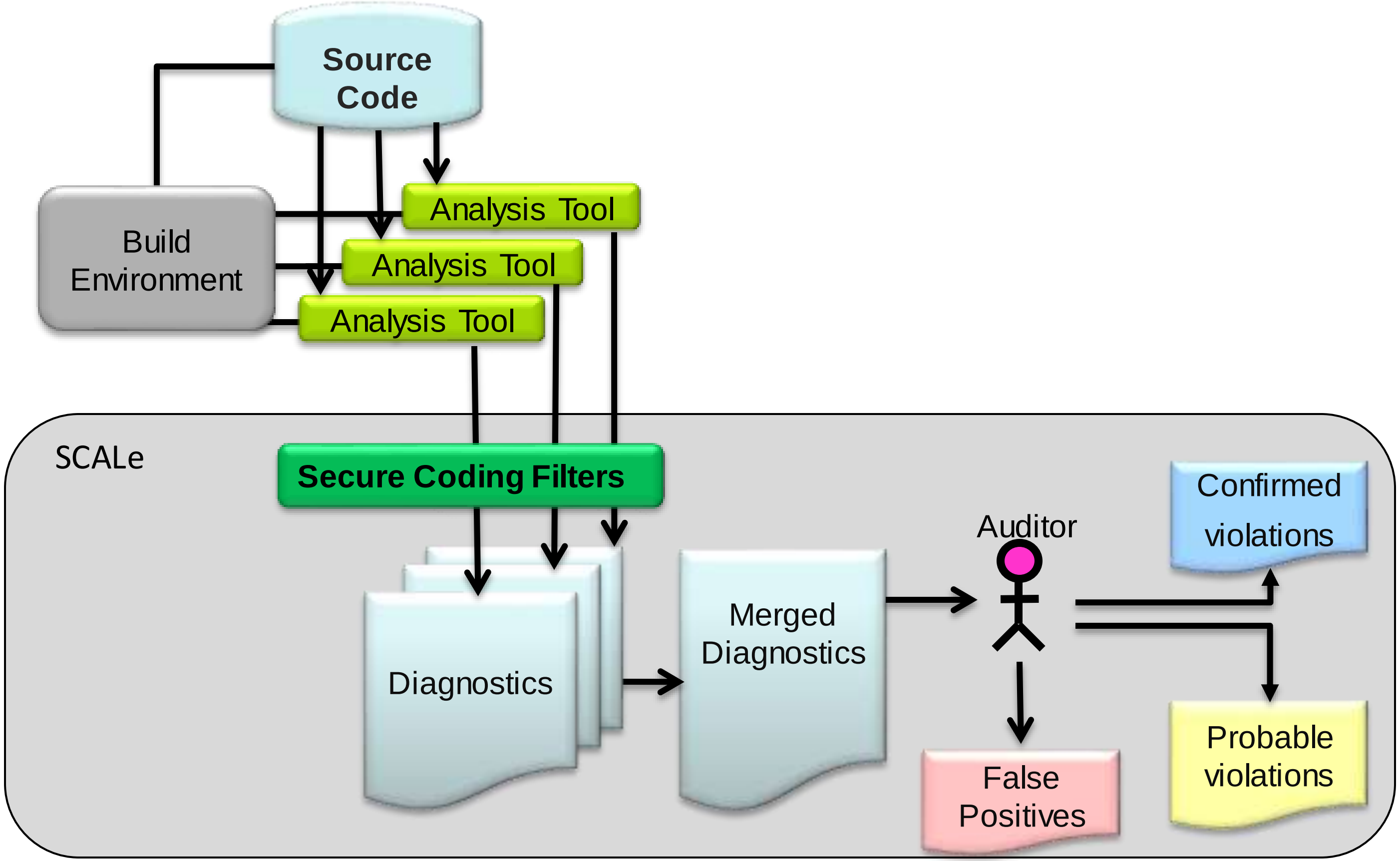
Source Code Analysis Laboratory (SCALe)

- 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.

SCALe Secure Coding Conformance Process



Secure Coding Training and 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

Secure Coding Research

Automated Code Repair (ACR)

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

Semantic Equivalence Checker for Binary Static Analysis

- Evaluating the effectiveness of using source static analysis tools on decompiled binary.

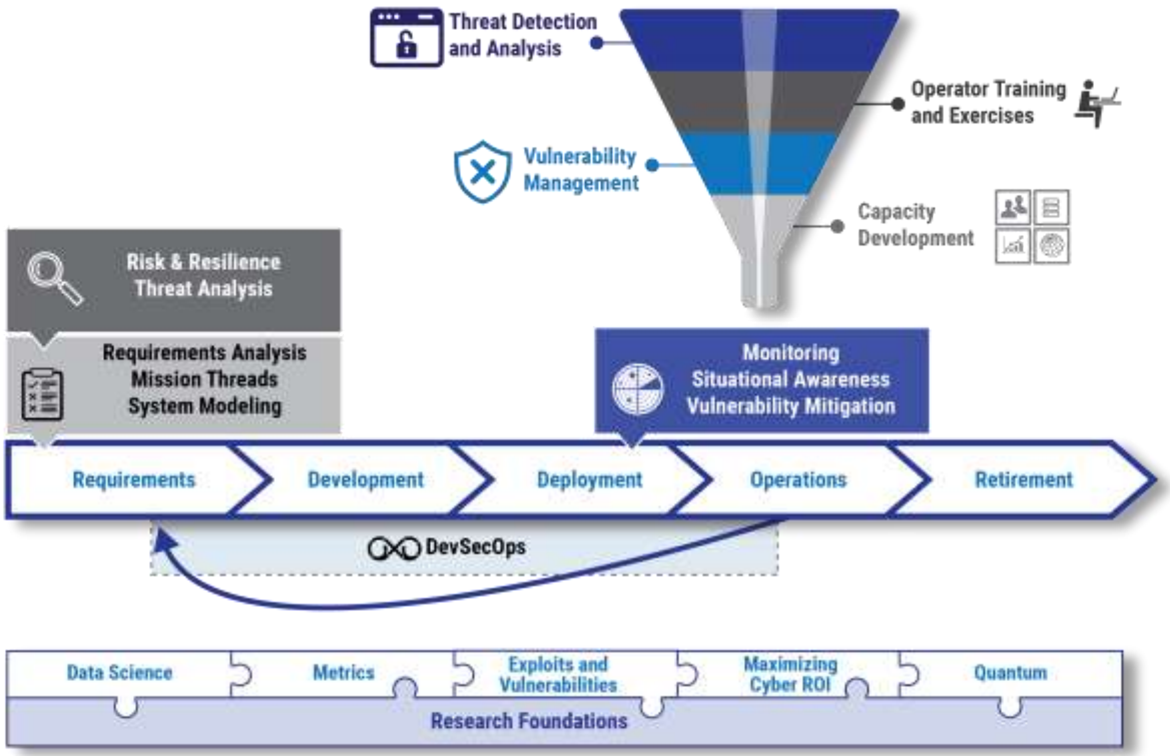
Automated Repair of Static Analysis Alerts (FY23-24)

- Automatically change code (or propose code changes) to resolve simple findings from Static Analysis tools, even if false positives. (reduce manual analysis when no cost)

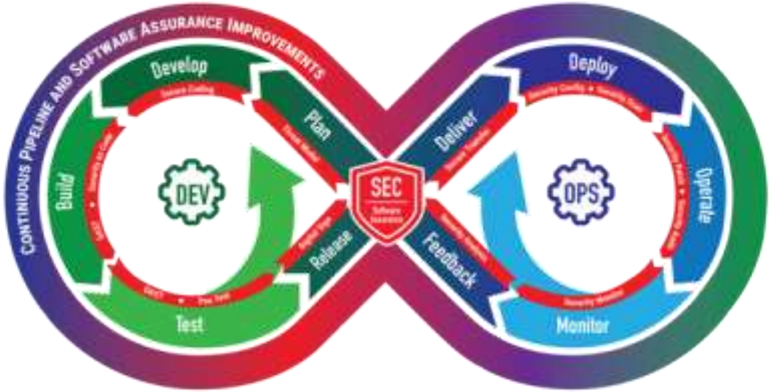
Detecting Inserted Malicious Code with Information Flows (FY23-24)

- Develop tool to detect exfiltration of sensitive data and sensitive operations driven by external input

Improving the Full Lifecycle



Engineering for Cyber Awareness



- Security Requirements Engineering
- System Modeling and Analysis
- Mission Threads
- Code security and cyber supply chain
  - Binary analysis
  - SBOM
- Architecture Analysis and Acquisition Guidance
  - Zero-trust
- DevSecOps improvements
  - Platform Independent Model – DevSecOps maturity
  - cATO
- Risk and resilience models and planning

Backup

CERT Division – Birthplace of Cybersecurity

Trusted
Conducting research for the U.S. Government in a non-profit, public-private partnership

Valued
Collaborating with military, industry, and academia globally to innovate solutions

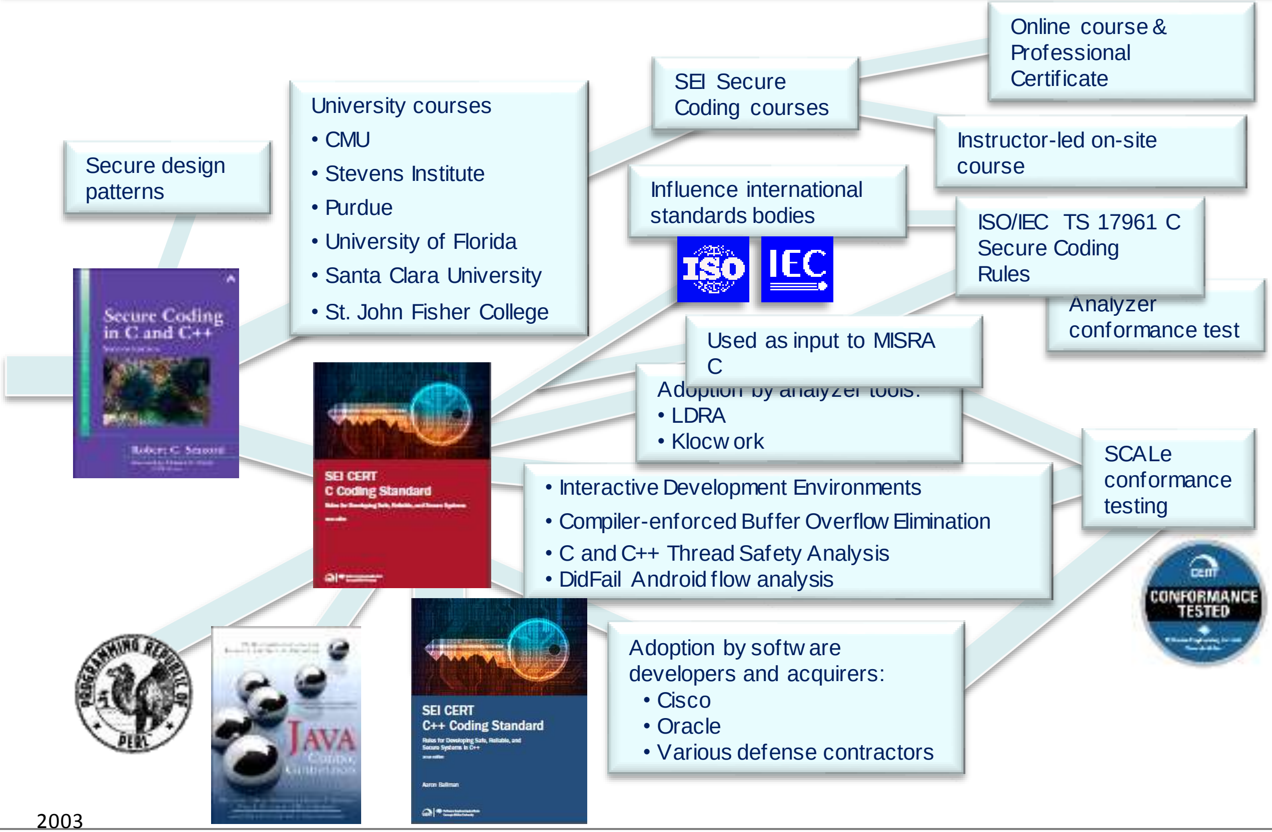
Relevant
Achieving technology and talent results for our mission partners

CERT Technical Areas

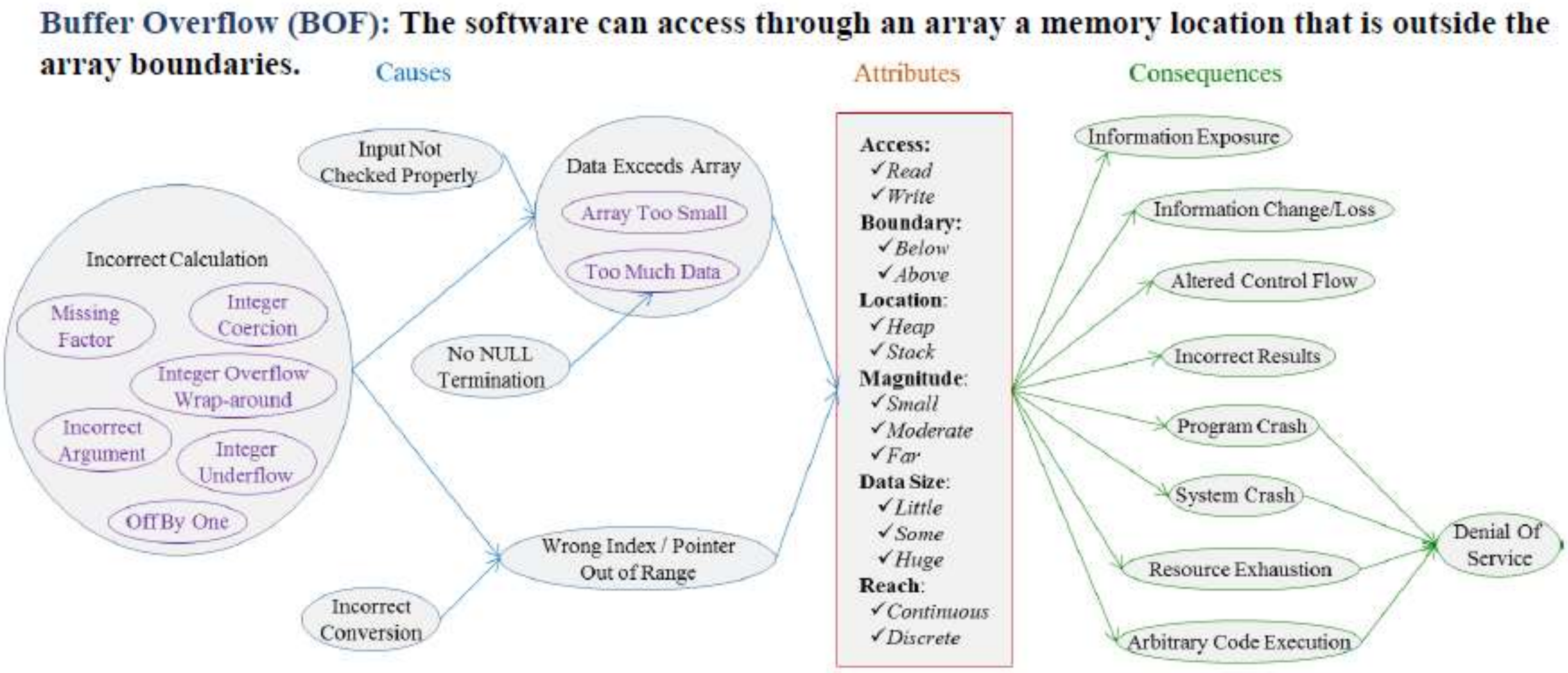
- Engineering for Cyber Resilience
- Measuring Risk and Optimizing Cybersecurity Investment
- Identifying and Countering Threats
- Cultivating Essential Skills and Abilities
- Situational Awareness and Network Analysis
- Cyber Operations Development

Secure Coding History

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

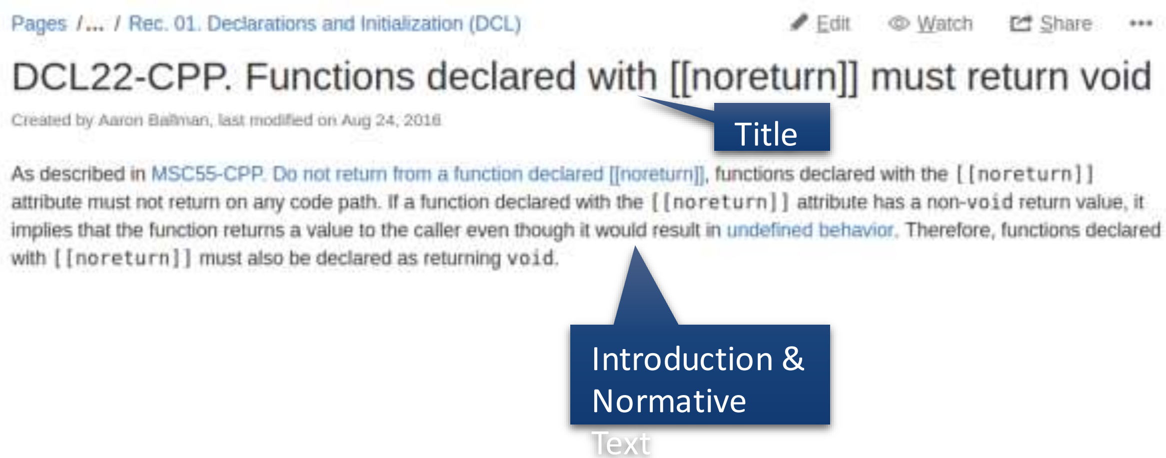


Buffer overflow has many causes



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

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

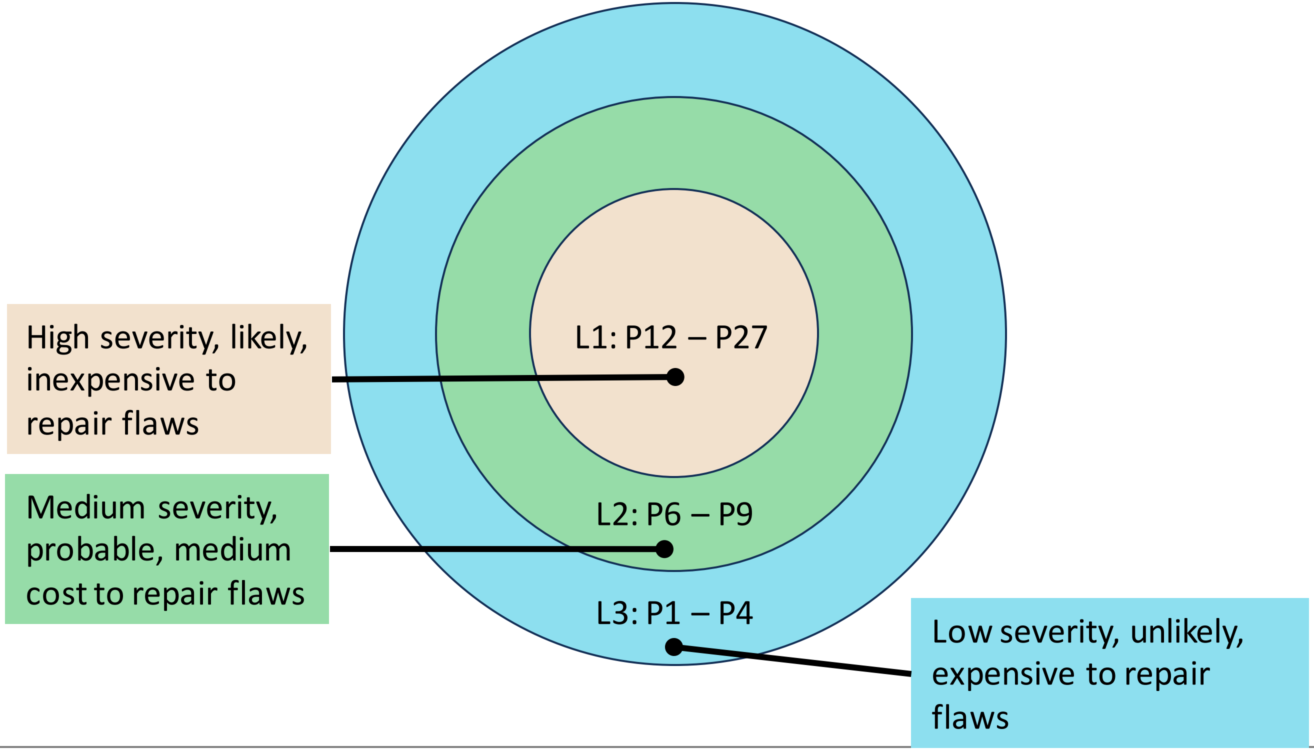
Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

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



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"

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.

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

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

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.

Automated Code Repair (ACR) tool as a black box

Input: Buildable codebase

Output: Repaired source code, suitable for committing to repository

We support C and plan to have limited support for C++

ACR Tool

Envisioned use of tool

- Use before every release build
- Use occasionally for debugging builds
- Intended for ordinary developers
- Can be a tool in the DevOps toolchain
- Can be used for legacy code and for new code

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)`

Research progression:

- FY16 Integer Overflow: (start + i) → UADD (start, i)
- FY17 Inference of Memory Bounds (out of bounds reads like HeartBleed): abort or warning
- F18-20 - Memory Safety: Fat Pointers
- FY21 - Combined Analysis for Source Code and Binaries for Software Assurance: Semantic Equivalence Checking of Decompiled Code (LLVM Focus)
- FY22 - Decompilation for Software Assurance: Analysis and Repair of Correctly Decompiled Functions (Ghidra Focus)

Why repair of source code instead of as a compiler pass?

Repair of source code	Repair as a compiler pass
Easily audited (if desired)	Must trust the tool.
Repairs can easily be tweaked to improve performance, if necessary.	Difficult to remediate performance issues caused by repair.
Changes to source code are frequent and easily handled.	Changes to the build process may be more difficult and error-prone.
Okay to do slow, heavy-weight static analysis; produces a persistent artifact.	Slowing down every test build is not okay.

Automated Code Repair (ACR) for Memory Safety

Software vulnerabilities constitute a major threat to DoD.

Memory violations are among the most common and most severe types of vulnerabilities.

- 15% of CVEs in the NIST NVD and 24% of critical-severity CVEs.
- iPhone iOS CVE-2019-7287 (exploited by Chinese government, according to https://techcrunch.com/2019/08/31/china-google-iphone-uyghur/)
- Android Stagefright (2015)
- CloudBleed (2017)

Huge volume of code is in use by DoD, with unknown number of vulnerabilities.

Automated Code Repair (ACR) for Memory Safety

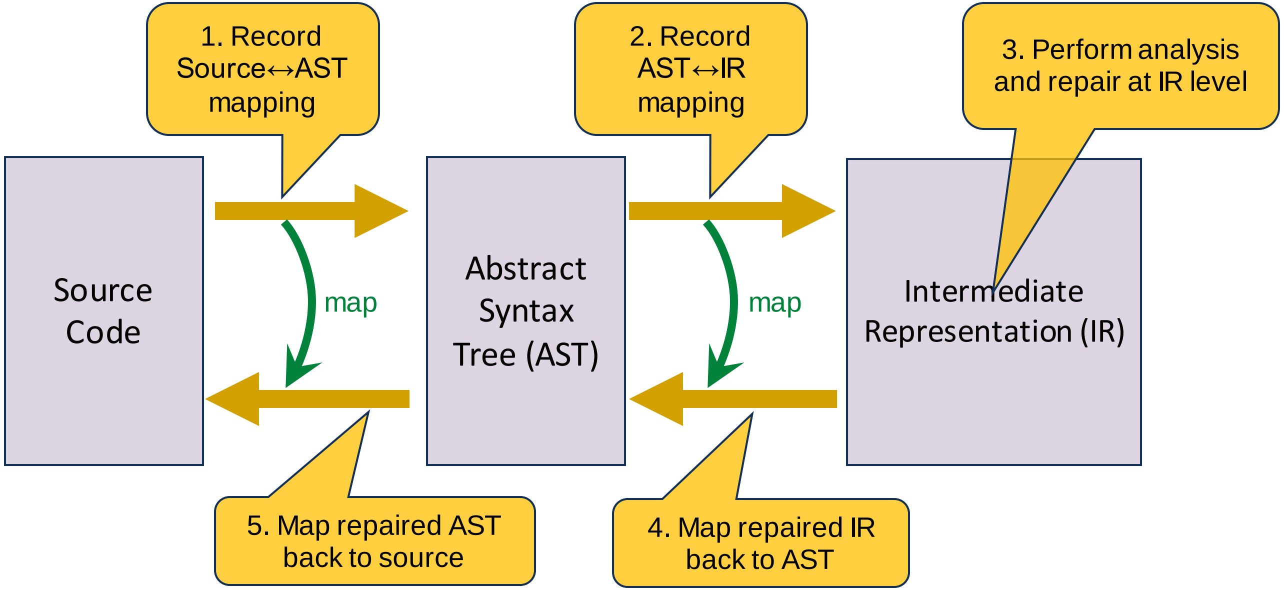
Solution: Automatically repair source code to assure memory safety.

- Abort program before memory violation.

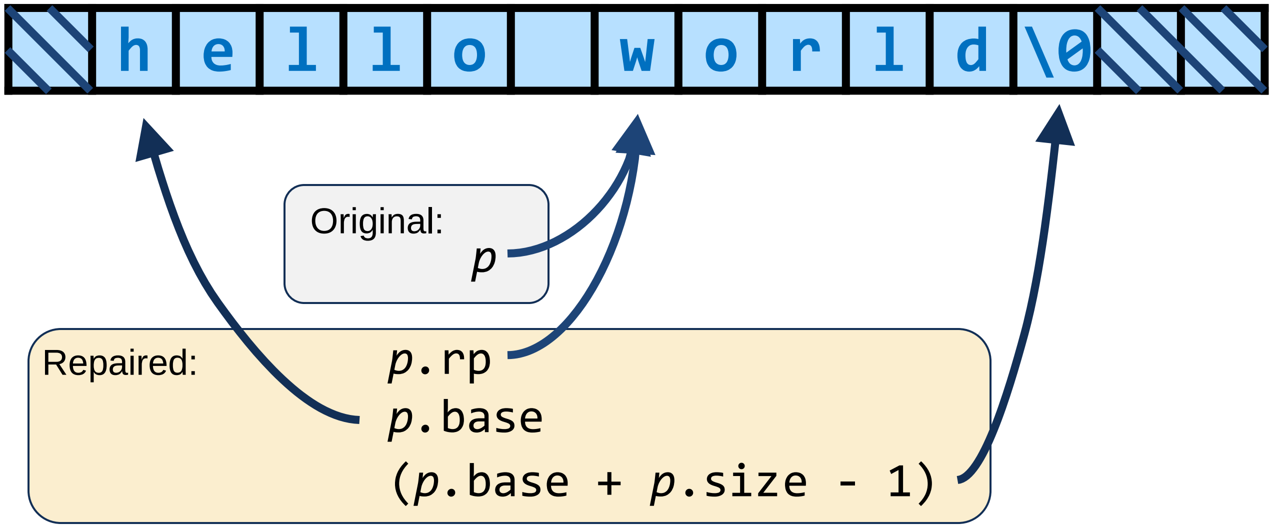
Approach:

- Transform source code to an intermediate representation (IR), retaining mapping.
- Try to assure that each memory access:
  - is within bounds (spatial memory safety), and
  - is not to a deallocated region (temporal memory safety — future work for FY20)
- If unable to assure, repair code to ensure memory safety.
  - Use fat pointers to store bounds information where possible.
- Map the repairs at the IR level back to source code.

Source Code Repair Pipeline



Fat pointer example



Example of tool output

Original Source Code

```c


#define BUF_SIZE 256
char nondet_char();

int main() {
    char* p = malloc(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *p = c;
        p = p + 1;
    }
    return 0;
}
```

Repaired Source Code

```c
#include "fat_header.h"
#include "fat_stdlib.h"
#define BUF_SIZE 256
char nondet_char();

int main() {
    FatPtr_char p = fatmalloc_char(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *bound_check(p) = c;
        p = fatp_add(p, 1);
    }
    return 0;
}
```

Integer Overflow

In 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

```
inline static size_t UADD(size_t lop, size_t rop) {
    size_t result;
    bool flag = __builtin_add_overflow(lop, rop, &result);
    if (flag) {result = SIZE_MAX;}
    return result;
}
```

- 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*]".

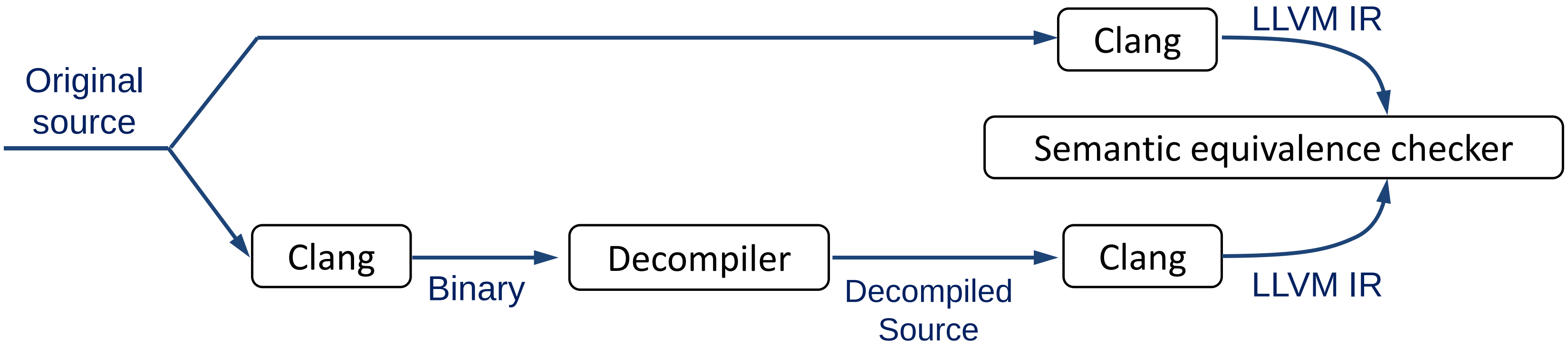
Solution & Approach: Static analysis to find & evaluate likely bounds.
(E.g., re-usable buffer: guess that upper bound for reading is the last position written.)

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).

Semantic Equivalence Checker

- The FY21/22 Project is doing semantic equivalence checking at the LLVM IR level:



- The LENS approach is sufficient for the FY21 goal (TRL < 5), but cannot be used for binaries for which the source code in unavailable.
- This proposed MTP will check equivalence at the machine-code level, enabling it to be used in the absence of original source code, as in a DoD environment (TRL ≥ 5).

Envisioned use of tool in practice by DoD