From Secure Coding to Secure Software

Robert Schiela
Mark Sherman

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Why Software Security?

Developed nations’ economies and defense depend, in large part, on the reliable execution of software.

Software is ubiquitous, affecting all aspects of our personal and professional lives.

Software vulnerabilities are equally ubiquitous, jeopardizing:

- personal identities
- intellectual property
- consumer trust
- business services, operations, and continuity
- critical infrastructures & government
Software and security failures are rampant

**Toyota Is Recalling Millions of Prius Hybrids to Fix a Software Bug**

*Source: Gizmo, Feb 12, 2014*

**iPhone Software Security Flaws Exposed**

*Source: Financial Times Limited, Feb 25, 2014*

**eBay Suffers Massive Security Breach, All Users Must Change Their Passwords**

*Source: New York Times, Feb 11, 2014*

*Source: Forbes (online), May 21, 2014; The Telegraph, May 22, 2014*
Software and security failures are expensive


Polling Question 1

What programming language are you most concerned about using securely?

- Ada
- Assembly
- C
- C++
- C#
- Java
- Java-Script
- Objective-C
- Perl
- PHP
- Python
- PL/SQL or SQL
- Ruby
- Swift
- Visual Basic
- Other
- Little to none developed in-house
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
• Uncontrolled Format String
• 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 Software Development

Secure software development starts with understanding insecure coding practices, and how these may be exploited.

Insecure designs can lead to “intentional errors”, that is, the code is correctly implemented but the resulting software contains a vulnerability.

Secure designs require an understanding of functional and non-functional software requirements.

Secure coding requires an understanding of implementation specifics.
Sources of Software Insecurity

Absent or minimal consideration of security during all life cycle phases
Complexity, inadequacy, and change
Incorrect or changing assumptions
Not thinking like an attacker
Flawed specifications & designs
Poor implementation of software interfaces
Unintended, unexpected interactions
  • with other components
  • with the software’s execution environment
Inadequate knowledge of secure coding practices
Polling Question 2

Does your organization use a coding standard for security?

• Yes
• No
• Maybe?
Coding rules – 2016 Edition

• Collected wisdom of programmers and tools vendors
  • Fed by community wiki started in Spring 2006
  • Over 1,500 registered contributors

• C Coding Standards
  Available as downloadable report
  http://cert.org/secure-coding/products-services/secure-coding-download.cfm

• Java Coding Standards
  Available as book

• C++, Perl, and “Current Standards”
  Available on Secure Coding Wiki
  https://www.securecoding.cert.org/
CWE Guidance

CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Description
The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

Detection Methods

Automated Static Analysis
This weakness can often be detected using automated static analysis tools. Modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis generally does not account for environmental considerations when reporting out-of-bounds memory operations. This can make it difficult for users to determine which warnings should be investigated first. For example, an analysis tool might report buffer overflows that originate from command line arguments in a program that is not expected to run with user input or other special privileges.

Effectiveness: High
detection techniques for buffer-related errors are more mature than for most other weakness types.

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Manual Analysis

Manual analysis can be useful for finding this weakness, but it might not achieve desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large.

Automated Static Analysis - Binary / Precord

According to SDR, the following detection techniques may be useful:
Buffer Overflows

General Prevention Techniques

A number of general techniques to prevent buffer overflows include:

- Code auditing (automated or manual)
- Developer training – bounds checking, use of unsafe functions, and group standards
- Non-executable stacks – many operating systems have at least some support for this
- Compiler tools – StackShield, StackGuard, and Libsafe, among others
- Safe functions – use strncat instead of strcat, strncpy instead of strcpy, etc.
- Patches – Be sure to keep your web and application servers fully patched, and be aware of bug reports relating to applications upon which your code is dependent.
- Periodically scan your application with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.
Buffer overflow has many causes

Buffer Overflow (BOF): The software can access through an array a memory location that is outside the array boundaries.

Causes:
- Input Not Checked Properly
- Data Exceeds Array
  - Array Too Small
  - Too Much Data
- No NULL Termination
- Wrong Index / Pointer Out of Range
- Incorrect Conversion
- Incorrect Calculation
  - Missing Factor
  - Integer Coercion
  - Integer Overflow Wrap-around
  - Incorrect Argument
  - Integer Underflow
  - Off By One

Attributes:
- Access:
  - Read
  - Write
- Boundary:
  - Below
  - Above
- Location:
  - Heap
  - Stack
- Magnitude:
  - Small
  - Moderate
  - Far
- Data Size:
  - Little
  - Some
  - Huge
- Reach:
  - Continuous
  - Discrete

Consequences:
- Information Exposure
- Information Change/Loss
- Altered Control Flow
- Incorrect Results
- Program Crash
- System Crash
- Resource Exhaustion
- Denial Of Service
- Arbitrary Code Execution

Learning from rules and recommendations

Rules and recommendations in the secure coding standards focus to improve behavior

The “Ah ha” moment:
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
An methodology for rule creation

Exploit language ambiguities

Analyze vulnerable programs

Systematically test the rules

And still consult with experts
Examine language definitions and standards for undefined, unspecified and implementation-defined behavior

3.4.3 undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

3.4.4 unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

Source: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (ISO 9899 - Programming Languages – C draft)
Examine vulnerable code for patterns

Malware repository with millions of unique, tagged artifacts

CERT Secure Coding Team has evaluated over 100M LOC

Vulnerability Notes Database
Advisory and mitigation information about software vulnerabilities

CERT Knowledgebase
The CERT Knowledgebase is a collection of internet security information related to incidents and vulnerabilities. The CERT Knowledgebase houses the public Vulnerability Notes Database as well as two restricted-access components:

- Vulnerability Card Catalog contains descriptive and referential information regarding thousands of vulnerabilities reported to the CERT Coordination Center.
- Special Communications Database contains briefs that provide advance warning and important information about vulnerabilities, intruder activity, or other critical security threats.
Implement candidate rules and run against sample code

• Focus rule when possible to
  • maximize true positive of weakness (tag bad code)
  • minimize false negative of weakness (don’t tag good code)

• Write program to evaluate source code for particular rule

• Run program against collection of known bad source code and a
collection of other (suspected good) code to check sensitivity and
specificity of results
Experience with systematic testing

• Candidate rule typical evaluation
  • 10 iterations of proposed rule and associated checker
    • 7 internal evaluations
    • 3 external evaluations

• Each evaluation iteration carried out against > 10M lines of representative code
  • Variety of domains
  • Variety of code quality

• As part of creating C++ standard, general methodology applied to generate 46 rules and corresponding Clang C++ checkers
  • 19 by CERT researchers, 27 by others
Tapping into expert knowledge for developing CERT coding standards

Engage community

Consensus on vulnerability and mitigation

Tool vendor analysis
New Rule Example

EXP46-C – Do not use a bitwise operator with a Boolean-like operand

```c
if (!((getuid() & geteuid() == 0)) { 
   /* ... */
}
```

```c
if (!((getuid() && geteuid() == 0)) { 
   /* ... */
}
```

CWE-480, Use of incorrect operator
Updated Rule Example

ARR38-C – Guarantee that library functions do not form invalid pointers

if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* Silently discard per RFC 6520 */

CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-121, Stack-based Buffer Overflow
CWE-123, Write-what-where Condition
CWE-125, Out-of-bounds Read
CWE-805, Buffer Access with Incorrect Length Value
Development and Verification

- Mission Thread
- Threat Analysis
- Abuse Cases
- Architecture and Design Principles
- Coding Rules and Guidelines
- Testing, Validation and Verification
- Monitoring
- Breach Awareness

Sustainment

Engineering and Development

Deployment and Operations

Requirements and Acquisition
DISA STIG Requirements

Application Security STIG Requirements:

- APP3550: CAT I – not vulnerable to integer arithmetic issues
- APP3560: CAT I – does not contain format string vulnerabilities
- APP3570: CAT I – does not allow command injection
- APP3590.1: CAT I – does not have buffer overflows
- APP3590.2: CAT I – does not use functions known to be vulnerable to buffer overflows
- APP2060.1: CAT II – development team follows a set of coding standards
- APP2060.2: CAT II – development team creates a list of unsafe functions to avoid and include in coding standards
- APP2120.3: CAT II – developers are provided with training on secure design and coding practices on at least an annual basis

Adopting Secure Coding Practices

Secure Coding Infrastructure
• Defining Secure Coding Practices
• Influencing Language Standards
• Influencing Tool Vendors

Processes
• Coding Standards and Security Standards, Testing

Technology
• Tools: IDE’s and Analyzers
• Automated transformation and remediation

People
• Workforce Development
Risk assessment is performed using failure mode, effects, and criticality analysis.

### Risk Assessment

<table>
<thead>
<tr>
<th><strong>Severity</strong></th>
<th>How serious are the consequences of the rule being ignored?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>low</td>
</tr>
<tr>
<td>2</td>
<td>medium</td>
</tr>
<tr>
<td>3</td>
<td>high</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Likelihood</strong></th>
<th>How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unlikely</td>
</tr>
<tr>
<td>2</td>
<td>probable</td>
</tr>
<tr>
<td>3</td>
<td>likely</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Cost</strong></th>
<th>The cost of mitigating the vulnerability.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>high</td>
</tr>
<tr>
<td>2</td>
<td>medium</td>
</tr>
<tr>
<td>3</td>
<td>low</td>
</tr>
</tbody>
</table>
Priorities and Levels

- High severity, likely, inexpensive to repair flaws
- Med severity, probable, med cost to repair flaws
- Low severity, unlikely, expensive to repair flaws
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:

<table>
<thead>
<tr>
<th>Provably nonconforming</th>
<th>The code is provably nonconforming if one or more violations of a rule are discovered for which no deviation has been allowed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conforming</td>
<td>The code is conforming if no violations of a rule can be identified.</td>
</tr>
<tr>
<td>Provably conforming</td>
<td>Finally, the code is provably conforming if the code has been verified to adhere to the rule in all possible cases.</td>
</tr>
</tbody>
</table>

Evaluation violations of a particular rule ends when a “provably nonconforming” violation is discovered.
Polling Question 3

What testing does your organization perform on your software?

• Static Analysis
• Dynamic Analysis
• Both
• None
Tools encourage application of secure coding

Moving rules into IDEs improves application of secure coding:

• Early feedback corrects errors on introduction.
• Exceptions are understood in context.

Adoption of secure coding IDEs

• help deploy tools
• training on tools
• extend tools to meet targeted needs
Static Testing – Source code analysis tools

Secure Code Analysis Laboratory (SCALe)

- C, C++, Java, PERL, Python, Android rule conformance checking
- Thread safety analysis
- Information flows across Android applications
- Operating system call flows
SCALe Multitool evaluation

Improve expert review productivity by focusing on high priority violations

Filter select secure coding rule violations

• Eliminate irrelevant diagnostics
• Convert to common CERT Secure Coding rule labeling

Single view into code and all diagnostics

Maintain record of decisions
Polling Question 4

Do you use multiple static analysis tools?

• Yes, and we use a tool diagnostic aggregator
• Yes, but we review the tool diagnostics separately
• No, we just use one static analysis tool
• No, we don’t use static analysis tools
## Select SCALe Assessments

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

Have you taken some training on secure coding practices?

- Yes, self-taught
- Yes, through an online-delivered program
- Yes, through an in-person delivered program
- Yes, through my academic education
- No
Secure Coding Professional Certificates

Online Courses with Exam and Certificates for C/C++ and Java
2 Courses (Secure Software Concepts & Secure Coding) and Exam
Onsite, instructor-led courses available for groups
The Secure Coding course is designed for C and C++ developers. It encourages programmers to adopt security best practices and develop a security mindset that can help protect software from tomorrow’s attacks, not just today’s.

Topics

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

http://www.sei.cmu.edu/training/p63.cfm
Participants gain a working knowledge of common programming errors that lead to software vulnerabilities, how these errors can be exploited, and mitigation strategies to prevent their introduction.

Objectives

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

The Java Secure Coding Course is designed to improve the secure use of Java. Designed primarily for Java SE 8 developers, the course is useful to developers using older versions of the platform as well as Java EE and ME developers. Tailored to meet the needs of a development team, the course can cover security aspects of

<table>
<thead>
<tr>
<th>Trust and Security Policies</th>
<th>Object Orientation</th>
<th>Serialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation and Sanitization</td>
<td>Methods</td>
<td>The Runtime Environment</td>
</tr>
<tr>
<td>The Java Security Model</td>
<td>Vulnerability Analysis Exercise</td>
<td>Introduction to Concurrency</td>
</tr>
<tr>
<td>Declarations</td>
<td>Numerical Types in Java</td>
<td>in Java</td>
</tr>
<tr>
<td>Expressions</td>
<td>Exceptional Behavior</td>
<td>Advanced Concurrency Issues</td>
</tr>
<tr>
<td></td>
<td>Input/Output</td>
<td></td>
</tr>
</tbody>
</table>

http://www.sei.cmu.edu/training/p118.cfm
Polling Question 6

Are you more concerned about the secure code that you develop or acquire/procure?

• Software we develop
• Source code we acquire/procure
• Third-party libraries we acquire/procure
• Complete software we acquire/procure and integrate
• All of the above
## Evolution of software development

<table>
<thead>
<tr>
<th>Custom development – context:</th>
<th>Shared development – ISVs (COTS) – context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Software was limited</td>
<td>• Function largely understood</td>
</tr>
<tr>
<td>▪ Size</td>
<td>▪ Automating existing processes</td>
</tr>
<tr>
<td>▪ Function</td>
<td>• Grown beyond ability for using organization to develop economically</td>
</tr>
<tr>
<td>▪ Audience</td>
<td>• Outside of core competitiveness by acquirers</td>
</tr>
<tr>
<td>• Each organization employed developers</td>
<td></td>
</tr>
<tr>
<td>• Each organization created their own software</td>
<td></td>
</tr>
</tbody>
</table>

Supply chain: practically none

Supply chain: software supplier
Development is now assembly

Collective development – context:
- Too large for single organization
- Too much specialization
- Too little value in individual components

Supply chain: long

Note: hypothetical application composition
Software supply chain for assembled software

Expanding the scope and complexity of acquisition and deployment

Visibility and direct controls are limited (only in shaded area)

Substantial open source contained in supply chain

• 90% of modern applications are assembled from 3rd party components
  • At least 75% of organizations rely on open source as the foundation of their applications
• Most applications are now assembled from hundreds of open source components, often reflecting as much as 90% of an application

Distributed development – context:
• Amortize expense
• Outsource non-differential features
• Lower acquisition (CapEx) expense

Supply chain: opaque

Sources: Geer and Corman, "Almost Too Big To Fail,"; login: (Usenix), Aug 2014; Sonatype, 2014 open source development and application security survey
Open source supply chain has a long path
Corruption in the tool chain already exists

- XcodeGhost corrupted Apple’s development environment

- Major programs affected
  - WeChat
  - Badu Music
  - Angry Birds 2
  - Heroes of Order & Chaos
  - iOBD2

Sources:
Open source is not secure

Heartbleed and Shellshock were found by exploitation

Other open source software illustrates vulnerabilities from cursory inspection

1.8 billion vulnerable open source components downloaded in 2015

26% of the most common open source components have high risk vulnerabilities

Reducing software supply chain risk factors

Software supply chain risk for a product needs to be reduced to acceptable level

- **Supplier Capability**: Supplier follows practices that reduce supply chain risks
- **Product Security**: Delivered or updated product is acceptably secure
- **Product Distribution**: Methods of transmitting the product to the purchaser guard against tampering
- **Operational Product Control**: Product is used in a secure manner
Supplier security commitment evidence

Supplier employees are educated as to security engineering practices
• Documentation for each engineer of training and when trained/retrained
• Revision dates for training materials
• Lists of acceptable credentials for instructors
• Names of instructors and their credentials

Supplier follows suitable security design practices
• Documented design guidelines
• Has analyzed attack patterns appropriate to the design such as those that are included in Common Attack Pattern Enumeration and Classification (CAPEC)
• Application of code signing techniques (interest in ISO 17960 – in early draft)
Evaluate a product’s threat resistance

What product characteristics minimize opportunities to enter and change the product’s security characteristics?

• Attack surface evaluation: Exploitable features have been identified and eliminated where possible
  - Access controls
  - Input/output channels
  - Attack enabling applications – email, Web
• Design and coding weaknesses associated with exploitable features have been identified and mitigated (CWE)
• Independent validation and verification of threat resistance
• Dynamic, Static, Interactive Application Security Testing (DAST, SAST, IAST)
• Delivery in or compatibility with Runtime Application Self Protection (RASP) containers
Establishing good product distribution practices

Recognize that supply chain risks are accumulated
• Subcontractor/COTS-product supply chain risk is inherited by those that use that software, tool, system, etc.

Apply to the acquiring organizations and their suppliers
• Require good security practices by their suppliers
• Assess the security of delivered products
• Address the additional risks associated with using the product in their context

Ideally open source is built with a compiler you trust
Maintain operational attack resistance

Who assumes responsibility for preserving product attack resistance with product deployment?

- Maintaining inventory of components
- Patching and version upgrades (component lifecycle management)
- Expanded distribution of usage
- Expanded integration

Usage changes the attack surface and potential attacks for the product

- Change in feature usage or risks
- Are supplier risk mitigations adequate for desired usage?
- Effects of vendor upgrades/patches and local configuration changes
- Effects of integration into operations (system of systems)
Where to start

<table>
<thead>
<tr>
<th>Anywhere</th>
</tr>
</thead>
<tbody>
<tr>
<td>No meaningful controls over what components are applications 76%</td>
</tr>
<tr>
<td>No coordination of security practices in various stages of the development life cycle 81%</td>
</tr>
<tr>
<td>No acceptance tests for third-party code 47%</td>
</tr>
</tbody>
</table>

Plenty of models to choose from

**BSIMM**: Building Security in Maturity Model

**CMMI**: Capability Maturity Model Integration for Acquisitions

**PRM**: SwA Forum Processes and Practices Group Process Reference Model

**RMM**: CERT Resilience Management Model

**SAMM**: OWASP Open Software Assurance Maturity Model

Contact Information

Robert Schiela
Technical Manager, Secure Coding
Telephone: +1 412.268.3736
Email: rschiela@cert.org

Mark Sherman
Technical Director, Cyber Security Foundations
Telephone: +1 412.268.9223
Email: mssherman@cert.org

Web Resources
http://www.sei.cmu.edu/
http://www.cert.org/
http://www.cert.org/secure-coding/
http://securecoding.cert.org/