Building Secure Software for Mission Critical Systems

Mark Sherman, PhD

Robert Schiela

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.

DM18-0096

Agenda

- **State of software**
- **Building software: the Secure Software Development Lifecycle**
  - Requirements
  - Development
  - Operations
- **Review**

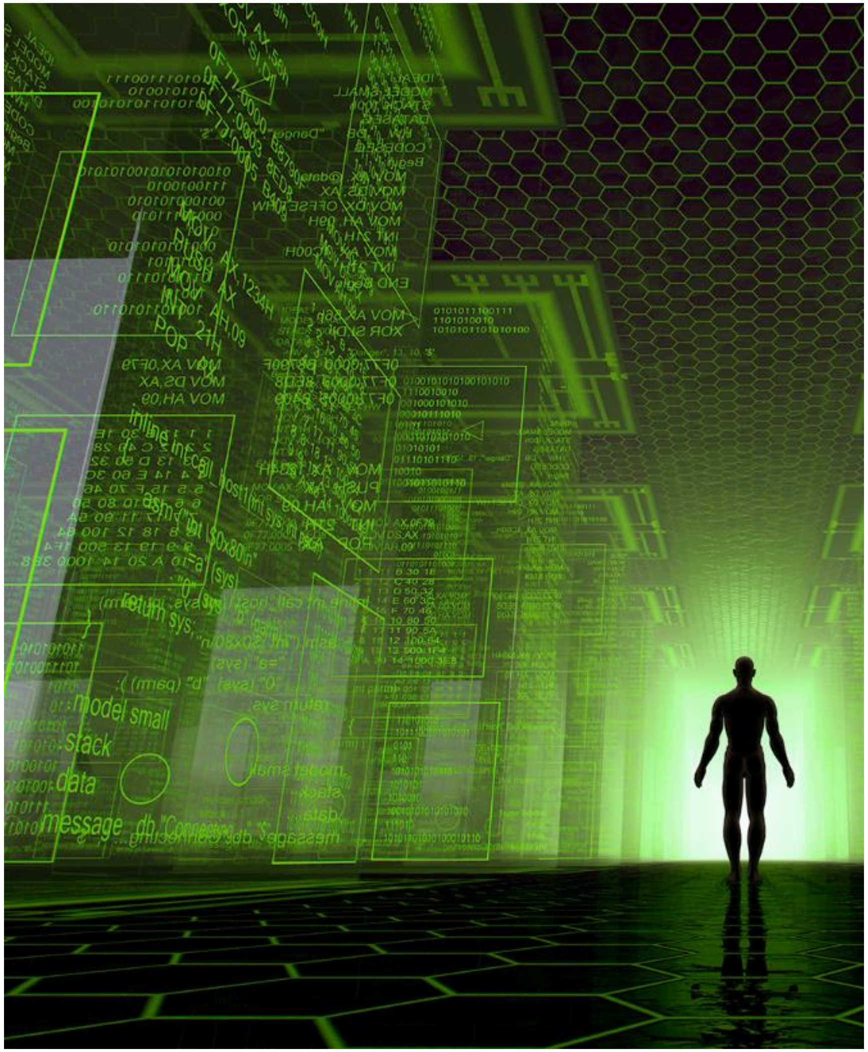
“Software is eating the world”

Marc Andreessen
Wall Street Journal
Aug 20, 2011

Software is the new Hardware

Source: http://www.wsj.com/articles/SB10001424053111903480904576512250915629460

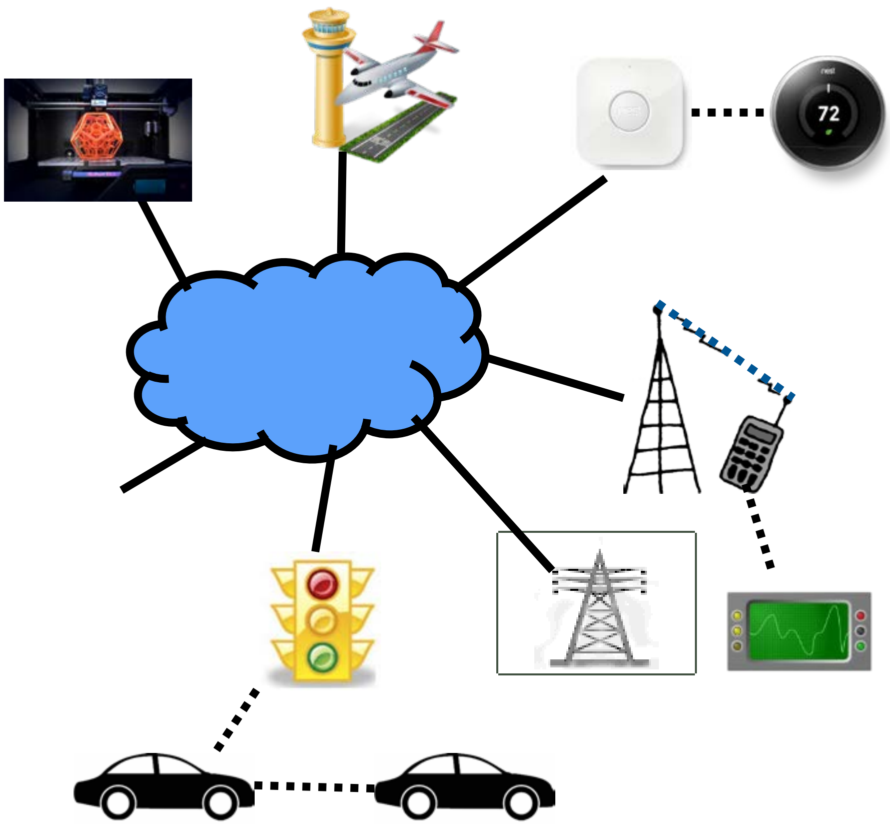
Software is the new hardware – IT



IT moving from specialized hardware to software, virtualized as

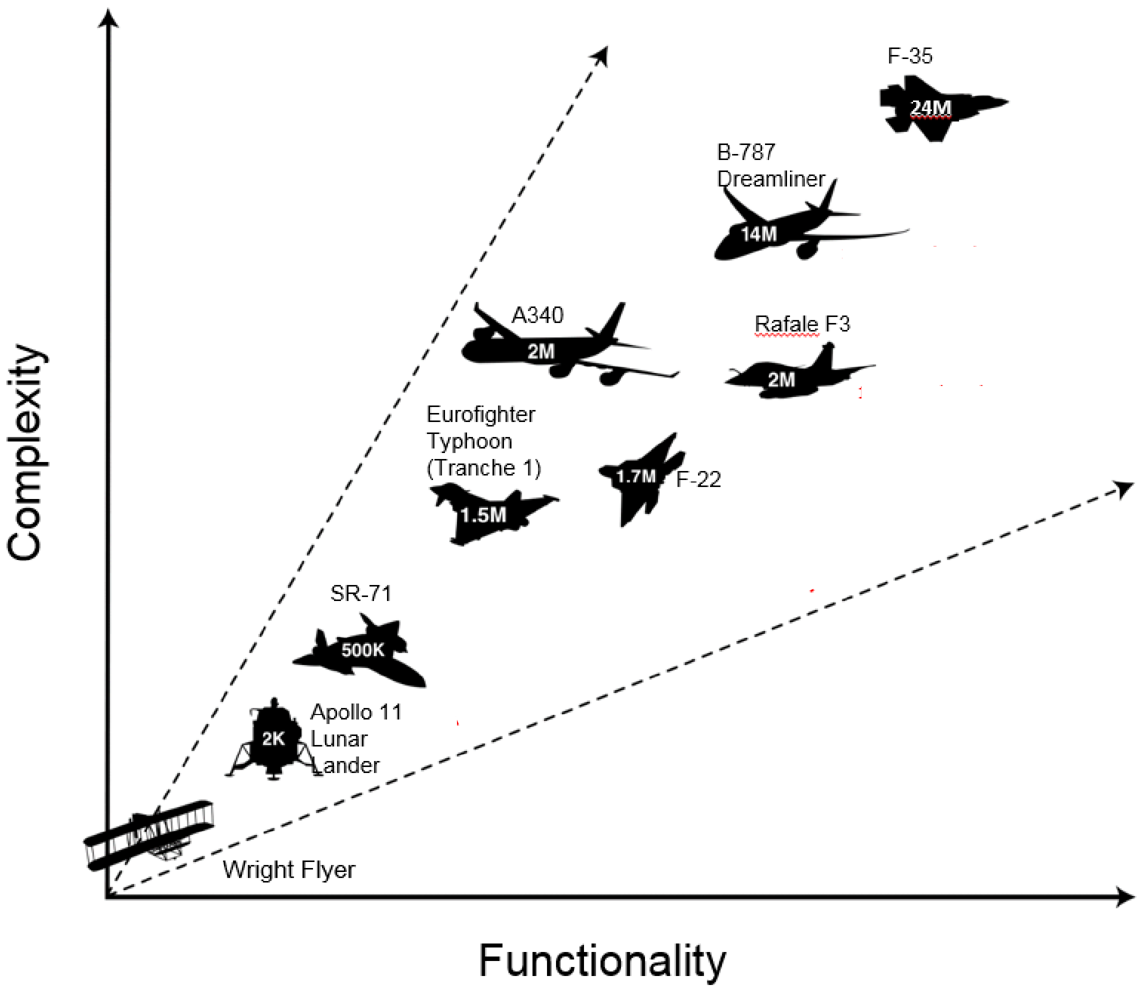
- Servers: virtual CPUs
- Storage: SANs
- Switches: Soft switches
- Networks: Software defined networks

Software is the new hardware – cyber physical



- Cellular
  - Main processor
  - Graphics processor
  - Base band processor (SDR)
  - Secure element (SIM)
- Automotive
  - Autonomous vehicles
  - Vehicle to infrastructure (V2I)
  - Vehicle to vehicle (V2V)
- Industrial and home automation
  - 3D printing (additive manufacturing)
  - Autonomous robots
  - Interconnected SCADA
- Aviation
  - Next Gen air traffic control
- Smart grid
  - Smart electric meters
  - Smart metering infrastructure
- Embedded medical devices

Mission function is increasingly delivered in software

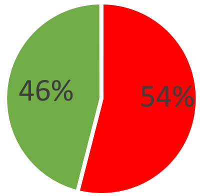


"The [F-35] aircraft relies on more than 20 million lines of code to "fuze" information from the JSF's radar, infrared cameras, jamming gear, and even other planes and ground stations to help it hunt down and hide from opponents, as well as break through enemy lines to blow up targets on the ground. …. But if the computer doesn't work, the F-35's greatest advertised advantages over existing rivals and future threats would suddenly become moot."
The Week, 2016

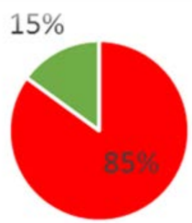
Source: Joseph Trevithick, http://theweek.com/articles/605165/f35-still-horribly-broken. Feb 26, 2016

Software vulnerabilities are ubiquitous

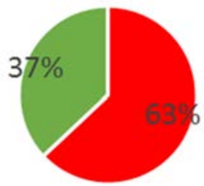
Existing Customer Premise Equipment (SOHO) typically vulnerable



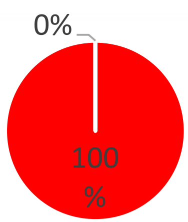
54% of tested routers are vulnerable to cross-site request forgery (CSRF)



85% of tested routers use non-unique default credentials



63% of tested routers are vulnerable to DNS spoofing attacks



100% of router firmware use BusyBox versions from 2011 or earlier and embedded Linux kernel versions from 2010 or earlier

Source: Land, J. "Systemic Vulnerabilities in Customer-Premises Equipment Routers," unpublished white paper, 2015

Steel furnaces have been successfully attacked

“Steelworks compromise causes massive damage to furnace.

One of the most concerning was a targeted APT attack on a German steelworks which ended in the attackers gaining access to the business systems and through them to the production network (including SCADA). The effect was that the attackers gained control of a steel furnace and this caused massive damages to the plant.”

Source: Sources: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2014.pdf?__blob=publicationFile; http://www.resilienceoutcomes.com/state-ict-security/

Electric grid under attack

BlackEnergy trojan strikes again: Attacks Ukrainian electric power industry

BY ROBERT LIPOVSKY IN COOPERATION WITH ANTON CHEREPANOV POSTED 4 JAN 2016 - 12:49PM

CYBERCRIME | TAGS | BLACKENERGY | UKRAINE

On December 23rd, 2015, around half of the homes in the Ivano-Frankivsk region in Ukraine (population around 1.4 million) were left without electricity for a few hours. According to the Ukrainian news media outlet TSN, the cause of the power outage was a "hacker attack" utilizing a "virus".

Source: http://www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacks-ukrainian-electric-power-industry/

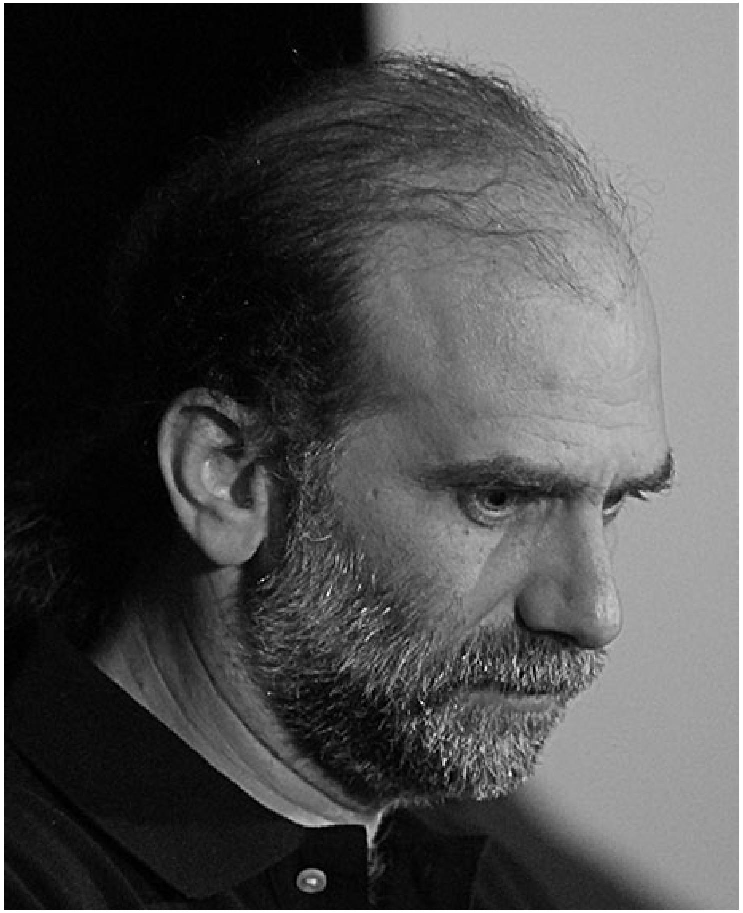
Weapons platforms potential cyber attack targets

“The [Joint Strike Fighter] aircraft relies on more than 20 million lines of code … In November 2015, the Pentagon canceled a cyber test because of worries it would, unsurprisingly, damage [the Autonomic Logistics Information System that identifies broken parts and other faults].”

The Week, 2016

Sources: https://www.dvidshub.net/image/935698/aerial-refueling-f-35-lightning-ii-joint-strike-fighters-eglin-afb-fla;
Joseph Trevithick, http://theweek.com/articles/605165/f35-still-horribly-broken. Feb 26, 2016

An ounce of prevention ….



"We wouldn't have to spend so much time, money, and effort on network security if we didn't have such bad software security."

Bruce Schneier in Viega and McGraw, "Building Secure Software," 2001

Source: Washington Post, March 19, 2014, http://www.washingtonpost.com/business/economy/toyota-reaches-12-billion-settlement-to-end-criminal-probe/2014/03/19/5738a3c4-af69-11e3-9627-c65021d6d572_story.html; http://www.greene-broillet.com/Articles/Toyotasuddenacceleration.shtml

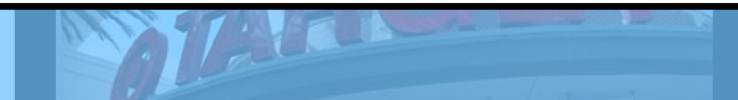
Software and security failures are expensive

Sections

The Washington Post

Business

Toyota reaches $1.2 billion settlement to end probe of accelerator problems



[GREENE BROILLET & WHEELER, LLP]

WHERE SUCCESS IS A TRADITION®

Toyota Sudden Acceleration Defect Case: $1.1 Billion Settlement

EMAIL FACEBOOK TWITTER SAVE MORE

Target on Friday revised the number of customers whose personal information was stolen in a widespread data breach during the holiday season, now reporting a range of 70 million to 110 million people.

The stunning figure represents about a third of all American adults at the low end, and is nearly three times as great as the company's original estimate at the upper end. The theft is one of the largest ever of retail data.

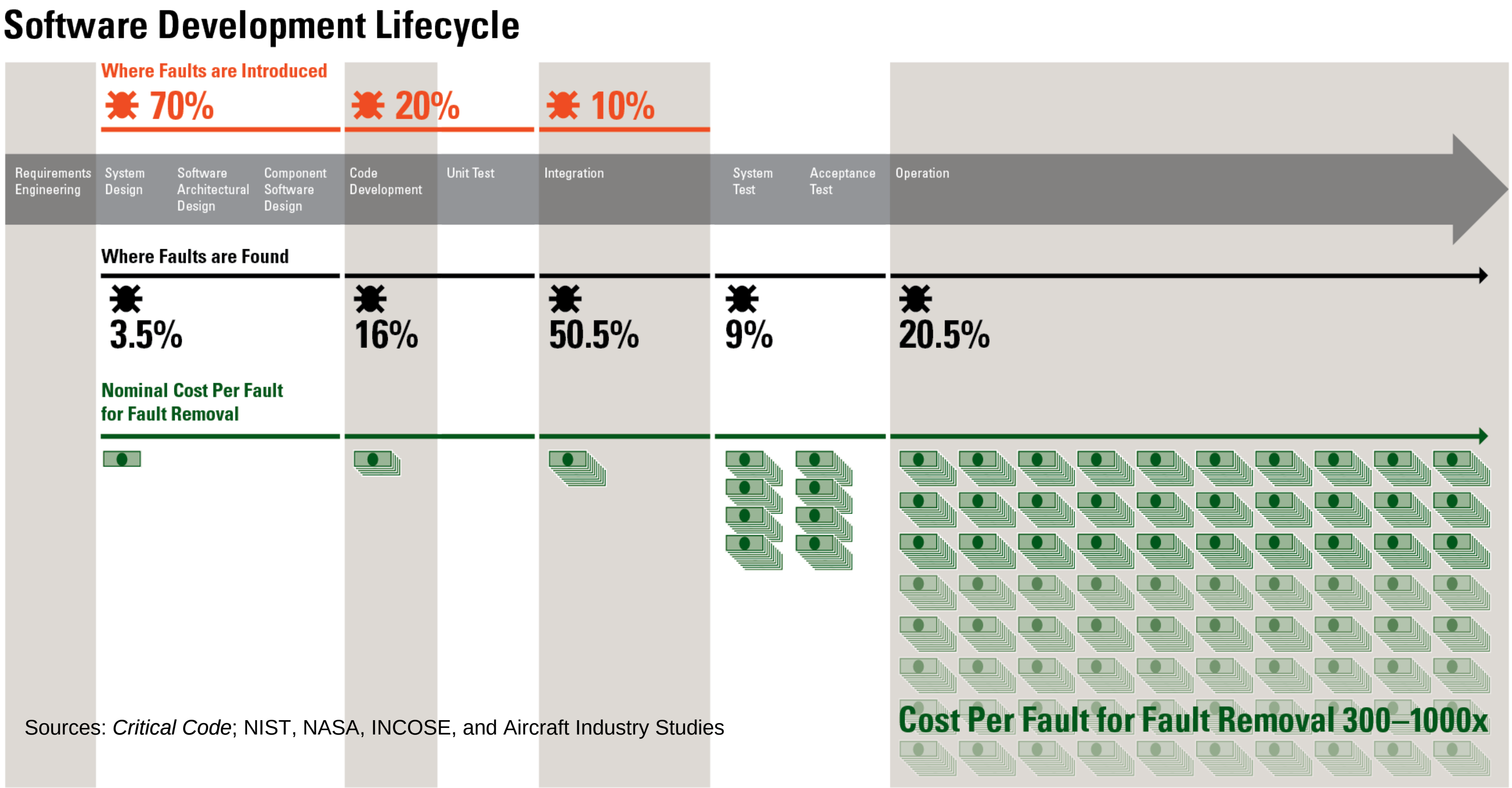
Source: New York Times, Jan 10, 2014

Average cost in a breach:
US$188 per record

Source: Ponemon Institute, "2013 Cost of Data Breach Study: Global Analysis", May 2013

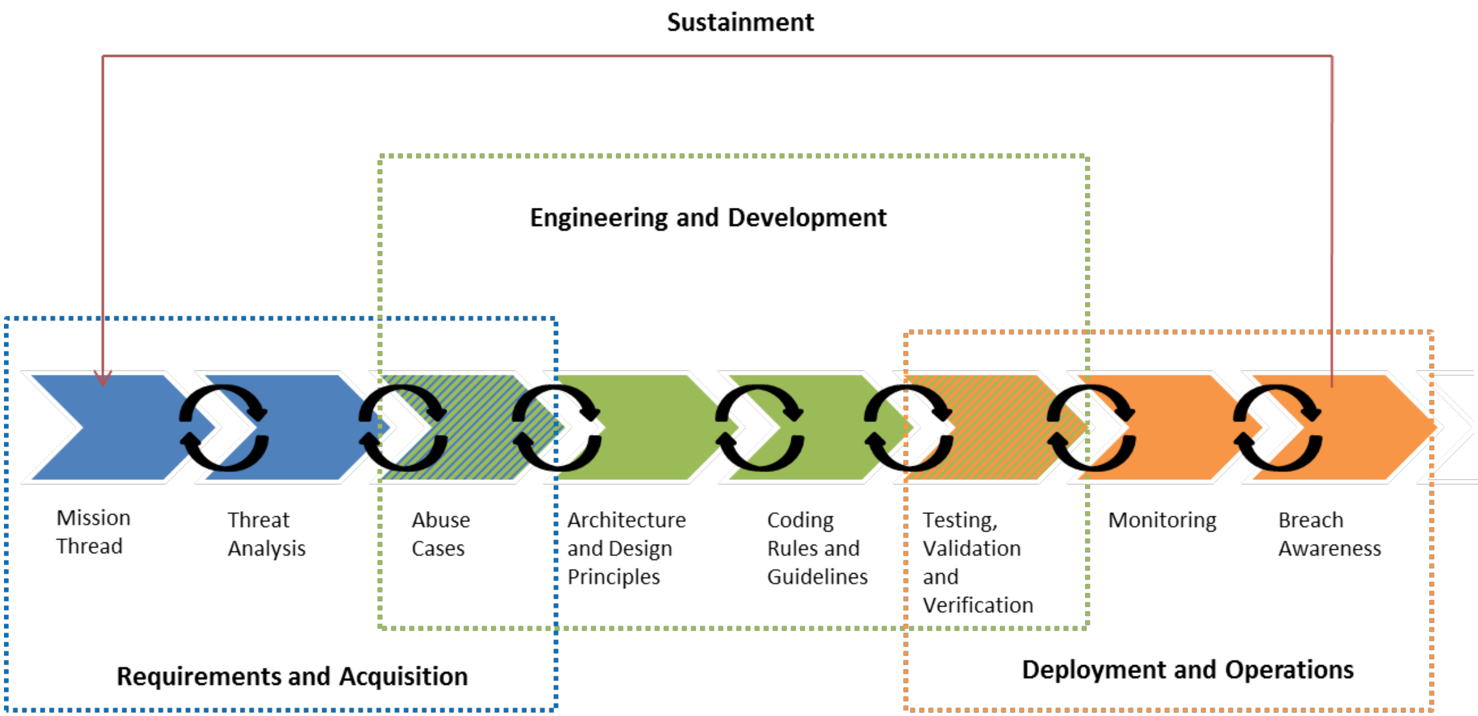
Catching software faults early saves money

Faults accounts for 30–50% percent of total software project costs

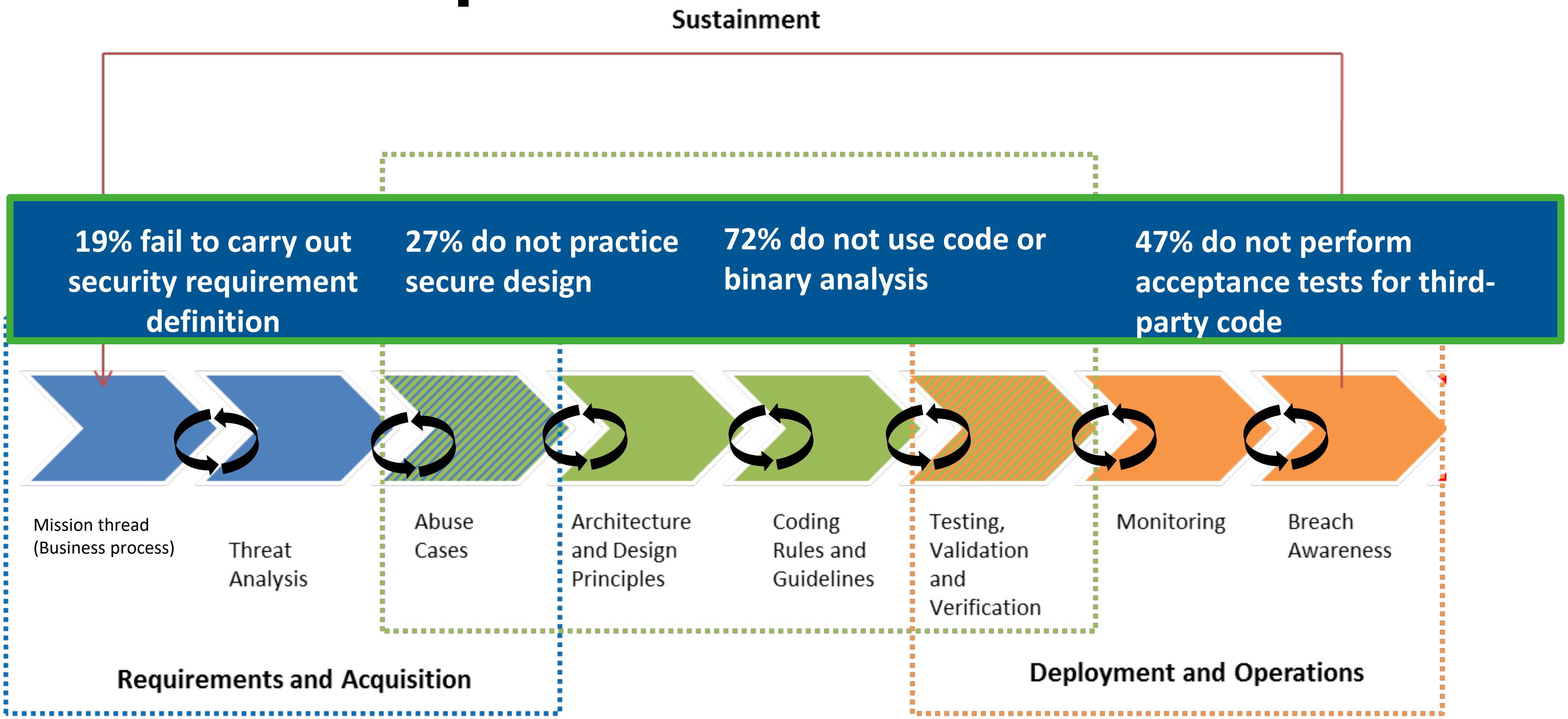


Sources: *Critical Code*; NIST, NASA, INCOSE, and Aircraft Industry Studies

Security is a lifecycle issue



Room for improvement



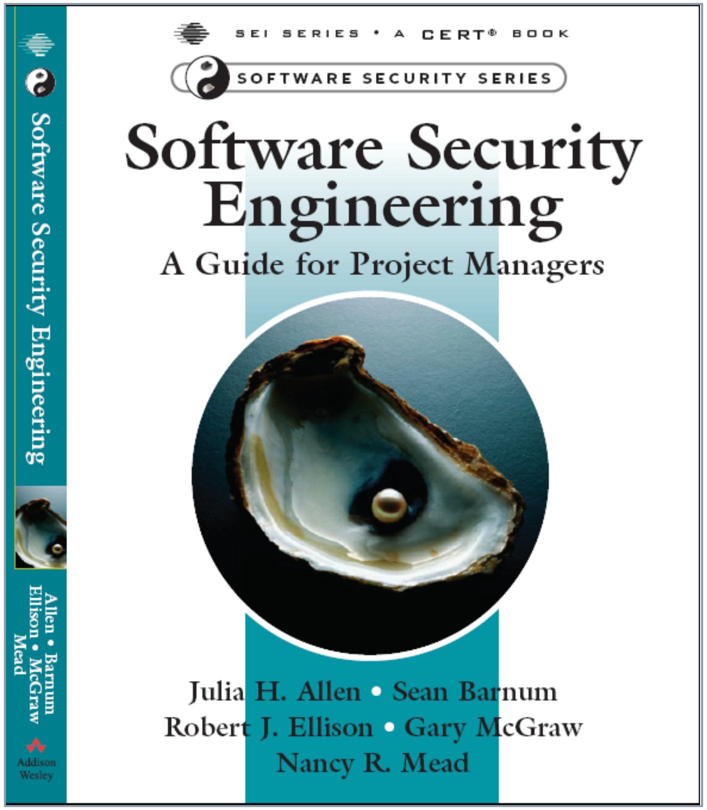
More than 81% do not coordinate their security practices in various stages of the development life cycle.

Sources: Forrester Consulting, “State of Application Security,” January 2011; Wendy Nather, Research Director, 451 Research, “Dynamic testing: Why Tools Alone Aren't Enough, March 25, 2015”

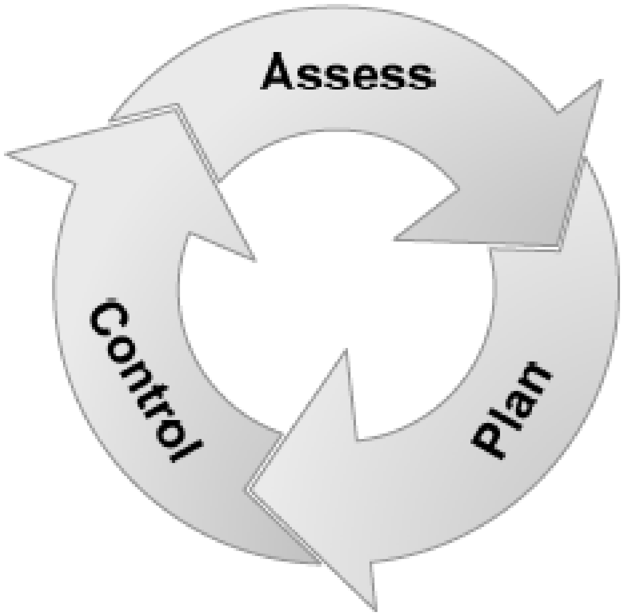
Software Security Engineering: A Guide for Project Managers

Contains an introduction to software security engineering and guidance for project managers

- Derives material from DHS SwA "Build Security In" web site
- Provides a process focus for projects delivering software-intensive products and systems



Organizational readiness: Mission Risk Diagnostic (MRD)



The MRD assesses risk in interactively complex, socio-technical systems

- Projects and programs
- Business processes and mission threads
- IT processes

MRD purpose:

- Gauge the extent to which a system is in position to achieve its mission and objective(s)

MRD assessment delivery:

- Expert-led assessment
- Self-assessment

Software Assurance Framework (SAF)

What

- Defines software assurance practices for acquiring and developing assured software products

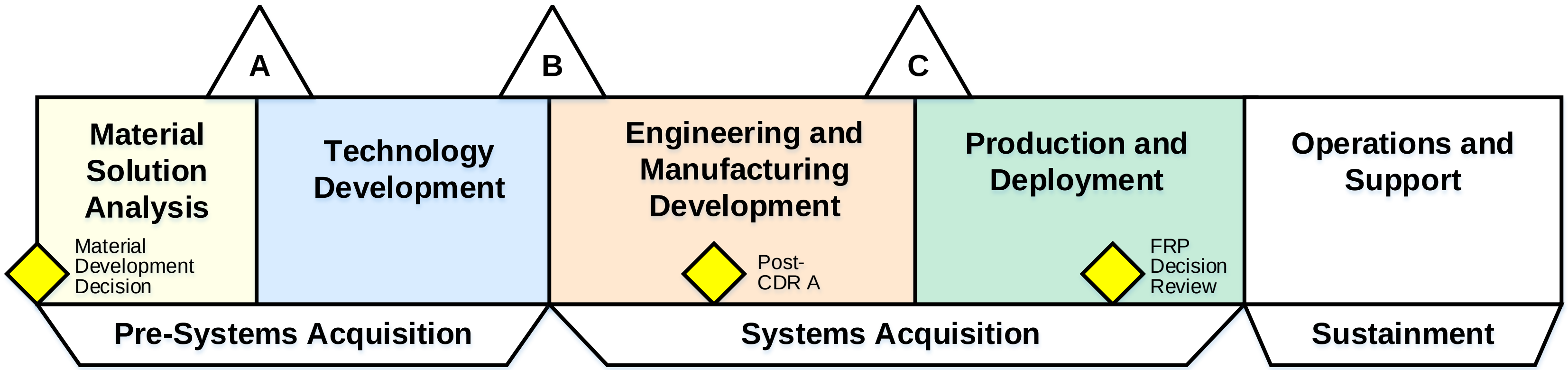
Why

- Improve software assurance practice in acquisition programs
- Enhance software assurance service provided by third parties

Benefits

- Establish confidence in a program's ability to acquire software-reliant systems across the life cycle and supply chain
- Reduce cybersecurity risk of deployed software-reliant systems

SAF: *Acquisition Lifecycle Focus*



- The DoD acquisition lifecycle is the organizing structure for the SAF.
- Best practices for software assurance are mapped to the lifecycle.
- The SAF is consistent with DoD and industry policies for software assurance (e.g., NIST 800-53, DoD 5000-2, BSIMM).

SAF: *Nine Practice Areas*

					Focus
1. Governance Infrastructure Practices					Governance Infrastructure
2. Materiel Solution Analysis (MSA) Practices	3. Technology Development (TD) Practices	4. Engineering and Manufacturing Development (EMD) Practices	5. Production and Deployment (PD) Practices	6. Operations and Support (O&S) Practices	Acquisition Lifecycle Assurance
7. Secure Software Development Practices		8. Secure Software Operation Practices			Software Security
9. Software Security Infrastructure Practices					Software Security Infrastructure

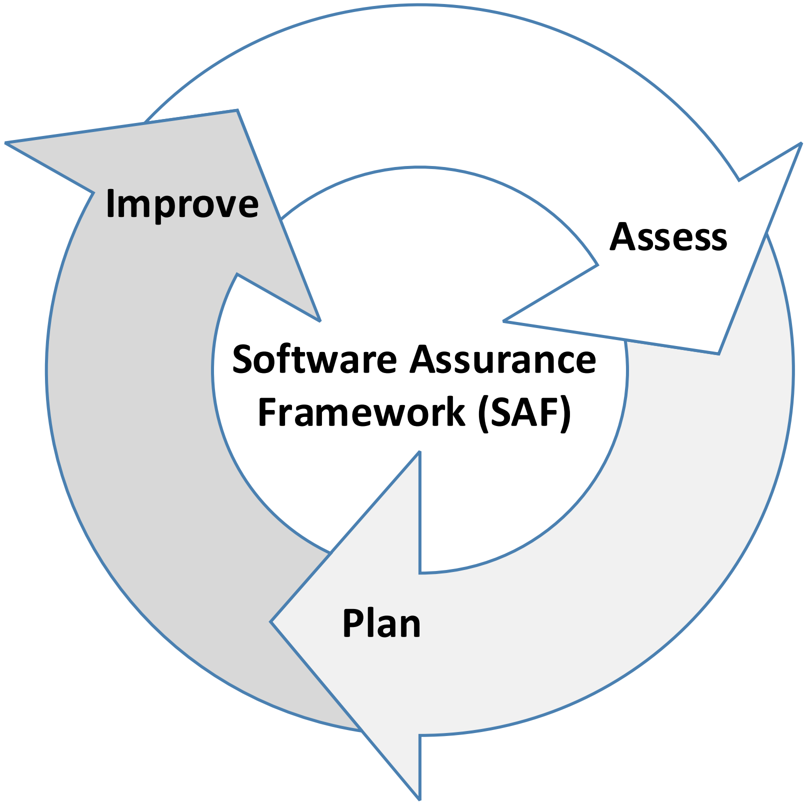
SAF: *Basis for Assessment and Improvement*

Acquisition Programs

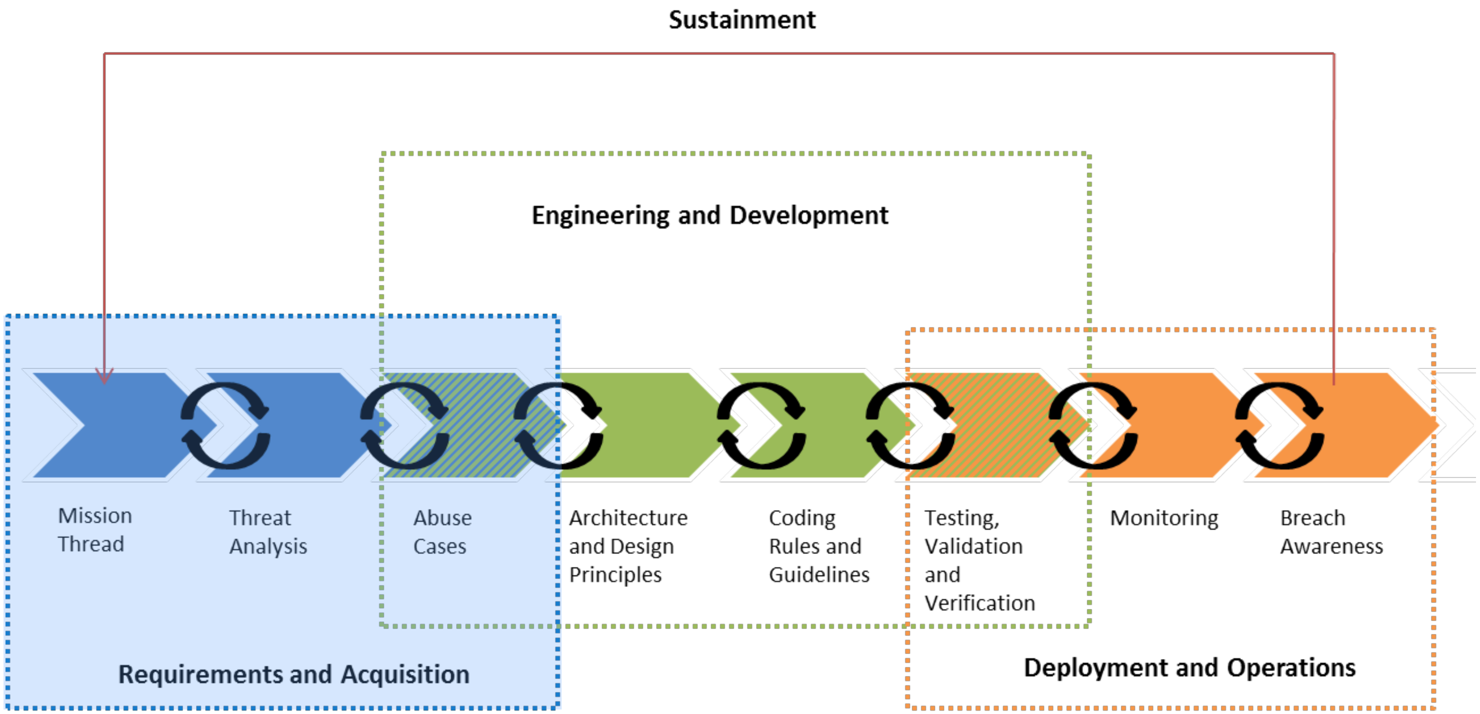
- Assess current software assurance practices
- Develop improvement plan
- Improve software assurance practices
- Supporting Program Protection Plans

Assurance Service Providers

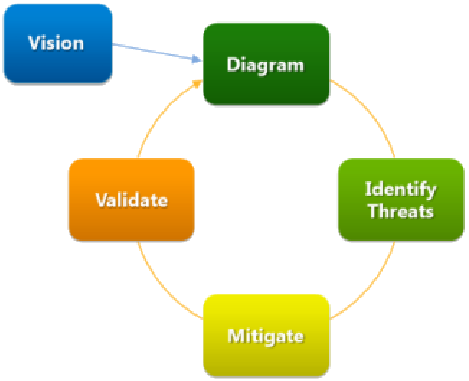
- Identify gaps in software assurance services currently provided
- Develop plan for new or enhanced software assurance services
- Provide new or enhanced software assurance services to constituents



Requirements



Threat analysis tools help derive abuse and misuse cases



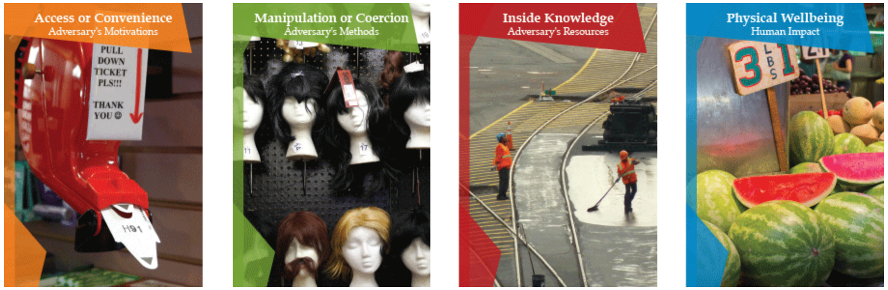
Microsoft SDL Threat Modeling Tool

STRIDE Threat Types

Desired Property	Threat	Definition
Authentication	Spoofing	Impersonating something or someone else
Integrity	Tampering	Modifying code or data without authorization
Non-repudiation	Repudiation	The ability to claim to have not performed some action against an application
Confidentiality	Information Disclosure	The exposure of information to unauthorized users
Availability	Denial of Service	The ability to deny or degrade a service to legitimate users
Authorization	Elevation of Privilege	The ability of a user to elevate their privileges with an application without authorization

16

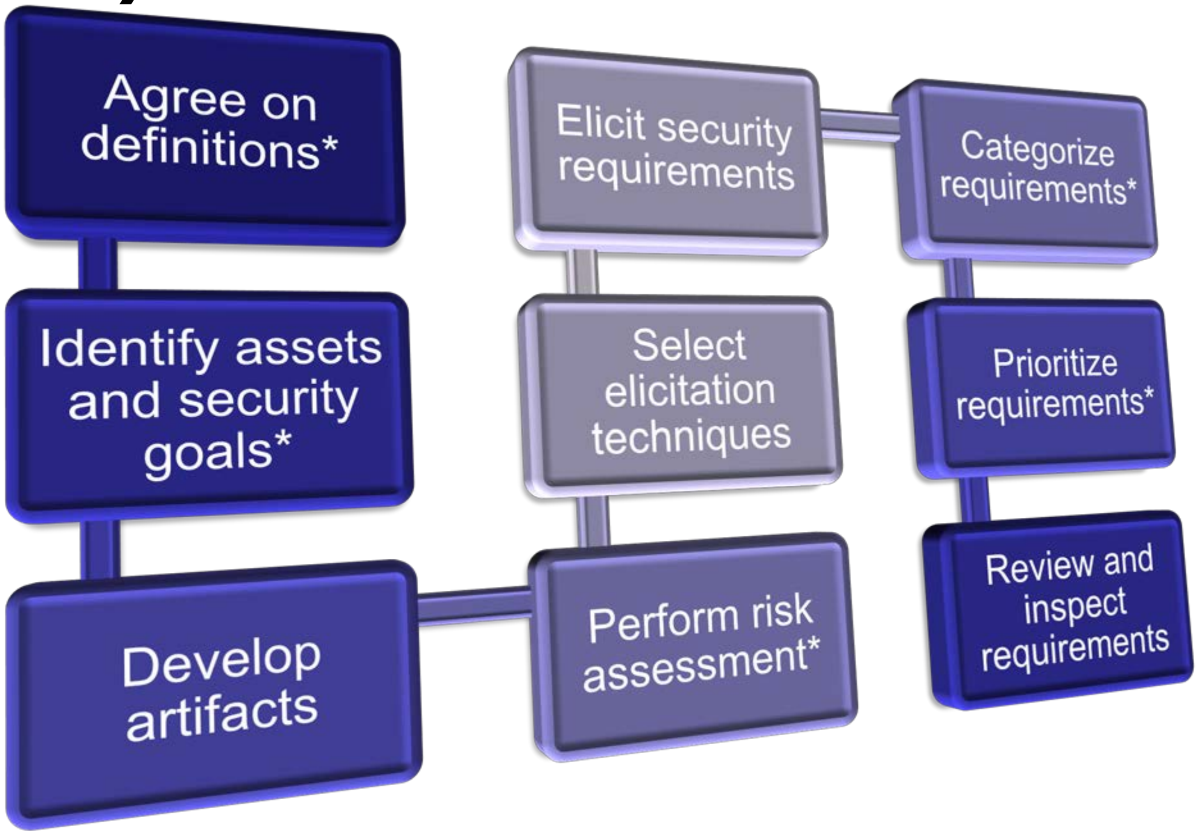
Microsoft STRIDE Threat Types



Denning, Friedman, Kohno
The Security Cards: Security Threat Brainstorming Toolkit

Jane Cleland-Huang's Persona non Grata
http://www.infoq.com/articles/personae-non-gratae

Security Quality Requirements Engineering (SQUARE)



A robust SQUARE tool is available for download from http://www.cert.org/sse/square.html
*SQUARE-Lite process

Embedded systems represent new classes of vulnerabilities

Embedded systems have different characteristics than IT systems

More and varied attack surfaces

- Sensors
- Multiple command-and-control masters
- Embedded firmware, FPGAs, ASICs
- Unique internal busses & controllers

Size, weight, power and latency demands tradeoff against defense-in-depth

Timing demands offer potential side channels

- Bit and clock cycle level operations
- Physical resources with real time sensors
- Safety-Critical Real-time OS

Confusion between failure resilience and attack

- Intermittent communications

Security approaches for IT systems do not cover embedded system security

Virus definitions and operating guidelines do not always apply

Firewalls and IDS/IPS of limited value

Centralized account control not possible

Network tools and assessment techniques unaware of embedded systems architecture and interfaces

- Unique and insecure protocols
- Maintenance backdoors
- Hardcoded credentials
- Unique architectures of embedded controllers

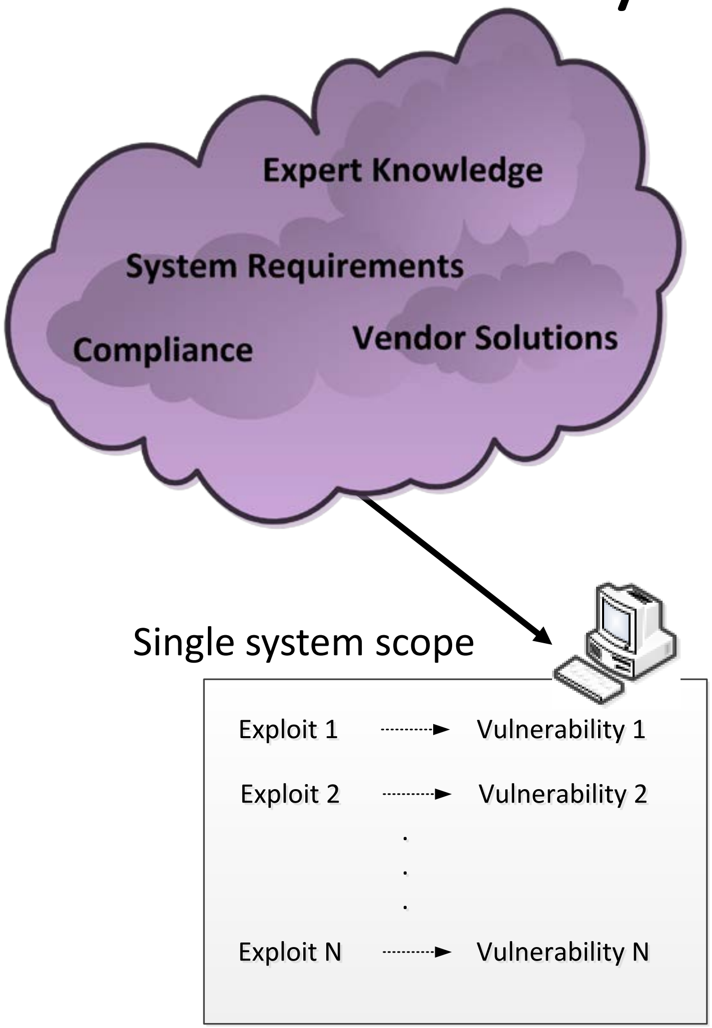
Unplanned connectivity and upgrades

Developers are not trained in software engineering

Programming for security is not the same as programming for safety

Safety strategy	Security view
Rely on physical models in fault trees	Attackers do not obey the laws of physics
Redundancy mitigates single failures	Attackers are not independent events
Fault trees collectively exhaustive	Attack trees depend on adversaries' creativity
Steady state behavior indicator of proper operation	APT (Advanced persistent threats) hide in steady state behavior
Deteriorating performance predicts maintenance for safety	Attackers cover their tracks
Microcontrollers and air gaps implement boundaries	Side channels open vulnerabilities

Need for multisystem risk analysis



Risk analysis is focused on a single system

- Standalone (i.e., single system) models have been developed
- Risk analysis considers the exploit of an individual vulnerability within a single system

Security risk identification techniques do not consider:

- Compositions of multiple vulnerabilities
- Cross-system security events/risks
- Impacts beyond the exploit of a single system (to the intended service and organization)

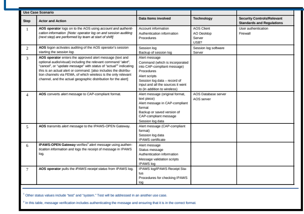
Need for systematic, multiple system evaluations

- Notation for expressing a security events and risks
- Take into account all context

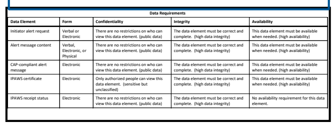
Security Engineering Risk Analysis approach

Comprehensive context

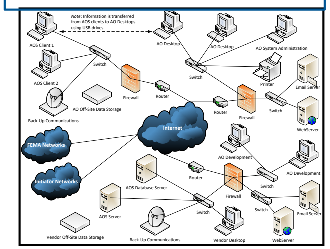
Use-Case View



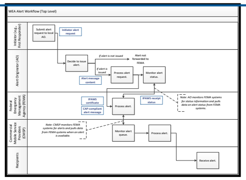
Data View



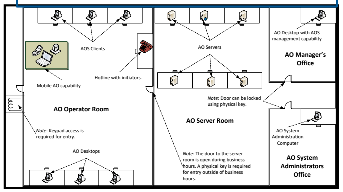
Network View



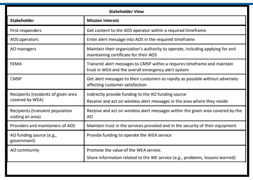
Workflow View



Physical View



Stakeholder View

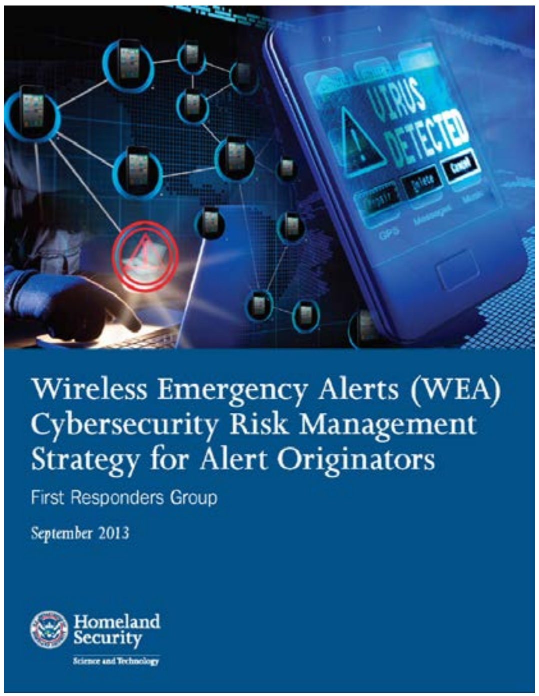


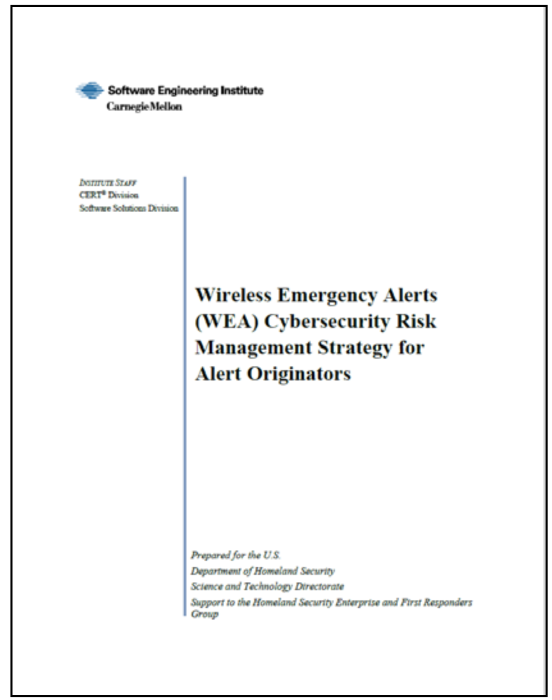
Determining actions

- Establish threat model
- Determine common system view
- Inspect connections between systems
- Evaluate
  - Consequences
  - Likelihood
  - Risk

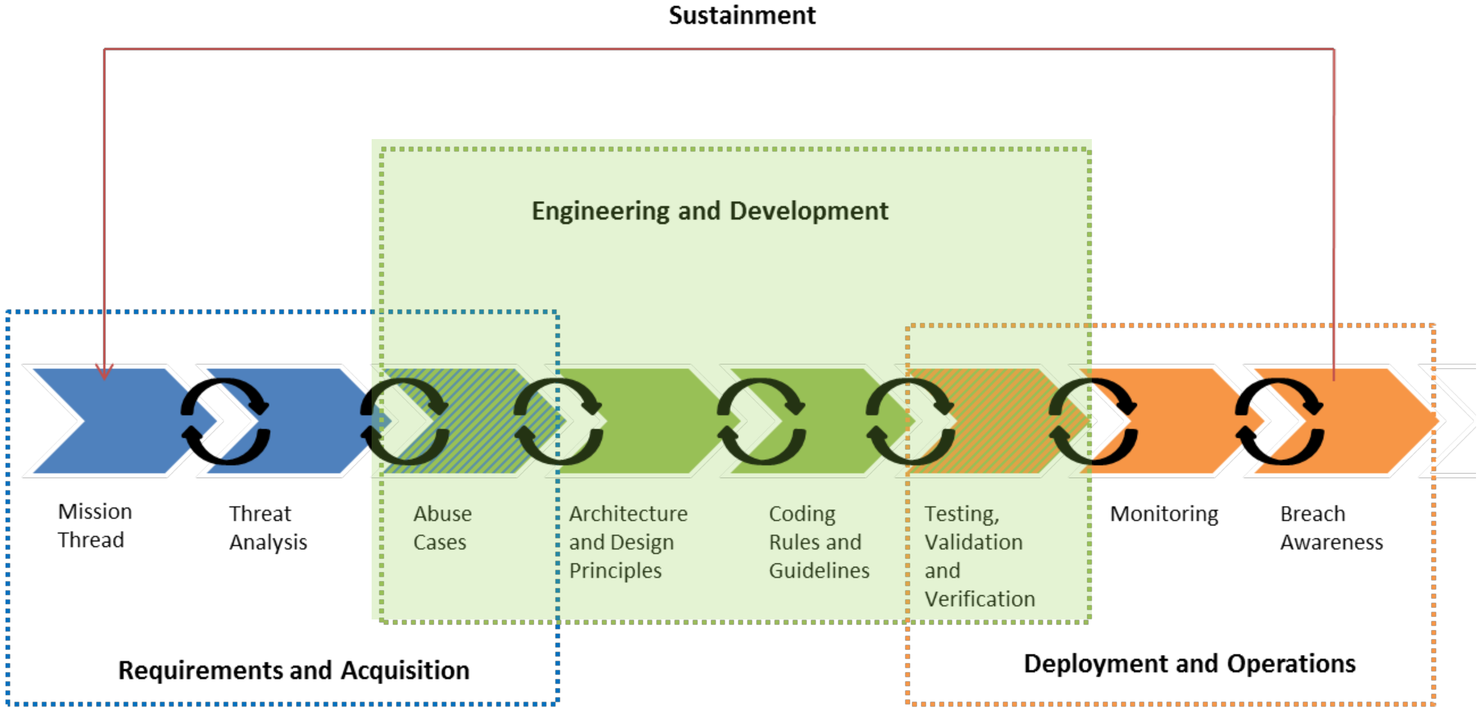
http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=427321

SERA applied to DHS’s Wireless Emergency Alerts system

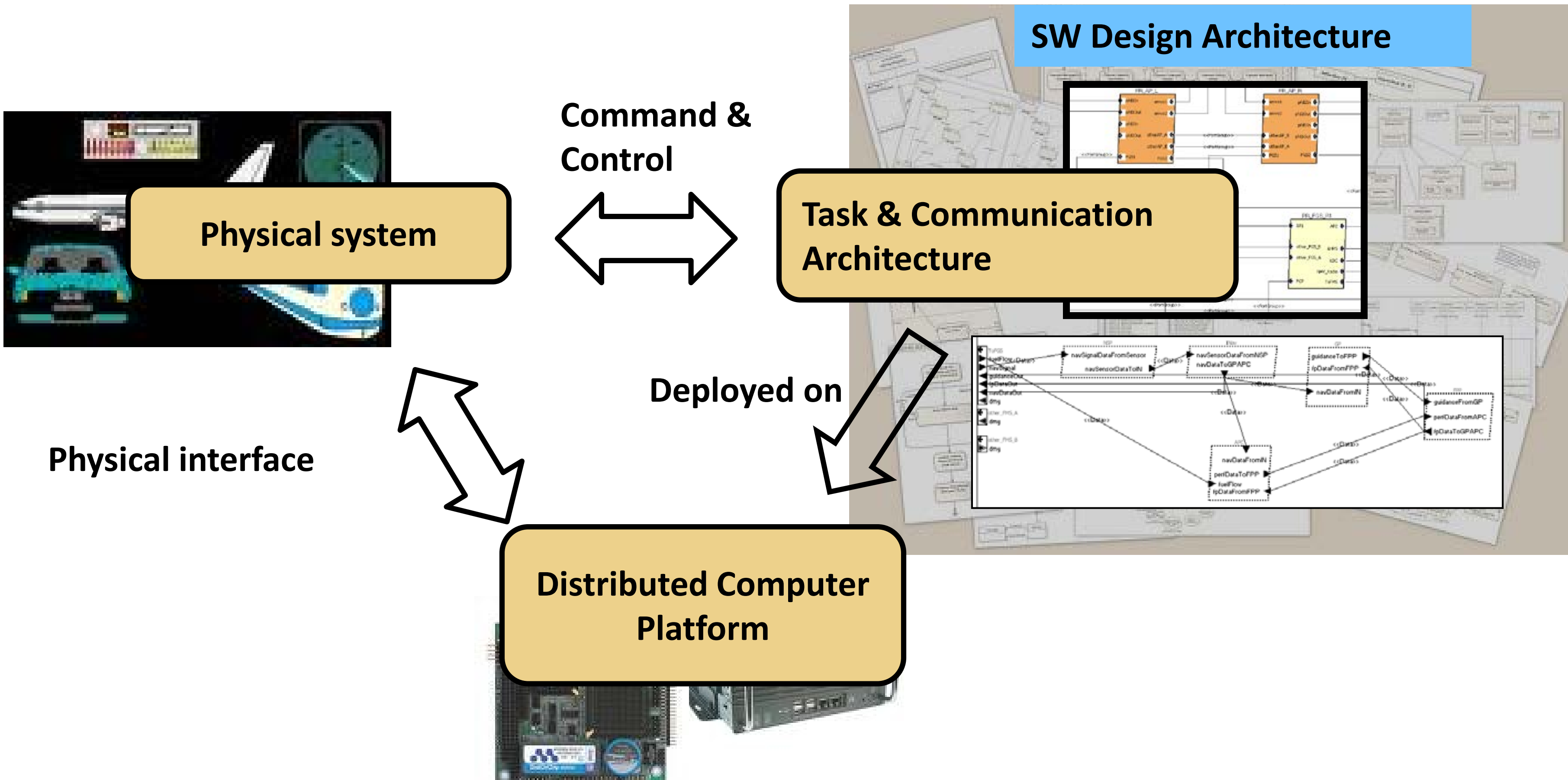




Development



Architecture Analysis & Design Language (AADL)



AADL Addresses Increasing Interaction Complexity and Mismatched Assumptions

Team Software Process

TSP is an agile, team-focused process for software and systems development.

The TSP strategy improves software engineering from the bottom up.

- Instills engineering discipline in software developers
- Builds high-performance trusted teams

TSP works in practice

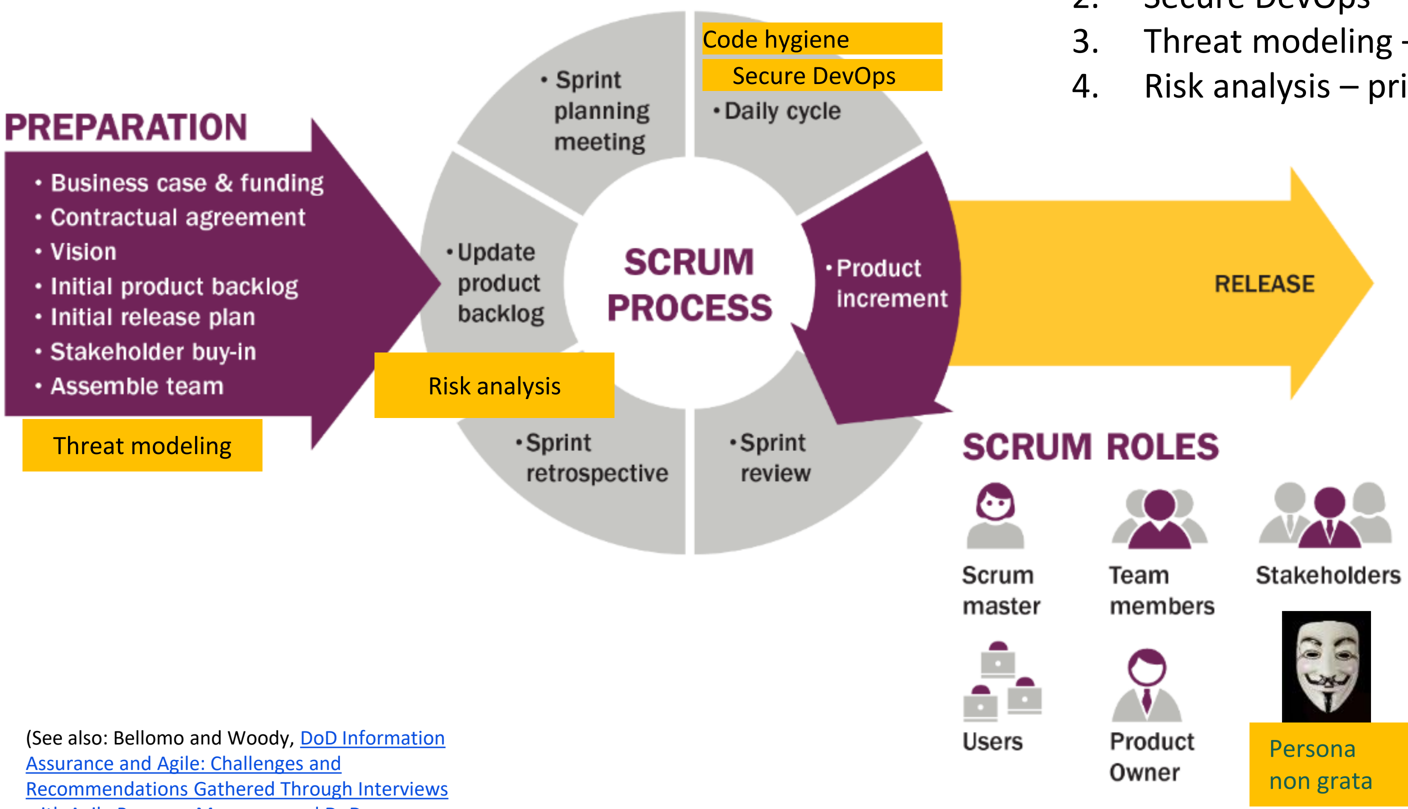
Performance Category	Typical TSP Result	Typical Industry Result
Effort estimation error	<10%	>30%
Schedule estimation error	<10%	>30%
Product quality (defects/KLOC)	0.01 to 0.5	1.0 to 7.0

Extending TSP with security

- Adding secure design
  - Minimize attack surfaces
  - Defense in depth for software development
- Adding secure coding
  - Adopting secure coding practices
- Tooling support for automated conformance checking
- Tracking security defects
  - Monitoring results of tests with respect to security

Integrating security into Agile (Scrum) development

1. Code hygiene – introduce secure coding
2. Secure DevOps – include security tools
3. Threat modeling – represent a new role
4. Risk analysis – prioritize in backlog

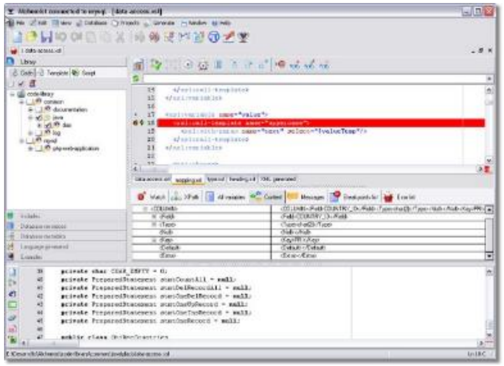


(See also: Bellomo and Woody, [DoD Information Assurance and Agile: Challenges and Recommendations Gathered Through Interviews with Agile Program Managers and DoD Accreditation Reviewers](http://repository.cmu.edu/cgi/viewcontent.cgi?article=1674&context=sei) (http://repository.cmu.edu/cgi/viewcontent.cgi?article=1674&context=sei)

Adoption of secure coding rules

Training

Integrated development environments



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 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

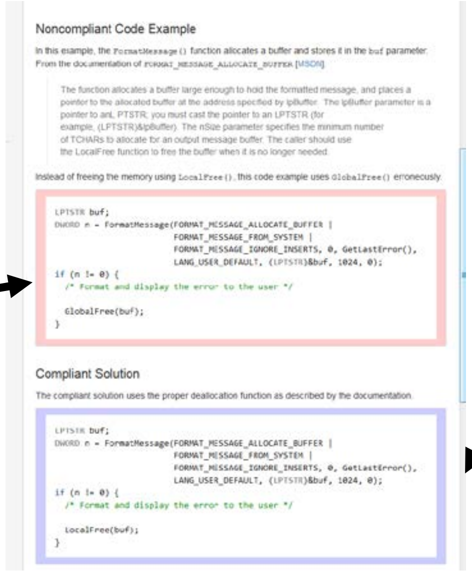
- https://www.securecoding.cert.org

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



Noncompliant Code Example

In this example, the `FormatMessage()` function allocates a buffer and stores it in the `buf` parameter. From the documentation of `FORMAT_MESSAGE_ALLOCATE_BUFFER` [MSDN]

> The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by lpBuffer. The lpBuffer parameter is a pointer to anL PTSTR; you must cast the pointer to an LPTSTR (for example, (LPTSTR)&lpBuffer). The nSize parameter specifies the minimum number of TCHARs to allocate for an output message buffer. The caller should use the LocalFree function to free the buffer when it is no longer needed.

Instead of freeing the memory using `LocalFree()`, this code example uses `GlobalFree()` erroneously.

```
LPTSTR buf;
DWORD n = FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS, 0, GetLastError(),
                        LANG_USER_DEFAULT, (LPTSTR)&buf, 1024, 0);
if (n != 0) {
  /* Format and display the error to the user */

  GlobalFree(buf);
}
```

Compliant Solution

The compliant solution uses the proper deallocation function as described by the documentation.

```
LPTSTR buf;
DWORD n = FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS, 0, GetLastError(),
                        LANG_USER_DEFAULT, (LPTSTR)&buf, 1024, 0);
if (n != 0) {
  /* Format and display the error to the user */

  LocalFree(buf);
}
```

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

Secure Coding in C/C++ Training

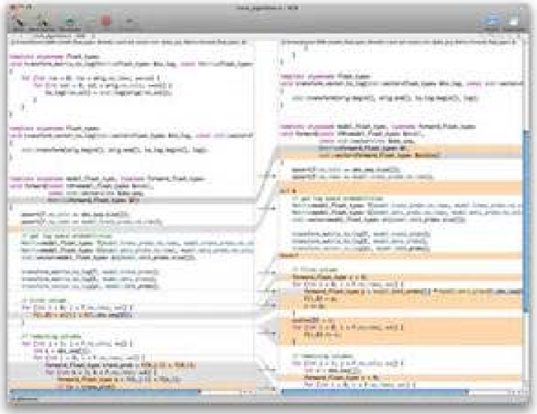
The Secure Coding course is designed for C and C++ developers. It encourages programmers to adopt security best practices and develop a security mindset that can help protect software from tomorrow's attacks, not just today's.

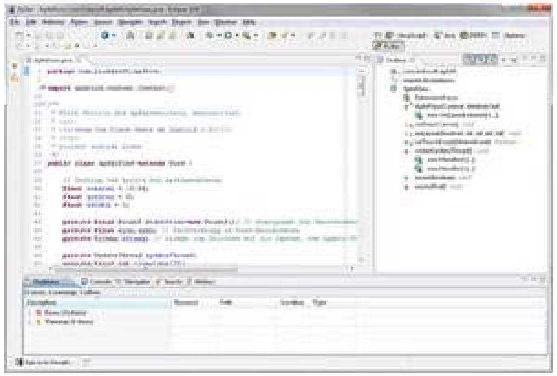
Topics

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

Additional information at ttp://www.sei.cmu.edu/training/p63.cfm

Tools encourage application of secure coding





Moving rules into IDE improves application of secure coding

- Early feedback corrects errors on introduction
- Exceptions are understood in context
- Feedback improves developer skill

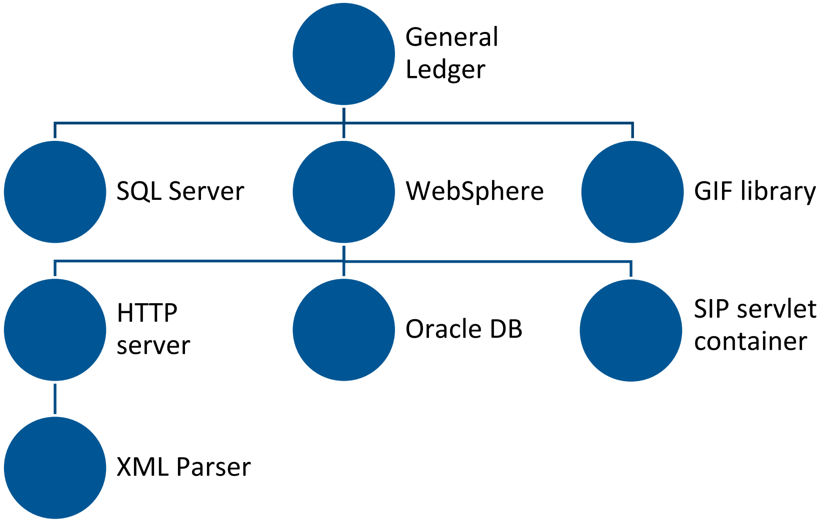
Target Clang static analyzer (C based languages)

- Widely used open source front end for popular compilers
- Integrated into Apple’s Xcode IDE

Target FindBugs (Java)

- Integrated into Eclipse and JDeveloper

Software is more assembled than built

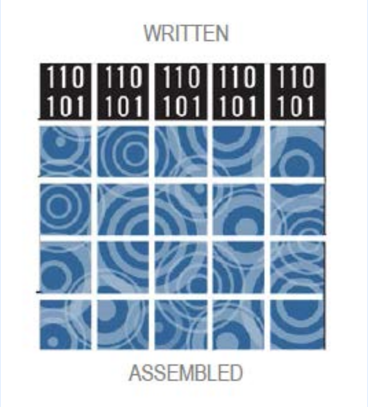


"Development" is now "assembly" using collective development

- Too large for single organization
- Too much specialization
- Too little value in individual components

Note: hypothetical application composition

The rise of open source



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

Distributed development – context:

- Amortize expense
- Outsource non-differential features
- Lower acquisition (CapEx) expense

Sources: Geer and Corman, “Almost Too Big To Fail,” ;login: (Usenix), Aug 2014; Sonatype, 2014 open source development and application security survey

The rise of open source

- 90% of
  assem
  - At le
    as th
- Most applications are now assembled from hundreds of open source components, often reflecting as much as 90% of an application

Distributed development – context:

ntial

)Ex)

“Developers are gorging themselves on an ever expanding supply of open source components”

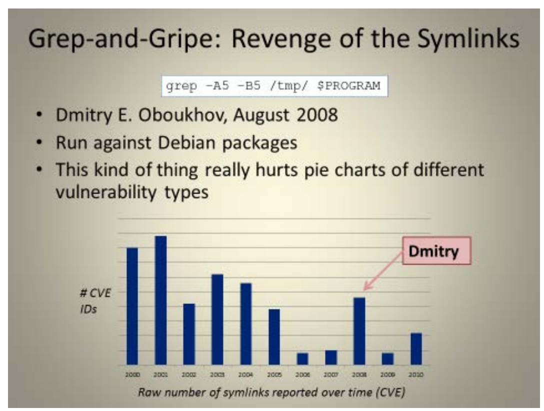
Sonatype, “2016 State of the Software Supply Chain”

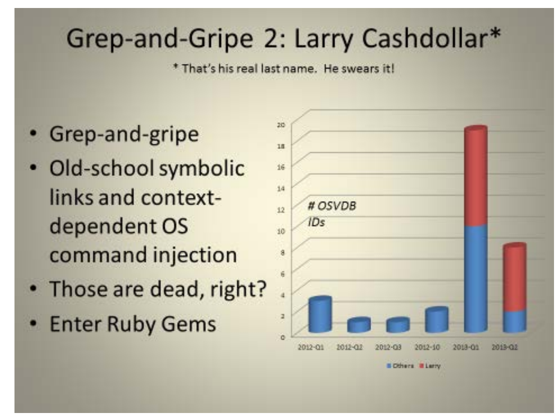
Sources: Geer and Corman, “Almost Too Big To Fail,” ;login: (Usenix), Aug 2014; Sonatype, 2014 open source development and application security survey

Open source is not secure

Heartbleed and Shellshock were found by exploitation

Other open source software illustrates vulnerabilities from cursory inspection





Sources: Steve Christey (MITRE) & Brian Martin (OSF), Buying Into the Bias: Why Vulnerability Statistics Suck, https://media.blackhat.com/us-13/US-13-Martin-Buying-Into-The-Bias-Why-Vulnerability-Statistics-Suck-Slides.pdf; Sonatype, Sonatype Open Source Development and Application Security Survey; Sonatype, 2016 State of the Software Supply Chain; Aspect Software "The Unfortunate Reality of Insecure Libraries," March 2012

Open sourc

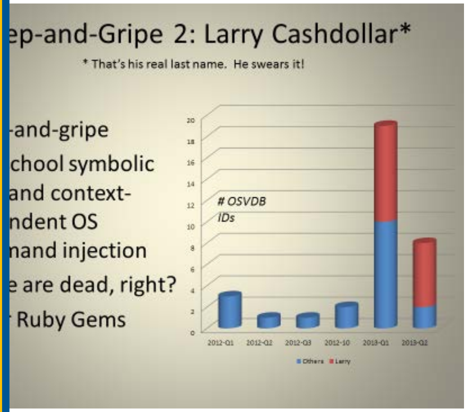
Heartbleed and Shellshock were fou by exploitation

Other open source software illustrates vulnerabilities from cu inspection

1.8 billion vulnerable open source components downloaded in 2015

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

On average, applications have 22.5 open source vulnerabilities



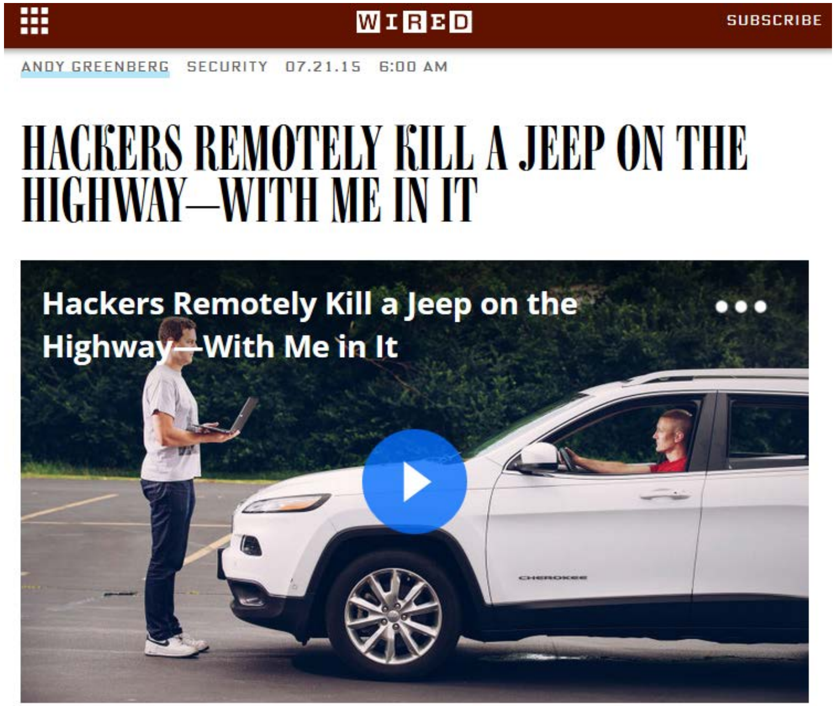
s Suck, https://media.blackhat.com/us-13/US-13-
ource Development and Application Security Survey;
ure Libraries," March 2012, Mike Pittenger, Black

Reducing software supply chain risk factors

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

Supplier Capability	**P**roduct Security	Product **D**istribution	**O**perational Product Control
Supplier follows practices that reduce supply chain risks	Delivered or updated product is acceptably secure	Methods of transmitting the product to the purchaser guard again tampering	Product is used in a secure manner

Connecting automotive systems to internet opens system to attack



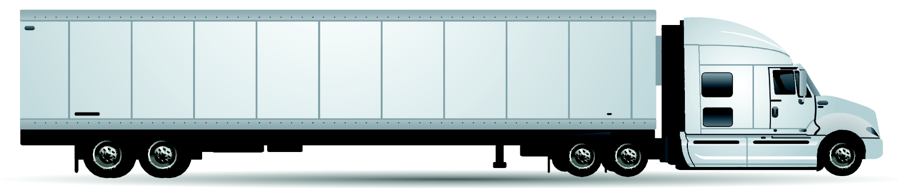
Extending systems opens vulnerabilities not anticipated

- Optimizations performed assuming one attack method
- Assumptions no longer hold with additional integrations

Studies suggest that new operational environments are a leading cause for introducing new vulnerabilities in existing systems.

Source: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
Clark, Frei, Blaze, Smith, "Familiarity Breeds Contempt: The Honeymoon Effect and the Role of Legacy Code in Zero-Day Vulnerabilities," ACSAC '10 Dec. 6-10, 2010, p. 251-260."

Machine-learning based systems increase exposures



"the [Tesla] car's driverless technology failed to detect the white side of the tractor-trailer against a brightly lit sky, so the brake wasn't activated."
-ABC7News, July 1, 2016

Operations are driven by high volume, high velocity sensor data

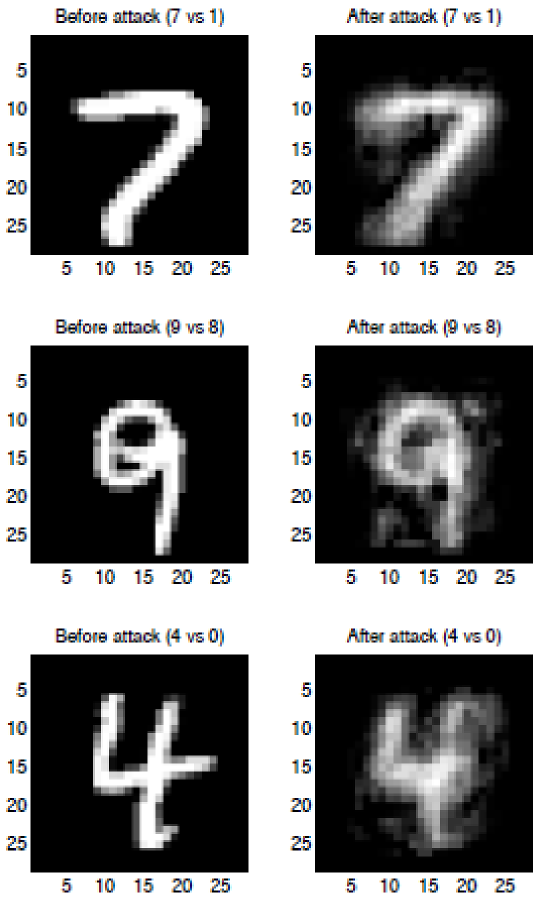
Decision making is based on "trained" models of behaviors

Conventional code development techniques of modest help

Understand the limits of training

Source: http://abc7news.com/automotive/tesla-self-driving-car-fails-to-detect-truck-in-fatal-crash/1410042/

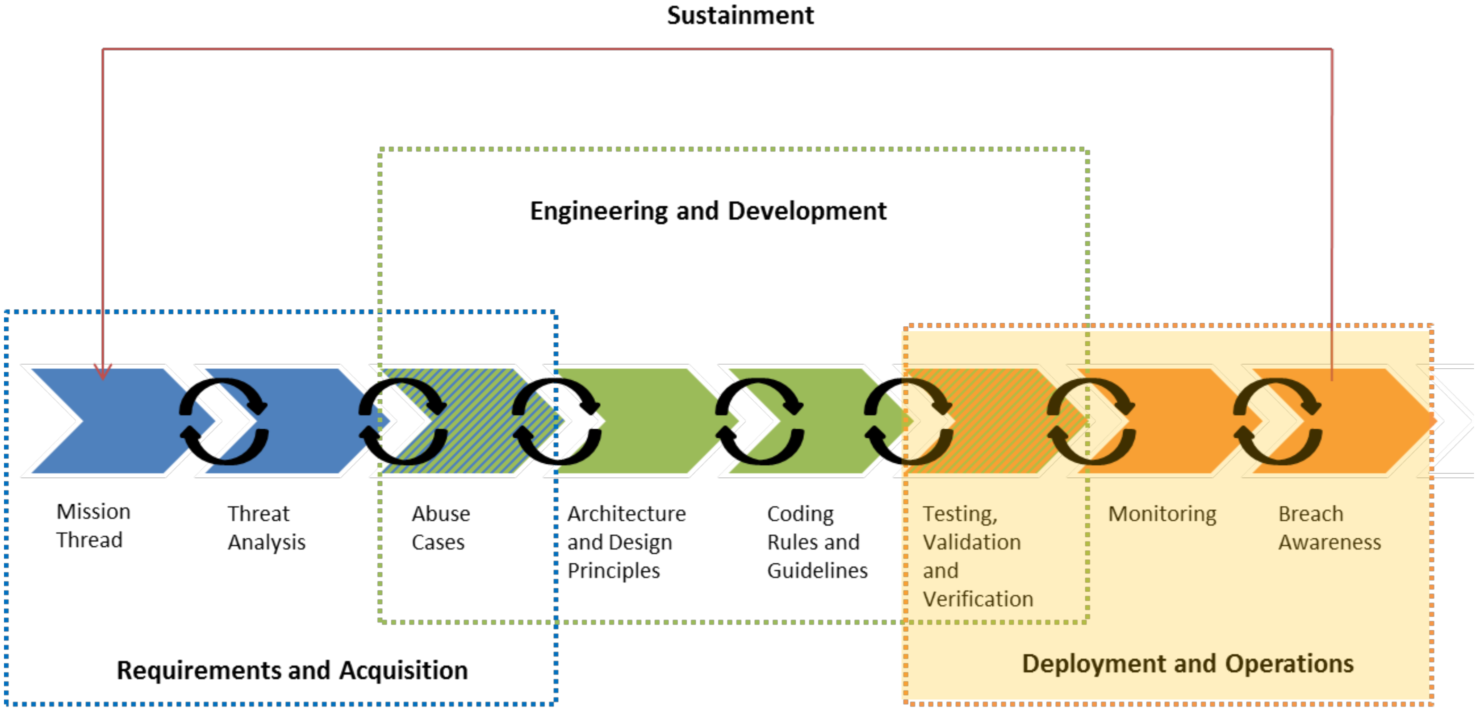
Recognizing and recovering poisoned systems



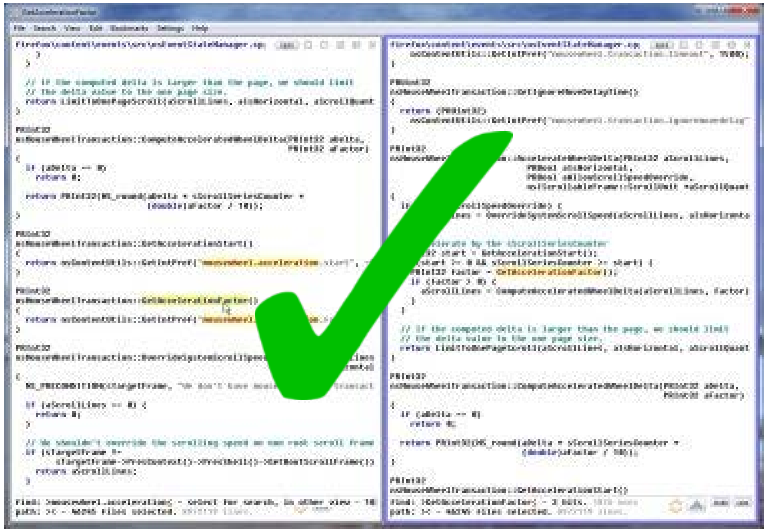
- “Chaff” and “noise” can emerge as vulnerabilities
- Defensive strategy based on “it is difficult to lie at scale”
- Tactics include consistency checks, such as
  - Multiple models in a single unit
  - Coordination among units
  - Coordination with environment

Source: Battista Biggio, Blaine Nelson, Pavel Laskov, Poisoning Attacks against Support Vector Machines, 2012, arxiv.org/abs/1206.6389

Deployment and operations



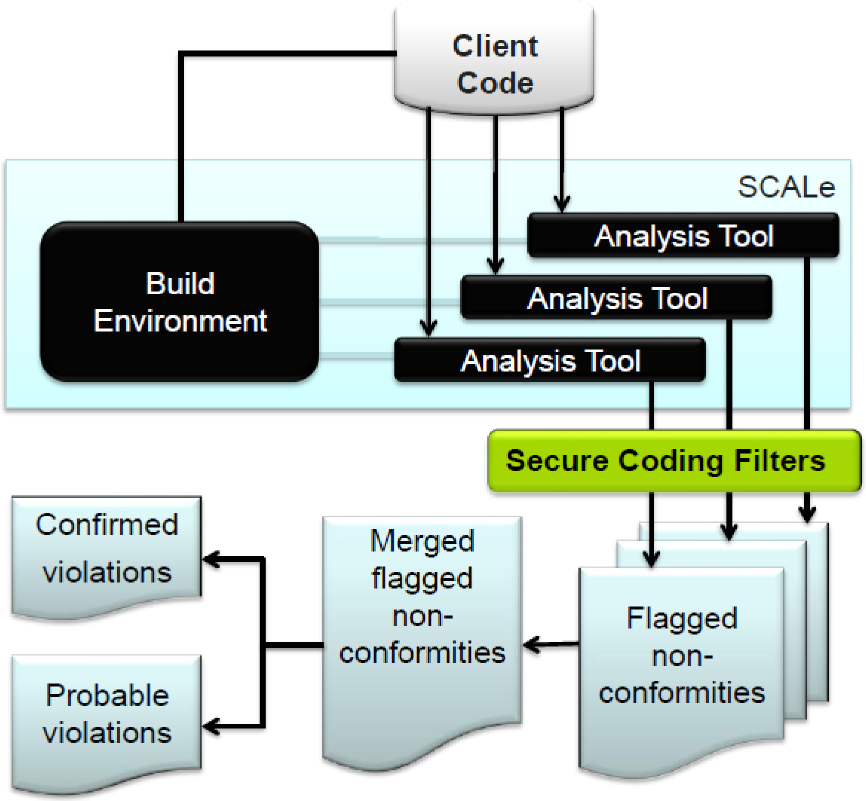
Static Testing – Source code analysis tools



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

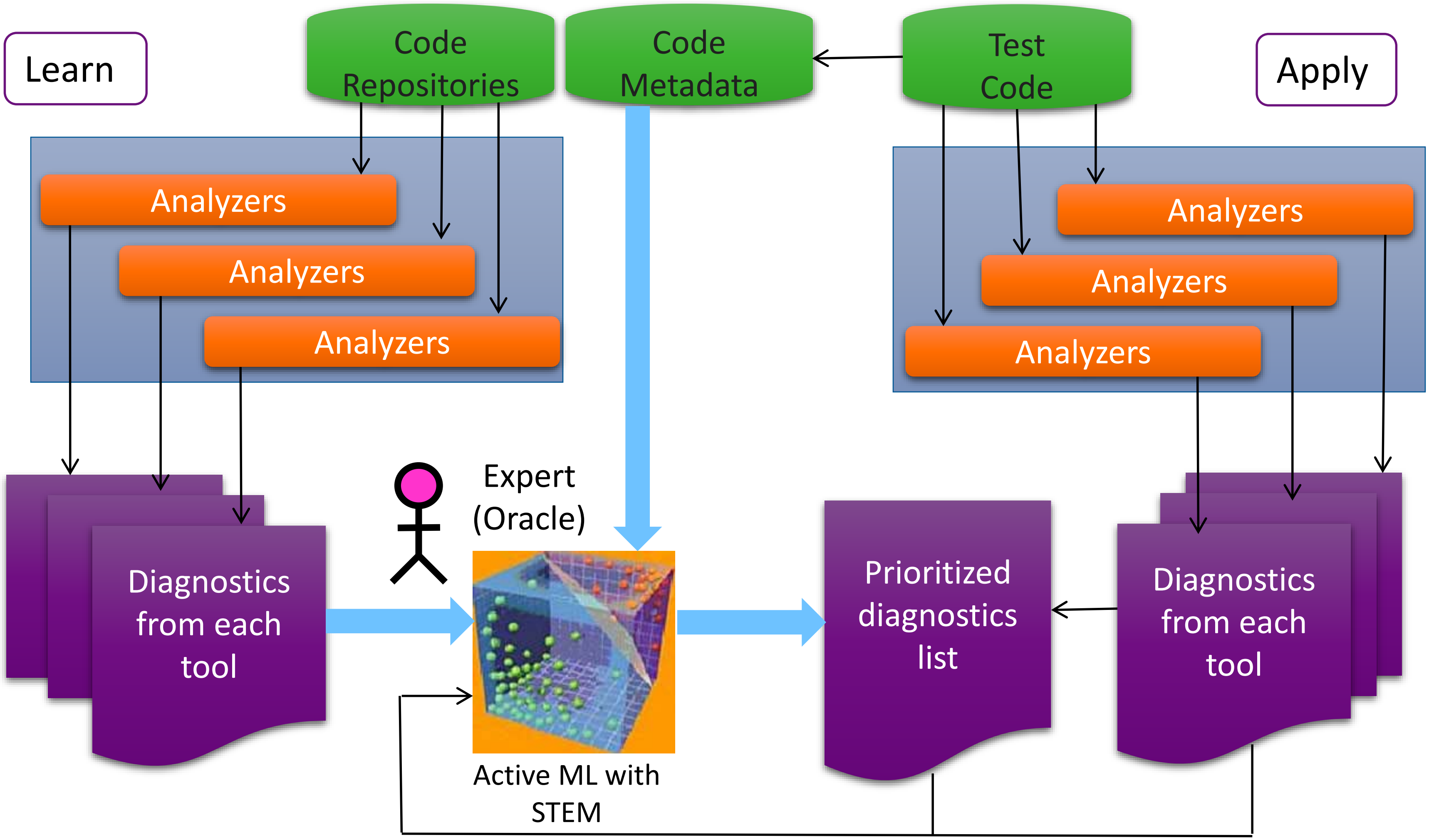
Filter select secure coding rule violations

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

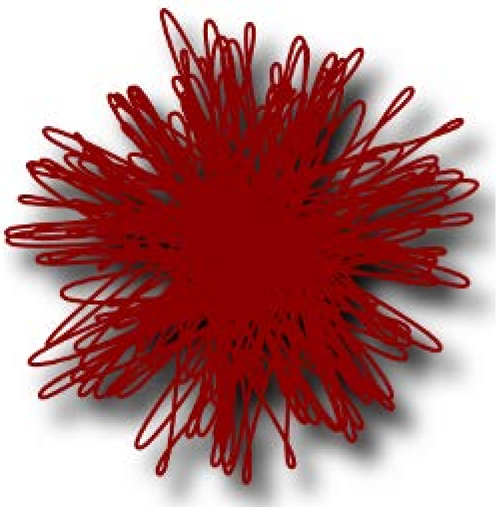
Single view into code and all diagnostics

Maintain record of decisions

Optimizing multitool evaluations



Dynamic testing and evaluation – fuzzing



Fuzz testing of attack surfaces

- Based on techniques used in CERT's Basic Fuzzing Framework (BFF)
- mutational fuzzing
- machine learning and evolutionary computing techniques
- adjusts its configuration parameters based on what it finds (or does not find) over the course of a fuzzing campaign

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.

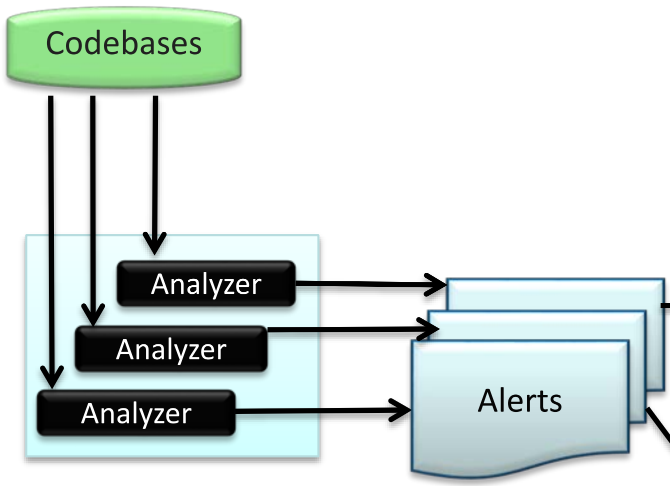
Sensitive Dataflow Analysis among Android App Sets

- Detecting tainted data flows across multiple Android components

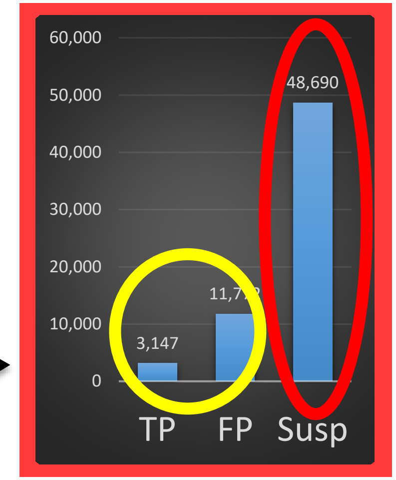
Integrating Secure Coding Rule analysis with Development Environments

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

Prioritizing Vulnerabilities



Today



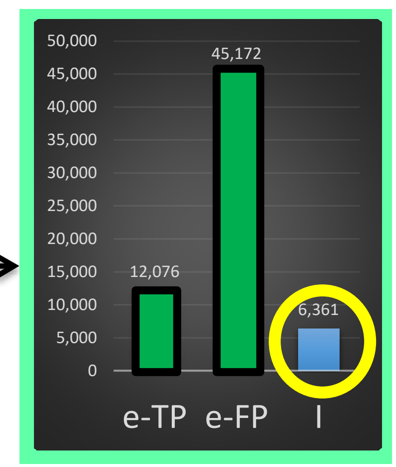
Many alerts left un-audited!

Project Goal

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

<u>Long-term goal</u>: 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"

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

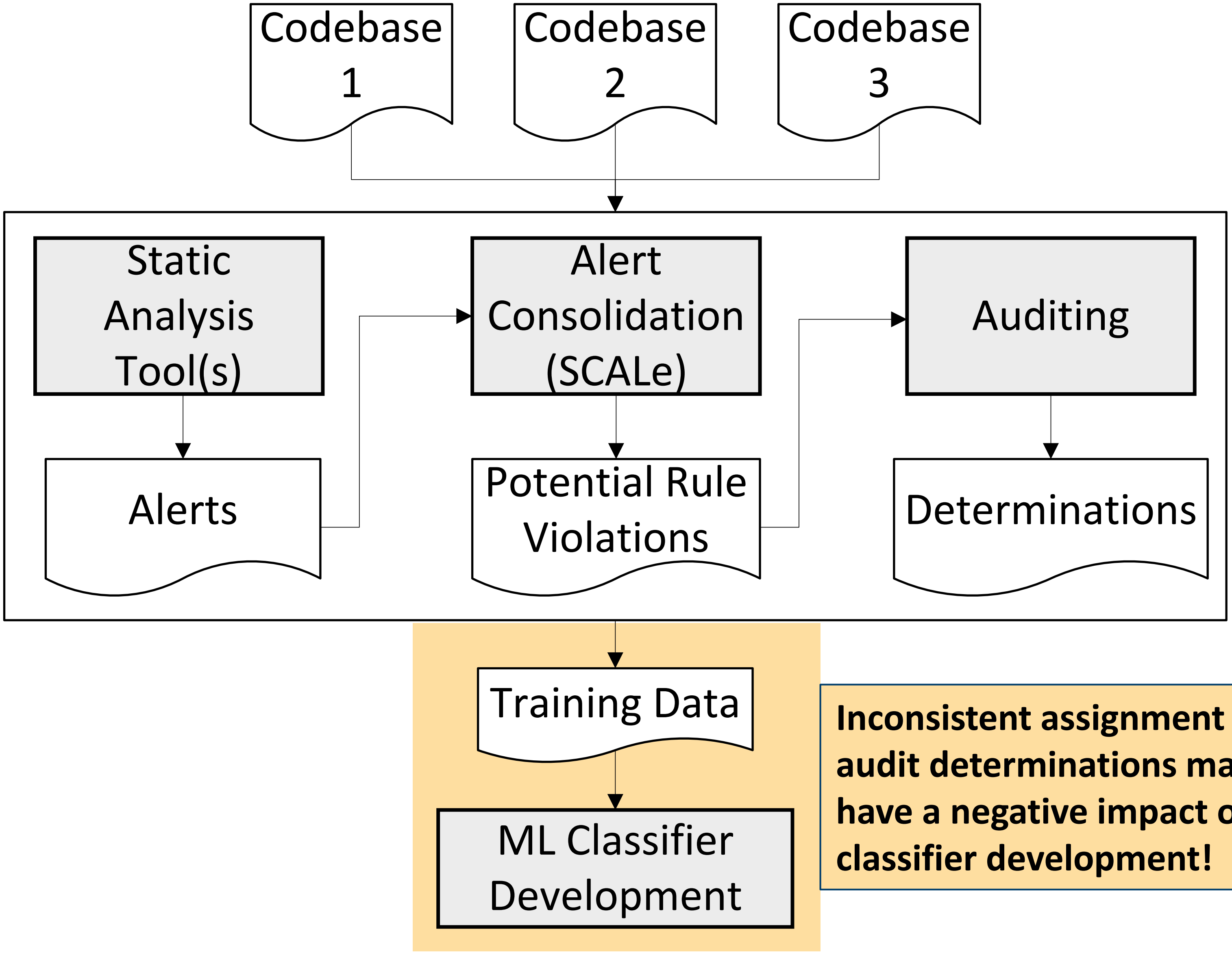
- Test audit skill: mentor-expert designation

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.

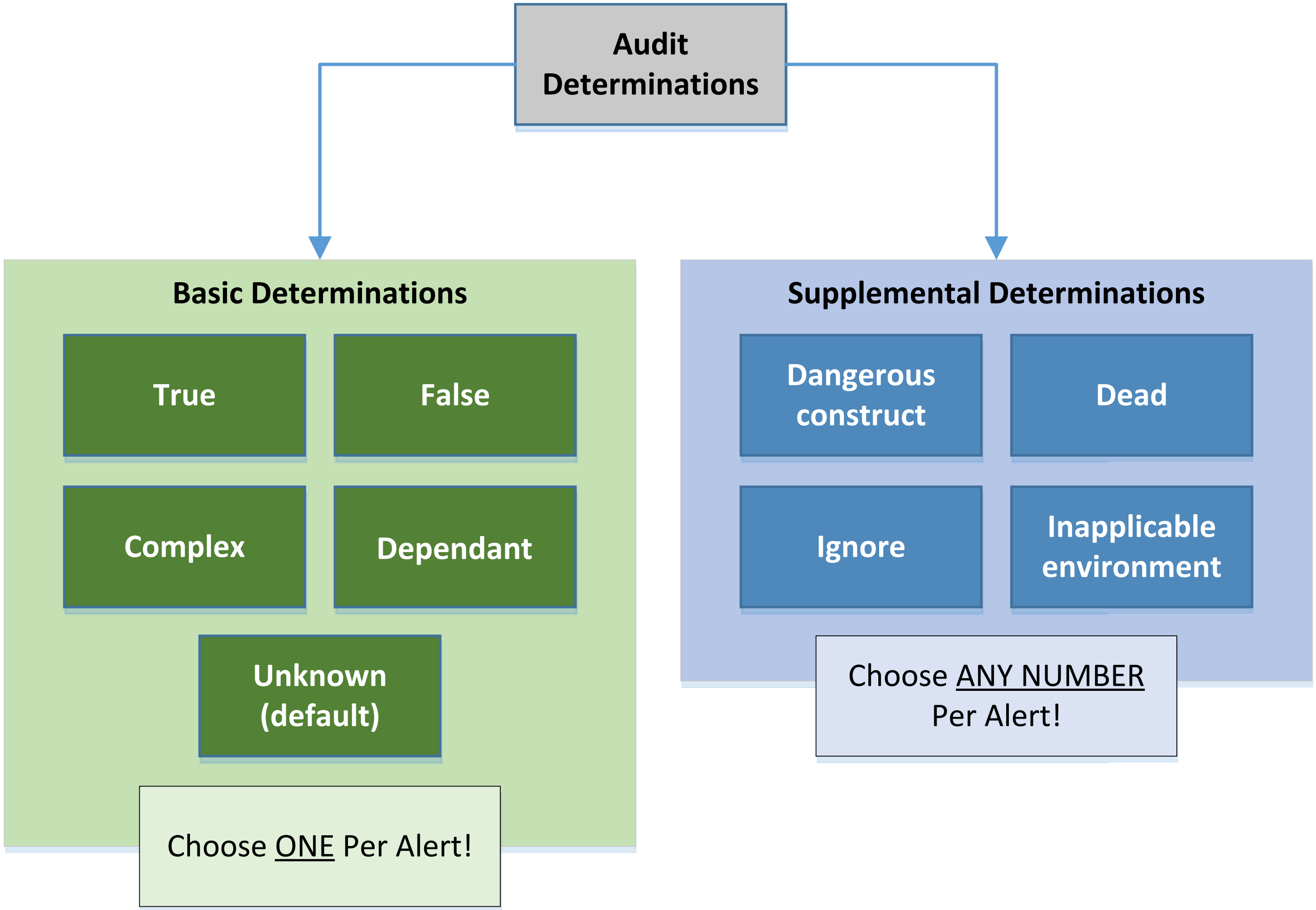
[2] Svoboda, Flynn, and Snavely. “Static Analysis Alert Audits: Lexicon & Rules”, IEEE Cybersecurity Development (SecDev), November 2016.

Background: Automatic Alert Classification



Inconsistent assignment of audit determinations may have a negative impact on classifier development!

Lexicon: Audit Determinations



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.

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

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

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).

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.

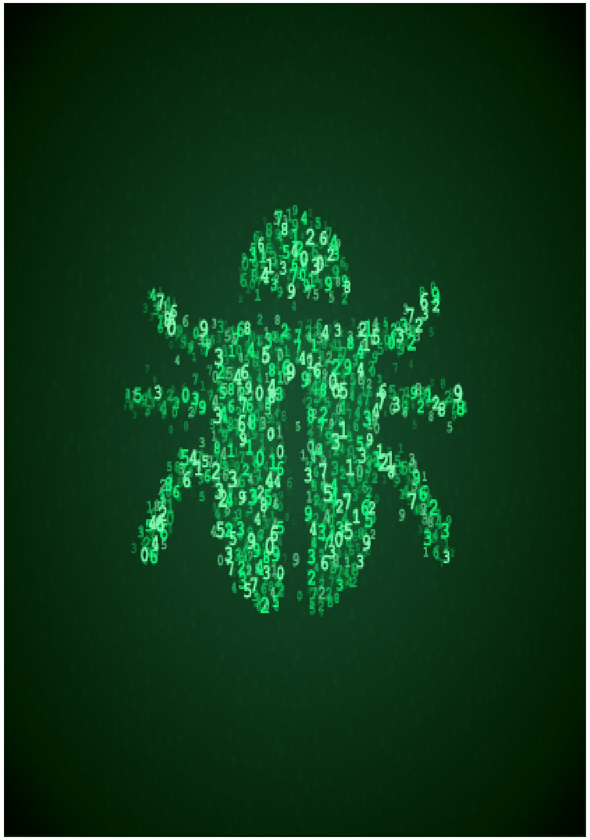
Formalizable Constraints (to be formally verified):

- 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.

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

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

```
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;
}
```

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

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

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

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).

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.

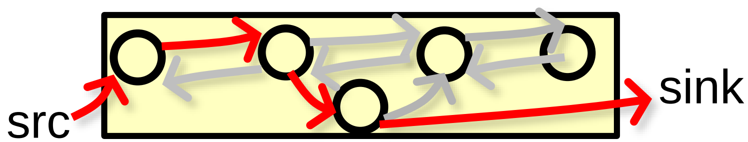
Approach: Add context sensitivity to analysis, to reduce false positives, while retaining analytical speed by using DidFail's fast 2-phase static analysis method (that summarizes potential flows of sensitive data per-app and quickly analyzes per-app-set).

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 <u>**across multiple Android components**</u> 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 <u>sets</u> (not just single apps)
- Fast user response: two-phase method uses phase-1 precomputation

Next: More precision using context sensitivity $\Longrightarrow$ fewer false alarms.

Analysis of Android App Sets: Sensitive Dataflow

Goal: enforce confidentiality and integrity

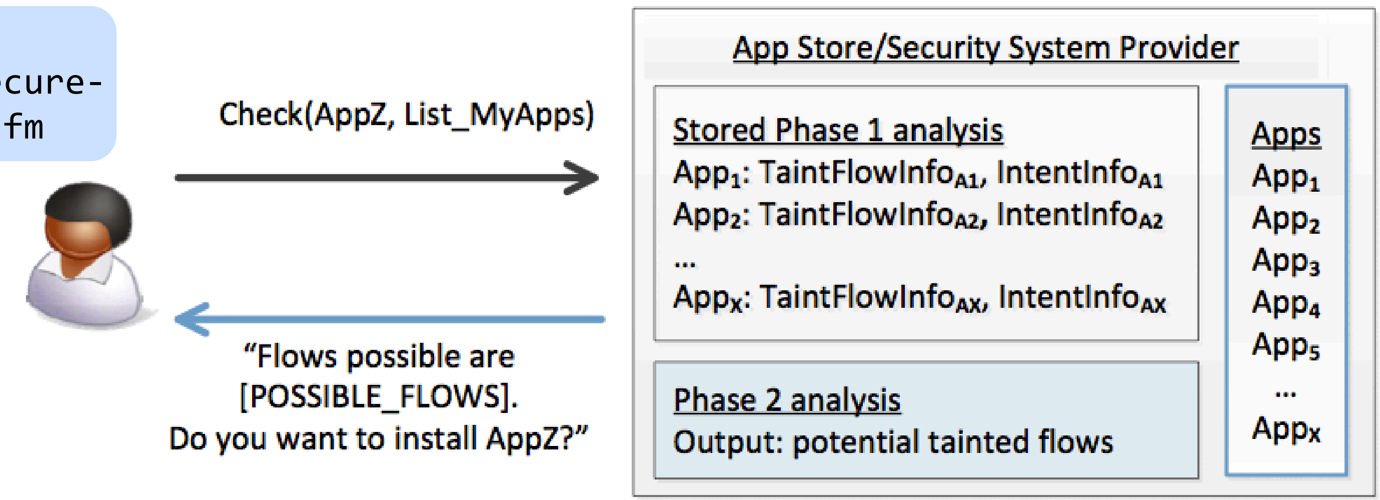
Cutting-edge Android **app set** static dataflow analysis "DidFail" combines precise **single-component taint analysis** and **intent analysis**.

- 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

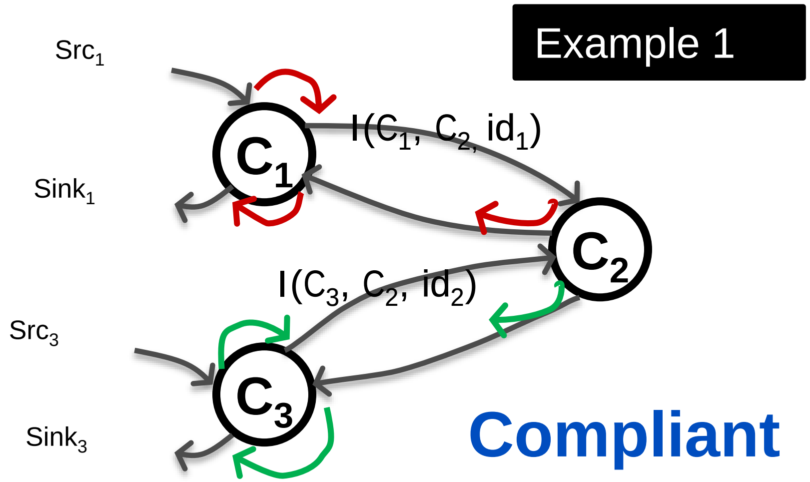
Source code and binaries:
`http://www.cert.org/secure-coding/tools/didfail.cfm`

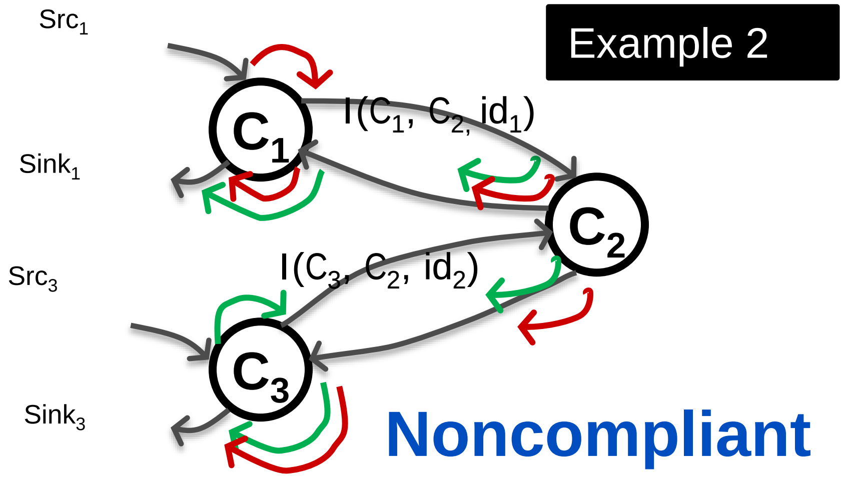
Next Work:
- More context sensitivity



Usability: Policies to Determine Allowed Flows

Policy: Prohibit flow from Src_1 to $Sink_3$

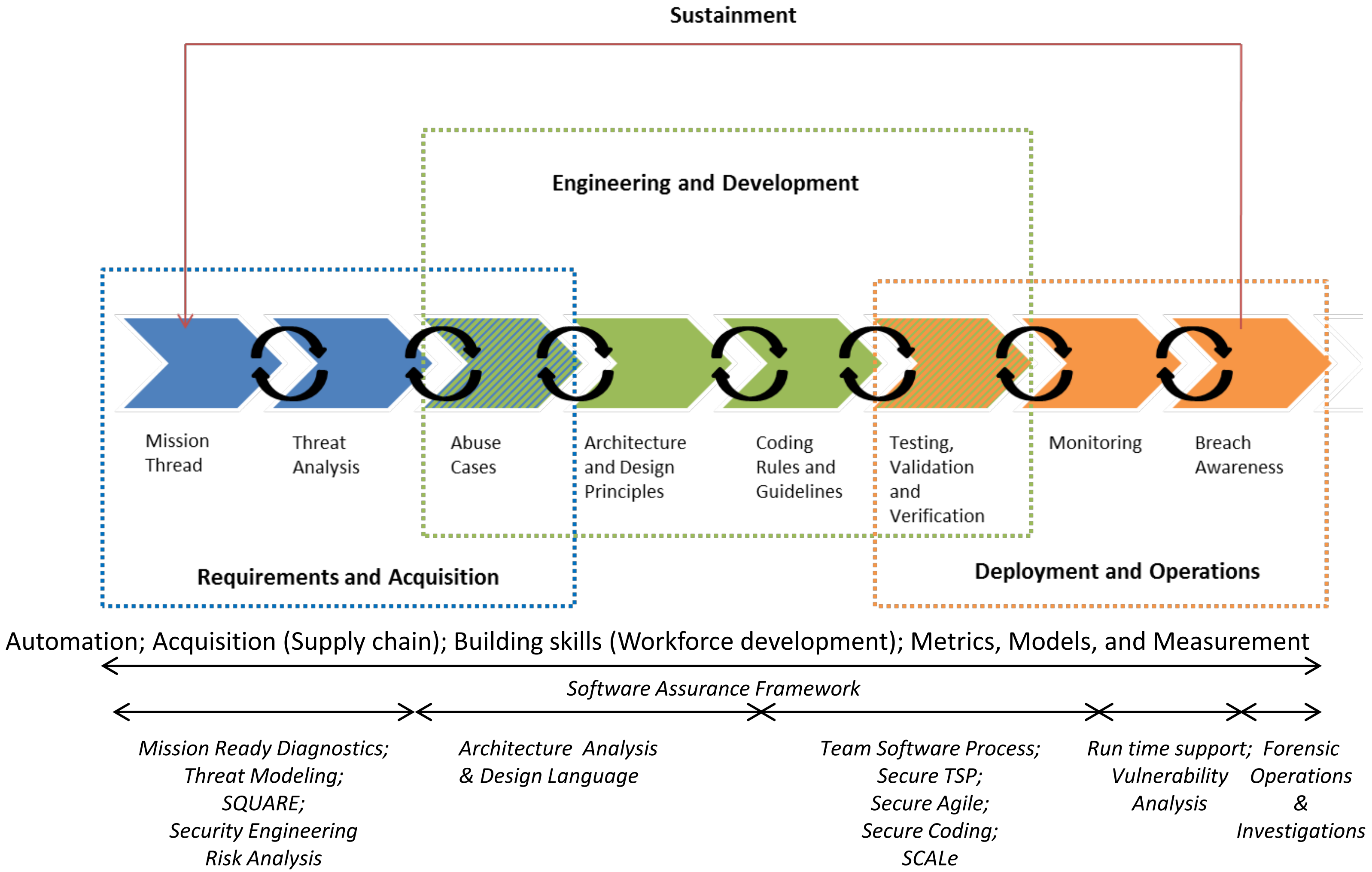




<u>Policies could come from:</u>

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

Review: Secure Software Development Lifecycle



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/

Contact Information

Robert Schiela

rschiela@sei.cmu.edu

Web Resources (CERT/SEI)

http://www.cert.org/

http://www.sei.cmu.edu/

http://securecoding.cert.org