IS 2620

IS 2620: Developing Secure Systems

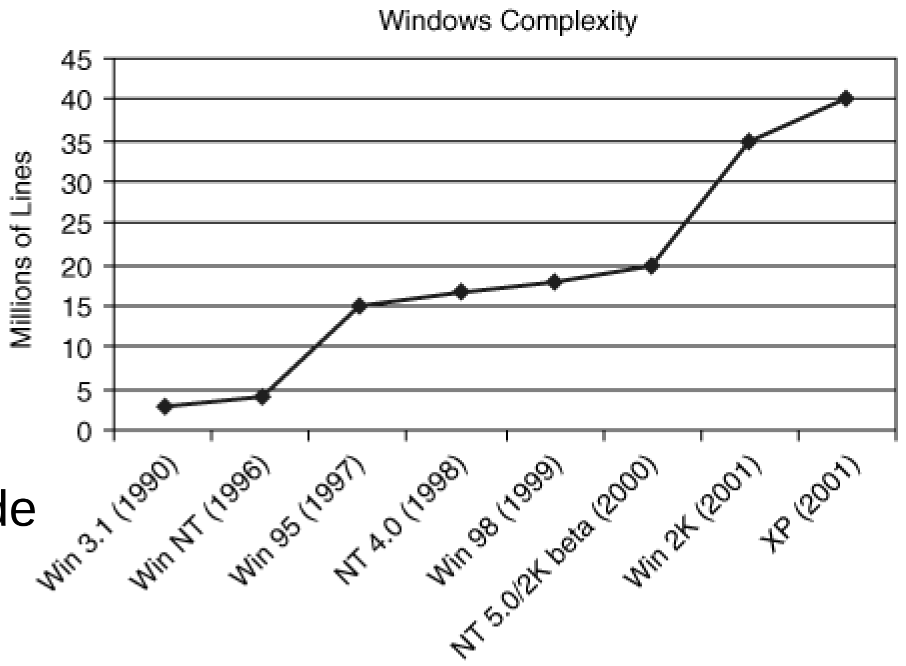
Building Security *In*

Lecture 2
Sept 8, 2017

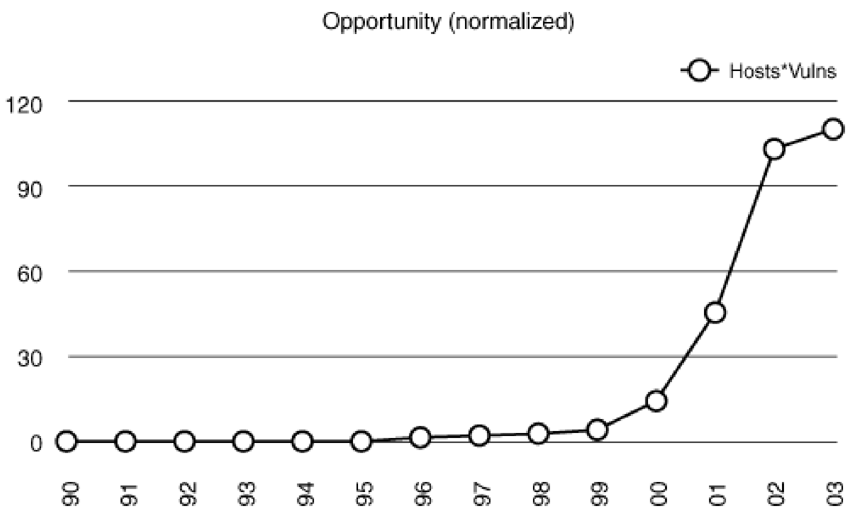
Recap: Trinity of trouble

- Three trends
  - Connectivity
    - Inter networked
    - Include SCADA (supervisory control and data acquisition systems)
    - Automated attacks, botnets
  - Extensibility
    - Mobile code – functionality evolves incrementally
    - Web/Os Extensibility
  - Complexity
    - XP is at least 40 M lines of code
    - Add to that use of unsafe languages (C/C++)

Bigger problem today .. And growing



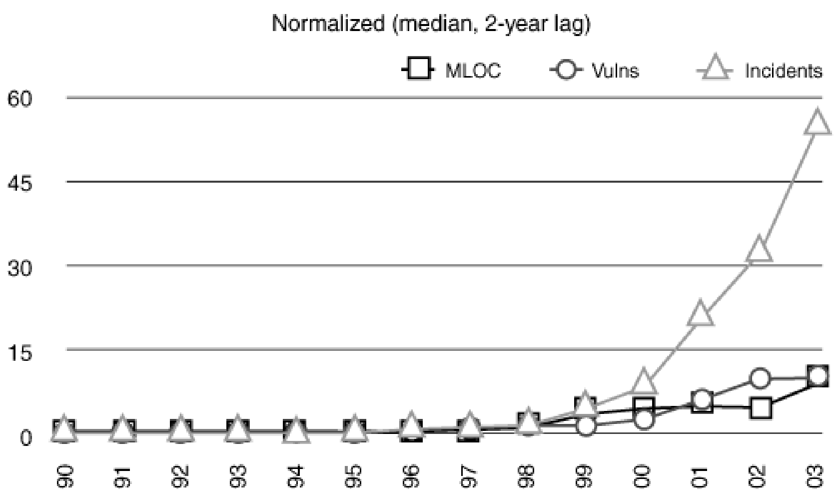
It boils down to …

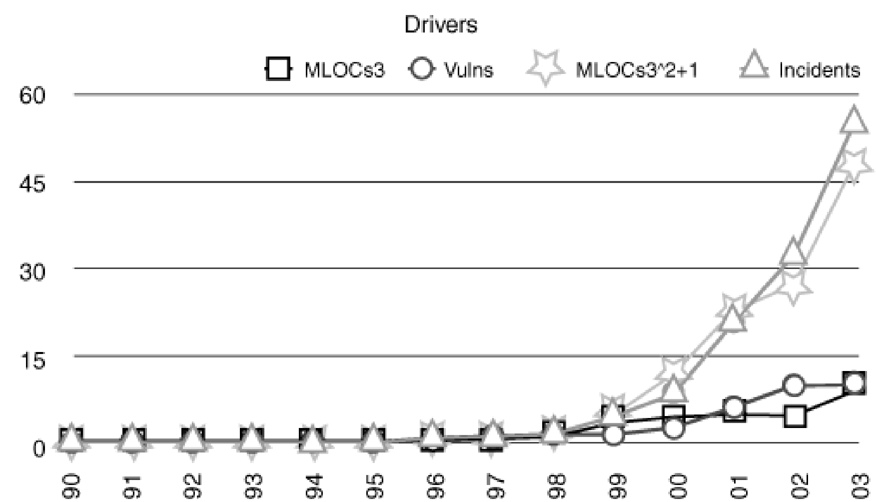


more code,

more bugs,

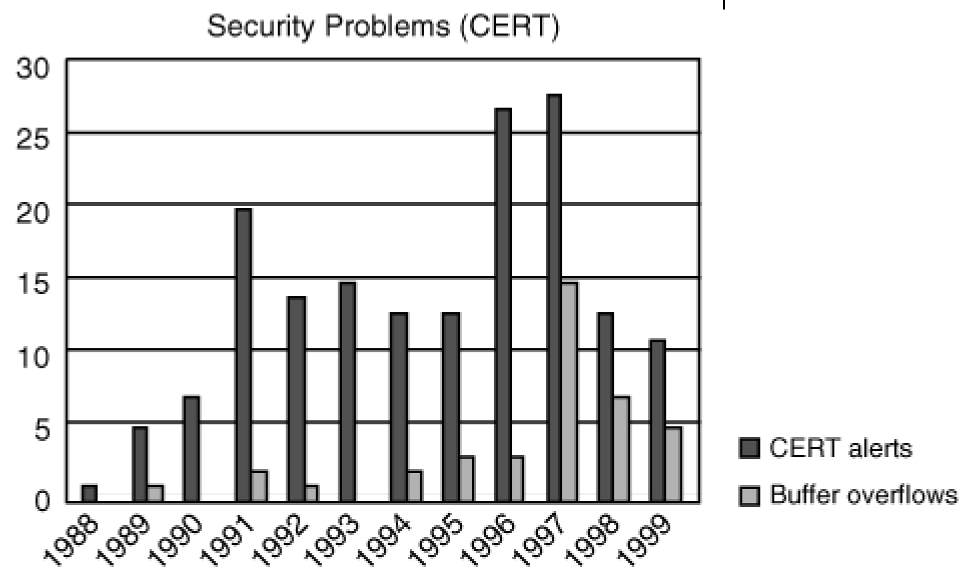
more security problems





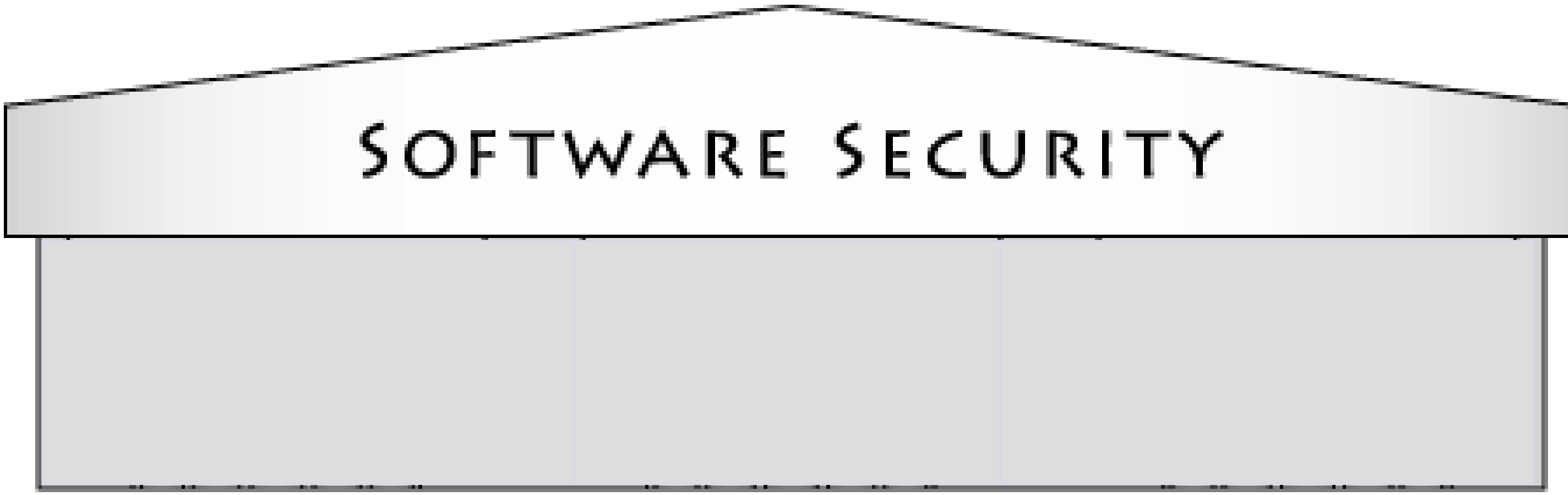
Security problems in software

- Defect
  - implementation and design vulnerabilities
  - Can remain dormant
- Bug
  - An implementation level software problem
- Flaw
  - A problem at a deeper level
- Bugs + Flaws
  - leads to Risk



Bug	Flaw
Buffer overflow: stack smashing Buffer overflow: one-stage attacks Buffer overflow: string format attacks Race conditions: TOCTOU Unsafe environment variables Unsafe system calls (fork(), exec(), system()) Incorrect input validation (black list vs. white list	Method over-riding problems (subclass issues) Compartmentalization problems in design Privileged block protection failure (DoPrivilege()) Error-handling problems (fails open) Type safety confusion error Insecure audit log design Broken or illogical access control (role-based access control [RBAC] over tiers) Signing too much code

Solution …
Three pillars of security



Pillar I:
Applied Risk management

- Architectural risk analysis
  - Sometimes called threat modeling or security design analysis
  - Is a best practice and is a touchpoint
- Risk management framework
  - Considers risk analysis and mitigation as a full life cycle activity

Pillar II:
Software Security Touchpoints

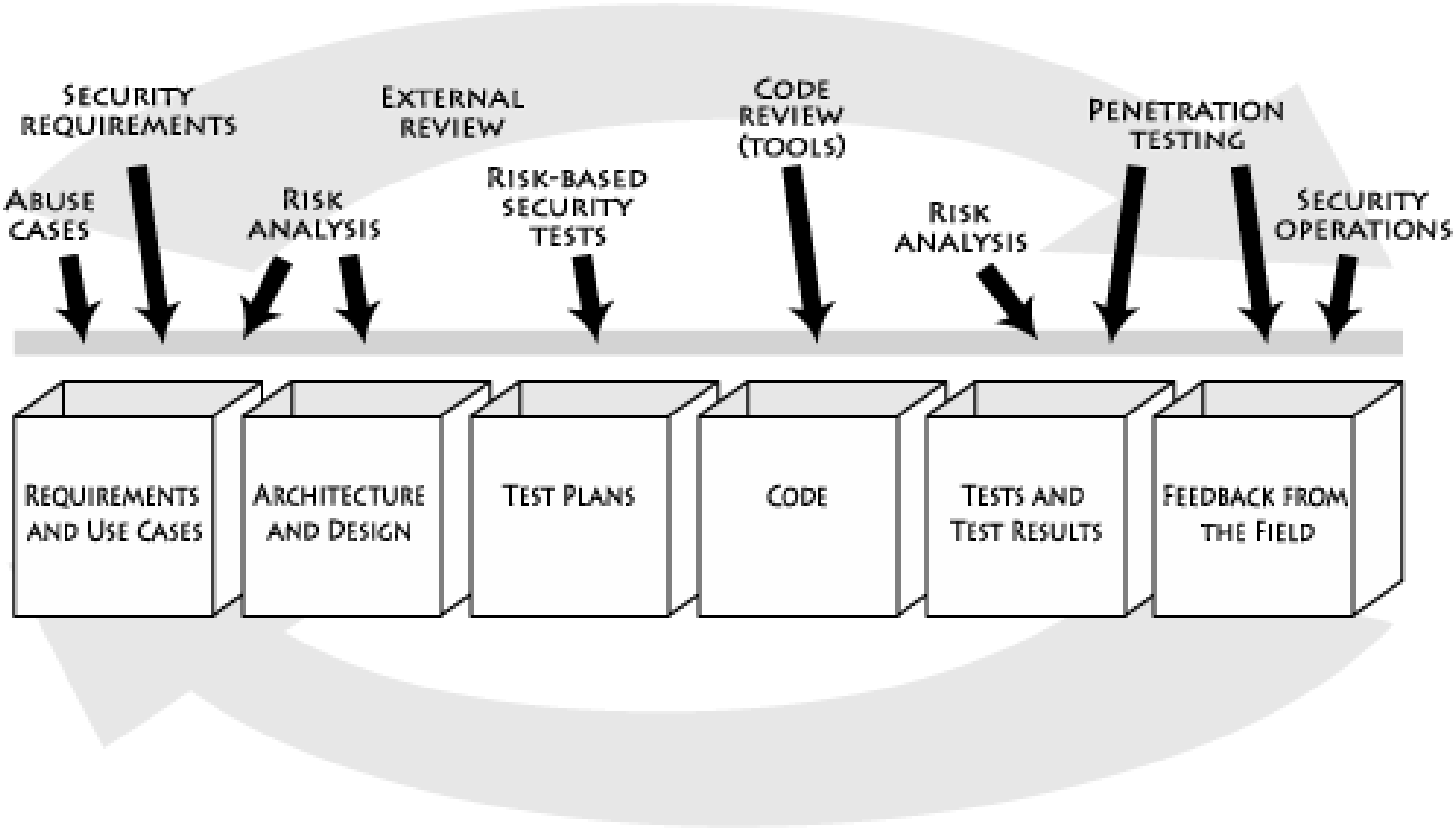
- "Software security is not security software"
  - Software security
    - is system-wide issues (security mechanisms and design security)
    - Emergent property
- Touchpoints in order of effectiveness (based on experience)
  1. Code review (bugs)
  2. Architectural risk analysis (flaws)
     - These two can be swapped
  3. Penetration testing
  4. Risk-based security tests
  5. Abuse cases
  6. Security requirements
  7. Security operations

Effectivemess

Pillar II: (contd.)

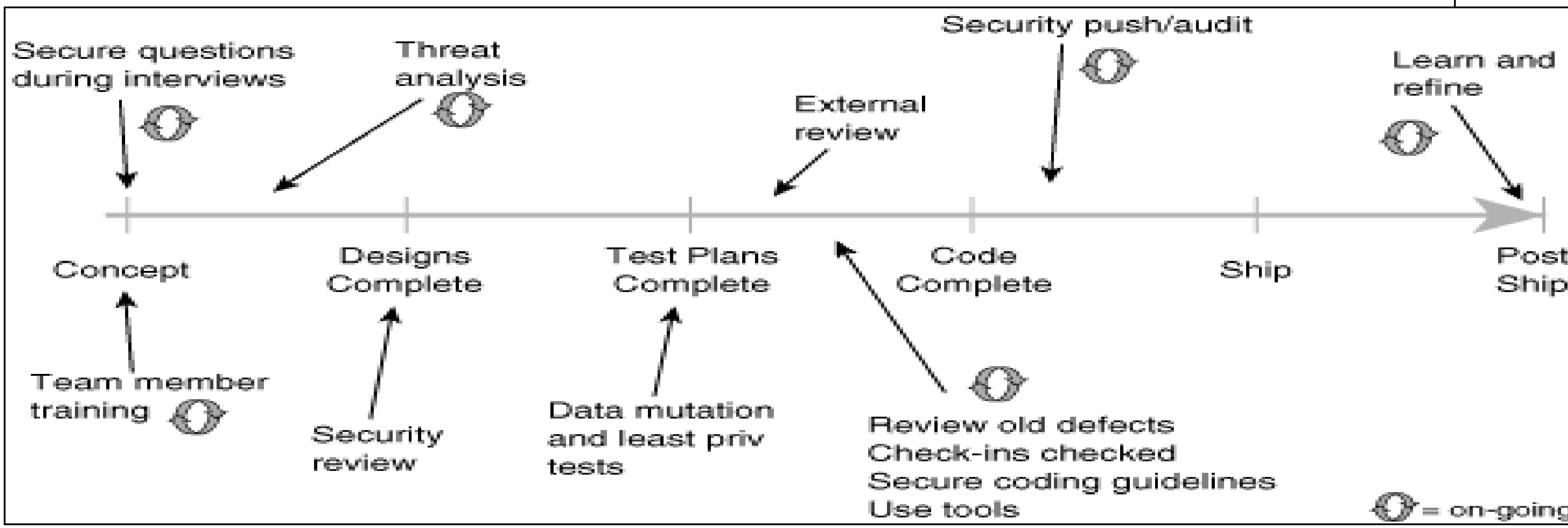
- Many organization
  - Penetration first
    - Is a reactive approach
- CR and ARA can be switched however skipping one solves only half of the problem
- Big organization may adopt these touchpoints simultaneously

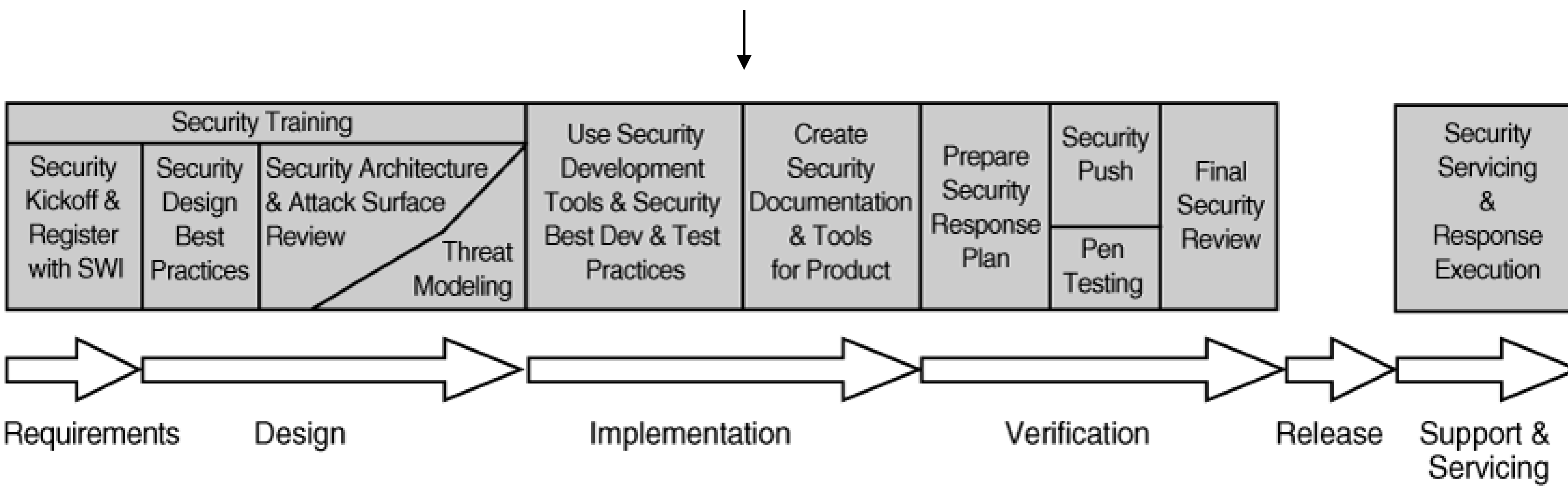
Pillar II: (contd.)



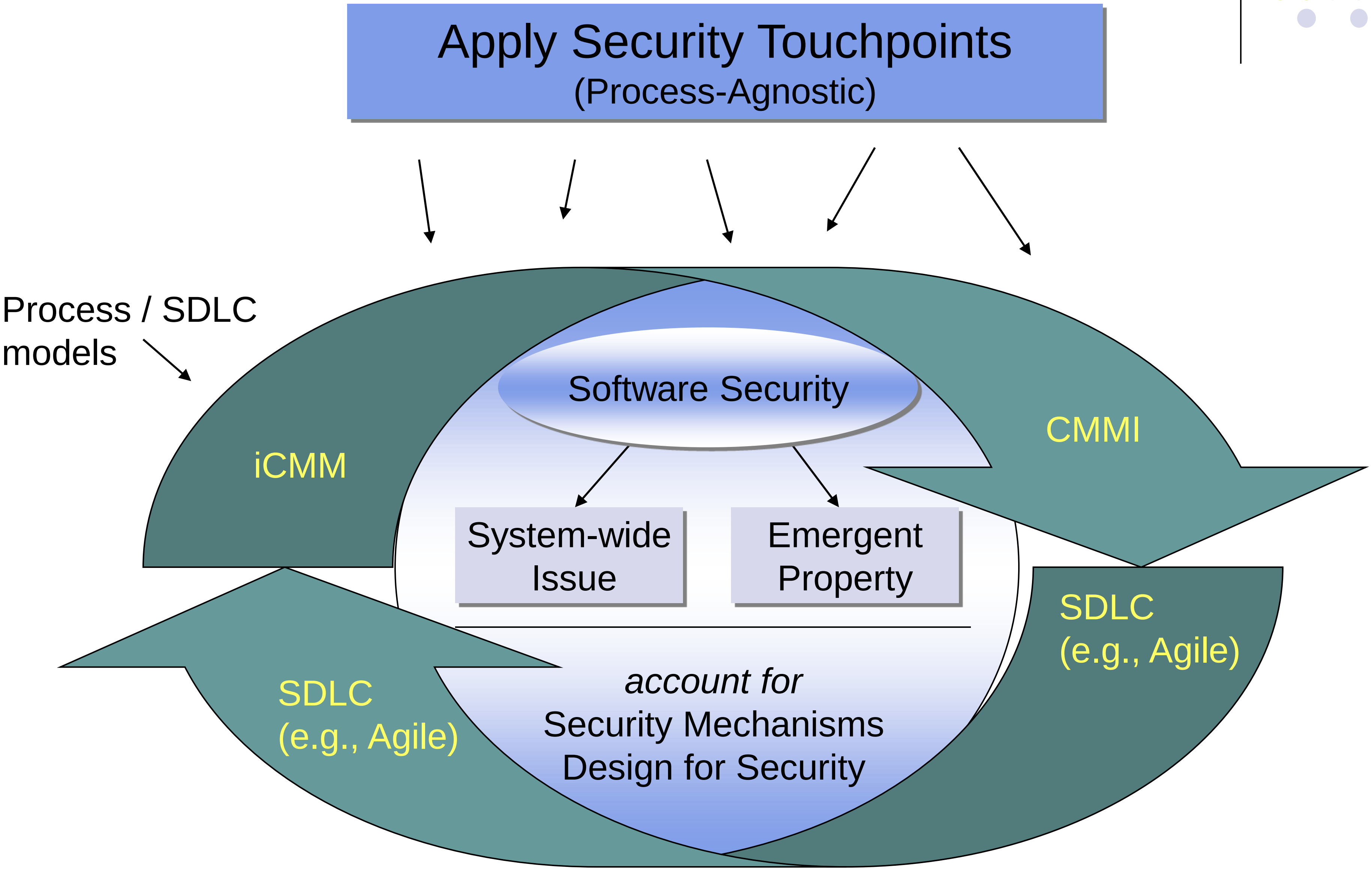
Software security best practices applied to various software artifacts

Pillar II: (contd.) Microsoft's move ..





Pillar II: (contd.)



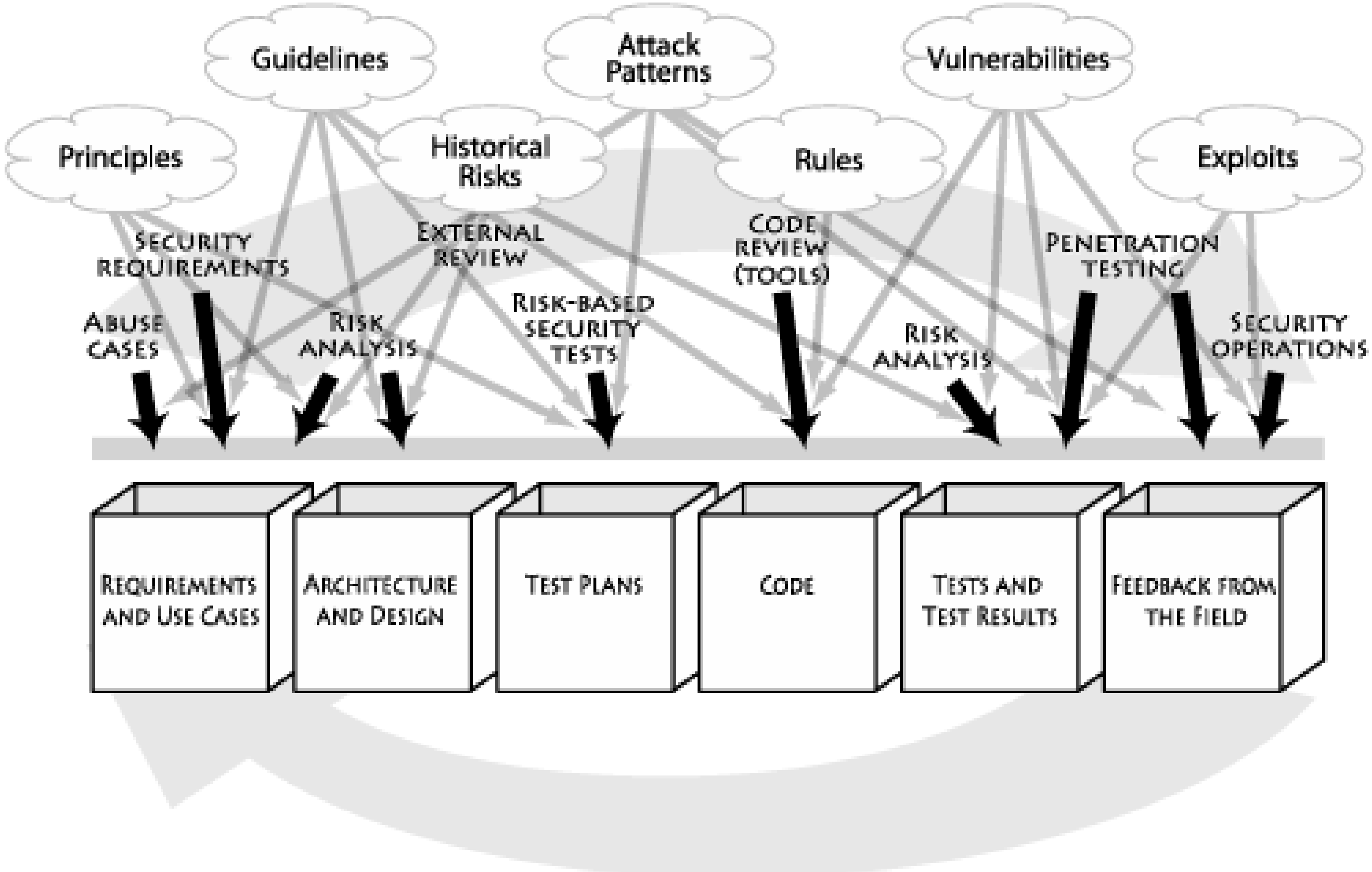
Pillar III: Knowledge

- Involves
  - Gathering, encapsulating, and sharing security knowledge
- Software security knowledge catalogs
  - Principles
  - Guidelines
  - Rules
  - Vulnerabilities
  - Exploits
  - Attack patterns
  - Historical risks

Can be put into three categories

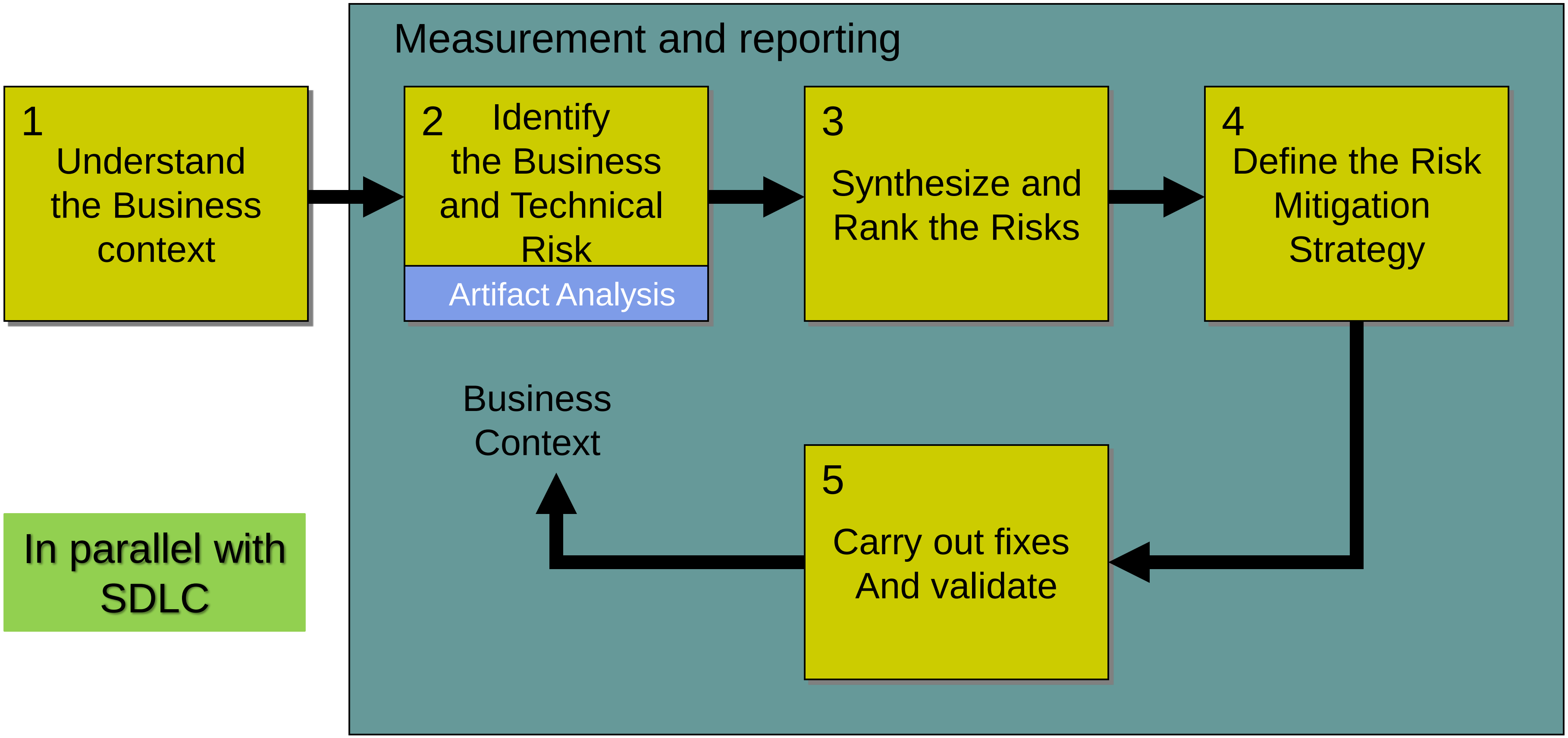
Prescriptive knowledge
Diagnostic knowledge
Historical knowledge

Pillar III: Knowledge catalogs to s/w artifacts



Risk management framework: Five Stages

- RMF occurs in parallel with SDLC activities



Stage 1: Understand Business Context

- Risk management
  - Occurs in a business context
  - Affected by business motivation
- Key activity of an analyst
  - Extract and describe business goals – clearly
    - Increasing revenue; reducing dev cost; meeting SLAs; generating high return on investment (ROI)
  - Set priorities
  - Understand circumstances
- Bottomline – answer the question
  - who cares?

Stage 2: Identify the business & technical risks

- Business risks have impact
  - Direct financial loss; loss of reputation; violation of customer or regulatory requirements; increase in development cost
- Severity of risks
  - Should be captured in financial or project management terms
- Key is –
  - tie technical risks to business context

Stage 3: Synthesize and rank the risks

- Prioritize the risks alongside the business goals
- Assign risks appropriate weights for resolution
- Risk metrics
  - Risk likelihood
  - Risk impact, severity
  - Number of risks mitigated over time

Stage 4: Risk Mitigation Strategy

- Need/Develop a coherent strategy
  - Mitigating risks
  - Cost effective -- account for
    - Cost
    - Implementation time
    - Completeness
    - Impact
    - Likelihood of success
- A mitigation strategy should
  - Be developed within the business context
  - Be based on what the organization can afford, integrate and understand
  - Must directly identify validation techniques

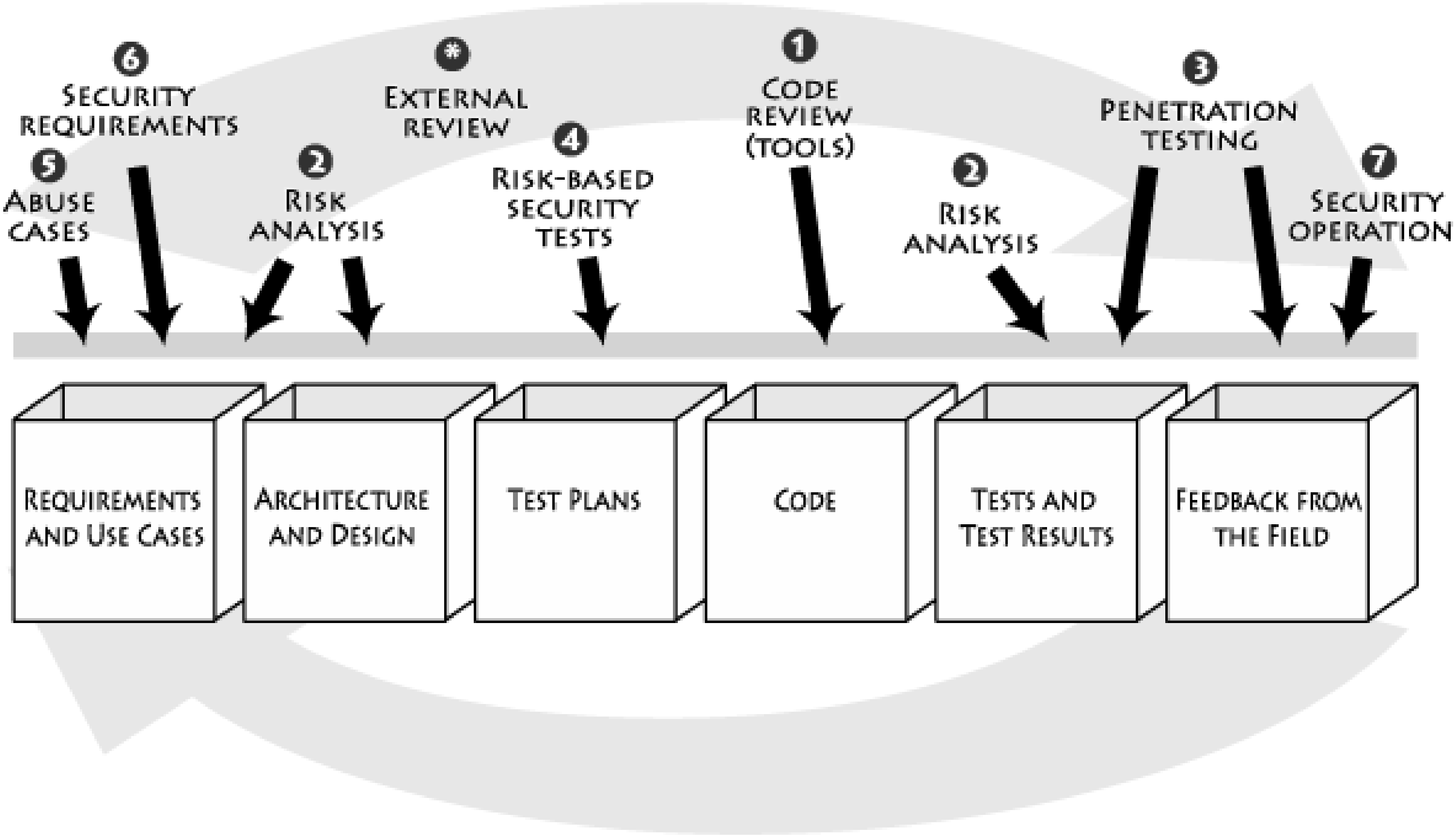
Stage 5: Carry out Fixes and Validate

- Execute the chosen mitigation strategy
    - Rectify the artifacts
  - Measure completeness
  - Estimate:
    - Progress, residual risks
- Validate that risks have been mitigated
  - Testing can be used to demonstrate
  - Develop confidence that unacceptable risk does not remain

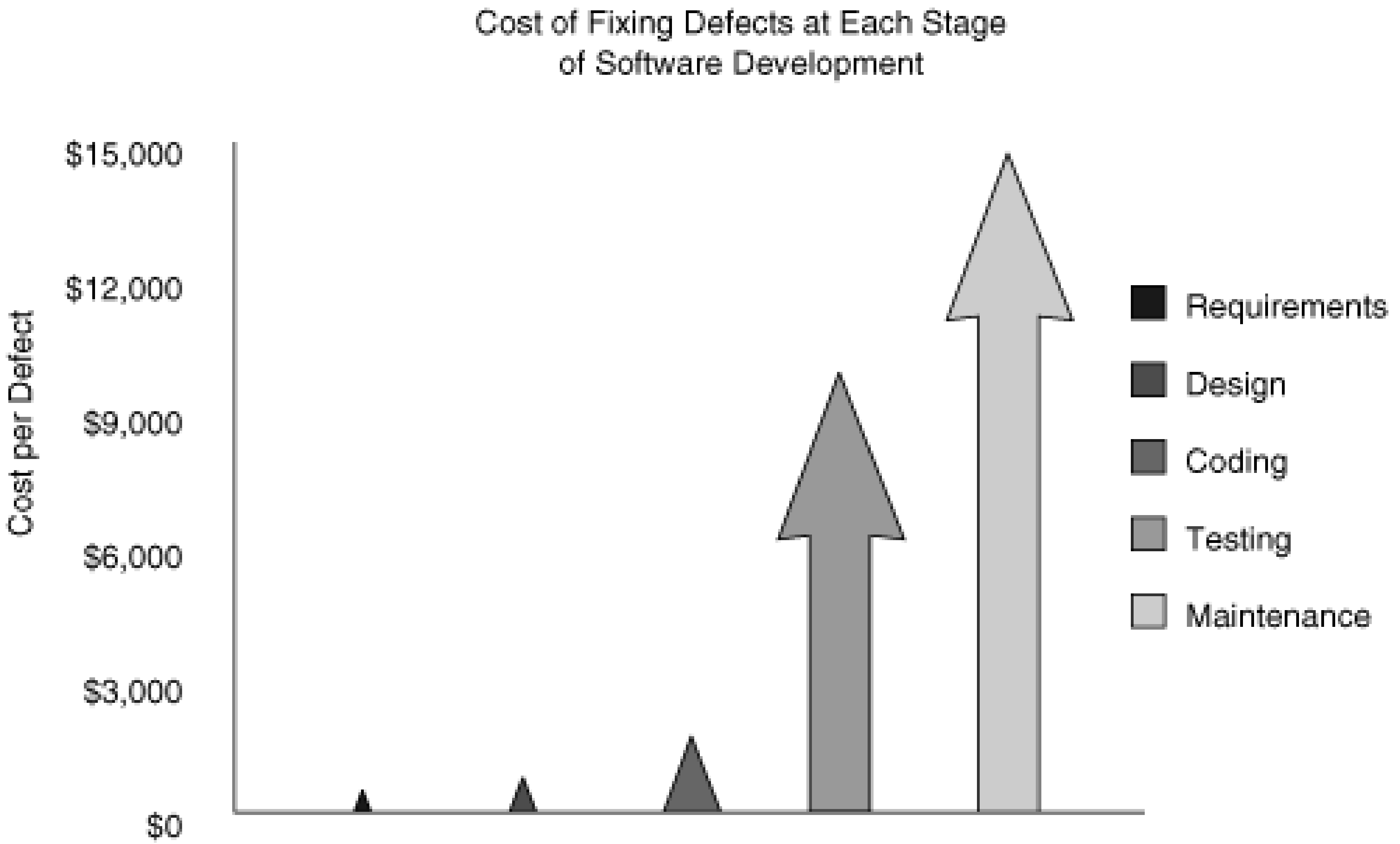
RMF - Multi-loop

- Risk management is a continuous process
  - Five stages may need to be applied many times
  - Ordering may be interleaved in different ways
    - Risk can emerge at any time in SDLC
      - One way – apply in each phase of SDLC
    - Risk can be found between stages
- Level of application
    - Primary – project level: Each stage must capture complete project
    - SDLC phase level
    - Artifact level
- RM is
    - Cumulative
    - At times arbitrary and difficult to predict

Seven Touchpoints



Cost of fixing defect at each stage



Code review

- Focus is on implementation bugs
  - Essentially those that static analysis can find
  - Security bugs are real problems – but architectural flaws are just as big a problem
    - Code review can capture ***only half of the problems***
  - E.g., Buffer overflow bug in a particular line of code
  - Architectural problems are very difficult to find by looking at the code
    - Specially true for today’s large software

Code review

- Taxonomy of coding errors
  - Input validation and representation
    - Some sources of problems
      - Metacharacters, alternate encodings, numeric representations
      - Forgetting input validation
      - Trusting input too much
      - Example: buffer overflow; integer overflow
  - API abuse (API represents contract between caller and callee)
    - E.g., failure to enforce principle of least privilege
  - Security features
    - Getting right security features is difficult
    - E.g., insecure randomness, password management, authentication, access control, cryptography, privilege management, etc.

Code review

- Taxonomy of coding errors
  - Time and state
    - Typical race condition issues
    - E.g., TOCTOU; deadlock
  - Error handling
    - Security defects related to error handling are very common
    - Two ways
      - Forget to handle errors or handling them roughly
      - Produce errors that either give out way too much information or so radioactive no one wants to handle them
    - E.g., unchecked error value; empty catch block

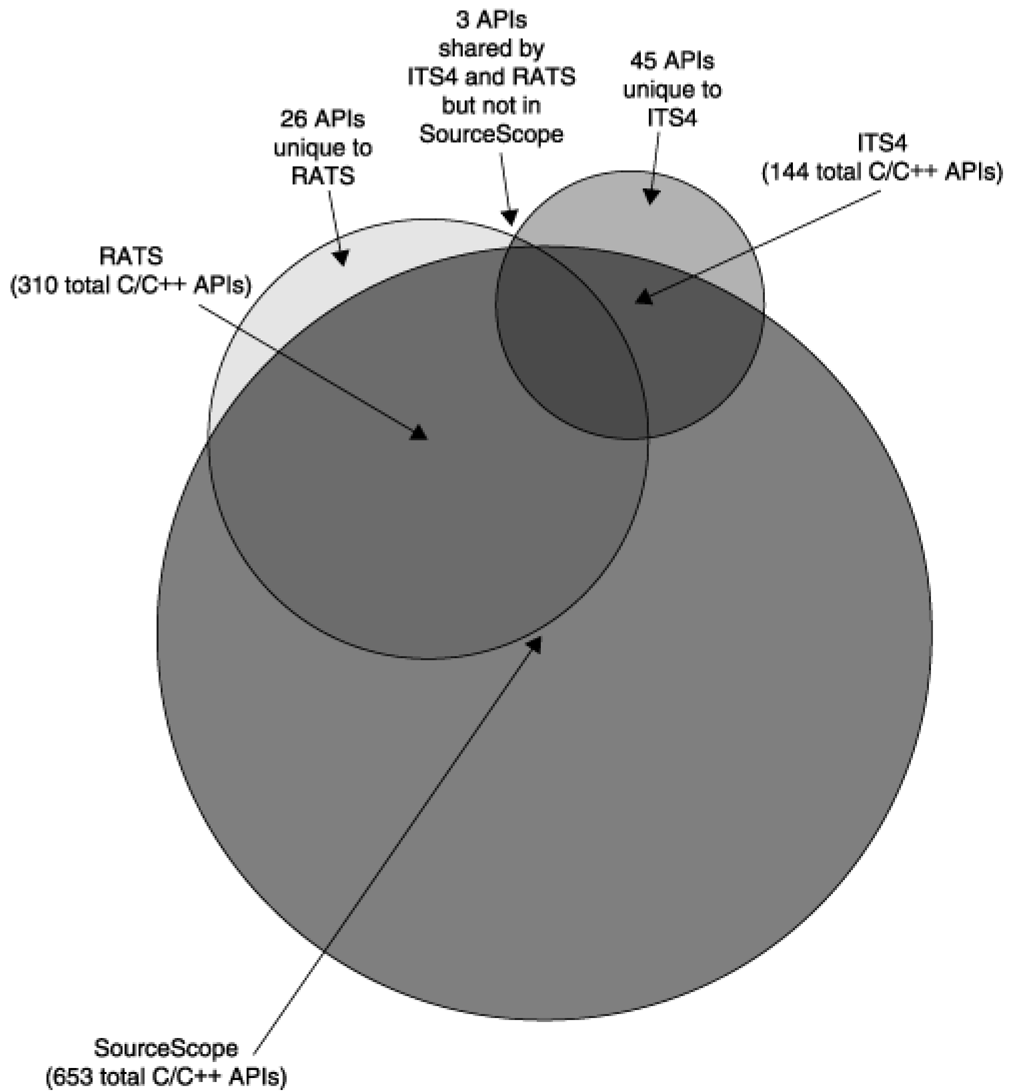
Code review

- Taxonomy of coding errors
  - Code quality
    - Poor code quality leads to unpredictable behavior
    - Poor usability
    - Allows attacker to stress the system in unexpected ways
    - E.g., Double free; memory leak
  - Encapsulation
    - Object oriented approach
    - Include boundaries
    - E.g., comparing classes by name
  - Environment
    - Everything outside of the code but is important for the security of the software
    - E.g., password in configuration file (hardwired)

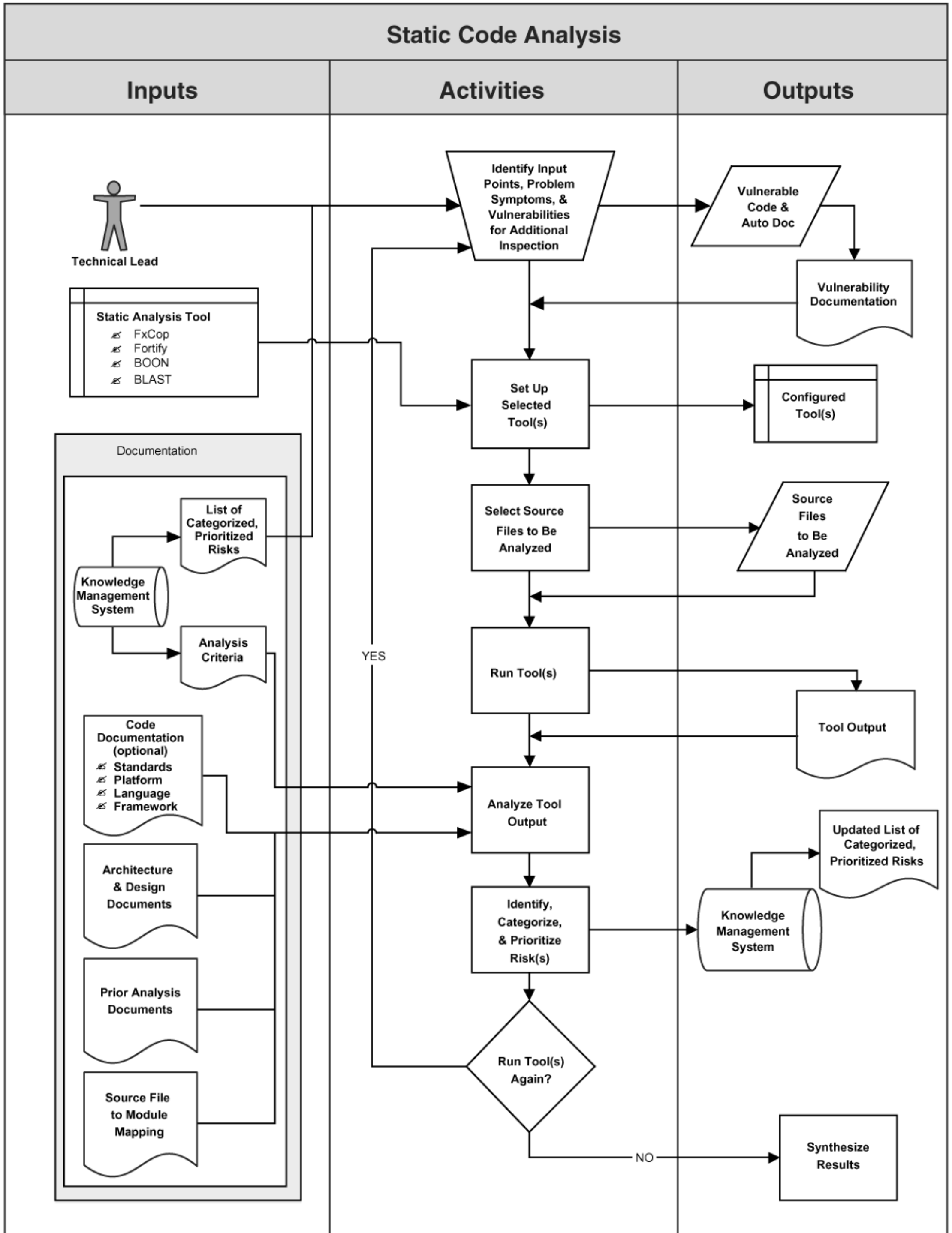
Code review

- Static analysis tools
  - False negative (wrong sense of security)
    - A sound tool does not generate false negatives
  - False positives
  - Some examples
    - ITS4 (It's The Software Stupid Security Scanner);
    - RATS; Flawfinder

Rules overlap



Cigital Static analysis process



Architectural risk analysis

- Design flaws
  - about 50% of security problem
  - Can’t be found by looking at code
    - A higher level of understanding required
- Risk analysis
  - Track risk over time
  - Quantify impact
  - Link system-level concerns to probability and impact measures
  - Fits with the RMF

ARA

- Three critical steps
  - Attack resistance analysis
  - Ambiguity analysis
  - Weakness Analysis

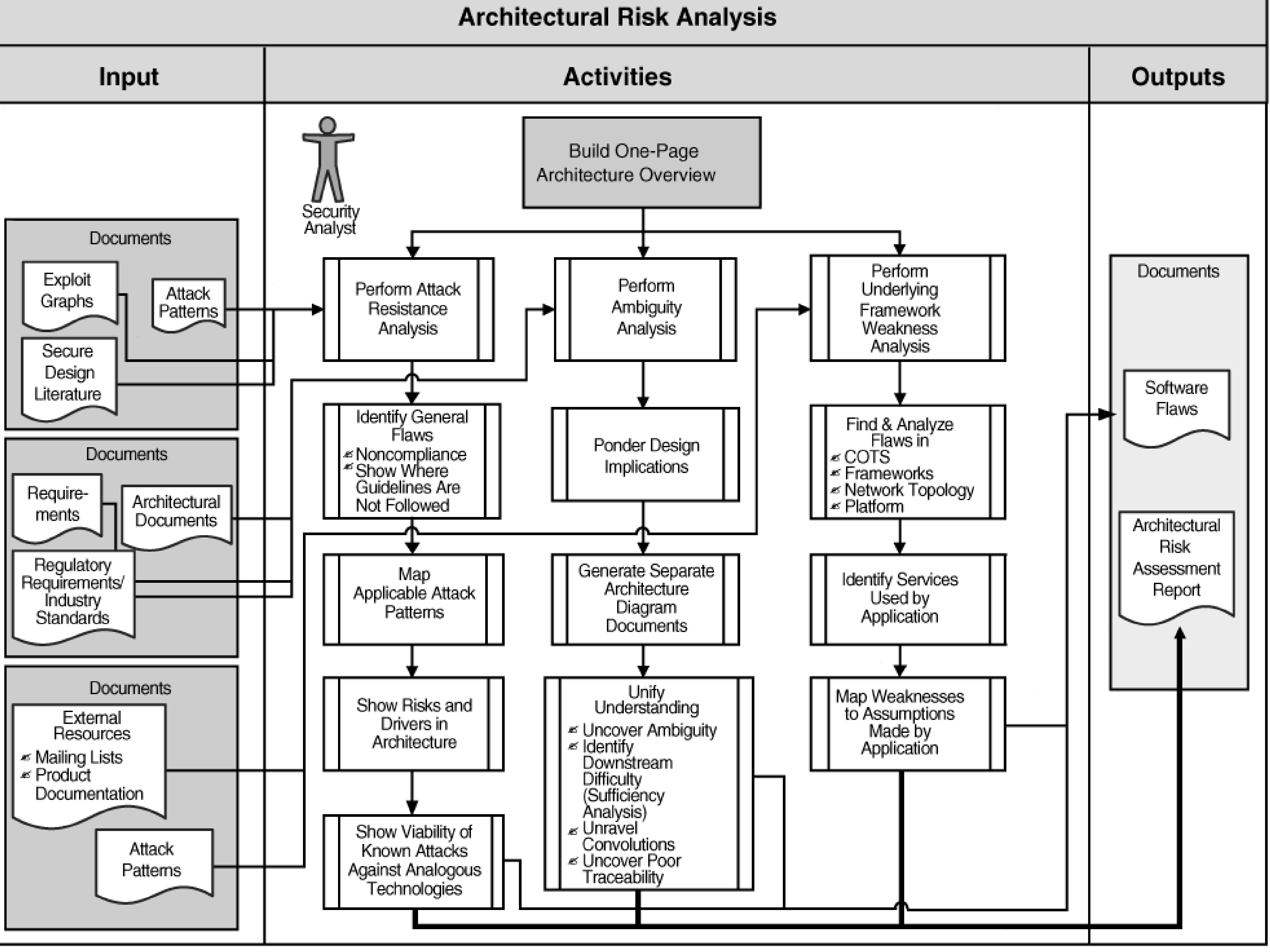
ARA process

- Attack resistance analysis
  - Steps
    - Identify general flaws using secure design literature and checklists
      - Knowledge base of historical risks useful
    - Map attack patterns using either the results of abuse case or a list of attack patterns
    - Identify risk based on checklist
    - Understand and demonstrate the viability of these known attacks
      - Use exploit graph or attack graph

- Note: particularly good for finding known problems

ARA process

- Ambiguity analysis
  - Discover new risks – creativity requried
  - A group of analyst and experience helps – use multiple points of view
    - Unify understanding after independent analysis
  - Uncover ambiguity and inconsistencies
- Weakness analysis
  - Assess the impact of external software dependencies
  - Modern software
    - is built on top of middleware such as .NET and J2EE
    - Use DLLs or common libraries
  - Need to consider
    - COTS
    - Framework
    - Network topology
    - Platform
    - Physical environment
    - Build environment


Architectural Risk Analysis
Input
Activities
Outputs
Security Analyst
Documents
Exploit Graphs
Attack Patterns
Secure Design Literature
Documents
Require-ments
Architectural Documents
Regulatory Requirements/ Industry Standards
Documents
External Resources
Mailing Lists
Product Documentation
Attack Patterns
Build One-Page Architecture Overview
Perform Attack Resistance Analysis
Perform Ambiguity Analysis
Perform Underlying Framework Weakness Analysis
Identify General Flaws
Noncompliance
Show Where Guidelines Are Not Followed
Ponder Design Implications
Find & Analyze Flaws in
COTS
Frameworks
Network Topology
Platform
Map Applicable Attack Patterns
Generate Separate Architecture Diagram Documents
Identify Services Used by Application
Show Risks and Drivers in Architecture
Unify Understanding
Uncover Ambiguity
Identify Downstream Difficulty (Sufficiency Analysis)
Unravel Convolutions
Uncover Poor Traceability
Map Weaknesses to Assumptions Made by Application
Show Viability of Known Attacks Against Analogous Technologies
Documents
Software Flaws
Architectural Risk Assessment Report

Software penetration testing

- Most commonly used today
- Currently
  - Outside $\rightarrow$ in approach
  - Better to do after code review and ARA
  - As part of final preparation acceptance regimen
  - One major limitation
    - Almost always a too-little-too-late attempt at the end of a development cycle
      - Fixing things at this stage
        - May be very expensive
        - Reactive and defensive

Software penetration testing

- A better approach
  - Penetration testing from the beginning and throughout the life cycle
  - Penetration test should be driven by perceived risk
  - Best suited for finding configuration problems and other environmental factors
  - Make use of tools
    - Takes care of majority of grunt work
    - Tool output lends itself to metrics
    - Eg.,
      - fault injection tools;
      - attacker's toolkit: disassemblers and decompilers; coverage tools monitors

Risk based security testing

- Testing must be
  - Risk-based
  - Grounded in both the system's architectural reality and the attacker's mindset
    - Better than classical black box testing
  - Different from penetration testing
    - Level of approach
    - Timing of testing
      - Penetration testing is primarily on completed software in operating environment; outside $\rightarrow$ in

Risk based security testing

- Security testing
  - Should start at feature or component/unit level testing
  - Must involve two diverse approaches
    - Functional security testing
      - Testing security mechanisms to ensure that their functionality is properly implemented
    - Adversarial security testing
      - Performing risk-based security testing motivated by understanding and simulating the attacker's approach

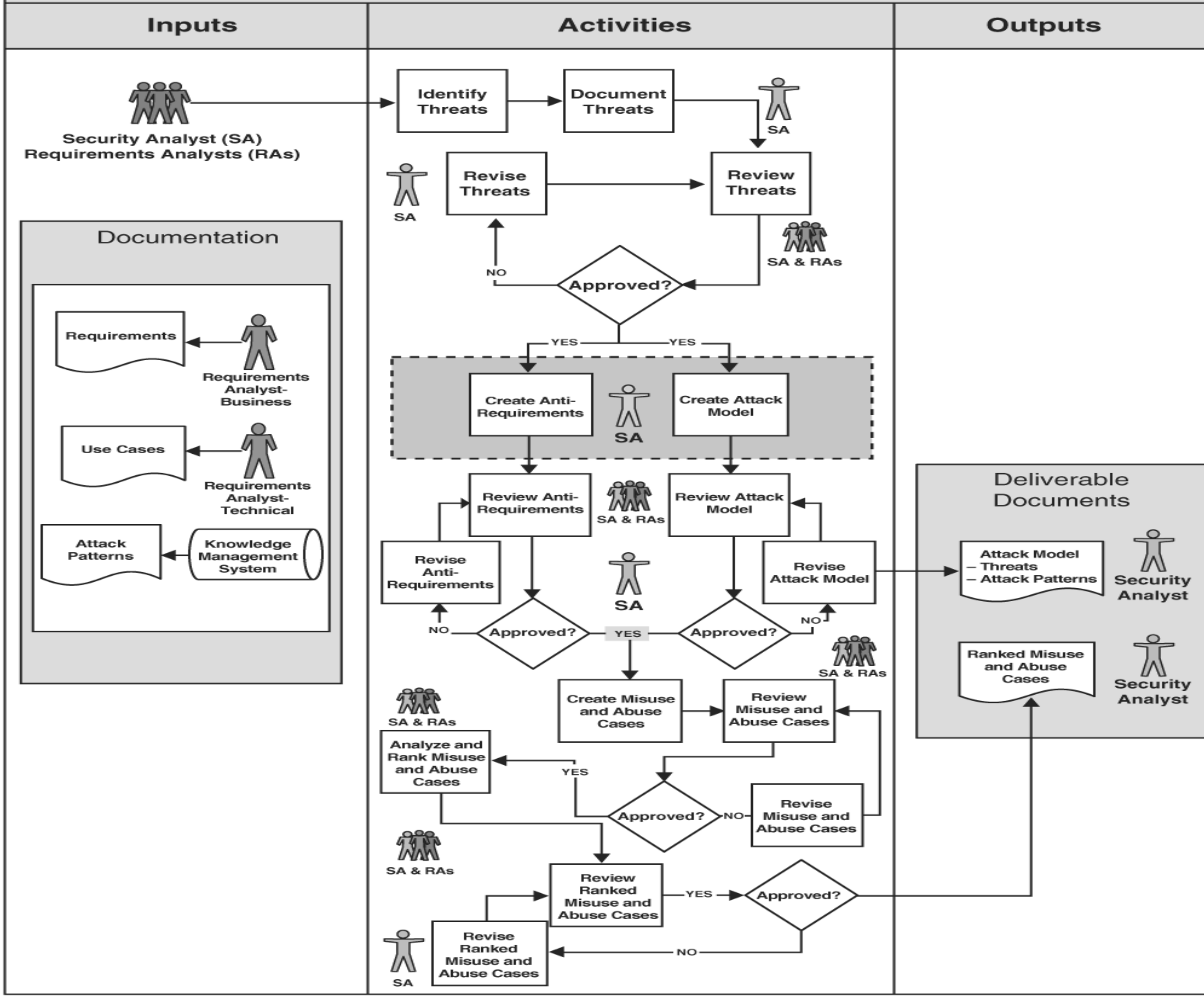
Abuse cases

- Creating anti-requirements
  - Important to think about
    - Things that you don’t want your software to do
    - Requires: security analysis + requirement analysis
  - Anti-requirements
    - Provide insight into how a malicious user, attacker, thrill seeker, competitor can abuse your system
    - Considered throughout the lifecyle
      - indicate what happens when a required security function is not included

Abuse cases

- Creating an attack model
  - Based on known attacks and attack types
  - Do the following
    - Select attack patterns relevant to your system – build abuse case around the attack patterns
    - Include anyone who can gain access to the system because threats must encompass all potential sources
  - Also need to model attacker

Abuse Cases



Inputs
Activities
Outputs
Security Analyst (SA)
Requirements Analysts (RAs)
Documentation
Requirements
Requirements Analyst-Business
Use Cases
Requirements Analyst-Technical
Attack Patterns
Knowledge Management System
Identify Threats
Document Threats
SA
Revise Threats
SA
Review Threats
SA & RAs
Approved?
NO
YES
YES
Create Anti-Requirements
SA
Create Attack Model
Review Anti-Requirements
SA & RAs
Review Attack Model
Revise Anti-Requirements
SA
Revise Attack Model
NO
Approved?
YES
Approved?
NO
SA & RAs
Create Misuse and Abuse Cases
Review Misuse and Abuse Cases
SA & RAs
Analyze and Rank Misuse and Abuse Cases
YES
Approved?
NO
Revise Misuse and Abuse Cases
SA & RAs
Review Ranked Misuse and Abuse Cases
YES
Approved?
NO
SA
Revise Ranked Misuse and Abuse Cases
Deliverable Documents
Attack Model
– Threats
– Attack Patterns
Security Analyst
Ranked Misuse and Abuse Cases
Security Analyst

Security requirements and operations

- Security requirements
  - Difficult tasks
  - Should cover both overt functional security and emergent characteristics
    - Use requirements engineering approach
- Security operations
  - Integrate security operations
    - E.g., software security should be integrated with network security

Handout: Coding Errors

- Input validation and representation
- API Abuse
- Secure Features
- Time and State
- Error Handling
- Code Quality
- Encapsulation
- Environment

Building Security In Maturity Model (BSIMM-V)

- Purpose:
  - quantify the activities carried out by *real software security initiatives*
- Requires
  - a framework to describe all of the initiatives uniformly.
  - Software Security Framework (SSF) and activity descriptions provide
    - a common vocabulary for explaining the salient elements of a software security initiative

Building Security In Maturity Model (BSIMM-V)

- How it was built
  - Software Security Framework
    - Based on knowledge of software security practices
  - Set of common activities
    - Based on interviews with executives in charge of software security interviews
  - Created scoreboards for each of the nine initiatives – reviewed by the participates

BSIMM Objectives

- The BSIMM is appropriate where business goals for software security include:
  - Informed risk management decisions
  - Clarity on what is “the right thing to do” for everyone involved in software security
  - Cost reduction through standard, repeatable processes
  - Improved code quality

Acknowledgement: Figures are from the BSIMM-V documents

Software Security Framework

- Twelve practices in four domains

The Software Security Framework (SSF)			
Governance	**Intelligence**	**SSDL Touchpoints**	**Deployment**
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

Practices that help organize, manage, and measure a software security initiative

Practices that result in collections of corporate knowledge used in carrying out software security activities

Practices that interface with traditional network security and software maintenance organizations

BSIMM-V

- Maturity model: a series of activities associated with each of the twelve practices; and goals of each practice

Domain	Business Goals
Gov	
Inte	
SSD	
Dep	

Domain	Practice	Business Goals
Governance	Strategy and Metrics	Transparency of expectations, Accountability for results
	Compliance and Policy	Prescriptive guidance for all stakeholders, Auditability
	Training	Knowledgeable workforce, Error correction
Intelligence	Attack Models	Customized knowledge
	Security Features and Design	Reusable designs, Prescriptive guidance for all stakeholders
	Standards and Requirements	Prescriptive guidance for all stakeholders
SSDL Touchpoints	Architecture Analysis	Quality control
	Code Review	Quality control
	Security Testing	Quality control
Deployment	Penetration Testing	Quality control
	Software Environment	Change management
	Configuration Management and Vulnerability Management	Change management

BSIMM Skeleton - assessment

- Detailed description of each activity is provided in the BSIMM document

	Objective	Activity	Level
	GOVERNANCE: STRATEGY AND METRICS — Planning, assigning roles and responsibilities, identifying software security goals, determining budgets, identifying metrics and gates.		
[SM1.1]	make the plan explicit	publish process (roles, responsibilities, plan), evolve as necessary	1
[SM1.2]	build support throughout organization	create evangelism role and perform internal marketing	
[SM1.3]	secure executive buy-in	educate executives	
[SM1.4]	establish SSDL gates (but do not enforce)	identify gate locations, gather necessary artifacts	
[SM1.6]	make clear who's taking the risk	require security sign-off	
[SM2.1]	foster transparency (or competition)	publish data about software security internally	2
[SM2.2]	change behavior	enforce gates with measurements and track exceptions	
[SM2.3]	create broad base of support	create or grow a satellite	
[SM2.5]	define success	identify metrics and use them to drive budgets	
[SM3.1]	know where all apps in your inventory stand	use an internal tracking application with portfolio view	3
[SM3.2]	create external support	run an external marketing program	

Other Skeletons

GOVERNANCE: TRAINING			
	Objective	Activity	Level
[T1.1]	promote culture of security throughout the organization	provide awareness training	1
[T1.5]	build capabilities beyond awareness	deliver role-specific advanced curriculum (tools, technology stacks, bug parade)	
[T1.6]	see yourself in the problem	create and use material specific to company history	
[T1.7]	reduce impact on training targets and build delivery staff	deliver on-demand individual training	
[T2.5]	educate/strengthen social network	enhance satellite through training and events	2
[T2.6]	ensure new hires enhance culture	include security resources in onboarding	
[T2.7]	create social network tied into dev	identify satellite through training	
[T3.1]	align security culture with career path	reward progression through curriculum (certification or HR)	3
[T3.2]	spread security culture to providers	provide training for vendors or outsourced workers	
[T3.3]	market security culture as differentiator	host external software security events	
[T3.4]	keep staff up-to-date and address turnover	require an annual refresher	
[T3.5]	act as informal resource to leverage teachable moments	establish SSG office hours	

Other Skeletons

INTELLIGENCE: ATTACK MODELS

Threat modeling, abuse cases, data classification, technology-specific attack patterns.

	Objective	Activity	Level
[AM1.1]	understand attack basics	build and maintain a top N possible attacks list	1
[AM1.2]	prioritize applications by data consumed/manipulated	create a data classification scheme and inventory	
[AM1.3]	understand the "who" of attacks	identify potential attackers	
[AM1.4]	understand the organization's history	collect and publish attack stories	
[AM1.5]	stay current on attack/vulnerability environment	gather attack intelligence	
[AM1.6]	communicate attacker perspective	build an internal forum to discuss attacks (T: standards/req)	
[AM2.1]	provide resources for security testing and AA	build attack patterns and abuse cases tied to potential attackers	2
[AM2.2]	understand technology-driven attacks	create technology-specific attack patterns	
[AM3.1]	get ahead of the attack curve	have a science team that develops new attack methods	3
[AM3.2]	arm testers and auditors	create and use automation to do what the attackers will do	

Other Skeletons

INTELLIGENCE: SECURITY FEATURES AND DESIGN

Security patterns for major security controls, middleware frameworks for controls, proactive security guidance.

	Objective	Activity	Level
[SFD1.1]	create proactive security guidance around security features	build and publish security features	1
[SFD1.2]	inject security thinking into architecture group	engage SSG with architecture	
[SFD2.1]	create proactive security design based on technology stacks	build secure-by-design middleware frameworks and common libraries (T: code review)	2
[SFD2.2]	address the need for new architecture	create SSG capability to solve difficult design problems	
[SFD3.1]	formalize consensus on design	form a review board or central committee to approve and maintain secure design patterns	3
[SFD3.2]	promote design efficiency	require use of approved security features and frameworks (T: AA)	
[SFD3.3]	practice reuse	find and publish mature design patterns from the organization	

Other Skeletons

INTELLIGENCE: STANDARDS AND REQUIREMENTS

Explicit security requirements, recommended COTS, standards for major security controls, standards for technologies in use, standards review board.

	Objective	Activity	Level
[SR1.1]	meet demand for security features	create security standards (T: sec features/design)	1
[SR1.2]	ensure that everybody knows where to get latest and greatest	create a security portal	
[SR1.3]	compliance strategy	translate compliance constraints to requirements	
[SR1.4]	tell people what to look for in code review	use secure coding standards	
[SR2.2]	formalize standards process	create a standards review board	2
[SR2.3]	reduce SSG workload	create standards for technology stacks	
[SR2.4]	manage open source risk	identify open source	
[SR2.5]	gain buy-in from legal department and standardize approach	create SLA boilerplate (T: compliance and policy)	
[SR3.1]	manage open source risk	control open source risk	3
[SR3.2]	educate third-party vendors	communicate standards to vendors	

Other Skeletons

SSDL TOUCHPOINTS: ARCHITECTURE ANALYSIS

Capturing software architecture diagrams, applying lists of risks and threats, adopting a process for review, building an assessment and remediation plan.

	Objective	Activity	Level
[AA1.1]	get started with AA	perform security feature review	1
[AA1.2]	demonstrate value of AA with real data	perform design review for high-risk applications	
[AA1.3]	build internal capability on security architecture	have SSG lead design review efforts	
[AA1.4]	have a lightweight approach to risk classification and prioritization	use a risk questionnaire to rank applications	
[AA2.1]	model objects	define and use AA process	2
[AA2.2]	promote a common language for describing architecture	standardize architectural descriptions (including data flow)	
[AA2.3]	build capability organization-wide	make SSG available as AA resource or mentor	
[AA3.1]	build capabilities organization-wide	have software architects lead design review efforts	3
[AA3.2]	build proactive security architecture	drive analysis results into standard architecture patterns (T: sec features/design)	

Other Skeletons

	SSDL TOUCHPOINTS: CODE REVIEW Use of code review tools, development of customized rules, profiles for tool use by different roles, manual analysis, ranking/measuring results.		
	Objective	Activity	Level
[CR1.1]	know which bugs matter to you	create top N bugs list (real data preferred) (T: training)	1
[CR1.2]	review high-risk applications opportunistically	have SSG perform ad hoc review	
[CR1.4]	drive efficiency/consistency with automation	use automated tools along with manual review	
[CR1.5]	find bugs earlier	make code review mandatory for all projects	
[CR1.6]	know which bugs matter (for training)	use centralized reporting to close the knowledge loop and drive training (T: strategy/metrics)	
[CR2.2]	drive behavior objectively	enforce coding standards	2
[CR2.5]	make most efficient use of tools	assign tool mentors	
[CR2.6]	drive efficiency/reduce false positives	use automated tools with tailored rules	
[CR3.2]	combine assessment techniques	build a factory	3
[CR3.3]	handle new bug classes in an already scanned codebase	build capability for eradicating specific bugs from the entire codebase	
[CR3.4]	address insider threat from development	automate malicious code detection	

Other Skeletons

SSDL TOUCHPOINTS: SECURITY TESTING

Use of black box security tools in QA, risk driven white box testing, application of the attack model, code coverage analysis.

	Objective	Activity	Level
[ST1.1]	execute adversarial tests beyond functional	ensure QA supports edge/boundary value condition testing	1
[ST1.3]	start security testing in familiar functional territory	drive tests with security requirements and security features	
[ST2.1]	use encapsulated attacker perspective	integrate black box security tools into the QA process	2
[ST2.4]	facilitate security mindset	share security results with QA	
[ST3.1]	include security testing in regression	include security tests in QA automation	3
[ST3.2]	teach tools about your code	perform fuzz testing customized to application APIs	
[ST3.3]	probe risk claims directly	drive tests with risk analysis results	
[ST3.4]	drive testing depth	leverage coverage analysis	
[ST3.5]	move beyond functional testing to attacker's perspective	begin to build and apply adversarial security tests (abuse cases)	

Other Skeletons

DEPLOYMENT: PENETRATION TESTING Vulnerabilities in final configuration, feeds to defect management and mitigation.			
	Objective	Activity	Level
[PT1.1]	demonstrate that your organization's code needs help too	use external penetration testers to find problems	1
[PT1.2]	fix what you find to show real progress	feed results to the defect management and mitigation system (T: config/vuln mgmt)	
[PT1.3]	create internal capability	use penetration testing tools internally	
[PT2.2]	promote deeper analysis	provide penentration testers with all available information (T: AA & code review)	2
[PT2.3]	sanity check constantly	schedule periodic penetration tests for application coverage	
[PT3.1]	keep up with edge of attacker's perspective	use external penetration testers to perform deep-dive analysis	3
[PT3.2]	automate for efficiency without losing depth	have the SSG customize penetration testing tools and scripts	

Other Skeletons

DEPLOYMENT: SOFTWARE ENVIRONMENT

OS and platform patching, Web application firewalls, installation and configuration documentation, application monitoring, change management, code signing.

	Objective	Activity	Level
[SE1.1]	watch software	use application input monitoring	1
[SE1.2]	provide a solid host/network foundation for software	ensure host and network security basics are in place	
[SE2.2]	guide operations on application needs	publish installation guides	2
[SE2.4]	protect apps (or parts of apps) that are published over trust boundaries	use code signing	
[SE3.2]	protect IP and make exploit development harder	use code protection	3
[SE3.3]	watch software	use application behavior monitoring and diagnostics	

Other Skeletons

DEPLOYMENT: CONFIGURATION MANAGEMENT AND VULNERABILITY MANAGEMENT Patching and updating applications, version control, defect tracking and remediation, incident handling.			
	Objective	Activity	Level
[CMVM1.1]	know what to do when something bad happens	create or interface with incident response	1
[CMVM1.2]	use ops data to change dev behavior	identify software defects found in operations monitoring and feed them back to development	
[CMVM2.1]	be able to fix apps when they are under direct attack	have emergency codebase response	2
[CMVM2.2]	use ops data to change dev behavior	track software bugs found in operations through the fix process	
[CMVM2.3]	know where the code is	develop an operations inventory of applications	
[CMVM3.1]	learn from operational experience	fix all occurrences of software bugs found in operations (T: code review)	3
[CMVM3.2]	use ops data to change dev behavior	enhance the SSDL to prevent software bugs found in operations	
[CMVM3.3]	ensure processes are in place to minimize software incident impact	simulate software crisis	
[CMVM3.4]	*engage external researchers in vulnerability discovery*	*operate a bug bounty program*	

Core BSIMM activities

- About 64% carried out

Twelve Core Activities Everybody Does

	Objective	Activity
[SM1.4]	establish SSDL gates (but do not enforce)	identify gate locations, gather necessary artifacts
[CP1.2]	promote privacy	identify PII obligations
[T1.1]	promote culture of security throughout the organization	provide awareness training
[AM1.2]	prioritize applications by data consumed/manipulated	create a data classification scheme and inventory
[SFD1.1]	create proactive security guidance around security features	build and publish security features
[SR1.1]	meet demand for security features	create security standards
[AA1.1]	get started with AA	perform security feature review
[CR1.4]	drive efficiency/consistency with automation	use automated tools along with manual review
[ST1.3]	start security testing in familiar functional territory	drive tests with security requirements and security features
[PT1.1]	demonstrate that your organization's code needs help too	use external penetration testers to find problems
[SE1.2]	provide a solid host/network foundation for software	ensure host and network security basics are in place
[CMVM1.2]	use ops data to change dev behavior	identify software bugs found in operations monitoring and feed them back to development

Summary

- Building Security In approach
- Building Security In Maturity Model approach