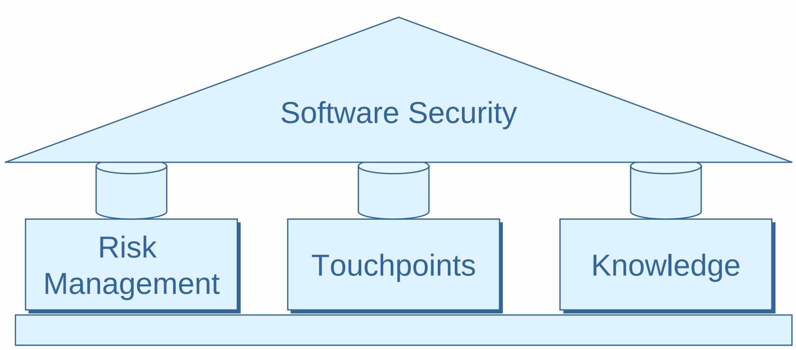
Software Security Touchpoints

March 27, 2006

Three pillars of security

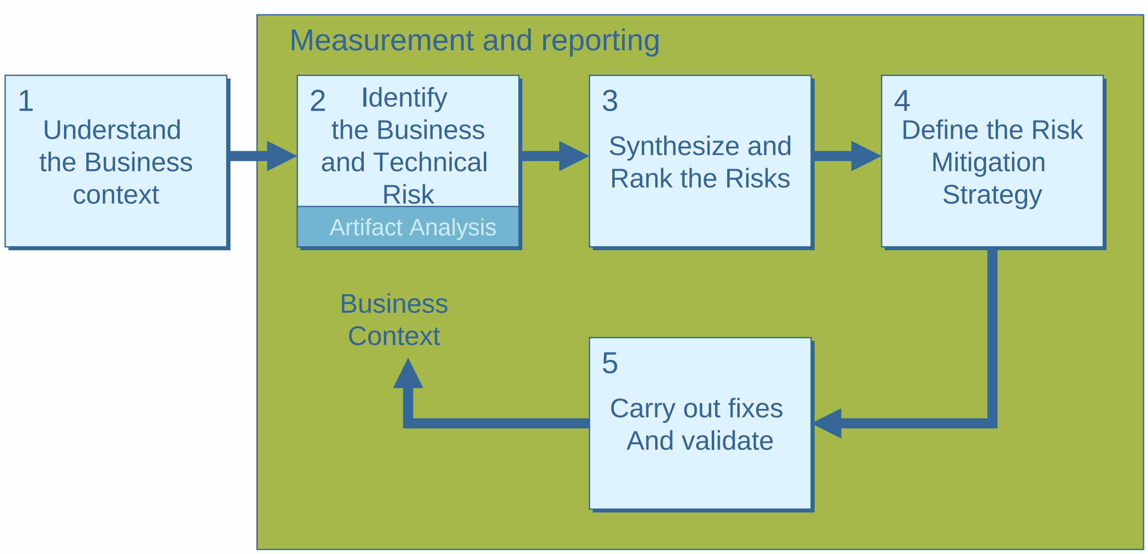


Applied risk management

- 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

Risk management framework

- RMF occurs in parallel with SDLC activities



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)
  - Code review (bugs)
  - Architectural risk analysis (flaws)
    - These two can be swapped
  - Penetration testing
  - Risk-based security tests
  - Abuse cases
  - Security requirements
  - Security operations

Software security touchpoints

- 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

Knowledge

- Software security knowledge catalogs
  - Principles
  - Guidelines
  - Rules
  - Vulnerabilities
  - Exploits
  - Attack patterns
  - Historical risks
- These can be grouped into following categories
  - Prescriptive knowledge
  - Diagnostic knowledge
  - Historical knowledge

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 source 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

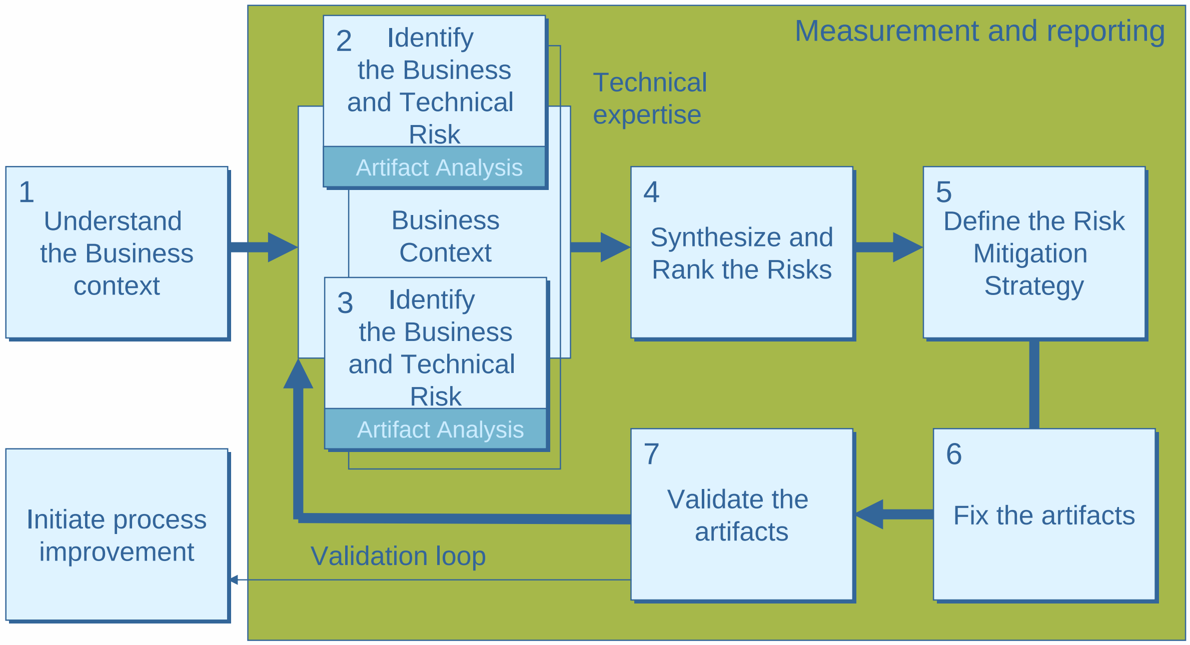
Cigital Static analysis process

- Figure 4-7

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 within RMF



ARA process

- Figure 5-4

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

Software penetration testing

- Most commonly used today
- Currently
  - – Outside->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->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

- Figure 8-1

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

Security requirements and operations

- Security requirements
  - Difficult tasks
  - Should over 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