Lecture 10: Assurance*

CS 392/681: Computer Security
Fall 2006

Nitesh Saxena

*Adopted from a previous lecture by Nasir Memon

Course Admin

- HW#5 being graded (almost done!)
- HW#6
  - Solution to be posted (sorry for delay!)
  - To be graded
- HW#7
  - Solution to be posted
  - To be graded
- HW#8 to be posted
  - Includes last lecture and this one
- Exam on 12/14?
- Last lecture’s ring policy issue
  - I need to check it still; may be I’ll write to Matt Bishop

1/30/2006
Today’s Lecture

- Quick Overview on Assurance
  - Read chapter 17 and slides
- Privacy and Anonymity using Mix Networks

Trust

- **Trustworthy** system has sufficient credible evidence leading one to believe that the system will meet a set of requirements
- **Trust** is a measure of trustworthiness relying on the evidence
- **Assurance** is confidence that a system meets its security requirements based on evidence provided by applying assurance techniques
Relationships

<table>
<thead>
<tr>
<th>Policy</th>
<th>Assurance</th>
<th>Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statement of requirements that explicitly defines the security expectations of the mechanism(s)</td>
<td>Provides justification that the mechanism meets policy through assurance evidence and approvals based on evidence</td>
</tr>
</tbody>
</table>

Problem Sources

1. Requirements definitions, omissions, and mistakes
2. System design flaws
3. Hardware implementation flaws, such as wiring and chip flaws
4. Software implementation errors, program bugs, and compiler bugs
5. System use and operation errors and inadvertent mistakes
6. Willful system misuse
7. Hardware, communication, or other equipment malfunction
8. Environmental problems, natural causes, and acts of God
9. Evolution, maintenance, faulty upgrades, and decommissions
Examples

- Challenger explosion
  - Sensors removed from booster rockets to meet accelerated launch schedule
- Deaths from faulty radiation therapy system
  - Hardware safety interlock removed
  - Flaws in software design
- Bell V22 Osprey crashes
  - Failure to correct for malfunctioning components; two faulty ones could outvote a third
- Intel 486 chip
  - Bug in trigonometric functions

Role of Requirements

- *Requirements* are statements of goals that must be met
  - Vary from high-level, generic issues to low-level, concrete issues
- *Security objectives* are high-level security issues
- *Security requirements* are specific, concrete issues
Types of Assurance

- Policy Assurance
  - Security Policy is a requirement
  - Security Model is a means of detecting/preventing errors, omissions in security policy
  - Policy Assurance: Evidence that security policy is complete/consistent/sound
    - Achieved through use of model

Design Assurance

- Evidence that Design meets Security Policy
  - Validation / verification techniques
Implementation Assurance

- Evidence that the implementation meets the design
- Primarily based on standard software engineering practice

Operational / Administrative Assurance

- Evidence establishing system sustains the security policy requirements during installation, configuration, and day-to-day operation
  - Best: evidence that system *can’t* enter non-secure state
- Least Privilege, Separation of Privilege
- Training, documentation
Life Cycle

- Conception
- Manufacture
- Deployment
- Fielded Product Life

Design and implementation refinement

Security requirements

Design

Implementation

Assurance justification
Conception

- Idea
  - Decisions to pursue it
- Proof of concept
  - See if idea has merit
- High-level requirements analysis
  - What does “secure” mean for this concept?
  - Is it possible for this concept to meet this meaning of security?
  - Is the organization willing to support the additional resources required to make this concept meet this meaning of security?

Manufacture

- Develop detailed plans for each group involved
  - May depend on use; internal product requires no sales
- Implement the plans to create entity
  - Includes decisions whether to proceed, for example due to market needs
Deployment

- Delivery
  - Assure that correct masters are delivered to production and protected
  - Distribute to customers, sales organizations
- Installation and configuration
  - Ensure product works appropriately for specific environment into which it is installed
  - Service people know security procedures

Fielded Product Life

- Routine maintenance, patching
  - Responsibility of engineering in small organizations
  - Responsibility may be in different group than one that manufactures product
- Customer service, support organizations
- Retirement or decommission of product
Waterfall Life Cycle Model

- Requirements definition and analysis
  - Functional and non-functional
  - General (for customer), specifications
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

Relationship of Stages
Other Models

- **Exploratory programming**
  - Develop working system quickly
  - Used when detailed requirements specification cannot be formulated in advance, and adequacy is goal
  - No requirements or design specification, so low assurance

- **Prototyping**
  - Objective is to establish system requirements
  - Future iterations (after first) allow assurance techniques

More Models

- **Formal transformation**
  - Create formal specification
  - Translate it into program using correctness-preserving transformations
  - Very conducive to assurance methods

- **System assembly from reusable components**
  - Depends on whether components are trusted
  - Must assure connections, composition as well
  - Very complex, difficult to assure
Even More Models

- Extreme programming
  - Rapid prototyping and “best practices”
  - Project driven by business decisions
  - Requirements open until project complete
  - Programmers work in teams
  - Components tested, integrated several times a day
  - Objective is to get system into production as quickly as possible, then enhance it
  - Evidence adduced *after* development needed for assurance

Security: Built In or Add On?

- Think of security as you do performance
  - You don’t build a system, then add in performance later
    - Can “tweak” system to improve performance a little
    - Much more effective to change fundamental algorithms, design
  - You need to design it in
    - Otherwise, system lacks fundamental and structural concepts for high assurance
Reference Validation Mechanism

- **Reference monitor** is access control concept of an abstract machine that mediates all accesses to objects by subjects.
- **Reference validation mechanism** (RVM) is an implementation of the reference monitor concept.
  - Tamperproof
  - Complete (always invoked and can never be bypassed)
  - Simple (small enough to be subject to analysis and testing, the completeness of which can be assured)
    - Last engenders trust by providing assurance of correctness

Examples

- **Security kernel** combines hardware and software to implement reference monitor
- **Trusted computing base** (TCB) is all protection mechanisms within a system responsible for enforcing security policy
  - Includes hardware and software
  - Generalizes notion of security kernel
Adding On Security

- Key to problem: analysis and testing
- Designing in mechanisms allow assurance at all levels
  - Too many features adds complexity, complicates analysis
- Adding in mechanisms makes assurance hard
  - Gap in abstraction from requirements to design may prevent complete requirements testing
  - May be spread throughout system (analysis hard)
  - Assurance may be limited to test results

Example

- 2 AT&T products
  - Add mandatory controls to UNIX system
  - SV/MLS
    - Add MAC to UNIX System V Release 3.2
  - SVR4.1ES
    - Re-architect UNIX system to support MAC
Comparison

- Architecting of System
  - SV/MLS: used existing kernel modular structure; no implementation of least privilege
  - SVR4.1ES: restructured kernel to make it highly modular and incorporated least privilege

Comparison

- File Attributes (inodes)
  - SV/MLS added separate table for MAC labels, DAC permissions
    - UNIX inodes have no space for labels; pointer to table added
    - Problem: 2 accesses needed to check permissions
    - Problem: possible inconsistency when permissions changed
    - Corrupted table causes corrupted permissions
  - SVR4.1ES defined new inode structure
    - Included MAC labels
    - Only 1 access needed to check permissions
Without adequate design/implementation, all our work for naught

In reality, what we’ve studied shows how to get good requirements

Turning these into good systems beyond the realm of security expert

Solution: insist on use of appropriate software engineering methodologies

- Take CS510, ECE574 for more

Mistakes will be made

- Must they lead to security violations?

Solution: Risk Mitigation

Definitions:

- Threat: Potential occurrence leading to undesirable consequences
- Vulnerability: Weakness enabling threat
- Exploit: Method for Threat to use Vulnerability

All must occur for a violation to happen
Risk Mitigation

- Threat-based
  - Enumerate threats
  - For each threat, eliminate possibility of exploitable vulnerability
- Vulnerability-based
  - Formal verification
  - Testing
  - Architecture / design
- Exploit-based

Security in Layered Architectures

- Systems built in layers
- “Perfect” mechanism at high layer doesn’t prevent vulnerabilities beneath
  - Limits threats to lower layers
  - Simpler security abstractions
Evaluating Assurance

- How do we gather *evidence* that system meets security requirements?
- Process-based techniques: Was system constructed using proper methods?
  - SEI CMM
  - ISO 9000
- System Evaluation
  - Requirements Tracing
  - Representation Correspondence
  - Reviews
  - Formal Methods

Assurance in Requirements
Definition and Analysis

A *threat* is a potential occurrence that can have an undesirable effect on the system assets or resources. It is a danger that can lead to undesirable consequences.

A vulnerability is a weakness that makes it possible for a threat to occur.

Security Objectives - Every identified threat must be addressed by some countermeasure that mitigates it.
Justifying Requirements

Example application

- Threat T1 – A person not authorized to use the system gains access to the system and its facilities by impersonating an authorized user.
- Requirement IA1 – A user is permitted to begin a session only with valid unique identifier and concomitant authentication using a password.
- Requirement IA2 – Before first user/system interaction in a session a successful identification and authentication takes place.

Example (contd.)

Assumptions

- A1 – The product must be configured such that only an approved group of users has physical access to the system.
- A2 – Only authorized users may physically remove from the system the media that contains authentication data
- A3 – Users do not disclose passwords to others
- A4 – Passwords generated by admin are distributed in a secure manner.
Example (contd.)

- Now we say threat T1 is addressed by IA1, IA2, A1, A2, A3, and A4.
- We also need a list of informal justifications saying why threat T1 is addressed by the set of requirements and assumptions.
- We need a list like this for all threats!!

Assurance During Design

- Modularity and layering can simplify design
- A system should be broken down into components. Subcomponents, Modules and Functions.
- Design documentation is very important.
Documentation for security analysis

- Security Functions: High level descriptions of the functions that enforce security. Like authentication, encryption, etc.
  - Description of individual security functions
  - Overview of the set of security functions
  - Mapping to requirements

Documentation (contd.)

- External Interfaces: How system security enforcement functions control mechanisms through which users access system resources.
  - Component overview
  - Data descriptions
  - Interface descriptions
Documentation (contd.)

- Internal design – shows internal structure and functions of the components of the system. For each component you have
  - Overview of parent component
  - Detailed description of component
  - Security relevance of component.

Assurance in Implementation and Integration

- Without adequate implementation and integration, all our work for naught
- In reality, what we’ve studied shows how to get good requirements and design
- Turning these into good systems beyond the realm of security expert?
- Solution: insist on use of appropriate software engineering methodologies
  - Take SE courses!!
Process Based Techniques

- Software Engineering Institute Capability Maturity Model (SEI CMM)
  - Specifies levels of process maturity
  - Criteria to evaluate level of an organization
- ISO 900[0-?] similar
  - More directed to manufacturing than software
- Configuration Management
  - Log/track changes
  - Ensure process followed
  - Regression testing / update, release control

System Evaluation

- Requirements Tracing
  - Track requirement to mechanism
  - Ensures nothing forgotten
  - *Doesn’t ensure it is correct*
- Representation Correspondence
  - Requirements tracing between levels
- Validating Correctness:
  - Informal arguments
  - Formal verification
    - May use automated tools
System Evaluation: Reviews

- Formal Process of "passing" on specification / design / implementation
  - Team evaluates component
  - Provides independent evidence that component meets requirements
- Review is a structured process
  - Materials presented to reviewers
  - Reviewers evaluate using agreed on methods
  - Review meeting: collect comments and discuss
  - Report: List of comments, reviewer agreement/disagreement

Implementation Management

- Assume a secure design
  - How to ensure implementation will be secure?
- Constrained Implementation Environment
  - Strong typing
  - Built-in buffer checks
  - Virtual machines
- Coding Standards
  - Restrict how language is used
  - Meeting standards eliminates use of "unsafe" features
Implementation Management: Configuration Management

- Control changes made
  - Development
  - Production / operation
- Version control and tracking
  - Audit
- Change Authorization
- Enforce integration procedures
- Automated production tools

Configuration Management: CVS

- Concurrent Versions System (CVS)
  - Commonly used in DoD, elsewhere?
  - Client-Server / network approach
- CVS tree: “official” versions at server
- Check-out: Get a local copy of a version
- Check-in: merges your updates into tree
  - Creates new version
  - Forces you to comment why changed
  - Flags conflicts
  - Ignores files you’ve created that aren’t in official tree

http://www.cvshome.org
Testing

- Functional (specification based) vs. Structural (code based) testing
- Levels: Unit, System, Independent
- Security Testing:
  - Functional
  - Structural
  - Requirements (separate from functional?)
- Automated Test Suites

Process Guidance Working Group Test Model

- Test Matrix: Maps requirements to lower levels
  - At lowest level, test assertion
  - Used to develop test cases
- Divides checks into six areas
  - Discretionary Access Control
  - Privileges
  - Identification and Authorization
  - Object Reuse
  - Audit
  - System Architecture Constraints
Top-Level Matrix: OS Example

<table>
<thead>
<tr>
<th>Component</th>
<th>DAC</th>
<th>PRIV</th>
<th>I&amp;A</th>
<th>OR</th>
<th>Audit</th>
<th>Arch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Process Control</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>File Management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Audit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>I/O interfaces</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>I/O device drivers</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>IPC management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Memory management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1/30/2006
Module 9 - Assurance.

PGWG Test Model

- Each row generates lower level matrix
- Continue until test assertions possible
  - Verify only root can use \textit{stime} successfully
  - Verify audit record generated for call to \textit{stime}
- Develop test case specification for each assertion
  - Call \textit{stime} as root: time should change, audit generated
  - Call \textit{stime} as non-root: no change, fail, audit generated
- Develop test for each specification
Operation/Maintenance

- Fixes / maintenance
  - Hot fix: quick solution
    - Possibly security testing only
    - May limit functionality
  - Regular fix: more thorough testing
    - Reintroduce functionality while maintaining security

- Procedures to track flaws
  - Reporting
  - Test to detect flaw
  - Regression test: ensure flaw not “unfixed”

Formal Methods

- Software verification beyond scope of course
  - But important to achieve security

- Limited software verification
  - Verify the security subsystems
  - *Confine* the rest
Protocol Verification

- Generating protocols that meet security specifications
- Assumes cryptography secure
  - But cryptography not enough

Key Points

- Assurance critical for determining trustworthiness of systems
- Different levels of assurance, from informal evidence to rigorous mathematical evidence
- Assurance needed at all stages of system life cycle
- Building security in is more effective than adding it later