Systems with Assurance
Evaluation
Auditing

Lecture 10

November 6, 2003

Threats and Vulnerabilities

- Threat
  - A potential occurrence that can have an undesirable effect on the system assets of resources
    - Results in breaches in confidentiality, integrity, or a denial of service
    - Example: outsider penetrating a system is an outsider threat (insider threat?)
    - Need to identify all possible threats and address them to generate security objectives
- Vulnerability
    - A weakness that makes it possible for a threat to occur

Architectural considerations

- Determine the focus of control of security enforcement mechanism
  - Operating system: focus is on data
  - Applications: more on operations/transactions
- Centralized or Distributed
  - Distribute them among systems/components
  - Tradeoffs?
  - Generally easier to "assure" centralized system
- Security mechanism may exist in any layer

Architectural considerations Example: Four layer architecture

- Application layer
  - Transaction control
- Services/middleware layer
  - Support services for applications
  - Eg., DBMS, Object reference brokers
- Operating system layer
  - Memory management, scheduling and process control
- Hardware
  - Includes firmware

Architectural considerations

- Select the correct layer for a mechanism
  - Controlling user actions may be more effective at application layer
  - Controlling file access may be more effective at the operating system layer
  - Recall PEM!
- How to secure layers lower to target layer
  - Application security means OS security as well
  - Special-purpose OS?
  - All DBMSs require the OS to provide specific security features

Build or Add?

- Security is an integral part of a system
  - Address security issues at system design phase
  - Easy to analyze and assure
- Reference monitor (total mediation!)
  - Mediates all accesses to objects by subjects
- Reference validation mechanism must be–
  1. Tamperproof
  2. Never be bypassed
  3. Small enough to be subject to analysis and testing – the completeness can be assured
- Security kernel
  - Hardware + software implementing a RM

Trusted Computing Base

- TCB consists of all protection mechanisms within a computer system that are responsible for enforcing a security policy
- TCB monitors four basic interactions
  - Process activation
  - Execution domain switching
  - Memory protection
  - I/O operation
- A unified TCB may be too large
  - Create a security kernel

Security Policy Requirements

- Can be done at different levels
- Specification must be
  - Clear
    - "meet C2 security"
  - Unambiguous
    - "users must be identified and authenticated"
  - Complete
- Methods of defining policies
  - Extract applicable requirements from existing security standards (e.g. Common Criteria)
  - Create a policy based on threat analysis
  - Map the system to an existing model
- Justify the requirements: *completeness* and *consistency*

Design assurance

- Identify design flaws
  - Enhances trustworthiness
  - Supports implementation and operational assurance
- Design assurance technique employs
  - Specification of requirements
  - Specification of the system design
  - Process to examine how well the design meets the requirement

Techniques for Design Assurance

- Modularity & Layering
  - Well defined independent modules
  - Simplifies and makes system more understandable
  - Data hiding
  - Easy to understand and analyze
- Different layers to capture different levels of abstraction
  - Subsystem (memory management, I/O subsystem, credit-card processing function)
  - Subcomponent (I/O management, I/O drivers)
  - Module: set of related functions and data structure
- Use principles of secure design

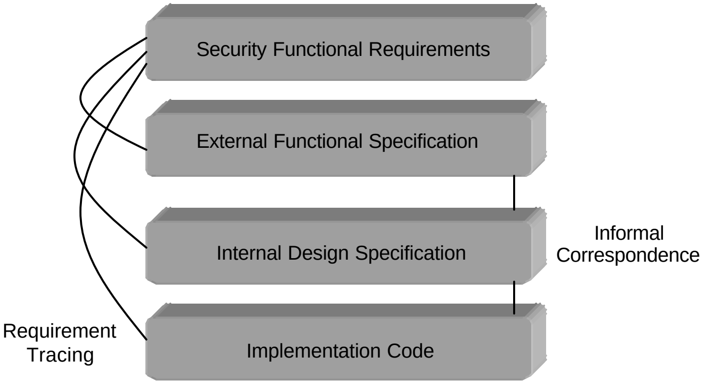
Design Documents

- Design documentation is an important component in life cycle models
- Documentation must specify
  - Security functions and approach
    - Describe each security function
    - Overview of a set of security functions
    - Map to requirements (tabular)
  - External interfaces used by users
    - Parameters, syntax, security constraints and error conditions
    - Component overview, data descriptions, interface description
  - Internal design with low-level details
    - Overview of the component
    - Detailed description of the component
    - Security relevance of the component

Design meets requirements?

- Techniques needed
  - To prevent requirements and functionality from being discarded, forgotten, or ignored at lower levels of design
- Requirements tracing
  - Process of identifying specific security requirements that are met by parts of a description
- Informal Correspondence
  - Process of showing that a specification is consistent with an adjacent level of specification

Requirement mapping and informal correspondence



Design meets requirements?

- Informal arguments
  - Protection profiles
    - Define threats to systems and security objectives
    - Provide rationale (an argument)
  - Security targets
    - Identifies mechanisms and provide justification
- Formal methods: proof techniques
  - Formal proof mechanisms are usually based on logic (predicate calculus)
  - Model checking
    - Checks that a model satisfies a specification

Design meets requirements?

- Review
  - When informal assurance technique is used
  - Usually has three parts
    - Reviews of guidelines
    - Conflict resolution methods
    - Completion procedures

Implementation considerations for assurance

- Modularity with minimal interfaces
- Language choice
  - C programs may not be reliable
    - Pointers – memory overwrites
    - Not much error handling
    - Writing past the bounds of memory and buffers
      - Notorious for Buffer overflow!
  - Java
    - Designed to support secure code as a primary goal
    - Ameliorates C security risks present in C
    - Sandbox model (mobile code security)

Assurance through Implementation management

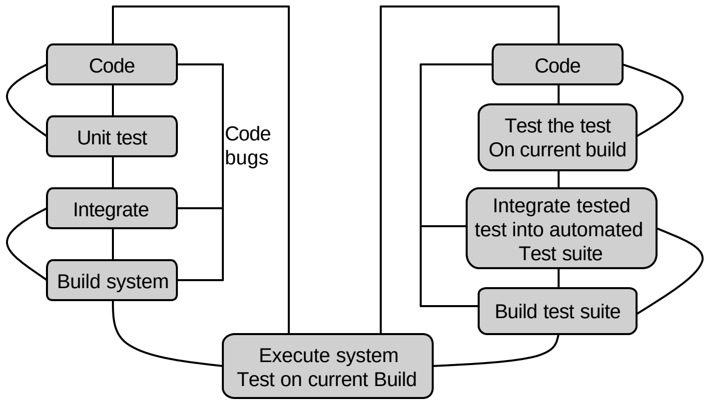
- Configuration management tools
  - Control of the refinement and modification of configuration items such as source code, documentation etc.
  - CM system functions
    - Version control and tracking
    - Change authorization
    - Integration procedures
    - Tools for product generation

    CVS?

Implementation meets Design?

- Security testing
  - Functional testing (FT) (black box testing)
    - Testing of an entity to determine how well it meets its specification
  - Structural testing (ST) (white box testing)
    - Testing based on an analysis of the code to develop test cases
- Testing occurs at different times
  - Unit testing (usually ST): testing a code module before integration
  - System testing (FT): on integrated modules
  - Security testing: product security
    - Security functional testing (against security issues)
    - Security structural testing (security implementation)
    - Security requirements testing

Code development and testing



Operation and maintenance assurance

- Bugs in operational phase need fixing
- Hot fix
  - Immediate fix
  - Bugs are serous and critical
- Regular fix
  - Less serious bugs
  - Long term solution after a hot fix

Evaluation

What is Formal Evaluation?

- Method to achieve *Trust*
  - Not a guarantee of security
- Evaluation methodology includes:
  - Security requirements
  - Assurance requirements showing how to establish security requirements met
  - Procedures to demonstrate system meets requirements
  - Metrics for results (level of trust)
- Examples: TCSEC (Orange Book), ITSEC, CC

Formal Evaluation: Why?

- Organizations require assurance
  - Defense
  - Telephone / Utilities
  - "Mission Critical" systems
- Formal verification of entire systems not feasible
- Instead, organizations develop formal evaluation methodologies
  - Products passing evaluation are trusted
  - Required to do business with the organization

TCSEC: The Original

- Trusted Computer System Evaluation Criteria
  - U.S. Government security evaluation criteria
  - Used for evaluating commercial products
- Policy model based on Bell-LaPadula
- Enforcement: Reference Validation Mechanism
  - Every reference checked by compact, analyzable body of code
- Emphasis on Confidentiality
- Metric: Seven trust levels:
  - D, C1, C2, B1, B2, B3, A1
  - D is "tried but failed"

TCSEC Class Assurances

- C1: Discretionary Protection
  - Identification
  - Authentication
  - Discretionary access control
- C2: Controlled Access Protection
  - Object reuse and auditing
- B1: Labeled security protection
  - Mandatory access control on limited set of objects
  - Informal model of the security policy

TCSEC Class Assurances *(continued)*

- B2: Structured Protections
  - Trusted path for login
  - Principle of Least Privilege
  - Formal model of Security Policy
  - Covert channel analysis
  - Configuration management
- B3: Security Domains
  - Full reference validation mechanism
  - Constraints on code development process
  - Documentation, testing requirements
- A1: Verified Protection
  - Formal methods for analysis, verification
  - Trusted distribution

How is Evaluation Done?

- Government-sponsored independent evaluators
  - Application: Determine if government cares
  - Preliminary Technical Review
    - Discussion of process, schedules
    - Development Process
    - Technical Content, Requirements
  - Evaluation Phase

TCSEC: Evaluation Phase

- Three phases
  - Design analysis
    - Review of design based on documentation
  - Test analysis
  - Final Review
- Trained independent evaluation
  - Results presented to Technical Review Board
  - Must approve before next phase starts
- Ratings Maintenance Program
  - Determines when updates trigger new evaluation

TCSEC: Problems

- Based heavily on confidentiality
  - Did not address integrity, availability
- Tied security and functionality
- Base TCSEC geared to operating systems
  - TNI: Trusted Network Interpretation
  - TDI: Trusted Database management System Interpretation

Later Standards

- CTCPEC – Canada
- ITSEC – European Standard
  - Did not define criteria
  - Levels correspond to strength of evaluation
  - Includes code evaluation, development methodology requirements
  - Known vulnerability analysis
- CISR: Commercial outgrowth of TCSEC
- FC: Modernization of TCSEC
- FIPS 140: Cryptographic module validation
- Common Criteria: International Standard
- SSE-CMM: Evaluates developer, not product

ITSEC: Levels

- E1: Security target defined, tested
  - Must have informal architecture description
- E2: Informal description of design
  - Configuration control, distribution control
- E3: Correspondence between code and security target
- E4: Formal model of security policy
  - Structured approach to design
  - Design level vulnerability analysis
- E5: Correspondence between design and code
  - Source code vulnerability analysis
- E6: Formal methods for architecture
  - Formal mapping of design to security policy
  - Mapping of executable to source code

ITSEC Problems:

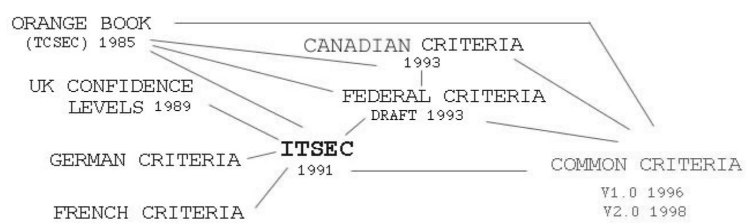
- No validation that security requirements made sense
  - Product meets goals
  - But does this meet user expectations?
- Inconsistency in evaluations
  - Not as formally defined as TCSEC

Common Criteria

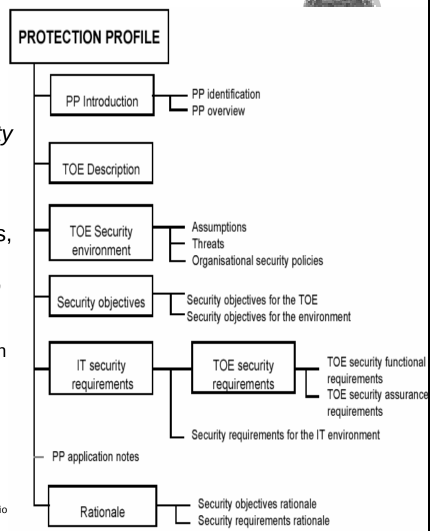
- Replaced TCSEC, ITSEC

1. CC Documents
   - Functional requirements
   - Assurance requirements
   - Evaluation Assurance Levels (EAL)
2. CC Evaluation Methodology
   - Detailed evaluation guidelines for each EAL
3. National Scheme (Country specific)

Common Criteria: Origin



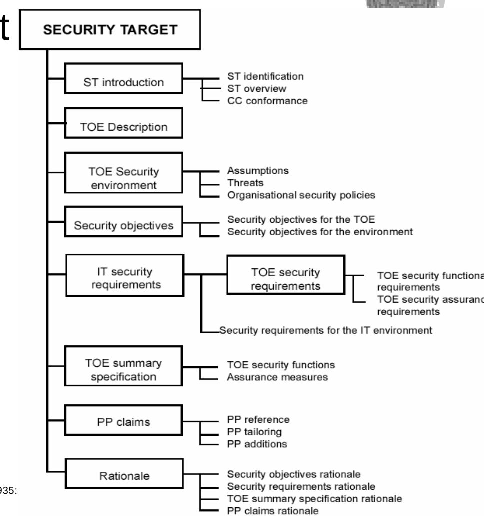
CC Evaluation 1: Protection Profile



Implementation independent, domain-specific set of security requirements

- Narrative Overview
- Product/System description
- Security Environment (threats, overall policies)
- Security Objectives: System, Environment
- IT Security Requirements
  - Functional requirements drawn from CC set
  - Assurance level
- Rationale for objectives and requirements

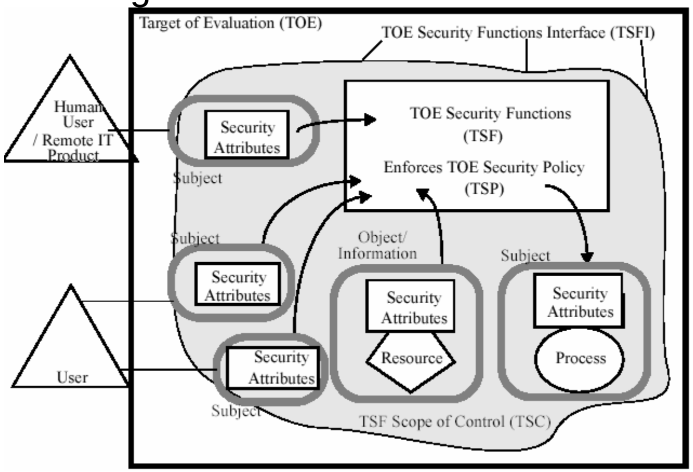
CC Evaluation 2: Security Target



Specific requirements used to evaluate system

- Narrative introduction
- Environment
- Security Objectives
  - How met
- Security Requirements
  - Environment and system
  - Drawn from CC set
- Mapping of Function to Requirements
- Claims of Conformance to Protection Profile

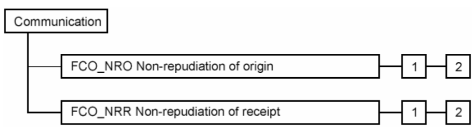
Security Functional Requirement Paradigm



Common Criteria: Functional Requirements

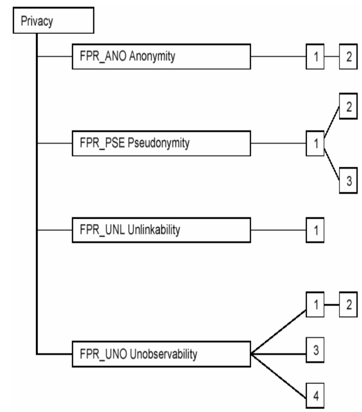
- 362 page document
- 11 Classes
  - Security Audit, Communication, Cryptography, User data protection, ID/authentication, Security Management, Privacy, Protection of Security Functions, Resource Utilization, Access, Trusted paths
- Several families per class
- Lattice of components in a family

Class Example: Communication



- Non-repudiation of origin
  1. Selective Proof. Capability to request verification of origin
  2. Enforced Proof. All communication includes verifiable origin

Class Example: Privacy

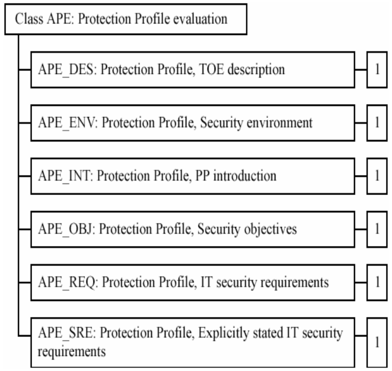


1. Pseudonymity
   1. The TSF shall ensure that [assignment: *set of users and/or subjects*] are unable to determine the real user name bound to [assignment: *list of subjects and/or operations and/or objects*]
   2. The TSF shall be able to provide [assignment: *number of aliases*] aliases of the real user name to [assignment: *list of subjects*]
   3. The TSF shall [selection: *determine an alias for a user, accept the alias from the user*] and verify that it conforms to the [assignment: *alias metric*]
2. Reversible Pseudonimity
   1. …
3. Alias Pseudonimity
   1. …

Common Criteria: Assurance Requirements

- 216 page document
- 10 Classes
  - Protection Profile Evaluation, Security Target Evaluation
  - Configuration management, Delivery and operation, Development, Guidance, Life cycle, Tests, Vulnerability assessment
  - Maintenance
- Several families per class
- Lattice of components in family

Example: Protection Profile Evaluation



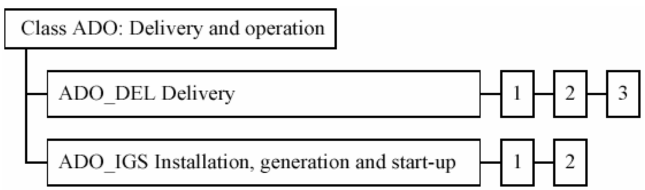
Security environment

- In order to determine whether the IT security requirements in the PP are sufficient, it is important that the security problem to be solved is clearly understood by all parties to the evaluation.

1. Protection Profile, Security environment, Evaluation requirements
   - Dependencies: No dependencies.
   - Developer action elements:

- The PP developer shall provide a statement of TOE security environment as part of the PP.
  - Content and presentation of evidence elements:
- ….

Example:
Delivery and Operation



Installation, generation and start-up

A. Installation, generation, and start-up procedures
   - Dependencies: AGD_ADM.1 Administrator guidance
B. Developer action elements:
   - The developer shall document procedures necessary for the secure installation, generation, and start-up of the TOE.
C. Content and presentation of evidence elements:
   - The documentation shall describe the steps necessary for secure installation, generation, and start-up of the TOE.
D. …..

Common Criteria:
Evaluation Assurance Levels

1. Functionally tested
2. Structurally tested
3. Methodically tested and checked
4. Methodically designed, tested, and reviewed
5. Semi-formally designed and tested
6. Semi-formally verified design and tested
7. Formally verified design and tested

Common Criteria: Evaluation Process

- National Authority authorizes evaluators
  - U.S.: NIST accredits commercial organizations
  - Fee charged for evaluation
- Team of four to six evaluators
  - Develop work plan and clear with NIST
  - Evaluate Protection Profile first
  - If successful, can evaluate Security Target

Common Criteria: Status

- About 80 registered products
  - Only one at level 5 (Java Smart Card)
  - Several OS at 4
  - Likely many more not registered
- New versions appearing on regular basis

Auditing

What is Auditing?

- Logging
  - Recording events or statistics to provide information about system use and performance
- Auditing
  - Analysis of log records to present information about the system in a clear, understandable manner

Auditing goals/uses

- User accountability
- Damage assessment
- Determine causes of security violations
- Describe security state for monitoring critical problems
  - Determine if system enters unauthorized state
- Evaluate effectiveness of protection mechanisms
  - Determine which mechanisms are appropriate and working
  - Deter attacks because of presence of record

Problems

- What to log?
  - looking for violations of a policy, so record *at least* what will show such violations
  - Use of privileges
- What do you audit?
  - Need not audit everything
  - Key: what is the policy involved?

Audit System Structure

- Logger
  - Records information, usually controlled by parameters
- Analyzer
  - Analyzes logged information looking for something
- Notifier
  - Reports results of analysis

Logger

- Type, quantity of information recorded controlled by system or program configuration parameters
- May be human readable or not
  - If not, usually viewing tools supplied
  - Space available, portability influence storage format

Example: RACF

- Security enhancement package for IBM’s MVS/VM
- Logs failed access attempts, use of privilege to change security levels, and (if desired) RACF interactions
- View events with LISTUSERS commands

Example: Windows NT

- Different logs for different types of events
  - *System event* logs record system crashes, component failures, and other system events
  - *Application event* logs record events that applications request be recorded
  - *Security event* log records security-critical events such as logging in and out, system file accesses, and other events
- Logs are binary; use *event viewer* to see them
- If log full, can have system shut down, logging disabled, or logs overwritten

Windows NT Sample Entry

Date:	2/12/2000	Source:	Security
Time:	13:03	Category:	Detailed Tracking
Type:	Success	EventID:	592
User:	WINDSOR\Administrator		
Computer:	WINDSOR		

Description:
A new process has been created:
New Process ID: 2216594592
Image File Name:
\Program Files\Internet Explorer\IEXPLORE.EXE
Creator Process ID: 2217918496
User Name: Administrator
FDomain: WINDSOR
Logon ID: (0x0,0x14B4c4)

[would be in graphical format]

Analyzer

- Analyzes one or more logs
  - Logs may come from multiple systems, or a single system
  - May lead to changes in logging
  - May lead to a report of an event

  - Using *swatch* to find instances of *telnet* from *tcpd* logs:
    `/telnet/&!/localhost/&!/*.site.com/`
  - Query set overlap control in databases
    - If too much overlap between current query and past queries, do not answer
  - Intrusion detection analysis engine (director)
    - Takes data from sensors and determines if an intrusion is occurring

Notifier

- Informs analyst, other entities of results of analysis
- May reconfigure logging and/or analysis on basis of results
- May take some action

Examples

- Using *swatch* to notify of *telnet*s
  `/telnet/&!/localhost/&!/*.site.com/mail staff`
- Query set overlap control in databases
  - Prevents response from being given if too much overlap occurs
- Three failed logins in a row disable user account
  - Notifier disables account, notifies sysadmin

Designing an Audit System

- Essential component of security mechanisms
- Goals determine what is logged
  - Idea: auditors want to detect violations of policy, which provides a set of constraints that the set of possible actions must satisfy
  - So, audit functions that may violate the constraints
- Constraint p_i : *action* $\Rightarrow$ *condition*

Example: Bell-LaPadula

- Simple security condition and *-property
  - S reads $O \Rightarrow L(S) = L(O)$
  - S writes $O \Rightarrow L(S) = L(O)$
  - To check for violations, on each read and write, must log $L(S)$, $L(O)$, action (read, write), and result (success, failure)
  - Note: need *not* record S, O!
    - In practice, done to identify the object of the (attempted) violation and the user attempting the violation

Remove Tranquility

- New commands to manipulate security level must also record information
  - S reclassify O to $L(O')$ => $L(O) = L(S)$ and $L(O') = L(S)$
  - Log $L(O)$, $L(O')$, $L(S)$, action (reclassify), and result (success, failure)
  - Again, need not record O or S to detect violation
    - But needed to follow up …

Example: Chinese Wall

- Subject S has $COI(S)$ and $CD(S)$
  - $CD_H(S)$ is set of company datasets that S has accessed
- Object O has $COI(O)$ and $CD(O)$
  - $san(O)$ iff O contains only sanitized information
- Constraints
  - S reads $O \Rightarrow COI(O)$? $COI(S) \vee \exists O'(CD(O') \in CD_H(S))$
  - S writes $O \Rightarrow (S \text{ canread } O) \wedge \neg\exists O'(COI(O) = COI(O') \wedge S \text{ canread } O' \wedge \neg san(O'))$

Recording

- S reads $O \Rightarrow COI(O)$? $COI(S) \lor \exists O'(CD(O') \in CD_H(S))$
  - Record $COI(O)$, $COI(S)$, $CD_H(S)$, $CD(O')$ if such an O' exists, action (read), and result (success, failure)
- S writes $O \Rightarrow (S \text{ canread } O) \land \neg\exists O'(COI(O) = COI(O') \land S \text{ canread } O' \land \neg san(O'))$
  - Record $COI(O)$, $COI(S)$, $CD_H(S)$, plus $COI(O')$ and $CD(O')$ if such an O' exists, action (write), and result (success, failure)

Implementation Issues

- Show non-security or find violations?
  - Former requires logging initial state as well as changes
- Defining violations
  - Does “write” include “append” and “create directory”?
- Multiple names for one object
  - Logging goes by *object* and not name
  - Representations can affect this (if you read raw disks, you’re reading files; can your auditing system determine which file?)

Syntactic Issues

- Data that is logged may be ambiguous
  - BSM: two optional text fields followed by two mandatory text fields
  - If three fields, which of the optional fields is omitted?
- Solution: use grammar to ensure well-defined syntax of log files

Example Grammar

entry	: date host prog [bad] user ["from" host] "to" user "on" tty
date	: daytime
host	: string
prog	: string ":"
bad	: "FAILED"
user	: string
tty	: "/dev/" string

- Log file entry format defined unambiguously
- Audit mechanism could scan, interpret entries without confusion

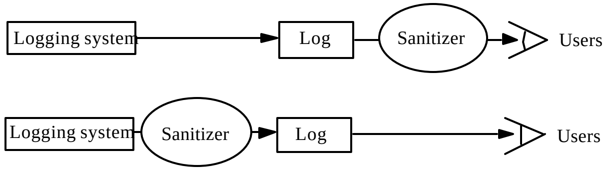
More Syntactic Issues

- Context
  - Unknown user uses anonymous *ftp* to retrieve file "/etc/passwd"
  - Logged as such
  - Problem: *which* /etc/passwd file?
    - One in system /etc directory
    - One in anonymous *ftp* directory /var/ftp/etc, and as *ftp* thinks /var/ftp is the root directory, /etc/passwd refers to /var/ftp/etc/passwd

Log Sanitization

- *U* set of users, *P* policy defining set of information *C*(*U*) that *U* cannot see; log sanitized when all information in *C*(*U*) deleted from log
- Two types of *P*
  - *C*(*U*) can't leave site
    - People inside site are trusted and information not sensitive to them
  - *C*(*U*) can't leave system
    - People inside site not trusted or (more commonly) information sensitive to them
    - Don't log this sensitive information

Logging Organization



- Top prevents information from leaving site
  - Users' privacy not protected from system administrators, other administrative personnel
- Bottom prevents information from leaving system
  - Data simply not recorded, or data scrambled before recording (Cryptography)

Reconstruction

- *Anonymizing sanitizer* cannot be undone
  - No way to recover data from this
- *Pseudonymizing sanitizer* can be undone
  - Original log can be reconstructed
- Importance
  - Suppose security analysis requires access to information that was sanitized?

Issue

- Key: sanitization must preserve properties needed for security analysis
- If new properties added (because analysis changes), may have to resanitize information
  - This *requires* pseudonymous sanitization or the original log

Example

- Company wants to keep its IP addresses secret, but wants a consultant to analyze logs for an address scanning attack
  - Connections to port 25 on IP addresses 10.163.5.10, 10.163.5.11, 10.163.5.12, 10.163.5.13, 10.163.5.14,
  - Sanitize with random IP addresses
    - Cannot see sweep through consecutive IP addresses
  - Sanitize with sequential IP addresses
    - Can see sweep through consecutive IP addresses

Generation of Pseudonyms

1. Devise set of pseudonyms to replace sensitive information
   - Replace data with pseudonyms that preserve relationship
   - Maintain table mapping pseudonyms to data
2. Use random key to encipher sensitive datum and use secret sharing scheme to share key
   - Used when insiders cannot see unsanitized data, but outsiders (law enforcement) need to
   - (t, n) –threshold scheme: requires *t* out of *n* people to read data

Application Logging

- Applications logs made by applications
  - Applications control what is logged
  - Typically use high-level abstractions such as:

    su: bishop to root on /dev/ttyp0
  - Does not include detailed, system call level information such as results, parameters, etc.

System Logging

- Log system events such as kernel actions
  - Typically use low -level events

```
3876 ktrace    CALL      execve(0xbfbff0c0,0xbfbff5cc,0xbfbff5d8)
3876 ktrace    NAMI      "/usr/bin/su"
3876 ktrace    NAMI      "/usr/libexec/ld-elf.so.1"
3876 su  RET   xecve  0
3876 su  CALL  __sysctl(0xbfbff47c,0x2,0x2805c928,0xbfbff478,0,0)
3876 su  RET   __sysctl 0
3876 su  CALL mmap(0,0x8000,0x3,0x1002,0xffffffff,0,0,0)
3876 su  RET   mmap 671473664/0x2805e000
3876 su  CALL geteuid
3876 su  RET   geteuid 0
```

  - Does not include high-level abstractions such as loading libraries (as above)

Contrast

- Differ in focus
  - Application logging focuses on application events, like failure to supply proper password, and the broad operation (what was the reason for the access attempt?)
  - System logging focuses on system events, like memory mapping or file accesses, and the underlying causes (why did access fail?)
- System logs usually much bigger than application logs
- Can do both, try to correlate them

Design

- *A posteriori* design
  - Need to design auditing mechanism for system not built with security in mind
- Goal of auditing
  - Detect *any* violation of a stated policy
    - Focus is on policy and actions designed to violate policy; specific actions may not be known
  - Detect actions *known* to be part of an attempt to breach security
    - Focus on specific actions that have been determined to indicate attacks

Detect Violations of Known Policy

- Goal: does system enter a disallowed state?
- Two forms
  - State-based auditing
    - Look at current state of system
  - Transition-based auditing
    - Look at actions that transition system from one state to another

State-Based Auditing

- Log information about state and determine if state is allowed
  - Assumption: you can get a snapshot of system state
  - Snapshot needs to be consistent
  - Non-distributed system needs to be quiescent

Example

- File system auditing tools (e.g. tripwire)
  - Thought of as analyzing single state (snapshot)
  - In reality, analyze many slices of different state unless file system quiescent
  - Potential problem: if test at end depends on result of test at beginning, relevant parts of system state may have changed between the first test and the last
    - Classic TOCTTOU flaw (time to check to time of use)

Transition-Based Auditing

- Log information about action, and examine current state and proposed transition to determine if new state would be disallowed
  - Note: just analyzing the transition may not be enough; you may need the initial state
  - Tend to use this when specific transitions *always* require analysis (for example, change of privilege)

Example

- TCP access control mechanism intercepts TCP connections and checks against a list of connections to be blocked
  - Obtains IP address of source of connection
  - Logs IP address, port, and result (allowed/blocked) in log file
  - Purely transition-based (current state not analyzed at all)

Detect Known Violations of Policy

- Goal: does a specific action and/or state that is known to violate security policy occur?
  - Assume that action *automatically* violates policy
  - Policy may be implicit, not explicit
  - Used to look for known attacks

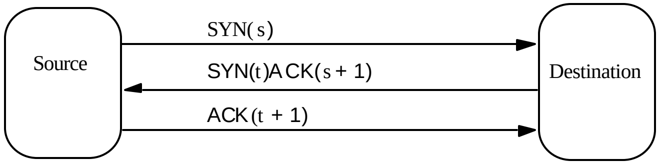
Example

- Land attack
  - Consider 3-way handshake to initiate TCP connection (next slide)
  - What happens if source, destination ports and addresses the same? Host expects ACK(t+1), but gets ACK(s+1).
  - RFC ambiguous:
    - p. 36 of RFC: send RST to terminate connection
    - p. 69 of RFC: reply with empty packet having current sequence number t+1 and ACK number s+1—but it receives packet and ACK number is incorrect. So it repeats this … system hangs or runs very slowly, depending on whether interrupts are disabled

3-Way Handshake and Land

Normal:
1. srcseq = *s*, expects ACK *s*+1
2. destseq = *t*, expects ACK *t*+1; src gets ACK *s*+1
3. srcseq = *s*+1, destseq = *t*+1; dest gets ACK *t*+1

Land:
1. srcseq = destseq = s, expects ACK s+1
2. srcseq = destseq = t, expects ACK t+1 but gets ACK s+1
3. Never reached; recovery from error in 2 attempted



Detection

- Must spot initial Land packet with source, destination addresses the same
- Logging requirement:
  - source port number, IP address
  - destination port number, IP address
- Auditing requirement:
  - If source port number = destination port number and source IP address = destination IP address, packet is part of a Land attack