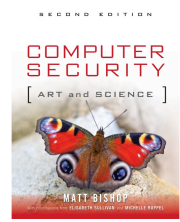
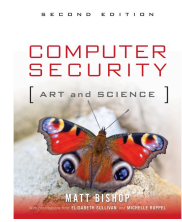
Building Systems with Assurance

Chapter 20



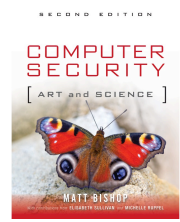
Overview

- Assurance in requirements definition, analysis
- Assurance in system and software design
- Assurance in implementation and Integration
- Assurance in operation and maintenance



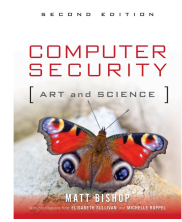
Threats and Goals

- *Threat* is a danger that can lead to undesirable consequences
- *Vulnerability* is a weakness allowing a threat to occur
- Each identified threat requires countermeasure
  - Unauthorized people using system mitigated by requiring identification and authentication
- Often single countermeasure addresses multiple threats



Architecture

- Where do security enforcement mechanisms go?
  - Focus of control on operations or data?
    - Operating system: typically on data
    - Applications: typically on operations
  - Centralized or distributed enforcement mechanisms?
    - Centralized: called by routines
    - Distributed: spread across several routines



Layered Architecture

- Security mechanisms at any layer
  - Example: 4 layers in architecture
    - *Application layer*: user tasks
    - *Services layer*: services in support of applications
    - *Operating system layer*: the kernel
    - *Hardware layer*: firmware and hardware proper
- Where to put security services?
  - Early decision: which layer to put security service in

Security Services in Layers

- Choose best layer
  - User actions: probably at applications layer
  - Erasing data in freed disk blocks: OS layer
- Determine supporting services at lower layers
  - Security mechanism at application layer needs support in all 3 lower layers
- May not be possible
  - Application may require new service at OS layer; but OS layer services may be set up and no new ones can be added

Security: Built In or Add On?

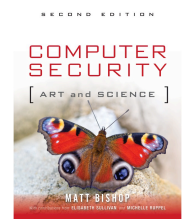
- Think of security as you do performance
  - You don't build a system, then add in performance later
    - Can "tweak" system to improve performance a little
    - Much more effective to change fundamental algorithms, design
- You need to design it in
  - Otherwise, system lacks fundamental and structural 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 evidence of correctness

Examples

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

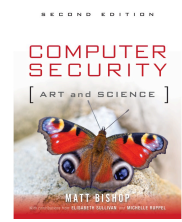


Adding On Security

- 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

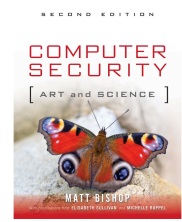
Example

- 2 AT&T products with same goal of adding 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

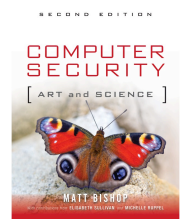


Comparison

- 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, DAC attributes
    - Only 1 access needed to check permissions

Requirements Assurance

- *Specification* describes of characteristics of computer system or program
- *Security specification* specifies desired security properties
- Must be clear, complete, unambiguous
  - Something like “meets C2 security requirements” not good: what *are* those requirements (actually, 34 of them!)

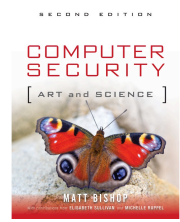


Example

- "Users of the system must be identified and authenticated" is ambiguous
  - Type of ID required—driver's license, token?
  - What is to be authenticated—user, representation of identity, system?
  - Who is to do the authentication—system, guard?
- "Users of the system must be identified to the system and must have that identification authenticated by the system" is less ambiguous
  - Under what conditions must the user be identified to the system—at login, time of day, or something else?

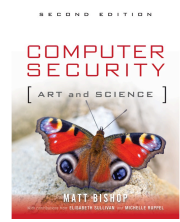
Example

- "Users of the system must be identified to the system and must have that identification authenticated by the system before the system performs any functions on behalf of that identity"
  - Type of identification is user name
  - User identification (name) to be authenticated
  - System to authenticate
  - Authentication to be done at login (before system performs any action on behalf of user)



Methods of Definition

- Extract applicable requirements from existing security standards
  - Tend to be semiformal
- Combine results of threat analysis with components of existing policies to create a new policy
- Map the system to existing model
  - If model appropriate, creating a mapping from model to system may be cheaper than requirements analysis



Example

- System X: UNIX system with MAC based on Bell-LaPadula Model
  - Mapping constructed in series of stages
  - Auditing also required

Example Stage 1

- Map elements, state variables of BLP to entities in System X
  - Subject set *S* in BLP → set of processes in System X
  - Object set *O* in BLP → set of inode objects, IPC objects, mail messages, processes as destinations, passive entities in System X
  - Right set *P* in BLP → set of rights of system functions in System X
    - Functions that create entities, write entities, have write w
    - Functions that read entities have right r
    - Functions that execute, search entities have right r
  - Access set *b* in BLP → types of access
    - Subjects can use rights r, w, a to access inode objects

Example Stage 1

- Access control matrix *a* for current state in BLP → current state of mandatory and discretionary controls in System X
- Functions f_s, f_o, and f_c in BLP → three functions in System X
  - *f*(*s*) is the maximum security level of subject *s*
  - *current-level*(*s*) is current security level of subject *s*
  - *f*(*o*) is the security level of object *o*
- Hierarchy H in BLP → differently for different objects in System X
  - Inode objects are hierarchical trees represented by the file system hierarchy
  - Other object types map to discrete points in the hierarchy

Example Stage 2

- Define BLP properties in language of System X and show each property is consistent with BLP
  - MAC property of BLP $\rightarrow$ user having over an object:
    - read access iff user's clearance dominates object's classification
    - write access over an object iff object's classification dominates user's clearance.
  - DAC property of BLP $\rightarrow$ user having access to object iff owner of object has explicitly granted that user access to object

Example Stage 2

- Label inheritance, user level changes specific to System X
  - Security level of newly created object inherited from creating subject
  - Security level of initial process at user login, security level of initial process after user level change, bounded by security level range defined for that user and for the terminal
  - Security level of newly spawned process inherited from parent, except for first process after a user level change
  - When user's level raised, child process does not inherit write access to objects opened by parent
  - When user's level lowered, all processes, accesses associated with higher privilege terminated

Example Stage 2

- Reclassification property of System X
  - Specially trusted users allowed to downgrade objects they own within constraints of user's authorizations.
- System X property of owner/group transfer allows ownership or group membership of process to be transferred to another user or group
- Status property is property of System X
  - Restricts visibility of status information available to users when they use standard System X set of commands

Example Stage 3

- Designers define System X rules by mapping System X system calls, commands, and functions to BLP rules
  - Simple security condition, *-property, and discretionary security property interpreted for each type of access
  - From these interpretations, designers can extract specific requirements for specific accesses to particular types of objects.

Example Stage 4

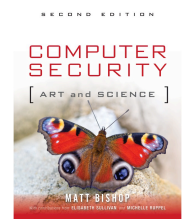
- Designers show System X rules preserve security properties
  - Show that the rules enforce the properties directly; or
  - Map the rules directly to a BLP rule or a sequence of BLP rules
    - 9 rules about current access
    - 5 rules about functions and security levels
    - 8 access permission rules
    - 8 more rules about subjects and objects
  - Designers must show that each rule is consistent with actions of System X.

Justifying Requirements

- Show policy complete and consistent
- Example: ITSEC suitability analysis
  - Map threats to requirements and assumptions
  - Describe how references address threat

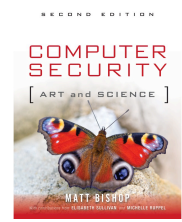
Example: System Y Evaluation

- 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 user session only if the user presents a valid unique identifier to the system and if the claimed identity of the user is authenticated by the system by authenticating the supplied password.
  - Requirement IA2: Before the first user/system interaction in a session, successful identification and authentication of the user take place.



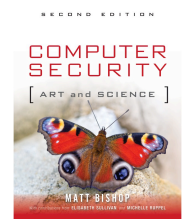
System Y Assumptions

- Assumption A1: The product must be configured such that only the approved group of users has physical access to the system.
- Assumption A2: Only authorized users may physically remove from the system the media on which authentication data is stored.
- Assumption A3: Users must not disclose their passwords to other individuals.
- Assumption A4: Passwords generated by the administrator shall be distributed in a secure manner.



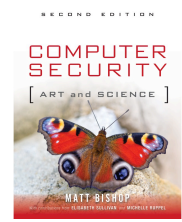
System Y Mapping

Threat	Security Target Reference
T1	IA1, IA2, A1, A2, A3, A4



System Y Justifications

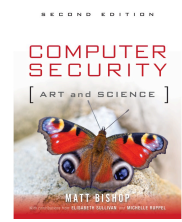
1. Referenced requirements and assumptions guard against unauthorized access.
   - Assumption A1 restricts physical access to the system to those authorized to use it.
   - Requirement IA1 requires all users to supply a valid identity and confirming password.
   - Requirement IA2 ensures that requirement IA1 cannot be bypassed.



System Y Justifications

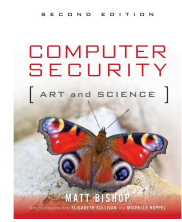
2. Referenced assumptions prevent unauthorized users from gaining access by using valid user's identity and password
   - Assumption A3 ensures that users keep passwords secret
   - Assumption A4 prevents unauthorized users from intercepting new passwords when those passwords are distributed to users
   - Assumption A2 prevents unauthorized access to authentication information stored on removable media.

These justifications provide an informal basis for asserting that, if the assumptions hold and the requirements are met, the threat is adequately handled.



Design Assurance

- Process of establishing that design of system sufficient to enforce security requirements
  - Specify requirements (see above)
  - Specify system design
  - Examine how well design meets requirements



Design Techniques

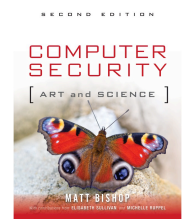
- Modularity
  - Makes system design easier to analyze
  - RVM: functions not related to security distinct from modules supporting security functionality
- Layering
  - Makes system easier to understand
  - Supports information hiding

Layering

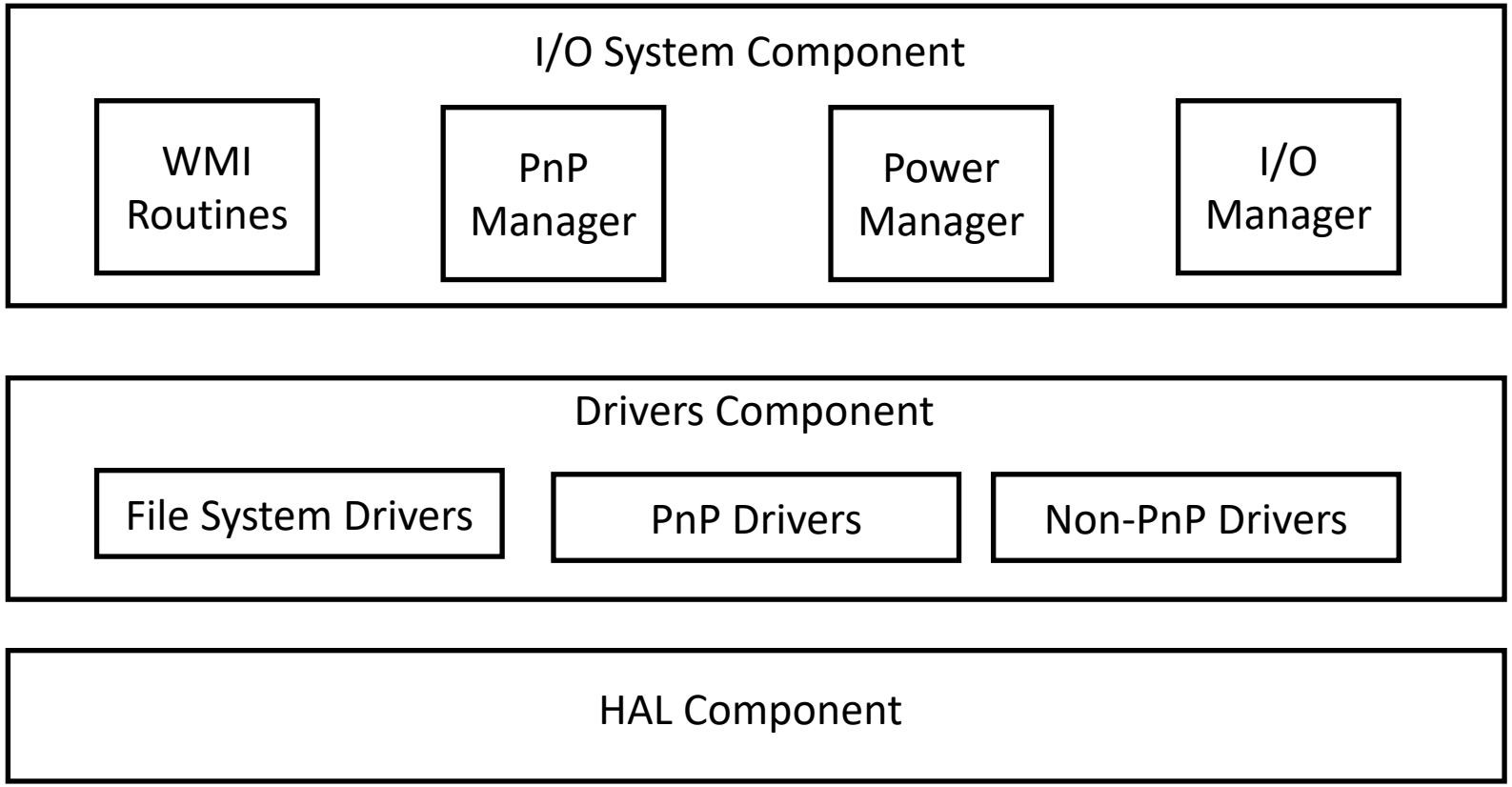
- Develop specifications at each layer of abstraction
  - *subsystem* or *component*: special-purpose division of a larger entity
    - Example: for OS, memory manager, process manager; Web store: credit card handlers
  - *subcomponent*: part of a component
    - Example: I/O component has I/O managers and I/O drivers as subcomponents
  - *module*: set of related functions, data structures

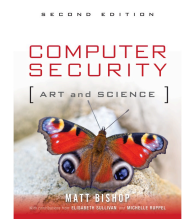
Example: Windows 10 and Windows Server 2016 I/O System

- 3 layer decomposition of components
  - I/O System Component
    - Windows Management Interface (WMI) routines
    - Plug and Play (PnP) manager
    - Power manager
    - I/O manager
  - Drivers Component
    - File system drivers
    - Plug and play drivers
    - Non-plug and play drivers
  - Hardware Abstraction Layer (HAL) component (no subcomponents)



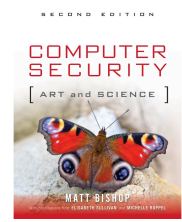
Example: Decomposition





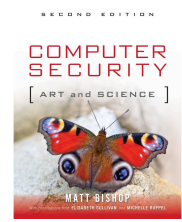
Example: More Details

- Subcomponents of file system drivers
  - Compact disk file system drivers (CDFS)
  - NT file system (NTFS)
  - Fast file allocation table file system (FAT)
  - Encrypting file system (EFS)
- Below this layer are module, function layers
- I/O system uses data stored in several places
  - Registry: database storing system configuration information
  - Driver installation files (INF)
  - Files storing digital signatures for drivers (CAT)



Design Document Contents

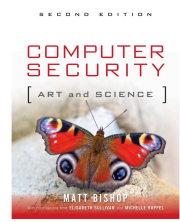
- Provide basis for analysis
  - Informal, semiformal, formal
- Must include:
  - *Security functions*: high-level descriptions of functions that enforce security and overview of protection approach
  - *External interfaces*: interfaces visible to users, how the security enforcement functions constrain them, and what the constraints and effects should be
  - *Internal design*: Design descriptions addressing the architecture in terms of the next layer of decomposition; also, for each module, identifies and describes all interfaces and data structures



Security Functions

Security functions summary specification identifies high-level security functions defined for the system; includes

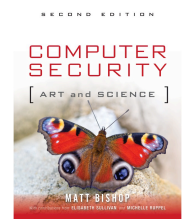
- *Description of individual security functions*, complete enough to show the intent of the function; tie to requirements
- *Overview of set of security functions* describing how security functions work together to satisfy security requirements
- *Mapping to requirements*, specifying mapping between security functions and security requirements.



External Interface

High-level description of external interfaces to system, component, subcomponent, or module

1. *Component overview* identifying the component, its parent, how the component fits into the design
2. *Data descriptions* identifying data types and structures needed to support the external interface descriptions specific to this component, and security issues or protection requirements relevant to data structures.
3. *Interface descriptions* including commands, system calls, library calls, functions, and application program interfaces as well as exception conditions and effects



Example 1

- Routine for error handling subsystem that adds an event to an existing log file

Interface Name

error_t add_logevent (handle_t handle, data_t event);

Input Parameters

handle	valid handle returned from previous call to *open_log*
event	buffer of event data with records in *logevent* format

Example 1 (*con't*)

Exceptions

- Caller lacks permission to add to EVENT file
- Inadequate memory to add to an EVENT file

Effects

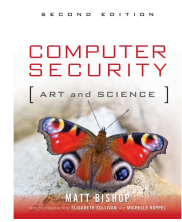
Event is added to EVENT log.

Output Parameters

```
status    status_ok           /* routine completed successfully */
          no_memory           /* insufficient memory (failed) */
          permission_denied   /* no permission (failed) */
```

Note

add_logevent is a user-visible interface



Example 2

- Interface for web user to change user password

Interface Name

User Manager / Change Password

Input Parameters

Old password	Current user's current password
New password	Current user's new password
Confirm new password	Current user's confirmation of new password
OK button	Used to submit change password request
CANCEL button	Used to cancel change password request and return to previous screen/window

Example 2 (*con't*)

Exceptions
- Caller does not have permission to submit change password request
- New password does not meet complexity requirements
- New password does not match confirmation password

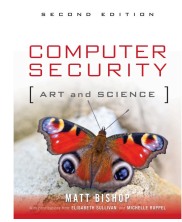
Effects
- Event is added to EVENT log
- If current password is correct, new password and confirmed password identical, and new password meets complexity requirements, user's password is changed

Output Parameters

Dialog box indicates password is changed, or password did not meet complexity requirements, or new and confirmed password did not match

Note

User Manager / Change Password is a user-visible interface



Internal Design

Describes internal structures and functions of components of system

1. *Overview of the parent component*; its high-level purpose, function, security relevance
2. *Detailed description of the component*; its features, functions, structure in terms of the subcomponents, all interfaces (noting externally visible ones), effects, exceptions, and error messages
3. *Security relevance of the component* in terms of security issues that it and its subcomponents should address

Example: Parent Component

- Documents high-level design of audit mechanism shown previously
- Audit component is responsible for recording accurate representation of all security-relevant events in the system and ensuring that integrity and confidentiality of the records are maintained.
  - *Audit view*: subcomponent providing authorized users with a mechanism for viewing audit records.
  - *Audit logging*: subcomponent records the auditable events, as requested by the system, in the format defined by the requirements
  - *Audit management*: subcomponent handling administrative interface used to define what is audited.

Example: Detailed Component Description

- Audit logging subcomponent records auditable events in a secure fashion. It checks whether requested audit event meets conditions for recording.
- Subcomponent formats audit record and includes all attributes of security-relevant event; generates the audit record in the predefined format
- Audit logging subcomponent handles exception conditions
  - Error writing to the log

Example

- Audit logging subcomponent uses one global structure:

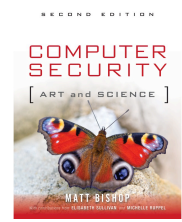
```
structure audit_config      /* defines configuration of */
                            /* which events to audit    */
```

- Audit logging subcomponent has two external interfaces:

```
add_logevent()              /* log an event */
logevent()                  /* ask to log event */
```

Example: Security Relevance

- Audit logging subcomponent monitors security-relevant events and records those events matching configurable audit selection criteria
  - Security-relevant events include attempts to violate security policy, successful completion of security-relevant actions



Low-Level Design

Focus on internal logic, data structures, interfaces; may include pseudocode

1. *Overview*, giving the purpose of the module and its interrelations with other modules, especially dependencies on other modules
2. *Security relevance of the module*, showing how the module addresses security issues
3. *Individual module interfaces*, identifying all interfaces to the module, and those externally visible.

Example: Overview of Module

- Audit logging subcomponent
  - Responsible for monitoring and recording security-relevant events
  - Depends on I/O system and process system components
- Audit management subcomponent
  - Depends on audit logging subcomponent for accurate implementation of audit parameters configured by audit management subcomponent
- All system components depend on audit logging component to produce their audit records

Example: Components Module Uses

- Audit logging subcomponent:

Variables

structure logevent_t	defines audit record
structure audit_ptr	current position in audit file
file_ptr audit_fd	file descriptor of audit file

Global structure

structure audit_config	defines configuration of which events are to be audited

External interfaces

add_logevent()	begin logging events of given type
logevent()	ask to log event

Example: Security Relevance of Module

- Audit logging subcomponent monitors security-relevant events, records those events matching the configurable audit selection criteria
  - Example: attempts to violate security policy
  - Example: successful completion of security-relevant actions
- Audit logging subcomponent must ensure no audit records are lost, and are protected from tampering

Example: Individual Module Interfaces

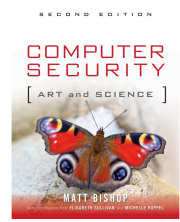
- *logevent*() only non-priviledged external interface

  verify function parameters
  call *check_selection_parameters* to determine if system has been configured to audit event
  if *check_selection_parameters* then
    call *create_logevent*
    call *write_logevent*
    return success or error number
  else
    return success

Example: Individual Module Interfaces (*con't*)

- *add_logevent*() available only to privileged users

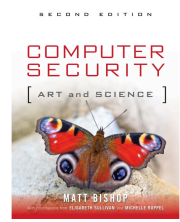
  verify caller has privilege/permission to use this function
  if caller does not have permission
    return permission_denied
  verify function parameters
  call *write_logevent* for each event record
  return success or error number from *write_logevent*



Internal Design

Show in which documents to put various designs to create a useful, readable, and complete set of documents

- *Introduction*: purpose, scope, target audience
- *Component overview*: identifies modules, data structures; how data is transmitted; security relevance and functionality
- *Detailed module designs*
  - *Module #1*: module's interrelations with other modules, local data structures, its control and data flows, security
    - *Interface Designs*: describes each interface
    - Interface 1a: security relevance, external visibility, purpose, effects, exceptions, error messages, and results



Example

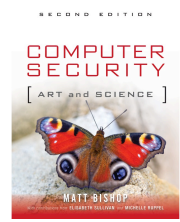
- Windows I/O System
  - High-level design document describes I/O system as a whole
    - Necessary descriptions of I/O System, Drivers, HAL
  - Describes first level of design decomposition
- Next level of decomposition (here only shown for I/O System)
  - High-level design document for I/O file drivers
  - Internal design specification for HAL component
- Internal design specifications for each subcomponent of I/O file drivers

Documentation and Specification

- Time, cost, efficiency may impact how complete set of documents prepared
- Different types of specifications
  - Modification Specifications
  - Security Specifications
  - Formal Specifications

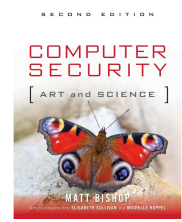
Modification Specifications

- Used when system built from previous versions or components
  - Specifications for these versions or components
  - Specifications for changes to, additions of, and methods for deleting modules, functions, components
- Developer understands the system upon which the new system is based



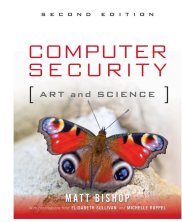
Security Considerations

- Security analysis must rest on specification of current system, not previous ones or changes only
  - If modification specifications are only ones, security analysis based upon incomplete specifications
  - If previous system has full security specifications, then analysis may be complete



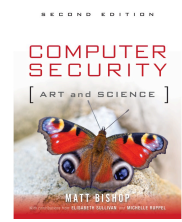
Security Specifications

- Used when design specifications adequate except for security issues
- Develop supplemental specifications to describe missing security functionality
  - Develop document that starts with security functions summary specification
  - Expand to address security issues of components, subcomponents, modules, functions



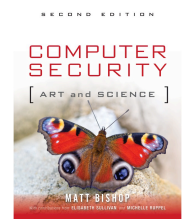
Example: System X

- Underlying UNIX system completely specified, including complete functional specifications and internal design specifications
  - Neither covered security well, let alone document new functionality
- Team supplemented existing documentation with security architecture document
  - Addresses deficiencies of existing documentation
  - Gives complete overview of each security function
  - Additional documentation describes external interface, internal design of all functions



Formal Specifications

- Any specification can be formal
- Written in formal language, with well-defined syntax and sound semantics
- Supporting tools allow checking
  - Parsers
  - Theorem provers

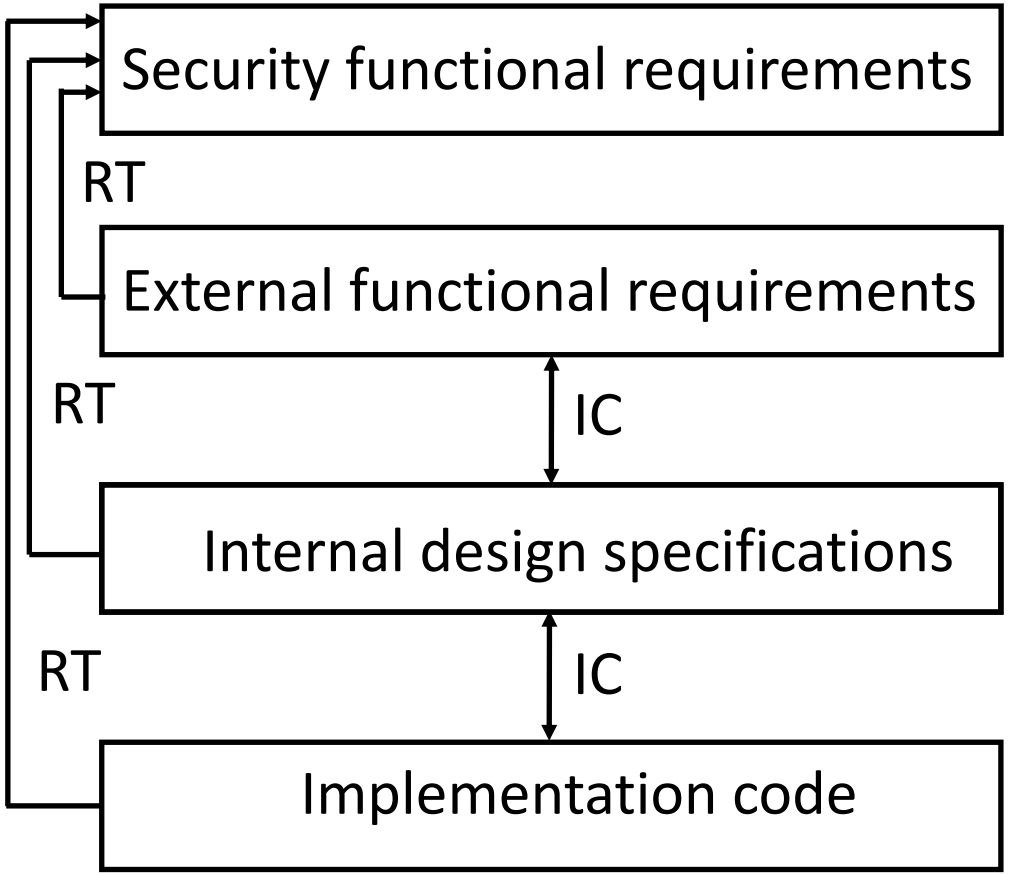


Justifications

- Formal techniques
  - Proofs of correctness, consistency
- Informal techniques
  - *Requirements tracing*: showing which specific security requirements are met by parts of a specification
  - *Informal correspondence* (also called *representation correspondence*): showing a specification is consistent with adjacent level of specification

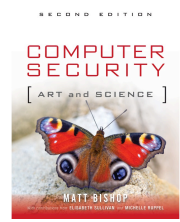
Requirements Mapping and Informal Correspondence

RT: requirements tracing
IC: Informal correspondence



Mappings Between Layers

- Informal techniques most appropriate when all levels of specification have identified requirements and all adjacent pairs of specifications have been shown to be consistent
  - Security functions summary specification and functional specification
  - Functional specification and high-level design specification
  - High-level design specification and low-level design specification
  - Low-level design specification and implementation code
- Doing third mapping may be difficult as difference in levels of abstraction can obscure relationship
  - Intermediate level often simplifies this



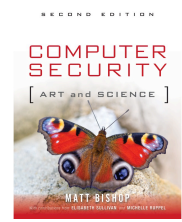
Example

- Family of specifications across several levels
- Security requirement R2 requires users of system be identified to system, and to have identification authenticated by system before use of any system functions
- Identification and authentication (I&A) high-level security-enforcing function from security functions summary specification:
    1. Users identify themselves to system using *login_ID* before they can use any system resources
    2. Users use password to authenticate their identity; system must accept password as authentic before any resources can be used
    3. Password must meet specific size, character constraints
- Interfaces *login*, *change_password* described in functional specification

Example

- Requirements mapping represented by table following explanation
  - In this example, only R2 maps to I&A
- Informal correspondence between functional, security functions summary specifications are:
  - *login* maps to items 1, 2 in description of I&A
  - *change_password* maps to items 2, 3 in description of I&A

Security requirements	**Function 1**	**I&A**	**...**	**Function *m***
R1				
R2		X		
...				
R*n*				

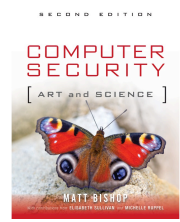


Informal Arguments

- Requirements tracing identifies components, modules, functions that meet requirements but not how well they are met
- *Informal arguments* uses approach similar to mathematical proofs

Example

- System W is a new version of an existing product
  - Previous version had good requirements, security functions summary, external functional, and design specifications
- System W added bug fixes, features (some large and pervasive)
- Developers created external functional specification, internal design specification documents for all modifications of the system
  - Each document defined scope to be modifications only
- Security analysts asked developers many questions
- Resulting combined security specification and analysis document addressed impacts of change on security of previous system



Example (*con't*)

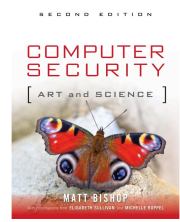
- Analysis document contained
  - Security analysis document containing individual documents for each of the different functional areas
  - System overview document
  - Test coverage analysis document
- Documentation semiformal, written in natural language with code excepts where practical
  - Design overview: gave high-level description of component, relevant security issues, impact on security
  - Requirements section: identified security functionality in module, traced it to applicable security functional requirements
  - Interface analysis: described new or impacted interfaces, mapped requirements to them, identified and documented security problems and made recommendations

Formal Methods

- Requirements tracing checks specifications satisfy requirements
- Specifiers intend to process specification using automated tools
  - Proof-based technology typically based on some form of logic (like predicate calculus); user constructs proof, proof checkers validate it
  - Model checking takes a security model and processes a specification to determine if it meets the model's constraints

Reviews of Assurance Evidence

- Reviewers given guidelines for review
- Other roles:
  - Scribe: takes notes
  - Moderator: controls review process
  - Reviewer: examines assurance evidence
  - Author: author of assurance evidence
  - Observer: observe process silently
- Important: managers may *only* be reviewers, and only then if their technical expertise warrants it

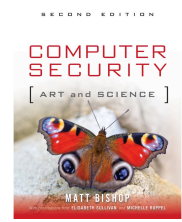


Setting Review Up

- Moderator manages review process
  - If not ready, moderator and author's manager discuss how to make it ready with author
  - May split it up into several reviews
  - Chooses team, defines ground rules
- Technical Review
  - Reviewers follow rules, commenting on any issues they uncover
    - May request moderator to stop review, send back to author
  - General and specific comments to author

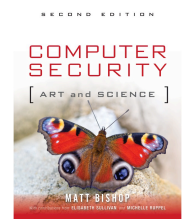
Review Meeting

- Moderator is master of ceremonies
  - Grammatical issues presented first
  - General and specific comments next
  - Goal is to collect comments on entity, *not* to resolve differences
  - Scribes write down comments and who made it (anyone can see it, help scribe, verify comment made)



Conflict Resolution

- After meeting, scribe creates Master Comment List
  - Reviewers mark “Agree” or “Challenge”
  - All comments that everyone “Agree”s are put on Official Comment List (OCL)
  - Rest must be resolved by reviewers
- Moderator, reviewers then:
  - Accept as is
  - Accept with changes on OCL
  - Reject



Conflict Resolution

- Author takes OCL, makes changes as sees fit
- Author then meets with reviewers
  - Explains how each comment made by reviewer was handled
  - All must be resolved to satisfaction of author, reviewer
- Review completed

Informal Review

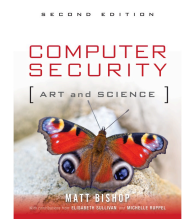
- Occurs sometimes due to quick pace of releases, bug fixes
  - Review process does not include moderator or scribe
  - Review may use electronic communications with one reviewer

Implementation Considerations for Assurance

- Make system modular, with minimum of interfaces
  - Interfaces are well-designed
  - Remove any non-security functionality from them, whenever possible
- Choice of programming language can affect assurance
  - Use one providing built-in features to avoid common flaws
    - Strong typing, built-in checks for buffer overflow, data hiding, error handling, etc.
  - Otherwise, develop and use appropriate coding standards and guidelines
    - Useful, but limited support for good code

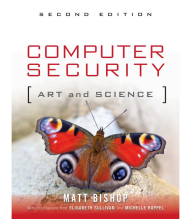
Implementation Management

- *Configuration management*: control of changes made in system's components, documentation, and testing throughout development, operational life
- Need processes, tools to do this effectively
- Configuration management system consists of:
  - Version control and tracking
  - Change authorization: restrict change check in to authorized people
  - Integration procedures
  - Product generation tools: generate the distribution version from authorized version



Justification

- Goal is to demonstrate implementation meets design
- Security testing
- Formal methods: used during coding processes, work best on small parts of a program performing well-defined tasks
  - We'll discuss these later (next chapter)



Testing

- Testing techniques
  - *Functional* (*black box*): testing to see how well entity meets its specifications
  - *Structural* (*white box*): testing based on analysis of code to develop test cases
- When to do testing
  - *Unit testing*: testing by developer on code module before integration
    - Usually structural testing
  - *System testing*: functional testing performed by integration team on integrated modules
    - May include structural testing
  - *Third-party* (*independent*) testing: functional testing by a group outside development organization

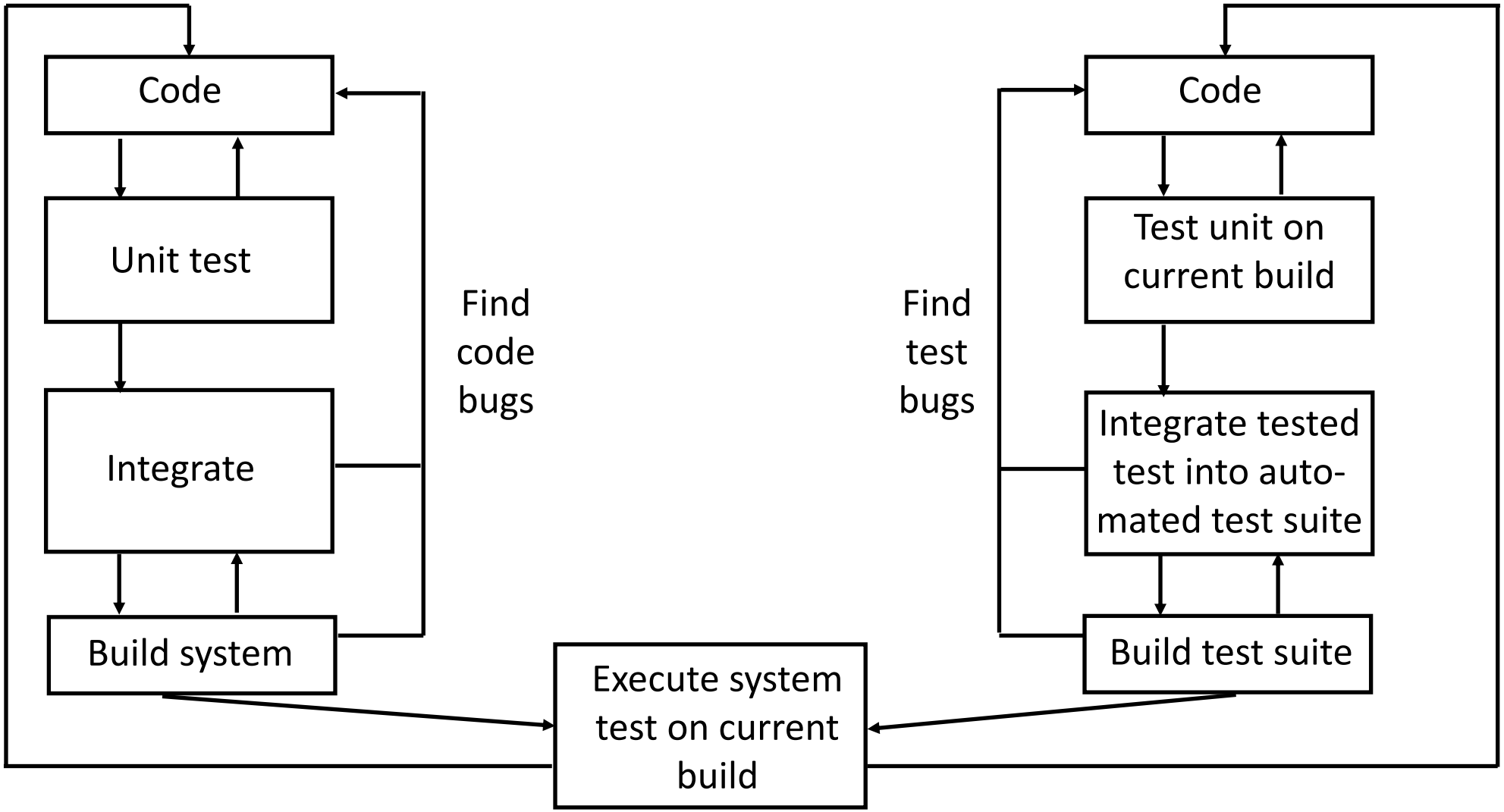
Security Testing

- Testing that addresses product security
  - *Security functional testing*: functional testing specific to security issues described in relevant specification
    - Focus is on pathalogical cases, boundary value issues, and so forth
  - *Security structural testing*: structural testing specific to security implementation found in relevant code
  - *Security requirements testing*: security functional testing specific to security requirements found in requirements specification
    - May overlap significantly with security functional testing
- Test coverage covers system security functions more consistently than ordinary testing
  - When completed, provides rigorous argument that all external interfaces have been completely tested

Security Testing

- Usually takes place at external interface level
  - Here, “interface” is point at which processing crosses security perimeter
  - Users access system through these
  - Therefore, violations of policy occur through these
- Parallel efforts, one by programming team, other by test team
- Security test suites ver large
  - Automated test suites essential

Code Development and Testing

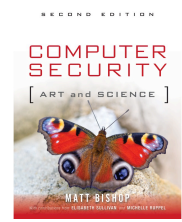


Plans and Reports

- Configuration management, documentation very important
  - Testers develop, document test plans, test specifications, test procedures, test results
- Writing test plans, specifications, procedures help authors examine, correct approaches
  - Provides assurance about test methodology
  - Enables analysis of test suite for correctness, completeness
- Reports identify which tests entity has passed, which it has failed
  - Watch out for failures due to automation (where automated test fails, but same test run independently of suite passes)

Security Testing Using PGWG

- PAT(Process Action Team) Guidance Working Group developed systematic approach to test development using successive decomposition of system, requirements tracing
- Methodology works well in system defined into successively smaller components
  - Requirements mapped to successively lower levels of design using *test matrices*
  - At lowest level, *test assertions* claim interfaces meet each requirement
  - Used to develop test cases
  - Includes documentation approach

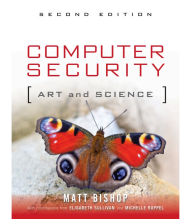


PGWG Test Matrices

- Two types of test matrices: high-level, low-level
- High level matrix
  - Rows are entity subsystems, major components
  - Columns are high-level security areas focused on functional requirements
    - Like access controls, integrity controls, cryptography
  - Cells give pointers to relevant documentation, lower-level test matrices
- Low level matrix
  - Rows are interfaces to subsystem, component
  - Columns represent security areas, their subdivisions, individual requirements
  - Cells contain test assertions, each of which apply to single interface and requirement
    - Any empty cells must be justified to show why requirement does not apply

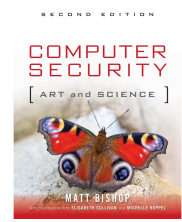
Example: Testing Security-Enhanced UNIX

- System includes file, memory, process, and IPC management, process control, I/O interfaces and devices
- Security functional requirement areas
  - Discretionary access control
  - Privileges, identification, authentication (I&A)
  - Object reuse protection
  - Security audit
  - System architecture constraints
- Testing uses interpretation of PGWG methodology
  - High-level matrix
  - Low-level matrices, 1 for each row of high-level matrix



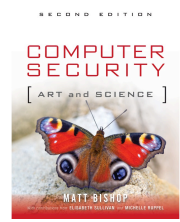
Example: High-Level Matrix

Security Requirement Area						
Component	DAC	Priv	I&A	OR	Audit	Arch
Process management					✓	
Process control	✓	✓		✓	✓	✓
File management	✓	✓		✓	✓	✓
Audit subsystem		✓	✓	✓	✓	✓
I/O subsystem interfaces	✓	✓	✓	✓	✓	
I/O device drivers		✓		✓	✓	✓
IPC management	✓	✓		✓	✓	✓
Memory management	✓	✓		✓	✓	✓



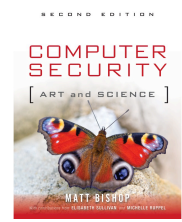
Example: Low-Level Matrix

System Call	DAC u/g/o	DAC ACL	Priv	I&A	OR	Security Audit	Logging	Isolation	Protection Domains
brk					✓			✓	✓
madvise								✓	✓
mmap	✓	✓			✓	✓	✓	✓	✓
mprotect	✓	✓				✓		✓	✓
msync								✓	✓
munmap			✓		✓	✓		✓	✓
plock	✓	✓	✓		✓	✓		✓	✓
vm-ctl	✓	✓	✓		✓	✓	✓	✓	✓



Test Assertions

- Created by identifying security-relevant, testable, analyzable conditions
  - Review design documentation for this
- PGWG methods for stating assertions
  - Develop statements describing behavior that must be verified
    - Example: “Verify that the calling process needs DAC write access permission to the parent directory of the file being created. Verify that if access is denied, the return error code is 2.”
  - Develop statements that tester must prove or disprove with tests
    - Example: “The calling process needs DAC write access permission to the parent directory of the file being created, and if access is denied, it returns error code 2.”
  - State assertions as claims embedded within structured specification format



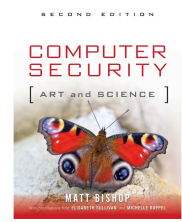
Test Specifications

- Test cases to verify truth of each assertion for each interface
- PGWG suggests:
  - High-level test specifications (HLTS) describe, specify test cases for each interface
  - Low-level test specifications (LLTS) provide information about each test case
    - Like setup and cleanup conditions, other environmental conditions

Example: HLTS for Interface *stime*()

High-level test specification includes assertion, test case specifications

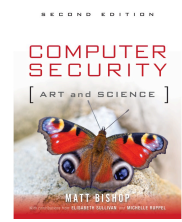
Assertion Number	Requirement Area and Number	Assertion	Relevant Test Cases
1	PRIV AC_1	Verify that only root can use system call *stime*() successfully	Stime_1, 2
2	PRIV AC_2	Verify audit record generated for every failed *stime*() call	Stime_1, 2
3	PRIV AC_3	Verify audit record generated for every successful *stime*() call	Stime_1, 2



Test Case Specifications

- Describe specific tests required to meet assertions

Test Case Name and Number	Is UserID = *root*?	Expected Results
Stime_1	Yes	Call to *stime*() should succeed; audit record should be generated noting successful attempt and new clock time
Stime_2	No	Call to *stime*() should fail; audit record should be generated noting failed attempt



LLTS for Stime_1

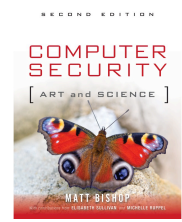
Test case name: K_MIS_stime_1

Test case description: Call *stime* as a non-*root* user to change system time; this should fail, verifying only *root* can use this call successfully

Expected result: *stime* call should fail with return value of –1, system clock should be unchanged, error number (*errno*) set to **EPERM**, audit record as shown below

Test specific setup:

1. Log in as a non-*root* user (*secusr1*)
2. Get the current system time

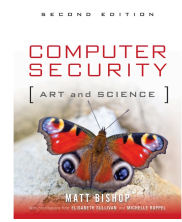


LLTS for Stime_1 (*con't*)

Algorithm:

1. Do the setup as above
2. Call *stime* to change system time to 10 min ahead of current time
3. If return value is –1, error number is **EPERM**, and current system time not new time given to *stime*, declare the test passed; otherwise, declare failed

Cleanup: If system time has changed, reduce current time to 10 minutes



LLTS for Stime_1 (*con't*)

Audit record field values for failure (success):

Authid	secusr1
RUID	secusr1
EUID	secusr1
RGID	scgrp1
EGID	secgrp1
Class	tune
Reason	Privilege failure (success)
Event	SETTHETIME_1
Message	Privilege failure (none)

Operation, Maintenance Assurance

- Bugs will be found during operation, requiring fixes
  - *Hot fix*: handle bugs immediately, sent out as quickly as possible
    - Used to fix bugs that immediately affect system security or operation
  - *Regular fix*: handle less serious bugs or give long-term solutions to bugs fixed by hot fix, usually collected until some condition arises and then sent out
    - Sent out as maintenance release or as “patch Tuesday” or some other way
- Well-defined procedures handle, track reported flaws
  - Include information about bug, such as description, remedial actions, severity, pointer to related configuration management entries, other documentation
  - Actions taken follow same security procedures used during original devlopment

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