ECS 235B Module 26 Implementation Assurance Techniques

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

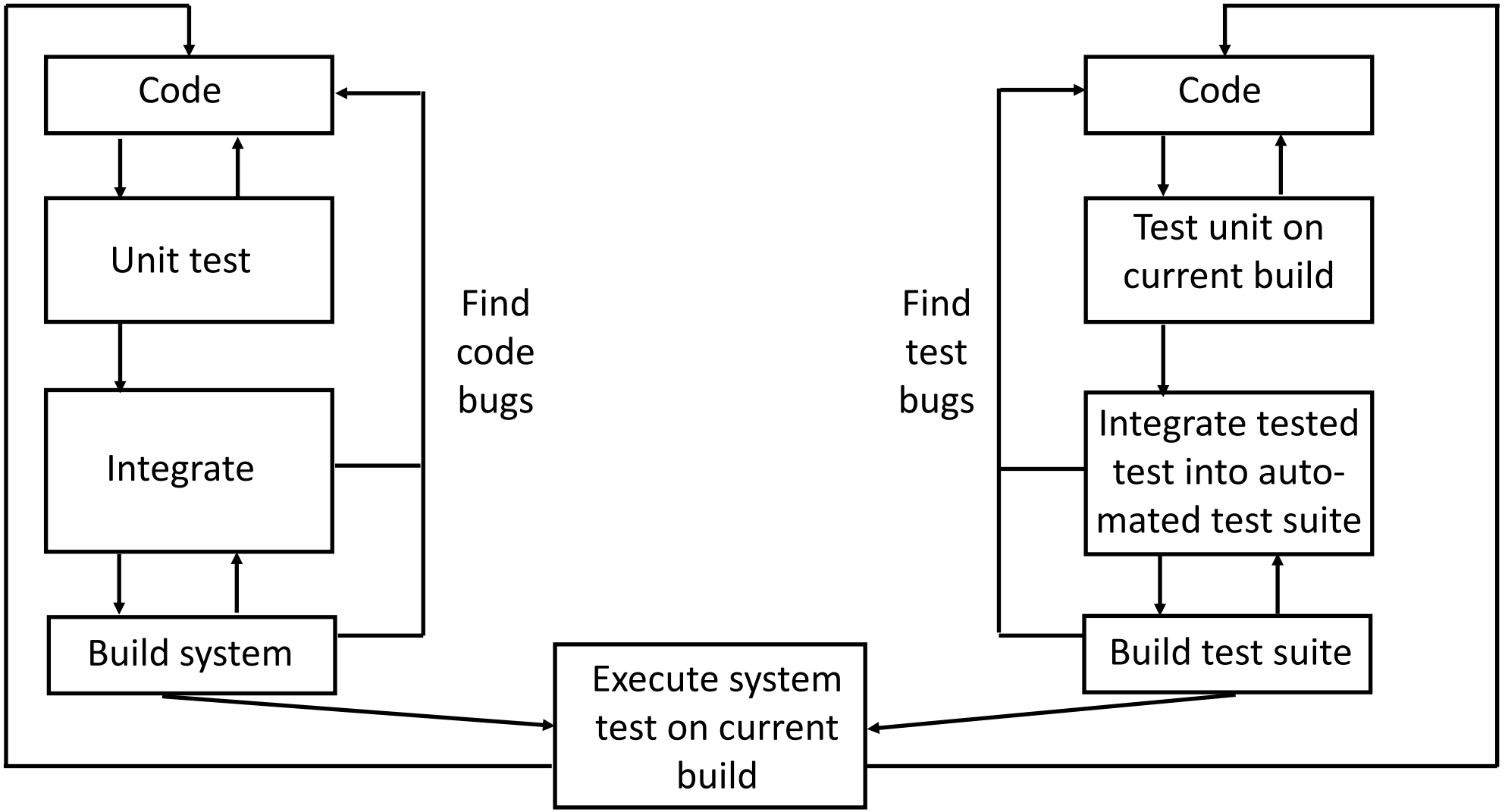
Security Testing

- 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