

Secure Operating Systems (CYB610)

Master of Science in Cybersecurity

Lecture 5 OS and Program Security

Dr. Ibrahim Gomaa

*Faculty of Informatics
Midocean University*



Chapter

05

Topics

This chapter covers the following topics and concepts:

- **Programs security**
- **Controls for Security**
- **Operating System Controls for Security**

Chapter

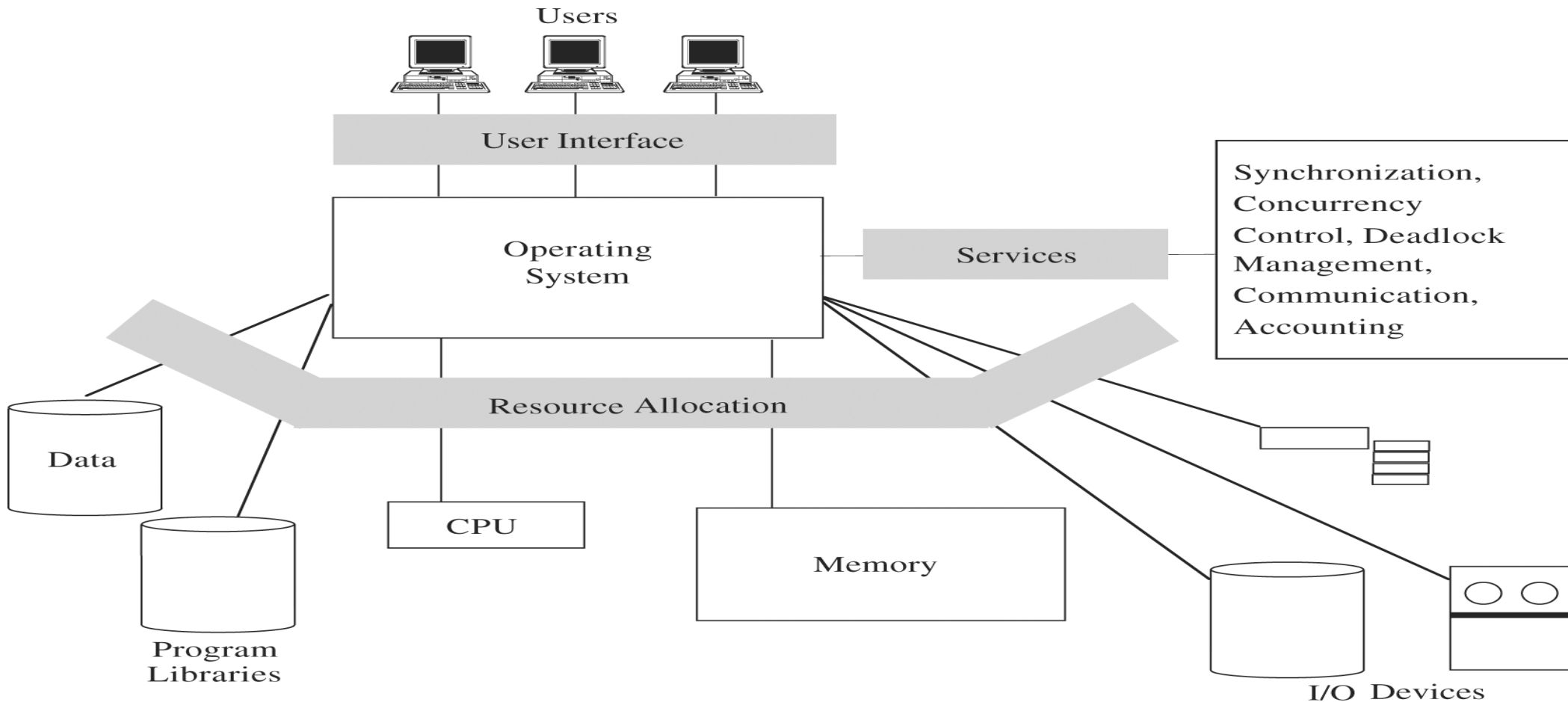
05

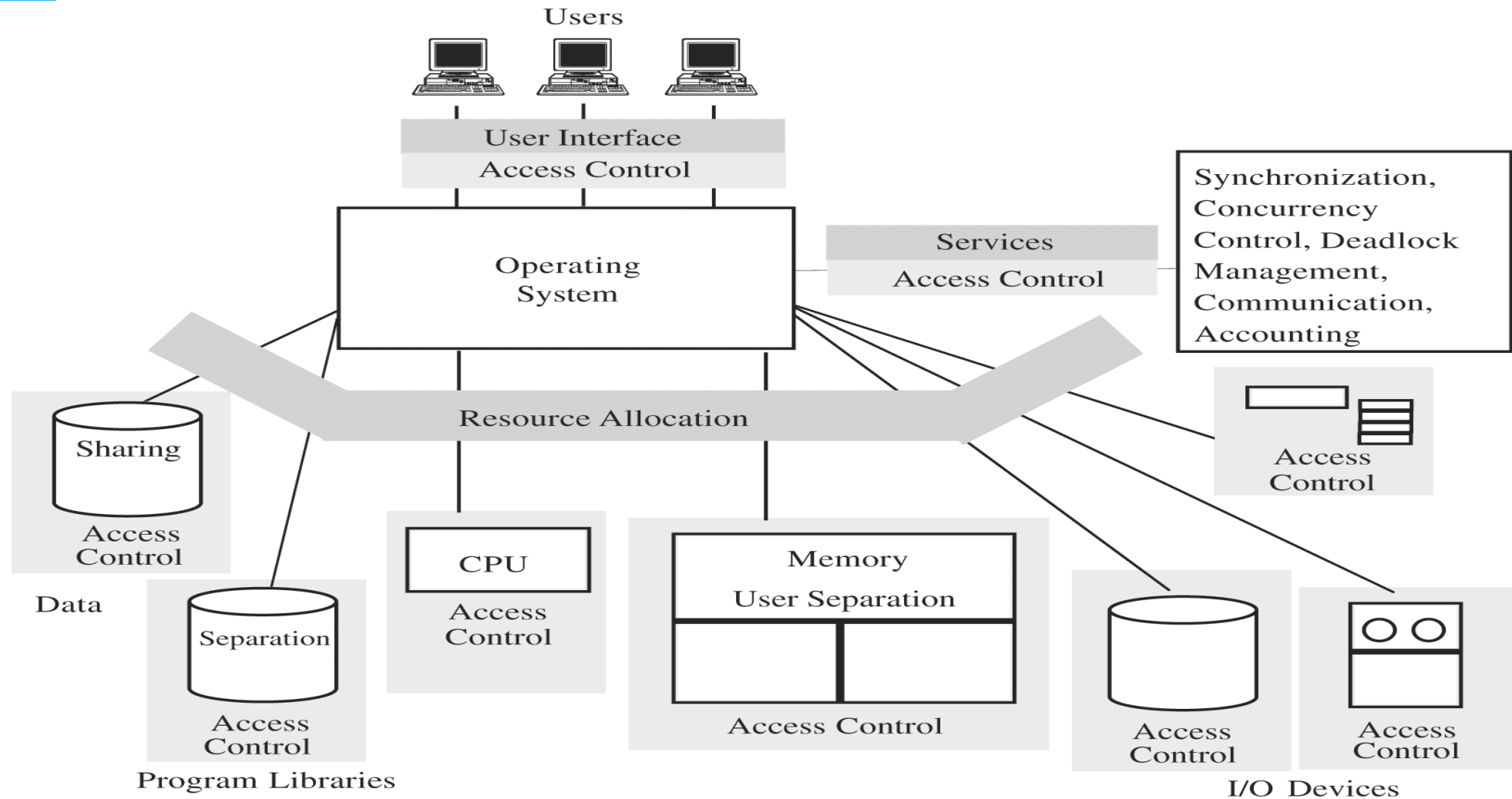
Goals

When you complete this chapter, you will be able to:

- **Know What is “Program security”**
- **Explain the Basic approaches to having secure programs**
- **Illustrate how to control security of programs during their development and maintenance**

Overview of an OS Functions





Programs security

Program is secure if we trust that it provides/enforces:

- Confidentiality
- Integrity
- Availability

What is “Program security?”

Depends on who you ask

- **user** - fit for his task
- **programmer** - passes all “his” tests
- **manager** - conformance to all specifications
- **Developmental criteria** for **program security** include:
 - Correctness of security & other requirements
 - Correctness of implementation
 - Correctness of testing

Fault tolerance terminology

- **Error** - may lead to a fault
- **Fault** - cause for deviation from the intended function
- **Failure** - system malfunction caused by fault

Note:

- Faults - seen by **insiders** (e.g., programmers)
- Failures - seen by **outsiders** (e.g., independent testers, users)

Error/fault/failure example:

Programmer's indexing error, leads to buffer overflow fault

Buffer overflow fault causes system crash (a failure)

Two categories of faults :

- **Permanent** faults
- **Transient** faults – can be much more difficult to diagnose

Basic approaches to having secure programs

1. Judging S/W security **by fixing** program **faults**
 - **Red Team / Tiger Team** tries to crack S/W
 - If programs withstands the attack $\Rightarrow$ security is good
2. Judging S/W security by testing program behavior
Run tests to **compare behavior vs. Requirements.**
 - Important: If a flaw detected as a failure (**an effect**), look for the underlying fault (**the cause**)
Recall: fault seen by insiders, failure – by outsiders
 - If possible, detect faults before they become failures

Basic approaches to having secure programs

- Any kind of fault/failure can cause a security incident
we must consider security consequences for all kinds of detected faults/failures
- Even inadvertent faults / failures
 - Inadvertent faults are the biggest source of security vulnerabilities exploited by attackers
- Testing only increases probability of eliminating faults

Basic approaches to having secure programs

3. Judging s/w security by program Security Analysis

- Best approach to judging s/w security
- Analyze what can go wrong
 - At every stage of program development!
 - From requirement definition to testing
 - After deployment
 - Configurations / policies / practices

Controls for Security

How to control security of programs during their development and maintenance?

Three types of controls for security (against program flaws):

1. **Developmental controls**
2. **Operating system controls**
3. **Administrative controls**

Controls for Security

- **Overview**

“Better to prevent than to cure”

- Preventing security flaws
 - We have seen a lot of possible security flaws
 - How to prevent (some of) them?
 - **Software engineering** concentrates on developing and maintaining high-quality s/w
 - We'll take a look at some techniques useful specifically for developing/ maintaining secure s/w

Developmental Controls for Security

- Nature of s/w development
 - Collaborative effort
 - Team of developers, each involved in ≥ 1 of stages:
 - Requirement specification
 - Regular req. specs: “do X”
 - Security req. specs: “do X and nothing more”

Developmental Controls for Security

- Design
- Implementation
- Testing
- Documenting at each stage
- Reviewing at each stage
- Managing system development thru all stages
- Maintaining deployed system (updates, patches, new versions, etc.)

Both **product** and **process** contribute to overall quality

Developmental Controls for Security

Fundamental principles of S/W engineering

1. Modularity
2. Encapsulation
3. Info hiding

Fundamental principles of S/W engineering

1. Modularity

Modules should be:

- Single-purpose - logically/functionally
- Small - for a human to grasp
- Simple - for a human to grasp
- Independent – high cohesion, low coupling
 - High cohesion – highly focused on (single) purpose
 - Low coupling – free from interference from other modules
- Modularity should improve correctness
 - Fewer flaws => better security

Fundamental principles of s/w engineering

2. Encapsulation

- Minimizing info sharing with other modules
=> Limited interfaces reduce # of covert channels
- Well documented interfaces
- “Hiding what should be hidden and showing what should be visible.”

Fundamental principles of s/w engineering

3. Information hiding

4.

- Module is a black box
 - Well defined function and I/O
- It's easy to know what a module does, but not how it does it.
- Reduces complexity, interactions, covert channels, ...
=> better security

Developmental Controls for Security

Many techniques for building a solid software

- Peer reviews
- Hazard analysis
- Testing
- Good design
- Risk prediction & management
- Static analysis
- Configuration management
- Additional developmental controls

Building a solid software

1) Peer Reviews - three types

- **Informal Reviews**
 - No formal process
 - May take the form of pair programming or a technical lead reviewing designs and code
 - Results may be documented
 - Varies in usefulness depending on the reviewers
 - Gain consensus on solutions before development
 - **Main purpose:** inexpensive way to get some benefit
- **Walk-throughs**
 - Developer walks team through code/document
 - Discover flaws in a single design document
 - May vary in practice from quite informal to very formal
 - **Main purposes:** learning, gaining understanding, finding defects
- **Inspection**
 - Formalized and detailed
 - Statistical measures used
 - **Main purpose:** finding defects

Various types of peer reviews can be highly effective

Building a solid software

2) Hazard Analysis

- systematic techniques to expose

potentially hazardous system states, including security vulnerabilities
- Components of HA
 - Hazard lists
 - What-if scenarios – identify non-obvious hazards
 - System-wide view (not just code)
 - Begins Day 1
 - Continues throughout SDLC (Software Development Life Cycle)
- Techniques
 - HAZOP – Hazard and Operability Studies
 - FMEA – Failure Modes and Effects Analysis
 - FTA – Fault Tree Analysis

Building a solid software

3) Testing – phases:

- Module/component/unit testing of individual modules
- Integration testing of interacting (sub)system modules
- (System) function testing checking against requirement specs
- (System) performance testing
- (System) acceptance testing – **with customer** against customer's requirements — on seller's or customer's premises
- (System) installation testing after installation on customer's system
- Regression testing after updates/changes to s/w
- **Types of testing**
 - Black Box testing – testers can't examine code
 - White Box / Clear box testing – testers can examine design and code, can see inside modules/system

Building a solid software

4) Good design

- Good design uses:
 - I. Modularity / encapsulation / info hiding
 - II. Fault tolerance
 - III. Consistent failure handling policies
 - IV. Design rationale and history
 - V. Design patterns

- I. Using **modularity / encapsulation / info hiding**
- II. Using fault tolerance for reliability and security
 - System tolerates component failures
 - **System more reliable than any of its components**
 - Different than for security, where system is as secure as its weakest component

Fault-tolerant approach:

- Anticipate faults (car: anticipate having a flat tire)
 - **Active fault detection** rather than **passive fault detection**
(e.g., by use of mutual suspicion: active input data checking)
- Use redundancy (car: have a spare tire)
- Isolate damage
- Minimize disruption (car: replace flat tire, continue your trip)

III. Using consistent failure handling policies

Each failure handled by one of 3 ways:

Retrying

Restore previous state, redo service using different “path”

E.g., use secondary code instead of primary code

Correcting

Restore previous state, correct something, run service using the same code as before

Reporting

Restore previous state, report failure to error handler, don't rerun service

Example — How fault-tolerance enhances security

If security fault destroys important data (availability in CIA), use **fault tolerant** to revert to backup data set

Good Design

IV. Using design rationale and history

Knowing it (including knowing design rationale and history for security mechanisms) helps developers modifying or maintaining system

V. Using design patterns

Knowing it enables looking for patterns showing what works best in which situation

Good Design

- Value of Good Design
 - Easy maintenance
 - Understandability
 - Reuse
 - Correctness
 - Better testing

=> translates into (saving) BIG bucks !

Building a solid software

5) Risk prediction & management

- Predict and manage risks involved in system development and deployment
 - Make plans to handle unwelcome events should they occur
- Risk prediction/mgmt are especially important for security
 - Because unwelcome and rare events can have security consequences
- Risk prediction/mgmt helps to select proper security controls (e.g., proportional to risk)

Building a solid software

6) Static analysis

- Before system is up and running, examine its design and code to locate security flaws
 - More than peer review
- Examines
 - Control flow structure (sequence in which instructions are executed, incl. iterations and loops)
 - Data flow structure (trail of data)
 - Data structures
- Automated tools available

Building a solid software

7) Configuration management

It is a process of controlling system modifications during development and maintenance
Offers security benefits by scrutinizing new/changed code

Problems with system modifications

One change interfering with other change

E.g., neutralizing it

Proliferation of different versions and releases

Older and newer

For different platforms

For different application environments (and/or customers categories)

Building a solid software

Reasons for software modification

- Corrective changes
 - To maintain control of system's day-to-day functions
- Adaptive changes
 - To maintain control over system's modifications
- Perfective changes
 - To perfect existing acceptable system functions
- Preventive changes
 - To prevent system's performance degradation to unacceptable levels

8) Additional developmental controls

8a) Learning from mistakes

Avoiding such mistakes in the future enhances security

8b) Proofs of program correctness

Formal methods to verify pgm correctness

Logic analyzer shows that:

- initial assertions about inputs...

- ... through implications of pgm statements...

- ... lead to the terminal condition (desired output)

Problems with practical use of pgm correctness proofs

- Esp. for large pgms/systems

Most successful for specific types of apps

- E.g. for communication protocols & security policies

**Even with all these developmental controls (1-8) –
still no security guarantees!**

1) Standards and guidelines for program development

Capture experience and wisdom from previous projects

Facilitate building higher-quality s/w (incl. more secure)

They include:

- Design Standard & Guidelines – design tools, languages, methodologies
- S&G for documentation, language, and coding style
- Programming S&G - incl. reviews, audits
- Testing S&G
- Configuration mgmt S&G

2) Security audits

- Check compliance with S&G

- Scare potential dishonest programmer from including illegitimate code (e.g., a trapdoor)

3) Separation of duties

- Break sensitive tasks into ≥ 2 pieces to be performed by different people (learned from banks)

- Example 1: modularity

 - Different developers for cooperating modules

- Example 2: independent testers

 - Rather than developer testing her own code

- Developmental controls not always used

OR:

Even if used, not foolproof

=> Need other, complementary controls, incl. OS controls

Such OS controls can protect against some program flaws

Trusted software

– code accurately developed and analyzed so we can trust that it does all and only what specs say.

- Trusted code establishes foundation upon which untrusted code runs
- Trusted code establishes security baseline for the whole system

In particular, OS can be trusted s/w

Key characteristics determining if OS code is trusted

1) Functional correctness

OS code consistent with specs

2) Enforcement of integrity

OS keeps integrity of its data and other resources even if presented with flawed or unauthorized commands

3) Limited privileges

OS minimizes access to secure data/resources

Trusted programs must have “need to access” and proper access rights to use resources protected by OS

Untrusted programs can’t access resources protected by OS

4) Appropriate confidence level

OS code examined and rated at appropriate trust level

Similar criteria used to establish if s/w other than OS can be trusted

Ways of **increasing security** if **untrusted** programs present:

- 1) Mutual suspicion
- 2) Confinement
- 3) Access log

1) **Mutual suspicion** between programs

Distrust other programs – treat them as if they were incorrect or malicious

- program protects its interface data
- With data checks, etc.

2) Confinement

OS can confine access to resources by suspected program

Example 1: strict compartmentalization

program can affect data and other programs *only* within its
compartment

Example 2: sandbox for untrusted programs

Can limit spread of viruses

3) Audit log / access log

Records who/when/how (e.g., for how long) accessed/used which objects

Events logged: logins/logouts, file accesses, program executions, device uses, failures, repeated unsuccessful commands (e.g., many repeated failed login attempts can indicate an attack)

Audit frequently for unusual events, suspicious patterns

It is a **forensic measure** not protective measure

Forensics – investigation to find who broke law, policies, or rules (a posteriori, not a priori)

Summary

- What is “Program security”
- Basic approaches to having secure programs
- Controls for Security
- Developmental Controls for Security
- Administrative Controls for Security
- Operating System Controls for Security



Thank You