DENNIS GUNAWAN

IF470 COMPUTER SECURITY

03 PROGRAM SECURITY

REVIEW: USER AUTHENTICATION

- Attack
  - Impersonation / Failed Authentication
- Vulnerability
  - Faulty or Incomplete Authentication
- Countermeasure
  - Strong Authentication

COURSE SUB LEARNING OUTCOMES (SUB-CLO)

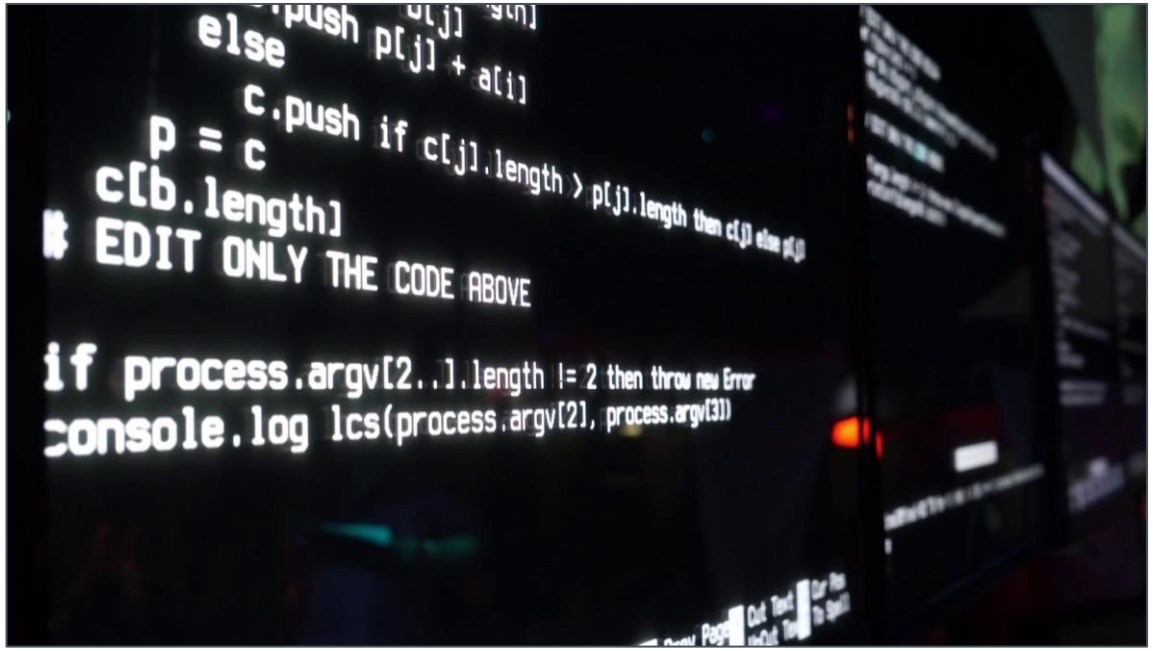
- Sub-CLO 3
  - Students are able to relate program security to real case in their daily life (C3)

OUTLINE

- Threat
  - Program Flaw Leads to Security Failing
- Vulnerability
  - Incomplete Mediation
  - Race Condition
  - Time-of-Check to Time-of-Use
  - Undocumented Access Point
- Ineffective Countermeasure
  - Penetrate-and-Patch
- Countermeasure
  - Identifying and Classifying Faults
  - Secure Software Design Elements
  - Secure Software Development Process
  - Testing
  - Defensive Programming

INTRODUCTION

- Programs and their computer code are the basis of computing
- Today's computer users sometimes write their own code, but more often they buy programs off the shelf
- All users gladly run programs all the time
  - Spreadsheets, music players, word processors, browsers, email handlers, games, simulators, and more



INTRODUCTION

- The programs have become more numerous and complex
  - Users are more frequently unable to know what the program is really doing or how
- Users seldom know whether the program they are using is producing correct results
  - What if a spreadsheet produces a result that is off by a small amount?
  - What if an automated drawing package doesn't align objects exactly?

INTRODUCTION

- Programs are written by fallible humans
- Program flaws can have 2 kinds of security implications
  - They can cause integrity problems leading to harmful output or action
  - They offer an opportunity for exploitation by a malicious actor
- Program correctness becomes a security issue

INTRODUCTION

Attackers are alert to

seemingly insignificant flaws

that can be exploited for larger effect

- When performing threat analysis, you should think of each possible outcome not just as a final result but also as a possible stepping stone to exploiting other vulnerabilities

PROGRAM FLAW LEADS TO SECURITY FAILING

- Common programming flaws
  - Exceeding the bounds of an array
  - Calculating subscripts from 0 instead of 1
  - Performing arithmetic on a pointer instead of the data to which it points
  - Storing a string value in a numeric location

PROGRAM FLAW LEADS TO SECURITY FAILING

- Programmers can be too optimistic
  - They tend to assume that
    - Data inputs are correct
    - Each update or correction is complete
    - Testing is effective
- Because these assumptions usually hold, programmers do not verify them and do not provide alternatives in case an assumption is mistaken

Given an assignment to compute the average of 2 input values

- Programmers may assume that
  - Both values are numbers
  - Their sum will not exceed the size of a computer word
  - They are represented in proper format (integer or floating point)
  - The module will receive exactly 2 inputs, not fewer or more

INCOMPLETE MEDIATION

Mediation

Verifying that the subject is authorized to perform the operation on an object

- Forgetting to ask "Who goes there?" before allowing the knight across the castle drawbridge is just asking for trouble

INCOMPLETE MEDIATION

http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=2009Jan17

- What would happen?
  - 1800Jan01?
  - 1800Feb30?
  - 2048Min32?
  - 1Aardvark2Many?
- Possibilities
  - The system would fail catastrophically
  - The receiving program would continue to execute but would generate a very wrong result
  - The processing server might have a default condition

INCOMPLETE MEDIATION

http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=2009Jan17

- The sensitive data (the parameter values) are in an exposed, uncontrolled condition
- The data values are not completely mediated

INCOMPLETE MEDIATION

http://www.things.com/order.asp?custID=101&part=555A&qy=20&price=10&ship=boat&shipcost=5&total=205

- This procedure leaves the parameter statement open for malicious tampering

http://www.things.com/order.asp?custID=101&part=555A&qy=20&price=1&ship=boat&shipcost=5&total=25

- From a security perspective, the most serious concern about this flaw was the length of time that it could have run undetected

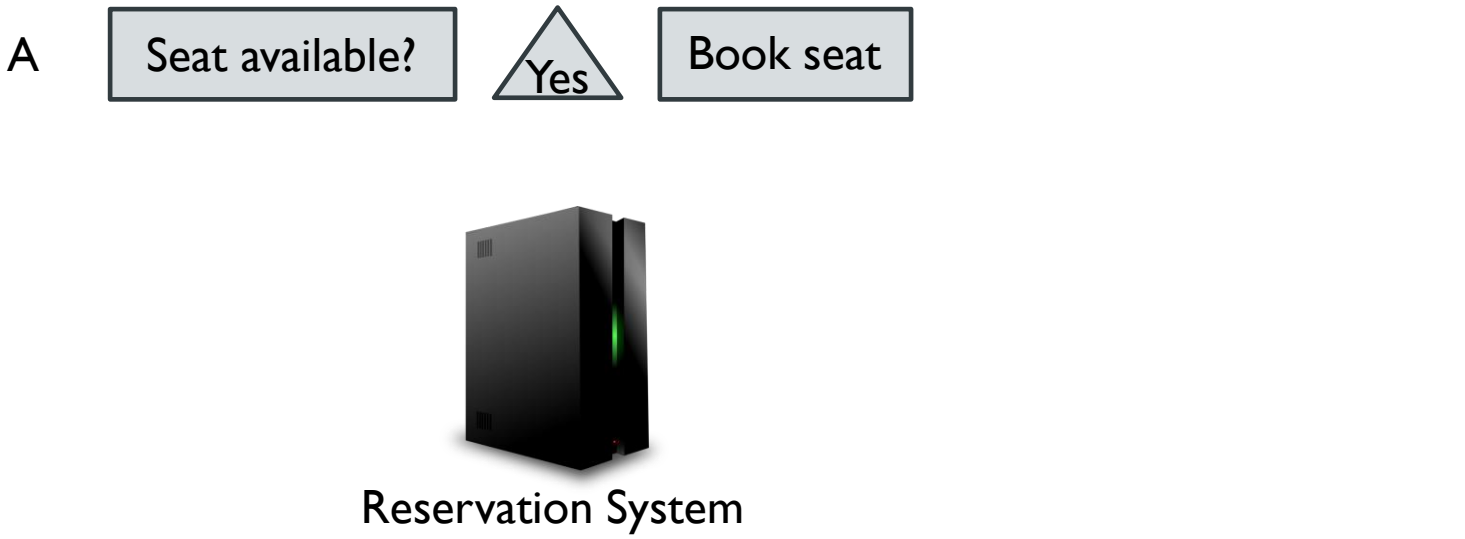
RACE CONDITION

Serialization Flaw

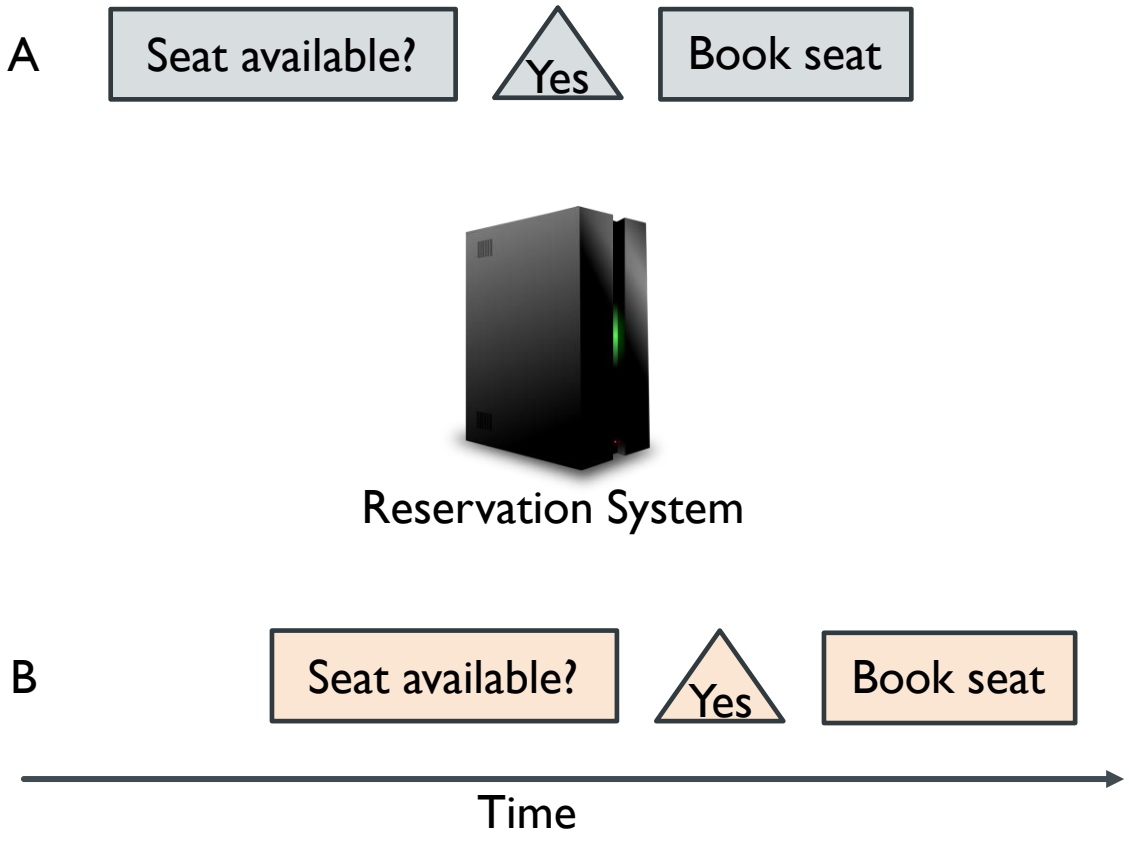
2 processes are competing within the same time interval, and the race affects the integrity or correctness of the computing tasks

- 2 processes execute concurrently, and the outcome of the computation depends on the order in which instructions of the processes execute

RACE CONDITION



RACE CONDITION



A race condition between 2 processes can cause inconsistent, undesired, and therefore wrong, outcomes

–

A failure of integrity

TIME-OF-CHECK TO TIME-OF-USE (TOCTTOU)

Consider a person buying a sculpture that costs $100. The buyer removes five $20 bills from a wallet, carefully counts them in front of the seller, and lays them on the table. Then the seller turns around to write a receipt. While the seller's back is turned, the buyer takes back one $20 bill. When the seller turns around, the buyer hands over the stack of bills, takes the receipt, and leaves with the sculpture.

- It exploits the delay between the two actions: check and use
- Between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check

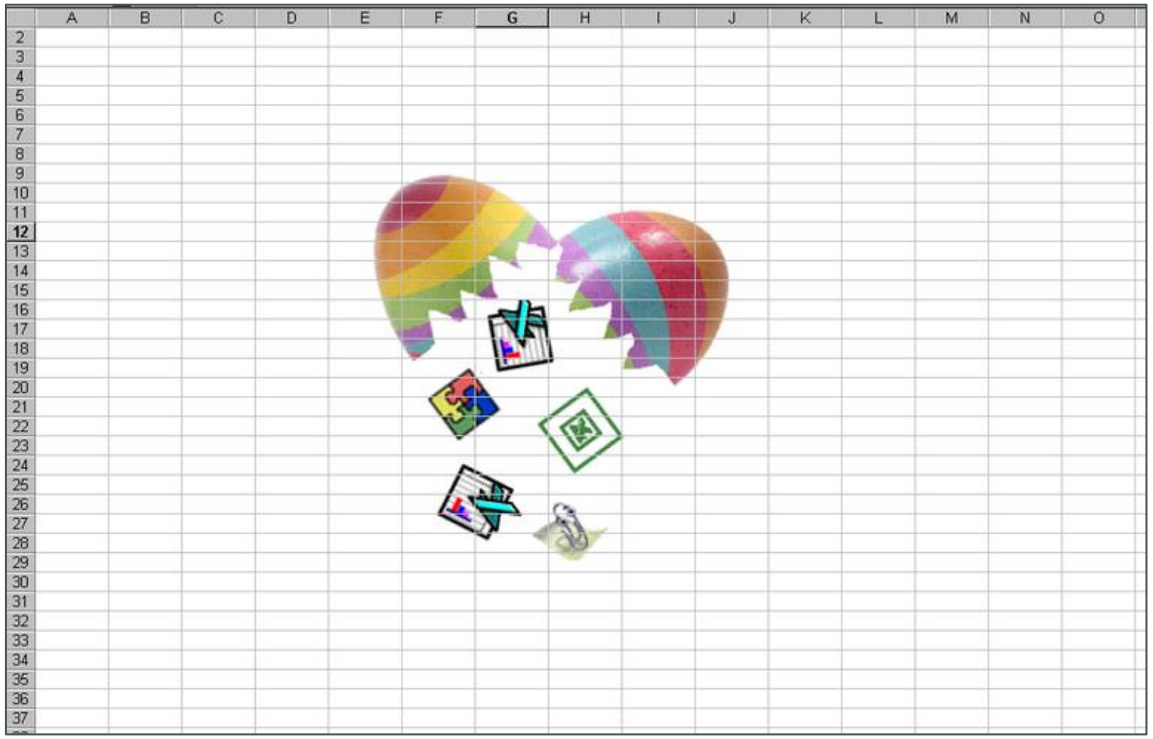
UNDOCUMENTED ACCESS POINT

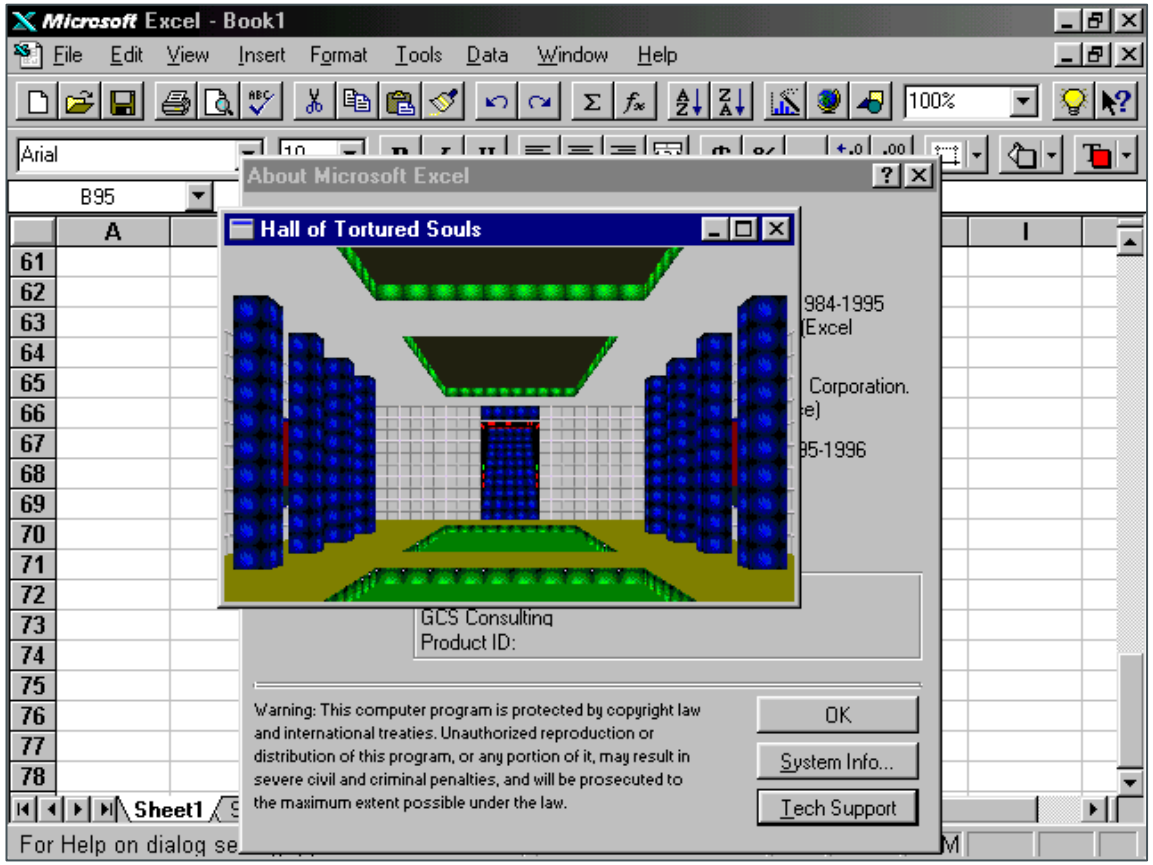
- During program development and testing, the programmer needs a way to access the internals of a module
- The programmer wants a special debug mode to test conditions
- Such an access point is called a backdoor or trapdoor

UNDOCUMENTED ACCESS POINT

- The programmer forgets to remove these entry points when the program moves from development to product
- The programmer decides to leave them in to facilitate program maintenance later
- The programmer may believe that nobody will find the specific entry
  - If there is a hole, someone is likely to find it

UNDOCUMENTED ACCESS POINT





SALAMI ATTACK

- The crook shaves a little from many accounts and puts these shavings together to form a valuable result
  - Like the meat scraps joined in a salami

- Why?
  - Computer computations are notoriously subject to small errors involving rounding and truncation
  - Malicious programmer

- Suppose an account generates $10.32 in interest
  - Would the account-holder notice if there were only $10.31? Or $10.21? Or $9.31?
  - Highly unlikely for a difference of $0.01, and not very likely for $1.01

PENETRATE-AND-PATCH

Search for and repair flaws

- 4 reasons why penetrate-and-patch is a misguided strategy
  - The pressure to repair a specific problem encourages developers to take a narrow focus on the fault itself and not on its context
  - The fault often has nonobvious side effects in places other than the immediate area of the fault
  - Fixing one problem often causes a failure somewhere else
  - The fault cannot be fixed properly because system functionality or performance would suffer as a consequence

IDENTIFYING AND CLASSIFYING FAULTS

It doesn't matter
whether the stone hits the pitcher
or the pitcher hits the stone,
it's going to be bad for the pitcher

- An inadvertent error can cause just as much harm to users and their data as can an intentionally induced flaw
- A system attack often exploits an unintentional security flaw to perform intentional damage

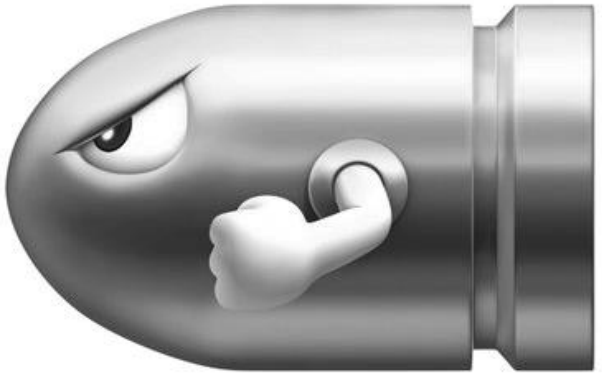
IDENTIFYING AND CLASSIFYING FAULTS

We do not have techniques to eliminate or address all program security flaws

- Program controls apply at the level of the individual program and programmer
  - Security is also about preventing certain actions: a "shouldn't do" list
    - Infeasible to list all of them
  - We cannot exhaustively test every state and data combination to verify a system's behavior
- Programming and software engineering techniques change and evolve far more rapidly than do computer security techniques

IDENTIFYING AND CLASSIFYING FAULTS

There is no “silver bullet”
to achieve security effortlessly



IDENTIFYING AND CLASSIFYING FAULTS

The inadvertent flaws fall into 6 categories

- Validation error (incomplete or inconsistent)
  - Permission checks
- Domain error
  - Controlled access to data
- Serialization and aliasing
  - Program flow order
- Inadequate identification and authentication
  - Basis for authorization
- Boundary condition violation
  - Failure on first or last case
- Other exploitable logic errors

IDENTIFYING AND CLASSIFYING FAULTS

- Each software problem (especially when it relates to security) has the potential not only for making software fail but also for adversely affecting a business or a life

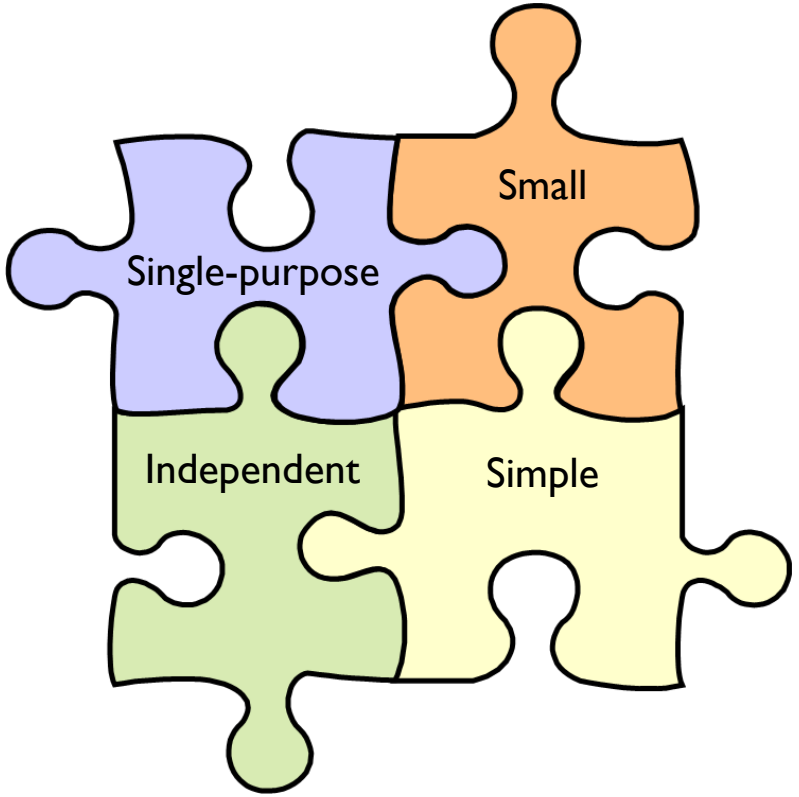
"One of the things we kept in mind during the course of our review is that in the conduct of space missions, you get only one strike, not three. Even if thousands of functions are carried out flawlessly, just one mistake can be catastrophic to a mission."

SECURE SOFTWARE DESIGN ELEMENTS

- Modularity

Modularization

The process of dividing a task into subtasks



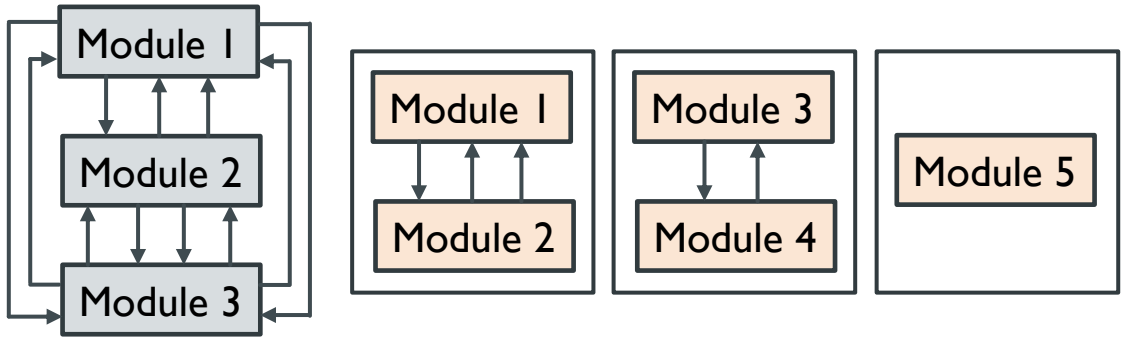
- Advantages
  - Maintenance
  - Understandability
  - Reuse
  - Correctness
  - Testing

SECURE SOFTWARE DESIGN ELEMENTS

- Modularity

A modular component usually has high cohesion and low coupling

- A highly cohesive component has a high degree of focus on the purpose
- Loosely coupled components are free from unwitting interference from other components



SECURE SOFTWARE DESIGN ELEMENTS

- Encapsulation
  - The technique for packaging the information (inside a component) in such a way as to hide what should be hidden and make visible what is intended to be visible
  - A component is affected only in known ways by other components in the system
  - Fewest interfaces possible are used

- Information Hiding
  - Other components' designers do not need to know how the module completes its function
  - Malicious developers cannot easily alter the components of others if they do not know how the components work

SECURE SOFTWARE DESIGN ELEMENTS

- Mutual Suspicion
  - Mutually suspicious programs operate as if other routines in the system were malicious or incorrect
  - A calling program cannot trust its called subprocedures to be correct, and a called subprocedure cannot trust its calling program to be correct

- Confinement
  - A technique used by an operating system on a suspected program to help ensure that possible damage does not spread to other parts of a system
  - A confined program is strictly limited in what system resources it can access

SECURE SOFTWARE DESIGN ELEMENTS

- Genetic Diversity
  - For reasons of convenience and cost, we often design systems with software or hardware (or both) from a single vendor
  - It is risky having many components of a system come from one source or rely on a single component

- Simplicity
  - Easier to understand, leave less room for error, and are easier to review for faults
  - All members of the design and implementation team can understand the role and scope of each element of the design
  - Facilitates correct maintenance

SECURE SOFTWARE DESIGN ELEMENTS

There are 2 ways of constructing a software design:

- One way is to make it so simple that there are obviously no deficiencies
- The other way is to make it so complicated that there are no obvious deficiencies

DESIGN PRINCIPLES FOR SECURITY

- Least Privilege
  - A subject should be given only those privileges that it needs in order to complete its task
- Open Design
  - The security of a mechanism should not depend on the secrecy of its design or implementation
- Economy of Mechanism
  - Security mechanisms should be as simple as possible
- Complete Mediation
  - All accesses to objects must be checked to ensure that they are allowed

DESIGN PRINCIPLES FOR SECURITY

- Permission Based (Fail-Safe Defaults)
  - Unless a subject is given explicit access to an object, it should be denied access to that object
- Least Common Mechanism
  - Mechanisms used to access resources should not be shared
- Separation of Privilege
  - A system should not grant permission based on a single condition
- Ease of use (Psychological Acceptability)
  - Security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present

TOP 10 SECURE CODING PRACTICES

- Validate input
- Heed compiler warnings
- Architect and design for security policies
- Keep it simple
- Default to deny
- Adhere to the principle of least privilege
- Sanitize data sent to other systems
- Practice defense in depth
- Use effective quality assurance techniques
- Adopt a secure coding standard

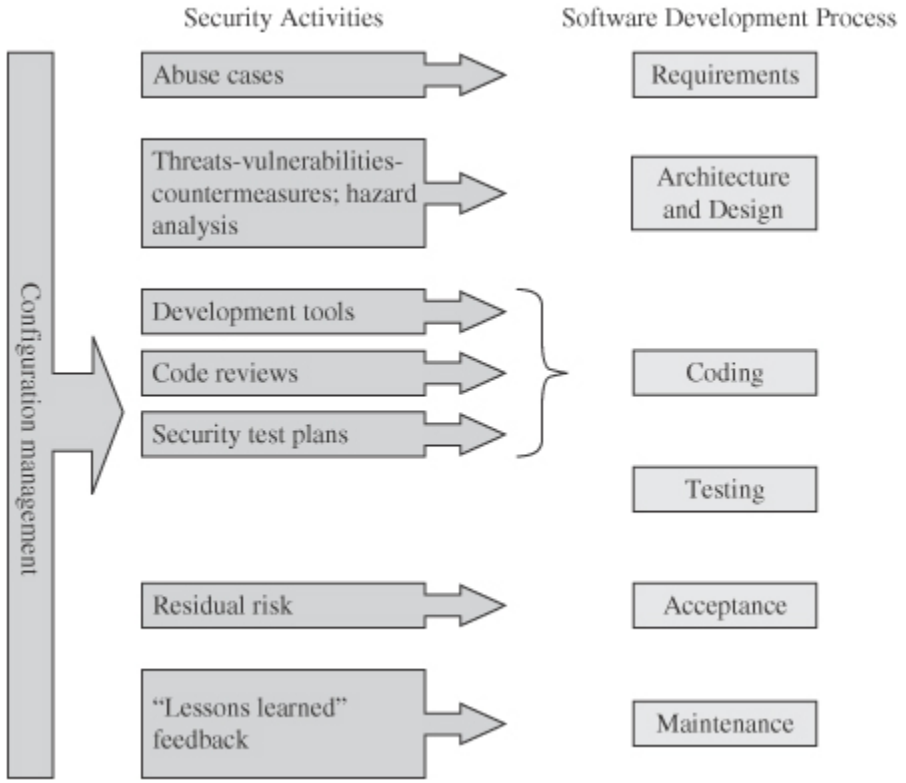
SECURE SOFTWARE DEVELOPMENT PROCESS

Software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product

- Development requires people who can do the following
  - Specify the system
  - Design the system
  - Implement the system
  - Test the system
  - Review the system at various stages
  - Document the system
  - Manage the system
  - Maintain the system

SECURE SOFTWARE DEVELOPMENT PROCESS

- Several key techniques for building "solid software"
  - Peer reviews
  - Hazard analysis
  - Good design
  - Prediction
  - Static analysis
  - Configuration management
  - Analysis of mistakes



SECURE SOFTWARE DEVELOPMENT PROCESS

- Peer Reviews

Sharing a product
with colleagues able to comment
about its correctness

- 3 types of peer reviews
  - Review
  - Walk-through
  - Inspection

SECURE SOFTWARE DEVELOPMENT PROCESS

- Peer Reviews
  - Keep careful track of what each reviewer discovers and how quickly he or she discovers it
  - Whether particular reviewers need training
    Whether certain kinds of faults are harder to find than others
  - Build a checklist of items to be sought in future reviews
  - A crafty programmer can conceal some of these flaws
    - The chance of discovery rises when competent programmers review the design and code, especially when the components are small and encapsulated

SECURE SOFTWARE DEVELOPMENT PROCESS

- Hazard Analysis

A set of systematic techniques intended to expose potentially hazardous system states

- Hazard analysis should begin when
  - You first start thinking about building a new system
  - Someone proposes a significant upgrade to an existing system
- Hazard analysis should continue throughout the system life cycle

SECURE SOFTWARE DEVELOPMENT PROCESS

- Hazard Analysis
  - A variety of techniques
    - Hazard and Operability Studies (HAZOP)
    - Failure Modes and Effects Analysis (FMEA)
    - Fault Tree Analysis (FTA)

	Known Cause	Unknown Cause
Known Effect	Description of system behavior	Deductive analysis FTA
Unknown Effect	Inductive analysis FMEA	Exploratory analysis HAZOP

SECURE SOFTWARE DEVELOPMENT PROCESS

- Good Design
  - Using a philosophy of fault tolerance
    - Isolating the damage caused by the fault and minimizing disruption to users
  - Capturing the design rationale and history
    - The reasons the system is built one way instead of another
  - Using design patterns
    - What designs work best in which situations

SECURE SOFTWARE DEVELOPMENT PROCESS

- Good Design
  - Having a consistent policy for handling failures

Retry

- Restore the system to its previous state
- Perform the service again, using a different strategy

Correct

- Restore the system to its previous state
- Correct some system characteristic
- Perform the service again, using the same strategy

Report

- Restore the system to its previous state
- Report the problem to an error-handling component
- Do not provide the service again

SECURE SOFTWARE DEVELOPMENT PROCESS

- Prediction

Predict the risks involved in building and using the system

- Risk prediction and management
- Decide which controls to use and how many
- What if our risk predictions are incorrect?
  - Defense in depth

SECURE SOFTWARE DEVELOPMENT PROCESS

- Static Analysis

Examine the characteristics of design and code

- Usually performed before peer review
- Several aspects of the design and code
  - Control flow structure
  - Data flow structure
  - Data structure

SECURE SOFTWARE DEVELOPMENT PROCESS

- Configuration Control
  - When we develop software, it is important to know who is making which changes to what and when
  - Reasons to change software
    - Corrective changes
      - Correct faults identified
    - Adaptive changes
      - Applied to the system after its initial fielding, to reflect changing user needs
    - Perfective changes
      - Perfect existing acceptable functions
    - Preventive changes
      - Address ways to avoid possible problems in the future

SECURE SOFTWARE DEVELOPMENT PROCESS

- Configuration Control

The process by which we control changes during development and maintenance

- 4 activities are involved in configuration management
  - Configuration identification
  - Change management
  - Configuration audit
  - Status accounting
- Performed by a configuration and change control board (CCB)
  - Representatives from all organizations with a vested interest in the system, perhaps including customers, users, and developers

SECURE SOFTWARE DEVELOPMENT PROCESS

- Configuration Control
  - Coordinating separate, related versions
    - Separate files
      - Different files for each release or version
    - Delta
      - The difference file which contains editing commands to describe the ways to transform the main version (particular version) into the variation
    - Conditional compilation
      - A single code component addresses all versions, relying on the compiler to determine which statements to apply to which versions

SECURE SOFTWARE DEVELOPMENT PROCESS

- Lessons from Mistakes
  - Document our decisions
    - What we decided to do and why
    - What we decided not to do and why
  - Information about the failures
    - How we found and fixed the underlying faults
  - Build checklists and codify guidelines

SECURE SOFTWARE DEVELOPMENT PROCESS

- Standards of documentation, language, and coding style

Layout of code on the page

Choices of names of variables

SECURE SOFTWARE DEVELOPMENT PROCESS

- Standards of Program Development

Security Audits

Check each project's compliance with standards and guidelines

SECURE SOFTWARE DEVELOPMENT PROCESS

- Process Standards
  - You have 2 friends

Sonya

Sonya is extremely well organized. She keeps lists of things to do, she always knows where to find a tool or who has a particular book, and everything is completed before its deadline.

Dorrie

Dorrie is a mess. She can never find her algebra book, her desk has so many piles of papers you cannot see the top, and she seems to deal with everything as a crisis because she tends to ignore things until the last minute.

  - Whom would you choose to organize and run a major project, a new product launch, or a multiple-author paper?

SECURE SOFTWARE DEVELOPMENT PROCESS

- Process Standards

Software built in an orderly manner
has a better chance of being good or secure than
software developed in a haphazard way

SECURE SOFTWARE DEVELOPMENT PROCESS

We should be skeptical of new technology's promise to make sweeping improvements

TESTING

- Goal: make the product failure free (eliminating the possibility of failure)
  - Realistically, testing will only reduce the likelihood or limit the impact of failures
- Find as many faults as possible and write up the findings
  - Developers can locate the causes and repair the problems if possible

TESTING

Developers grow trees

Testers manage forests

- Difficulty
  - Side effects, dependencies, unpredictable users, and flawed implementation bases (languages, compilers, infrastructure)
- We have to look for the hundreds of ways the program might go wrong

TESTING

- Types of Testing
  - Unit testing
  - Integration testing
  - Function test
  - Performance test
  - Acceptance test
  - Installation test
  - Regression testing
- 2 perspectives
  - Black-box testing
    - Testers cannot "see inside" the system
    - They apply particular inputs and verify that they get the expected output
  - Clear-box testing (white-box testing)
    - Testers can examine the design and code directly
    - Generate test cases based on the code's actual construction
- Key Ingredients
  - Product expertise
  - Coverage
  - Risk analysis
  - Domain expertise
  - Common vocabulary
  - Variation
  - Boundaries

EFFECTIVENESS OF TESTING

- Use different techniques to uncover different kinds of faults during development
  - It is not enough to rely on a single method for catching all problems

- Who does the testing?
  - From a security standpoint, independent testing is highly desirable
  - It may prevent a developer from attempting to hide something in a routine or keep a subsystem from controlling the tests that will be applied to it

LIMITATIONS OF TESTING

- Passing tests does not demonstrate the absence of problems
- Complete testing is complex and time consuming
- Testing only observable effects does not ensure any degree of completeness

LIMITATIONS OF TESTING

- Testing the internal structure of a product involves modifying the product by adding code to extract and display internal states
  - Affects the product's behavior
  - Can be a source of vulnerabilities
  - Can mask other vulnerabilities
- In testing real-time or complex systems, it is hard to reproduce and analyze problems reported
- Testing is almost always constrained by a project's budget and schedule

TESTING ESPECIALLY FOR SECURITY

- Penetration Testing

Ethical Hacking

Trying to crack
the system being tested

- Tiger team analysis
- As opposed to trying to break into the system for unethical reasons
- Resembles what an actual attacker might do

TESTING ESPECIALLY FOR SECURITY

Verification

Demonstrate formally the "correctness" of certain specific programs

- Verification checks the quality of the implementation

Validation

Assuring that the system developers have implemented all requirements

- Validation makes sure that the developer is building the right product according to the specification

DEFENSIVE PROGRAMMING

Offense sells tickets

Defense wins championships

- Defenders have to counter all possible attacks
  Attackers have only to find one weakness to exploit
- Program designers and implementers need not only write correct code but must also anticipate what could go wrong

DEFENSIVE PROGRAMMING

- Kinds of incorrect inputs
  - Value inappropriate for data type
    - Letters in a numeric field
    - M for a true / false item
  - Value unreasonable
    - 250 kilograms of salt in a recipe
  - Incorrect number of parameters
    - The system does not always protect a program from this fault
  - Value out of range for given use
    - A negative value for age
    - The date February 30
  - Value out of scale or proportion
    - A house description with 1 bedroom and 500 bathrooms
  - Incorrect order of parameters
    - A routine that expects age, sex, date, but the calling program provides sex, age, date

DEFENSIVE PROGRAMMING

Design by Contract

Documenting for each program module its preconditions, postconditions, and invariants

- Preconditions and postconditions are conditions necessary (expected, required, or enforced) to be true before the module begins and after it ends, respectively
- Invariants are conditions necessary to be true throughout the module's execution
- Each module comes with a contract
  - It expects the preconditions to have been met, and it agrees to meet the postconditions

REFERENCES

- Pfleeger, Charles P. and Shari Lawrence Pfleeger (2012), *Analyzing Computer Security*, 1st Edition, Prentice Hall.

NEXT WEEK: MALICIOUS CODE

- Threat
  - Malware – Virus, Trojan Horse, and Worm
- Technical Details
  - Malicious Code
- Vulnerability
  - Voluntary Introduction
  - Unlimited Privilege
  - Stealthy Behavior – Hard to Detect and Characterize
- Countermeasure
  - Hygiene
  - Detection Tools
  - Error Detecting and Error Correcting Codes
  - Memory Separation
  - Basic Security Principles

AS THE WORLD IS INCREASINGLY INTERCONNECTED, EVERYONE SHARES THE RESPONSIBILITY OF SECURING CYBERSPACE

Vision

To become an **outstanding** undergraduate Computer Science program that produces **international-minded** graduates who are **competent** in software engineering and have **entrepreneurial spirit** and **noble character**.

INFORMATIKA
UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA

Mission

1. To conduct studies with the best technology and curriculum, supported by professional lecturer
2. To conduct research in Informatics to promote science and technology
3. To deliver science-and-technology-based society services to implement science and technology