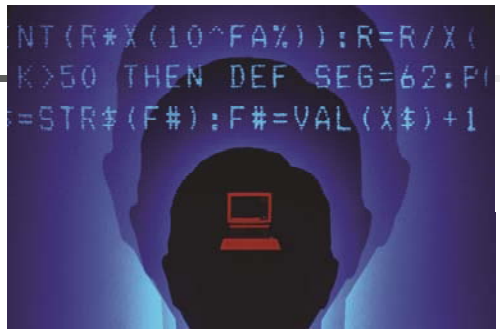
Secure Software Development



Chapter 18

Objectives

- Describe how secure coding can be incorporated into the software development process.
- List the major types of coding errors and their root cause.
- Describe good software development practices and explain how they impact application security.
- Describe how using a software development process enforces security inclusion in a project.

Compromises of Information

- Commonly caused by the exploitation of programming defects
  - Allows attackers to affect the confidentiality, integrity, and availability of information assets
- Security of information assets rests on the integrity and security of its software
  - Minor defects and flaws in programming lead to adverse consequences, such as the:
    - Exploitation of buffer overflows
    - Execution of remote procedure calls

Software Assurance as National Policy

- Improving software assurance practice is a national priority
  - Widespread problem that defects pose for protection of the critical information infrastructure
  - Software enables everything from our national defense and financial systems to the controls that regulate our pipelines and nuclear plants

What are the Aims of Software Assurance?

- Software assurance identifies and eliminates exploitable defects in the development, acquisition, and operation of software
- Software assurance guarantees that the products are:
  - Trustworthy
  - Predictable
  - Conformant
- Ensuring these qualities can be difficult because of the:
  - Complexity of modern computer systems
  - Increasing tendency for global outsourcing

Pair of Distinctions

- Vulnerabilities in software result from defects in specification, design, or programming
  - Introduced at three points in the process:
    - Design and development
    - Distribution
    - Updates and patches
- Software process
  - Development and sustainment process defined by a set of commonly agreed-on practices
  - Intended to build or enhance a software product

Software Process

- Software is a set of instructions that make the computer useful
  - Programmers write those instructions using a coding or programming language
  - Defects originate at different stages that include:
    - Failure to specify the product correctly
    - Incorrect design
    - Poor programming practice
    - Code-writing
    - Ineffective or inadequate review and testing

Software engineering

- Software engineering
  - The systematic development of software.
  - Universal requirement - the software works properly and performs the desired functions.
- Software engineering fits as many requirements as possible into a project management schedule timeline.
  - Analysts and developers work hard to get the functional elements correct.
  - However, the issue of non-functional requirements gets neglected entirely.

Software engineering

- Security in Software Engineering:
  - Traditionally, security is an add-on item incorporated after the functional requirements have been met.
  - Not viewed as an integral part of the software development lifecycle process.
- Security has been described as a non-functional requirement.
  - Usually placed into a category of secondary importance.
    - Now the trend is changing
- Trust is built upon an expectation that the software will work and continue to meet the requirements, and not change its behavior or functionality because of outside influences.

Software Engineering Process

- The challenge - Integration
  - There are two elements to achieve this objective.
    - First, the inclusion of security requirements and measures into the **specific process model** being used.
    - Second, the use of secure coding methods to prevent any possibility of security failures in the software being designed.

Software Engineering Process

- From requirements to system architecture to coding to testing, security is an imbedded property in all aspects of the process
- Several specific models have been developed to make the process of programming more effective and efficient.
- Some major models include:
  - The waterfall model
  - The spiral model
  - The evolutionary model
  - The agile model
  - The secure development lifecycle model (SDL)

Process Models

- Software Engineering Process Models
  - **The waterfall model.**
    - Characterized by a multistep process where the steps follow each other in a linear, one-way fashion, like water over a waterfall.
  - **The spiral model.**
    - Steps in phases that execute in a spiral fashion, repeating at different levels with each revolution of the model.
  - **The evolutionary model**
    - an iterative model designed to enable the construction of increasingly complex versions of a project.
  - **The agile model**
    - Iterative development, where requirements and solutions evolve through an ongoing collaboration between self-organizing cross-functional teams.

Process Models

- Independent of the method used, the process is about completing the requirements.
  - Opportunities exist independent of the model used to include
    - Security in the requirements process
    - Security awareness during architectural design, coding, and testing.

- **The secure development model (SDL)**
  - From a secure coding perspective, a secure development lifecycle (SDL) model is essential to success.
  - From requirements to system architecture to coding to testing, security is an embedded property in all aspects of the process.

Secure Development Lifecycle

- Firms have recognized the need for secure code.
- Security should be an issue that is addressed throughout the development process.
- The SDL accounts for security in each of its four major phases:
  - Requirements phase
  - Design phase
  - Coding phase
  - Testing phase

SDL Requirements Phase

- The requirements phase is the first step in a software development process model.
  - Define the specific requirements of the project.
  - The details for all end product requirements are documented
- Ensure the resultant software functions as desired.
- Items specifically regarding security should be enumerated during this step.
- Outcome of this phase is a document guiding security throughout the rest of the process.
- Adding security later tends to cost exponentially more than implementing it from the start.

SDL Requirements Phase

- Requirements should define specific security requirements if there is any expectation of them being designed into the project
- The **requirements process** is key to including security in software development.
  - Security-related items enumerated during the requirements process are visible throughout the rest of the software development process.

Security Considerations for Requirements Phase

- Analysis of security and privacy risk
- Authentication and password management
- Audit logging and analysis
- Authorization and role management
- Code integrity and validation testing
- Cryptography and key management
- Data validation and sanitization
- Network and data security
- Ongoing education and awareness
- Team staffing requirements
- Third-party component analysis

SDL Design Phase

- Coding without designing first is like building a house without using plans.
  - This might work fine on small projects, but as the scope grows, so do complexity and the opportunity for failure.
- Becomes more important as scope grows since complexity and chance of failure also grow.
- Design is a process involving trade-offs and choices,
  - The criteria used during the design decisions can have lasting impacts into program construction
- **Two secure coding principle are applied during the design phase:**
  - **Minimizing the attack surface area**
  - **Threat modeling**

Threat Modeling and Surface Area Minimization

- Attack surface minimization
  - A strategy to reduce the place where code can be attacked.
- Threat modeling
  - The process of analyzing threats and their effects on software in a granular fashion.
  - A communication tool designed to communicate to everyone on the development team the threats and dangers facing the code.
  - The output of the threat model process is a compilation of threats and how they interact with the software.
  - This information is communicated across the design and coding team, so that potential weaknesses can be mitigated before the software is released.

Threat Modeling Steps

1. Define scope
   1. Communicate what is in scope and out of scope with respect to the threat modeling effort. This includes both attacks and software components.
2. Enumerate assets
   1. List all of the component parts of the software being examined
3. Decompose assets
   1. Break apart the software into small subsystems composed of inputs and outputs. This is to simplify data flow analysis and to capture internal entry points.
4. Enumerate threats
   1. List all the threats to the software.

Important

Threat Modeling Steps

5. Classify threats
    1. Classify the threats by their mode of operation
6. Associate threats to assets
    1. Connect specific threats and modes to specific software subsystems
7. Score and rank threats
    5. Score each specific threat–asset pair and then rank them from most dangerous to least dangerous.
8. Create threat trees
    5. Create a graphical representation of the required elements for an attack vector
9. Determine and score mitigation
    5. Score the mitigation efforts associated with each attack vector

Important

Microsoft Threat Modeling Tool

http://www.microsoft.com/security/sdl/getstarted/threatmodeling.aspx

SDL Coding Phase

- Phase where the design is implemented.
- Software is checked for vulnerabilities using enumerations of known software vulnerabilities:
  - Common Weakness Enumeration (CWE)
  - Common Vulnerabilities and Exposures (CVE)
- Manual review is also used to reduce vulnerabilities.
- Static code analysis tools may be used to search software code for possible errors.

Major Programming Errors

- SANS & MITRE maintain a list of the 25 most dangerous programming errors in three categories:
  - Insecure interaction between components
  - Risky resource management
  - Porous defenses

  http://cwe.mitre.org/top25

- Common problems with erroneous code include:
  - **Buffer overflows**
  - **Improper input handling**
  - **Improper output handling**
  - **Least privilege problems**
  - **Injection vulnerabilities**
  - **Cryptographic failures**
  - **Language specific failures**

Buffer Overflows

- These vulnerabilities are relatively simple. The buffer used to hold program input is overwritten with data larger than the buffer.
- The root cause of this vulnerability is a mixture of two things:
  - Poor programming practice
  - Programming language weaknesses

Nearly half of all software exploits stem from buffer overflow.

CERT/CC at Carnegie Mellon University

Buffer Overflows

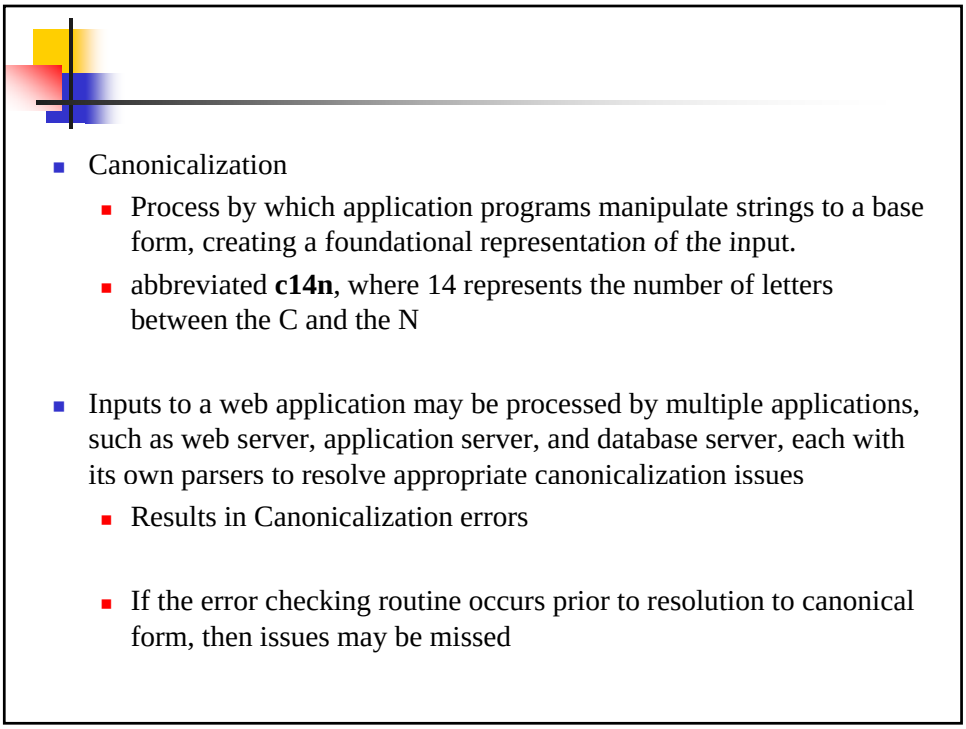
- Nearly half of all exploits of computer programs stem historically from some form of buffer overflow.
- The generic classification of buffer overflows includes many variants:
  - Static buffer overruns
  - Indexing errors
  - Format string bugs
  - Unicode and ANSI buffer size mismatches
  - Heap overruns

Countering Buffer Overflows

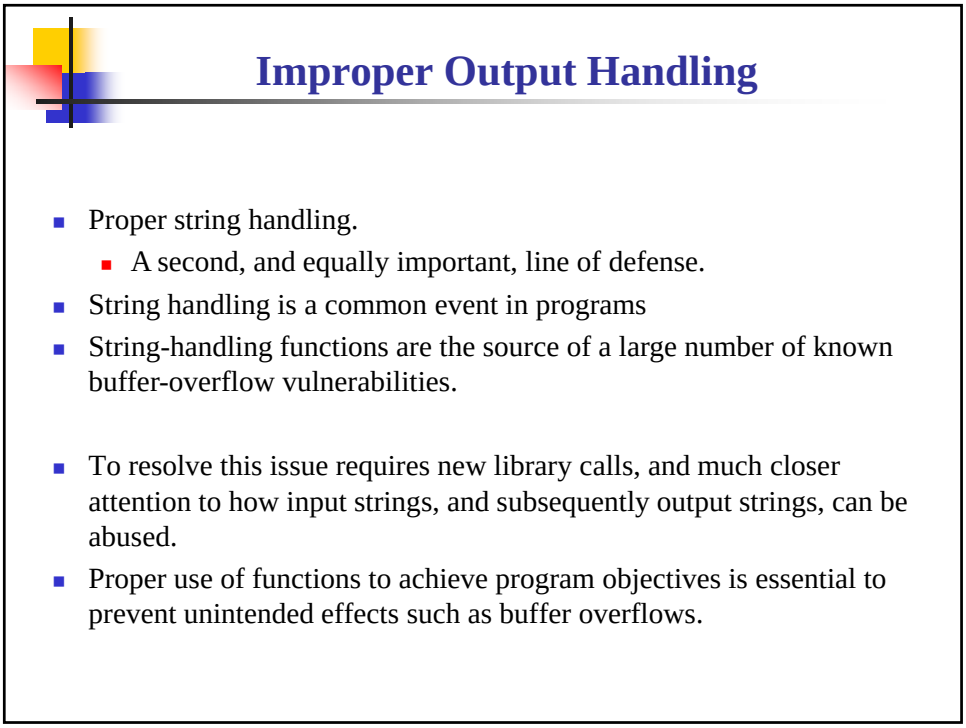
- Step 1 - Write solid code.
  - Regardless of the language used or source of input, treat all input from outside a function as hostile.
  - Validate all inputs as if they were hostile or were an attempt to force a buffer overflow.
- Step 2 - Proper string handling.
  - Strings are a common form of input
  - Because many string-handling functions have no built-in checks for string length, strings are frequently the source of exploitable buffer overflows
  - String handling is common in programs and is the source of a large number of known buffer overflow vulnerabilities.

Improper Input Handling

- Users have the ability to manipulate inputs
- It is up to the programmer to appropriately handle the input to prevent malicious entries from having an effect.
- In today's computing environment, a wide range of character sets is used.
  - Unicode allows multi-language support.
  - Character codesets allow multi-language capability.
  - Various encoding schemes, such as hex encoding are supported to allow diverse inputs.
- The net result of all these input methods is that there are numerous ways to create the same input to a program.
  - **How to deal- Canonicalization**

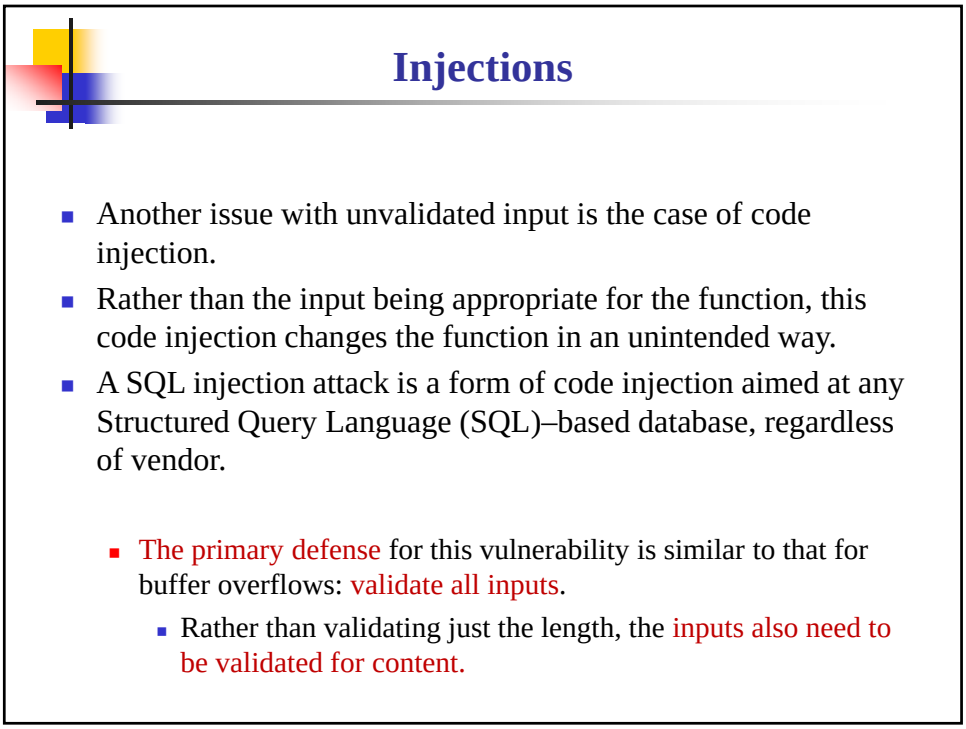


- Canonicalization
  - Process by which application programs manipulate strings to a base form, creating a foundational representation of the input.
  - abbreviated **c14n**, where 14 represents the number of letters between the C and the N
- Inputs to a web application may be processed by multiple applications, such as web server, application server, and database server, each with its own parsers to resolve appropriate canonicalization issues
  - Results in Canonicalization errors
  - If the error checking routine occurs prior to resolution to canonical form, then issues may be missed



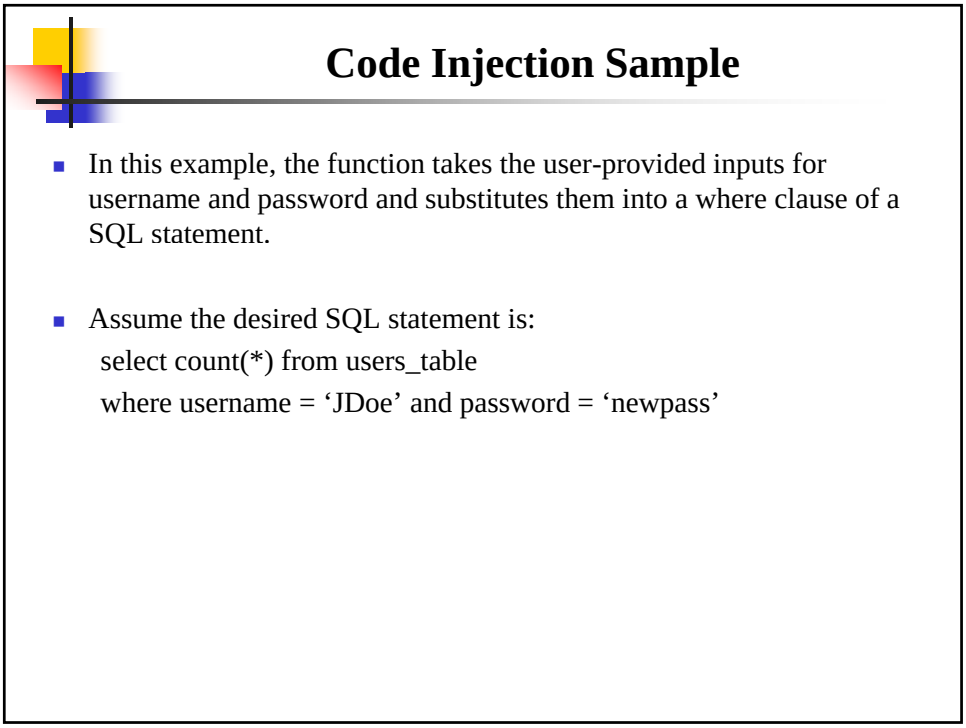
Improper Output Handling

- Proper string handling.
  - A second, and equally important, line of defense.
- String handling is a common event in programs
- String-handling functions are the source of a large number of known buffer-overflow vulnerabilities.
- To resolve this issue requires new library calls, and much closer attention to how input strings, and subsequently output strings, can be abused.
- Proper use of functions to achieve program objectives is essential to prevent unintended effects such as buffer overflows.



Injections

- Another issue with unvalidated input is the case of code injection.
- Rather than the input being appropriate for the function, this code injection changes the function in an unintended way.
- A SQL injection attack is a form of code injection aimed at any Structured Query Language (SQL)–based database, regardless of vendor.
  - The primary defense for this vulnerability is similar to that for buffer overflows: validate all inputs.
    - Rather than validating just the length, the inputs also need to be validated for content.



Code Injection Sample

- In this example, the function takes the user-provided inputs for username and password and substitutes them into a where clause of a SQL statement.
- Assume the desired SQL statement is:

  select count(*) from users_table

  where username = ‘JDoe’ and password = ‘newpass’

Code Injection Sample

- The values JDoe and newpass are provided by the user and simply inserted into the string sequence.

  select count(*)

  from users_table

  where username = ‘JDoe’ and password = ‘newpass’

- Though this seems functionally safe, it can be easily corrupted by using the sequence:
  - anything’ or ‘x’=‘x
- Since this changes the where clause to one that returns all records:

  select count(*)

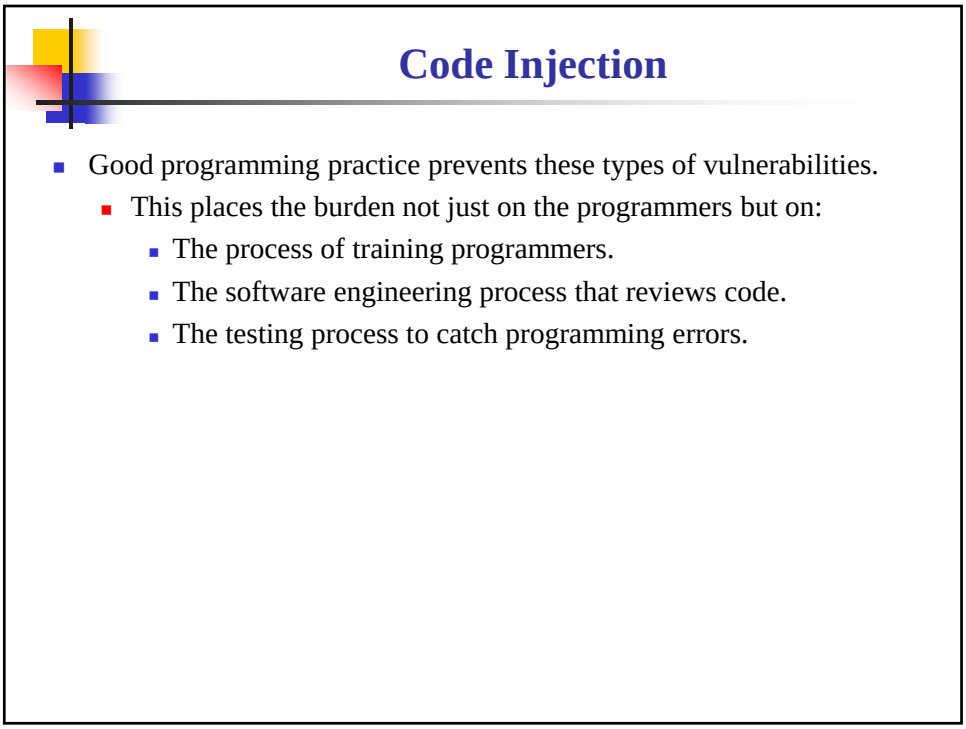
  from users_table

  where username = ‘JDoe’ and password = ‘anything’ or ‘x’=‘x’But

  Unlike the "real" query, which should return only a single item each time, this version will essentially return every item in the members database

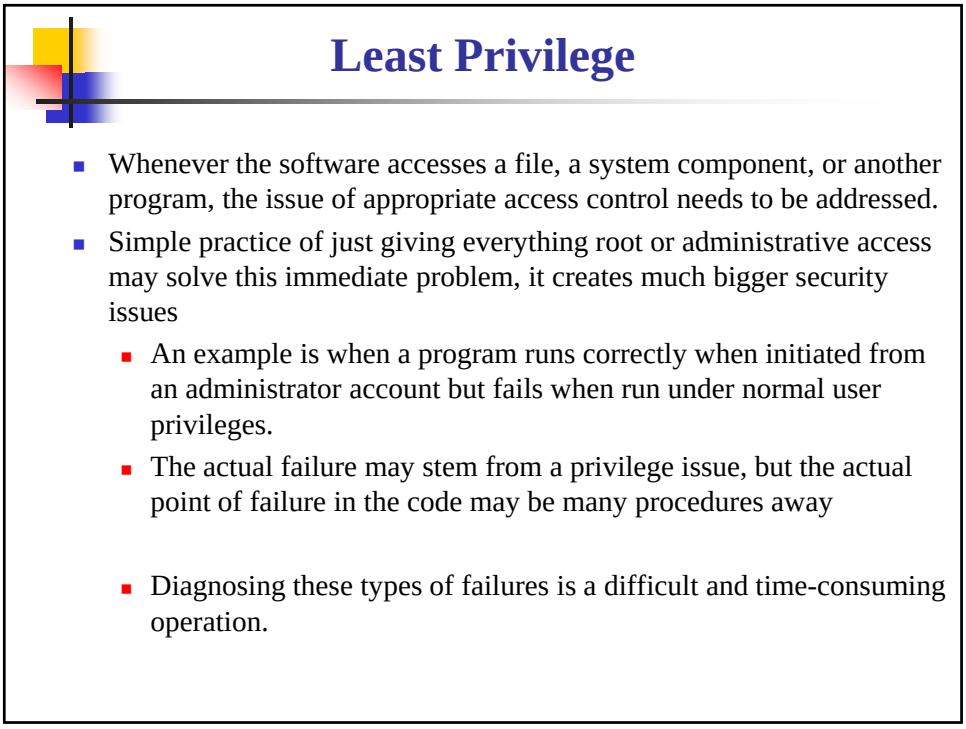
Testing for SQL Injection Vulnerability

- There are two main steps associated with testing for SQL injection vulnerability.
  - The first step is to confirm that the system is at all vulnerable.
    - This can be done using various inputs to test whether an input variable can be used to manipulate the SQL command. The following are common test vectors used:
      - ' or 1=1—
      - " or 1=1—
  - The second step is to use the error message information to attempt to perform an actual exploit against the database.



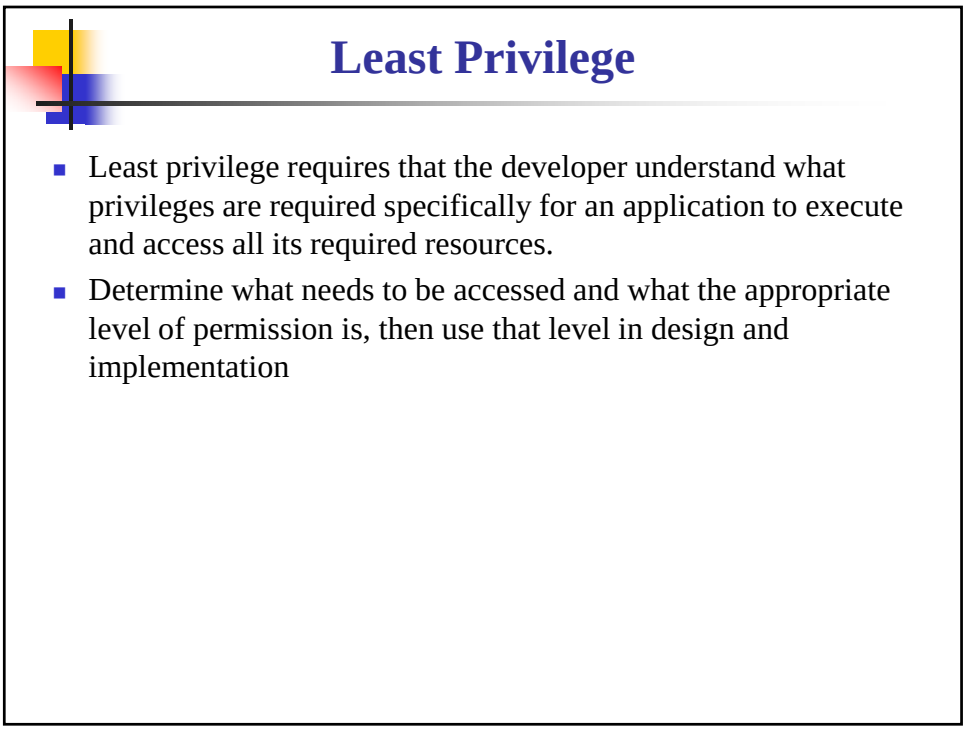
Code Injection

- Good programming practice prevents these types of vulnerabilities.
  - This places the burden not just on the programmers but on:
    - The process of training programmers.
    - The software engineering process that reviews code.
    - The testing process to catch programming errors.



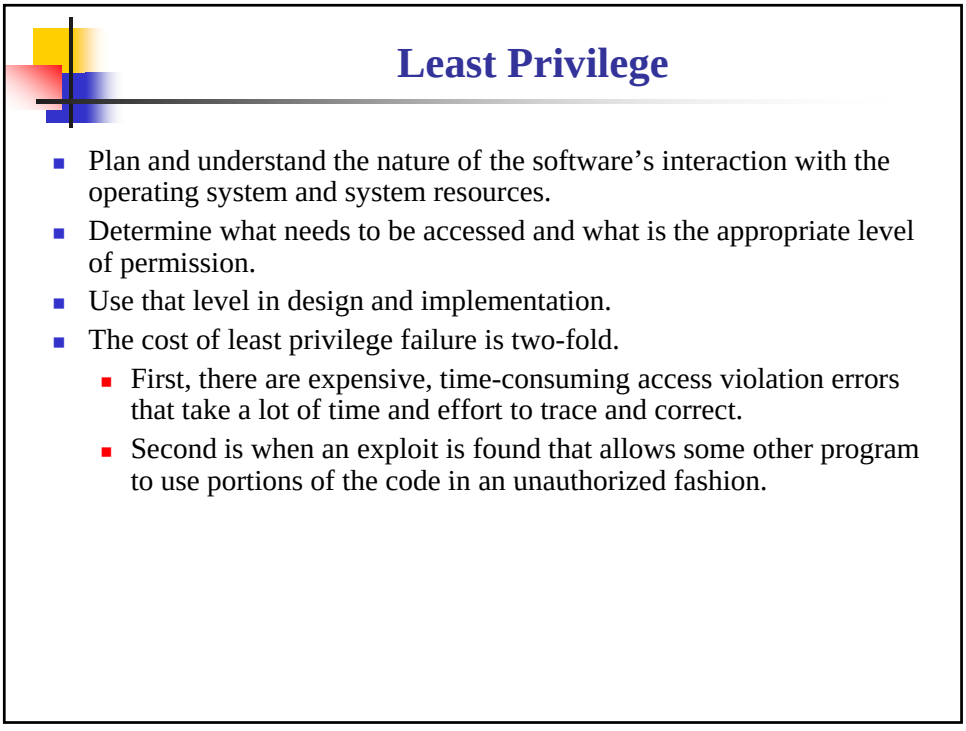
Least Privilege

- Whenever the software accesses a file, a system component, or another program, the issue of appropriate access control needs to be addressed.
- Simple practice of just giving everything root or administrative access may solve this immediate problem, it creates much bigger security issues
  - An example is when a program runs correctly when initiated from an administrator account but fails when run under normal user privileges.
  - The actual failure may stem from a privilege issue, but the actual point of failure in the code may be many procedures away
  - Diagnosing these types of failures is a difficult and time-consuming operation.



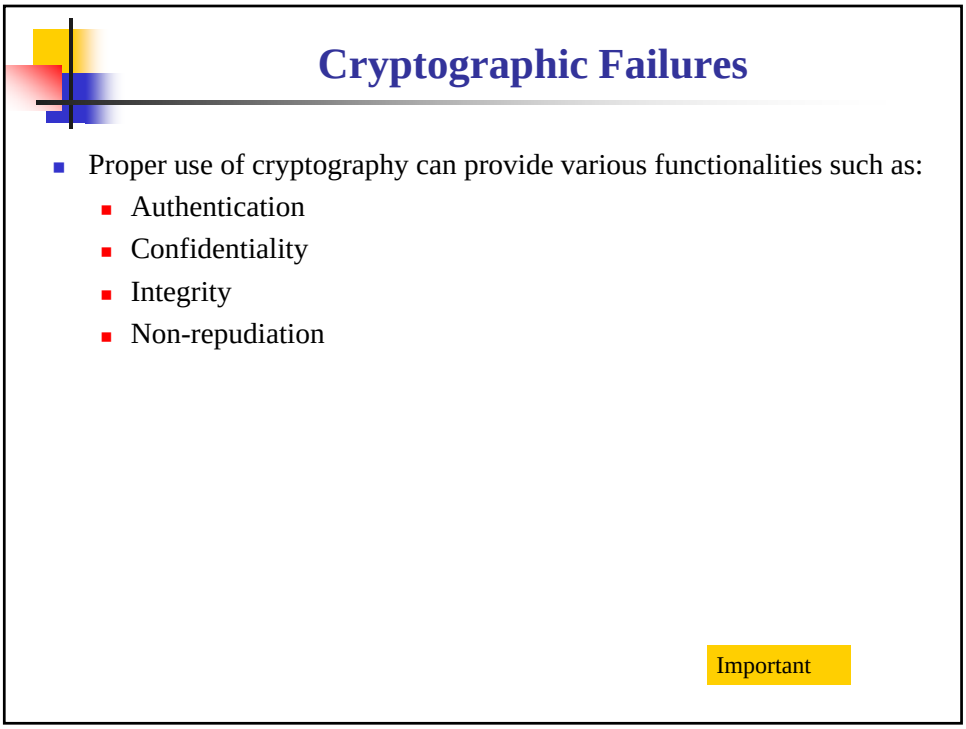
Least Privilege

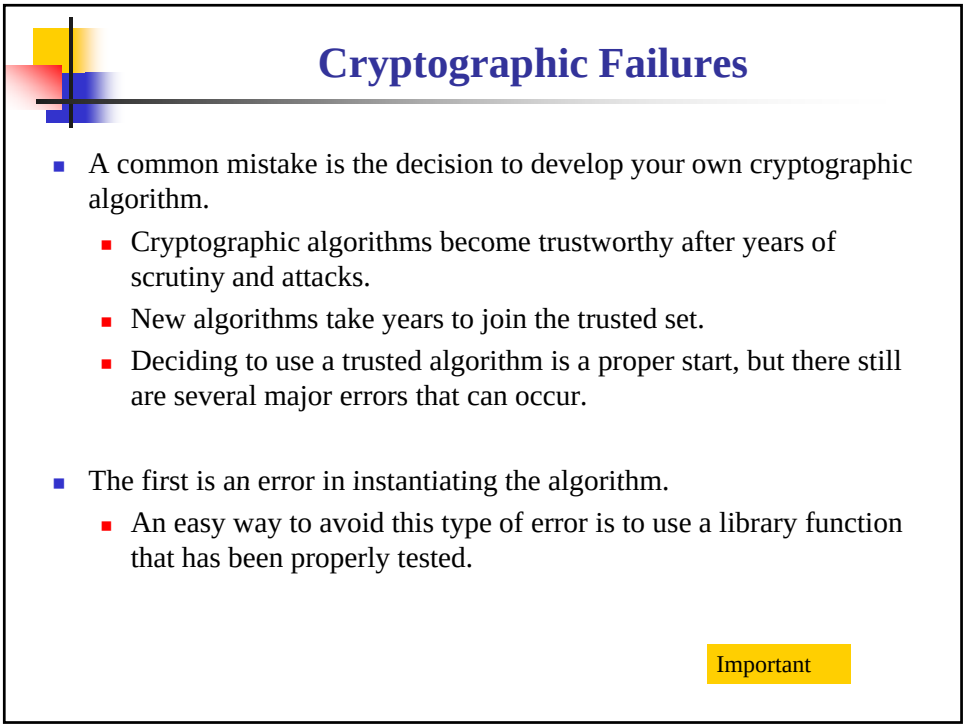
- Least privilege requires that the developer understand what privileges are required specifically for an application to execute and access all its required resources.
- Determine what needs to be accessed and what the appropriate level of permission is, then use that level in design and implementation



Least Privilege

- Plan and understand the nature of the software's interaction with the operating system and system resources.
- Determine what needs to be accessed and what is the appropriate level of permission.
- Use that level in design and implementation.
- The cost of least privilege failure is two-fold.
  - First, there are expensive, time-consuming access violation errors that take a lot of time and effort to trace and correct.
  - Second is when an exploit is found that allows some other program to use portions of the code in an unauthorized fashion.


Cryptographic Failures
Proper use of cryptography can provide various functionalities such as:
Authentication
Confidentiality
Integrity
Non-repudiation
Important


Cryptographic Failures
A common mistake is the decision to develop your own cryptographic algorithm.
Cryptographic algorithms become trustworthy after years of scrutiny and attacks.
New algorithms take years to join the trusted set.
Deciding to use a trusted algorithm is a proper start, but there still are several major errors that can occur.
The first is an error in instantiating the algorithm.
An easy way to avoid this type of error is to use a library function that has been properly tested.
Important

Cryptographic Failures

- Randomness:
  - Once you have an algorithm, and have chosen a particular instantiation, you need a random number to generate a random key since cryptographic functions use an algorithm and a key, the later being a digital number.
  - There are random functions built into the libraries of most programming languages.
    - These are pseudorandom number generators.
    - Although the distribution of output numbers appears random, it produces a reproducible sequence.
  - Using a cryptographic random number generator resolves this problem.

Important

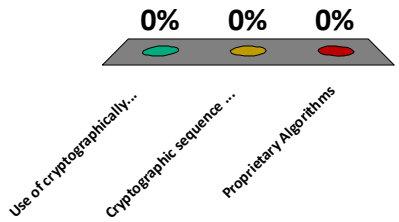
Use Only Approved Cryptographic Functions

- Always use vetted and approved libraries for all cryptographic work.
- Never create your own cryptographic functions, even when using known algorithms.
- The generation of a real random number is not a trivial task.

Important

Common cryptographic failures include which of the following

1. Use of cryptographically random numbers
2. Cryptographic sequence failures
3. Proprietary Algorithms



Cryptographic Failures

- Storing keys:
  - Storing private keys in areas where they can be recovered by an unauthorized person is the next source of potential failure.
  - Tools have been developed that can search code for 'random' keys and extract the key from the code or running process.
    - The bottom line– do not hard code secret keys in the code, as then they can be discovered.
  - Keys should be generated, and then passed by reference, minimizing the transfer of copies across a network or application.
  - Storing them in memory in a non-contiguous fashion is also important to prevent external detection and, again, trusted cryptographic library functions come to the rescue.

Important

Language-Specific Failures

- Modern programming languages are built around libraries that permit reuse and speed the development process.
- The development of many library calls and functions was done without regard to secure coding implications.
- Developing and maintaining a series of deprecated functions and prohibiting their use in new code, while removing them from old code when possible, is a proven path toward more secure code.

Important

Microsoft Recommended Deprecated C Functions

- Function families to deprecate/remove:
  - strcpy() and strncpy()
  - strcat() and strncat()
  - scanf()
  - sprint()
  - gets()
  - memcpy(), CopyMemory(), and RtlCopyMemory()
- Banned functions are easily handled via automated code reviews during the check-in process.

SDL Testing Phase

- The Testing phase
  - Last opportunity to determine that software performs properly before the end user experiences problems.
  - Testing can occur at each level of development, module, subsystem, system, and complete application.
    - Should be done as early as possible
  - The sooner errors are discovered and corrected, the lower the cost and the impact to project schedules.

Important

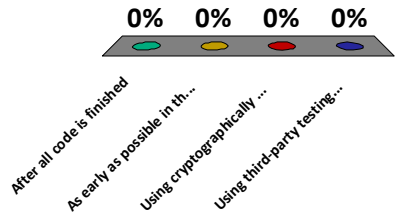
Testing

- The use of use cases to compare program responses to known inputs and comparison of the output to the desired output is a time-proven method of testing software.
  - The design of use cases to test specific functional requirements occurs based on the requirements determined in the requirements phase.
  - Providing additional security related to use cases is the process-driven way to ensure that the security specifics are also tested.
- Fuzzing often used to find errors in this phase.
  - Refers to a method used to test software that automates numerous input sequences to uncover possible exploits
- Other automated code-checking tools may be run in this phase to find errors.

Important

When is testing best accomplished

1. After all code is finished
2. As early as possible in the process
3. Using cryptographically random elements
4. Using third-party testing software



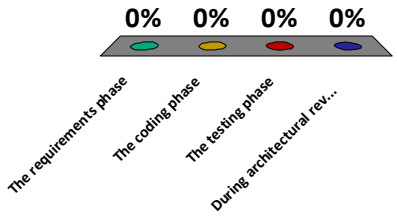
Good Practices

- A software development process that has security planning built-in will make a difference in the end result.
  - The process begins with requirements and ends with testing.
- Enumerating and defining the specific security requirements and how they are tested is a key element in building security into code.
- Making a 'code review' requirement, where a second programmer is walked through the functionality of code before release to testing can catch many errors.
- Security requirements are often included at the end in a project.
  - Putting the security requirements in the requirement phase and having the corporate backing to maintain an acceptable level of security functionality as a baseline solves many problems

Important

Security requirements are best defined in:

1. The requirements phase
2. The coding phase
3. The testing phase
4. During architectural review of the system



A Common Criteria: Basing Security Functionality on a Protection Profile

- Commonality of purpose makes it possible to use standard profiles of the functions
  - It serves as a consistent and reliable reference point for necessary behaviors to assure software
  - It is reusable because it contains the common policies, assumptions, and requirements
  - It facilitates and directs the tailoring of software and environmental security functions for a secure system

Common Criteria: Form of the Standard

- ISO 15408 standard, the "Common Criteria"
  - Useful model for developing a protection profile
  - Three parts to the standard include:
    - Part 1: Introduction and general model
    - Part 2: Security functional requirements
    - Part 3: Security assurance requirements

Common Criteria: Form of the Standard

- Common Criteria
  - Provides the security advice needed to address most ordinary threats
  - Provides advice about commonly accepted functions used to create trusted systems
  - Enumerates known software security attributes confirmable through direct observation
  - Provides an encyclopedic collection of standardized adaptable security properties
  - Supports the evaluation of software products by listing attributes to benchmark security behaviors
  - Because of its focus on observation - is understandable and easier to implement