بسم الله الرحمن الرحيم

KHAWAJAGAN Technologies

SP 2025

Secure Software Design and Development

CYC386

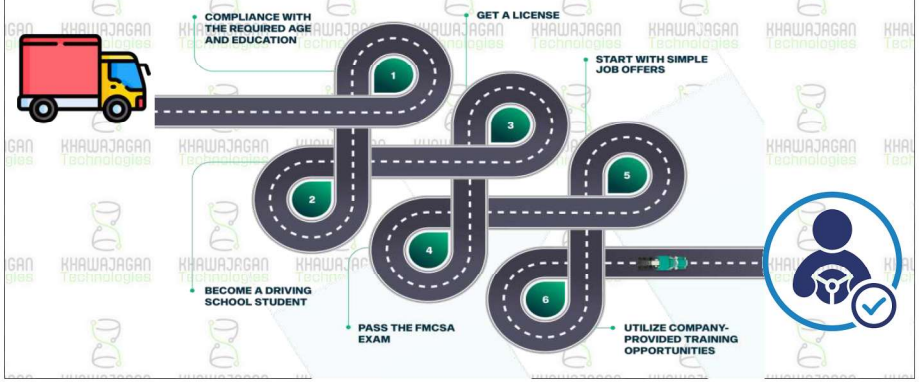
LECTURE 004 – 07-MAR-2025

FAISAL SHAHZAD

SP 2025

Defensive Programming

Defensive Programming



Defensive Programming

- Programming is not that much different from getting a driving license.
- Programs today are under regular attack if they happen to have any Internet access.
- Programs cannot defend themselves unless they're taught how to from the very first line of code typed.
- Role-based education in defensive programming techniques, is the avenue to reliably gaining those skills, and its importance cannot be overemphasized!

Defensive Programming

- Programmers today have an awesome responsibility to get it right the first time, because their work could adversely affect life and limb.
- Not that long ago, before software was used to control millions of kinetic machines and devices on our streets, in our homes, and in our bodies, when an application crashed or its control was lost, a restart usually fixed the issue.

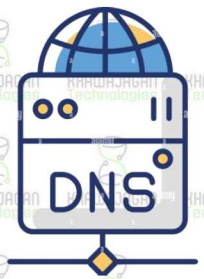
- Today when software crashes, it could indeed lead to a real crash that kills real people.

Defensive Programming

- Ethics also play an important role in today's software development world.
- In the case of autonomous vehicles, how should a program work in the face of an imminent threat that's unavoidable, protect the passenger in the vehicle, or minimize the damage outside the vehicle?
- Choices made in software dictate this type of behavior.
- Will consumers and users of self-driving cars ever know what those choices are before it's too late?
- With that sense of awesome responsibility in mind, guidance and paradigms for secure programming practices that improve software quality while enhancing its resilience features.

The Evolution Of Attacks

- Attack techniques evolve over time.
- Secure design and programming best practices have always been the best form of defense against these attacks—some of them have proven themselves capable of later defending attacks that had not yet been identified.
- Dan Kaminsky discovered in the summer of 2008 a serious vulnerability in the domain name servers (DNS) that could allow attackers to redirect clients to alternate servers of their own choosing, leading to potential misuse.



The Evolution Of Attacks

- Years earlier, a cryptographer named Daniel J. Bernstein looked at DNS security and decided that source port randomization was a smart design choice by the designers of DNS.
- The work-around that was rolled out following Kaminsky's discovery used the source port randomization feature to counter the problem.
- Bernstein did not know about Kaminsky's attack, but he understood and envisioned a general class of attacks and realized that this enhancement could protect against them.
- Consequently, the DNS program he wrote in 2000, djbdns, did not need any patching—it was already immune to Kaminsky's attack.

The Evolution Of Attacks

- This is what a good design looks like—it is not just secure against known attacks; it is also secure against unknown or zero-day attacks.
- Even with good designs, there are multiple ways a developer who implements the design can introduce vulnerabilities into the process, thus making the entire system insecure.
- This lecture explains some of the common vulnerabilities (with a focus on custom web applications that are the key targets of today's attacks) and provides some best practices and secure coding concepts to protect against them.

SP 2025

Threat and Vulnerability Taxonomies

MITRE's CWE

- Common Weakness Enumeration (CWE) is a formal list or dictionary of common software weaknesses that can occur in software's architecture, design, code, or implementation and can lead to exploitable security vulnerabilities.
- CWE is intended for both the development community and the security practitioners.
- CWE was created to serve as
  - A common language to describe software security weaknesses
  - A standard measuring stick for software security tools targeting these weaknesses;
  - A provider of a common baseline standard for weakness identification, mitigation, and prevention efforts.
- Software weaknesses are flaws, faults, bugs, vulnerabilities, and other errors in software implementation, code, design, or architecture that, if left unaddressed, could result in systems and networks being vulnerable to attack.

MITRE's CWE

- Some example software weaknesses include:
  - Buffer overflows
  - Format strings
  - Structure and validity problems
  - Common special element manipulations
  - Channel and path errors
  - Handler errors
  - User interface errors
  - Pathname traversal and equivalence errors
  - Authentication errors
  - Resource management errors
  - Insufficient verification of data
  - Code evaluation and injection
  - Randomness and predictability

OWASP Top 10

- The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted.
- "Community" includes corporations, educational organizations, and individuals from around the world with the focus on creating freely available articles, open methodologies, documentation, tools, and technologies to improve web software security.
- The OWASP Top 10 is a list of the 10 most severe web security issues as defined and regularly updated by the OWASP community:
  - Addresses issues with applications on the perimeter of an organization and accessed by external parties
  - Widely accepted and referenced as mandatory by:
    - U.S. Federal Trade Commission
    - U.S. Defense Information Systems Agency (U.S. Department of Defense)
    - Payment Card Industry Data Security Standard (PCI-DSS)

OWASP Top 10

- The OWASP Top 10 is a powerful awareness document for web application security with followingTop 10 Most Critical Web Application Security Risks:
  - Injection
  - Broken Authentication
  - Sensitive Data Exposure
  - XML External Entities (XXE)
  - Broken Access Control
  - Security Misconfiguration
  - Cross-Site Scripting (XSS)
  - Insecure Deserialization
  - Using Components with Known Vulnerabilities
  - Insufficient Logging & Monitoring

OWASP Top 10 - Injection

- Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query.
- The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

OWASP Top 10–Broken Authentication

- Application functions related to authentication and session management are often implemented incorrectly
- This allow attackers to compromise passwords, keys, or session tokens or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

OWASP T10–Sensitive Data Exposure

- Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII.
- Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes.
- Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

OWASP T10–XML External Entities

- Many older or poorly configured XML processors evaluate external entity references within XML documents.
- External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

OWASP T10–Broken Access Control

- Restrictions on what authenticated users are allowed to do are often not properly enforced.
- Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

OWASPT10–Security Misconfiguration

- Security misconfiguration is the most commonly seen issue.
- This is typically a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information.
- Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

OWASP T10–Cross Site Scripting XSS

- XSS flaws occur whenever an application either includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create HTML or JavaScript.
- XSS allows attackers to execute scripts in the victim's browser, which can hijack user sessions, deface websites, or redirect the user to malicious sites.

OWASP T10–Insecure Deserialization

- Insecure deserialization often leads to remote code execution.
- Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

OWASP T10–Known Vulnerabilities

- Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application.
- If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover.
- Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts

OWASPT10–Insufficient Logging

- Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data.

- Most breach studies show that time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

SP 2025

Input Validation / Handling

Input Validation / Handling

- Improper input handling is one of the most common weaknesses identified across applications today.
- Poorly handled input is a leading cause of critical vulnerabilities that exist in systems and applications.
- When attacking an application, a malicious user will attempt to supply unexpected data through its input parameters in order to cause an error condition or get their code to run.
- For this reason, every input data element received by the application should be validated before being used.

Input Validation / Handling

- A typical example of input that should be validated is the data the user supplies on a form's text fields.
- However, you should validate any input coming from the user's browser, because the attacker could supply malicious instructions from anywhere as input to gain unauthorized access to the application or its data.
- Your application will not know if the messages are coming from a browser or are being edited by a browser proxy.
- For this reason, any validation that's done on the client (browser) is easily circumvented, so it's essential that ALL official validation be performed on the server.

Input Validation / Handling

- Typical developers underestimate what really constitutes input to an application. There is usually much more than what the developer expects.
- For a web application, input could be any of the following:
  - All HTML form fields
  - Included form fields that do not seem to be modifiable by the user:
    - Checkboxes
    - Radio buttons
    - Select lists
    - Hidden fields
  - All links that contain hard-coded query string parameters
  - All cookies used by an application
  - HTTP response headers

Input Validation / Handling

- In general, the term input handing is used to describe functions such as validation, cleansing, sanitizing, filtering, encoding and/or decoding of input data.
- Applications receive input from various sources, including human users, software agents (browsers), files, and network/peripheral devices, to name a few.
- In the case of web applications, input can be transferred in various formats (name value pairs, JavaScript Object Notation [JSON], Simple Object Access Protocol [SOAP], web services, etc.) and obtained via URL query strings, POST data, HTTP headers, cookies, etc. .

Input Validation / Handling

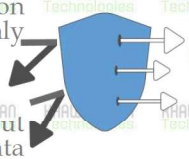
- We can obtain non-web application input via application variables, environment variables, the registry, configuration files, etc.
- Regardless of the data format or source/location of the input, all input from outside the application's security perimeter or trust boundary should be considered explicitly untrusted and potentially malicious.
- Applications that process untrusted input may become vulnerable to attacks such as buffer overflows, SQL injection, OS commanding, and denial of service, just to name a few.

Input Validation / Handling

- One of the key aspects of input handling is validating that the input satisfies a certain criterion.
- For proper validation, it is important to identify the form and type of data that is acceptable and expected by the application.
- Defining an expected format and usage of each instance of untrusted input is required to accurately define restrictions.
- Validation can include checks for variable-type safety (e.g., integer, floating point, text) and syntax correctness.

Input Validation / Handling

- String input should be checked for length (minimum and maximum number of characters) and "character set" validation, by which numeric input types such as integers and decimals can be validated against acceptable upper and lower bounds of values.
- When combining input from multiple sources, validation should be performed on the concatenated result and not only against the individual data elements alone.
- This practice helps avoid situations in which input validation may succeed when performed on individual data items but fail when done on a concatenated string from all the sources.



Client-Side vs. Server-Side Validation

- Developers make mistake to include validation routines in the client-side of an application using JavaScript functions as a sole means of performing bounds checking.
- Validation routines that are beneficial on the client side cannot be relied upon to provide a security control, because all data accessible on the client side is modifiable by a malicious user or attacker using proxy tool.
- This is true of any client-side validation checks in JavaScript and VBScript or external browser plug-ins such as Flash, Java, or ActiveX.
- The HTML Version 5 specification has added a new attribute "pattern" to the INPUT tag that enables developers to write regular expressions as part of the markup for performing validation checks.

Client-Side vs. Server-Side Validation

- Although this feature makes it even more convenient for developers to perform input validation on the client side without having to write any extra code, the risk from such a feature becomes significant when developers use it as the only means of performing input validation for their applications.
- Relying on client-side validation alone is bad practice. Although client-side validation is great for user interface (UI) and functional validation, it is not a substitute for server-side security validation.
- Performing validation on the server side is the ONLY way to assure the integrity of your validation controls.
- In addition, server-side validation routines will always be effective, regardless of the state of JavaScript execution on the browser.

Input Sanitization

- Sanitizing input can be performed by transforming input from its original form to an acceptable form via encoding or decoding.
- Common encoding methods used in web applications include HTML entity encoding and URL encoding schemes.
- HTML entity encoding serves the need for encoding literal representations of certain meta-characters to their corresponding character entity references.
- Character references for HTML entities are pre-defined and have the format "&name", where "name" is a case sensitive alphanumeric string.

Input Sanitization

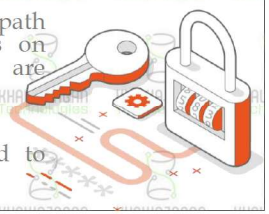
- A common example of HTML entity encoding is where "<" is encoded as < and ">" is encoded as >.
- URL encoding applies to parameters and their associated values that are transmitted as part of HTTP query strings.
- Likewise, characters that are not permitted in URLs are represented using their Unicode character set code point value, where each byte is encoded in hexadecimal as "%HH". For example, "<" is URL-encoded as "%3C" and ">" is URL-encoded as "%CE".

Input Sanitization

- There are multiple ways that input can be presented to an application.
- With web applications and browsers supporting multiple character encoding types, it has become commonplace for attackers to try to exploit inherent weaknesses in encoding and decoding routines.
- Applications requiring internationalization are a good candidate for input sanitization.
- One of the common forms of representing international characters is Unicode. Unicode transformations use the Universal Character Set (UCS), which consists of a large set of characters to cover symbols of almost all the languages in the world.
- Developer, rarely write routines that inspect every character within a Unicode string to confirm its validity. Such misrepresentation of characters enables attackers to spoof expected values by replacing them with visually or semantically similar characters from the UCS.

Canonicalization

- Canonicalization is another important aspect of input sanitization.
- Canonicalization deals with converting data with various possible representations into a standard "canonical" representation deemed acceptable by the application.
- Commonly known applications of canonicalization is "path canonicalization," in which file and directory paths on computer file systems or web servers (URL) are canonicalized to enforce access restrictions.
- Failure of such canonicalization mechanism can lead to directory traversal or path traversal attacks.



Canonicalization

- The concept of canonicalization is widely applicable and applies equally well to Unicode and XML processing routines.
- While expecting UTF-8 encoded characters, the application fails to sanitize and transform input supplied on the form in UTF-7 coding, leading to a cross-site scripting attack.
- Applications that are internationalized need to support multiple languages that cannot be represented using common ISO-8859-1 (Latin-1) character encoding.
- Languages such as Chinese and Japanese use thousands of characters and are therefore represented using variable-width encoding schemes. Improperly handled mapping and encoding of such international characters can also lead to canonicalization attacks.
- Based on input and output handling requirements, applications should identify acceptable character sets and implement custom sanitization routines to process and transform data specific to their needs.

SP 2025

Common Examples of Attacks Due to Improper Input Handling

Buffer Overflow

- A buffer overflow may occur when the length of the source variable input is not validated before being copied to the destination variable that's not set to accommodate it.
- The weakness is exploited when the size of "input" (source) exceeds the size of the destination, causing an overflow of the destination variable's address in memory.
- Sometime a buffer overflow (or overrun) error can force an application to stop operating and yields information about the error that can help an attacker formulate more effective future attacks that will succeed.

OS Commanding

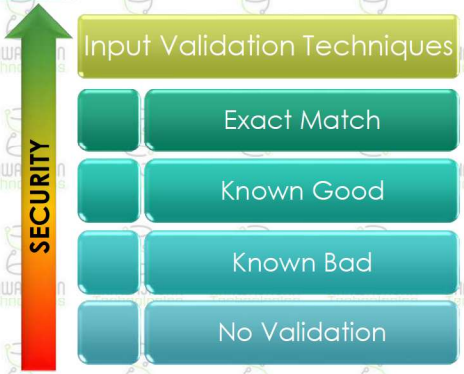
- OS commanding (command injection) is an attack technique used for un authorized execution of operating system commands. Improperly handled input from the user is one of the common weaknesses that can be exploited to run unauthorized commands.
-
- Consider a web application exposing a function showInfo() that accepts parameters "name" and "template" from the user and opens a file based on this input
  - e.g: http://example/cgi-bin/showInfo.pl?name=John&template=tmp1.txt
- As a result of improper or nonexistent input checking, by changing the template parameter value, an attacker can trick the web application into executing the command /bin/ls or open arbitrary files.
  - e.g: http://example/cgi-bin/showInfo.pl?name=John&template=/bin/ls|

SP 2025

Best Practices in Validating Input Data

Input Validation Techniques

- There are several techniques for validating input data, and each one has varying levels of security, with the better ones following the principle of use a positive security model, as illustrated in Figure



Exact Match Validation

- For exact match validation, the following elements are in play:
  - Data is validated against a list of explicit known values.
  - Requires the definition of all possible values that are considered valid input. e.g., a choice-based answer: "Yes" and "No."
  - Provides the strongest level of protection against malicious data, because it limits the attacker to choosing only between the accepted values.
  - May not be feasible when a large number of possible good values are expected—e.g., generic fields such as name or address.

Exact Match Validation

- Java example:

```
static boolean validateGender(String gender) {
if(gender.equals("Female"))
      return true;
else if(gender.equals("Male"))
      return true;
else
      return false;
}
```

Exact Match Validation

- .NET example:

```
static bool validateGender(String gender) {
if(gender.equals("Female"))
      return true;
else if(gender.equals("Male"))
      return true;
else
      return false;
}
```

Known Good Validation

- Known good validation is based on:
  - Whitelist validation
  - Data is validated against a list of allowable characters
  - Requires the definition of all characters that are accepted as valid input
  - Typically implemented using regular expressions (regex) to match known good data patterns

Known Good Validation

- Java example:

```
import java.util.regex.*;
static boolean validateUserFormat(String userName){
boolean isValid = false; //Fail by default
try{
// Verify that the UserName is 10 character alphanumeric
if (Pattern.matches("^[A-Za-z0-9]{10}$", userName))
      isValid=true;
}catch(PatternSyntaxException e){
 System.out.println(e.getDescription());
}
return isValid;
```

Known Good Validation

- .NET example:

```
using System.Text.RegularExpressions;
static bool validateUserFormat(String userName){
    bool isValid = false; //Fail by default
    // Verify that the UserName is 1-10 character alphanumeric
    isValid = Regex.IsMatch(userName, @"^[A-Za-z0-9]{10}$");
    return isValid;
}
```

Known Bad Validation

- For known bad validation, you'll find:
  - Blacklist validation (e.g. signatures for previously identified malware and viruses).
  - Data is validated against a list of characters that are deemed to be unacceptable.
  - Requires the definition of all characters that are considered dangerous to the application.
  - Useful for preventing specific characters from being accepted by the application.
  - Highly susceptible to evasion using various forms of character encoding.
  - Is the weakest method of validation against malicious data.

Known Bad Validation

- Java example:

```
import java.util.regex.*;
static boolean checkMessage(string messageText) {
boolean isValid = false; //Fail by default
try{
Pattern P = Pattern.compile("<|>", Pattern.CASE_INSENSITIVE |
Pattern.MULTILINE);
Matcher M = p.matcher(messageText);
if (!M.find())
     isValid = true;
}catch(Exception e){
     System.out.println(e.toString());
}
return isValid;
}
```

Known Bad Validation

- .NET example:

```
using System.Text.RegularExpressions;
static boolean checkMessage(string messageText){
    bool isValid = false; //Fail by default
    // Verify input doesn't contain any <, >
    isValid = !Regex.IsMatch(messageText, @"[><]");
    return isValid;
}
```

Handling Bad Input

- Three Possibilities of Handling
- Take No Action:
  - Most insecure way
- Escaping bad input:
  - The application attempts to fix the bad input data by encoding the malicious data in a "safe" format.
- Rejecting bad input:
  - The application rejects (discards) the input data and displays an error message to the user
- **Rejecting bad input is always considered better than escaping.**

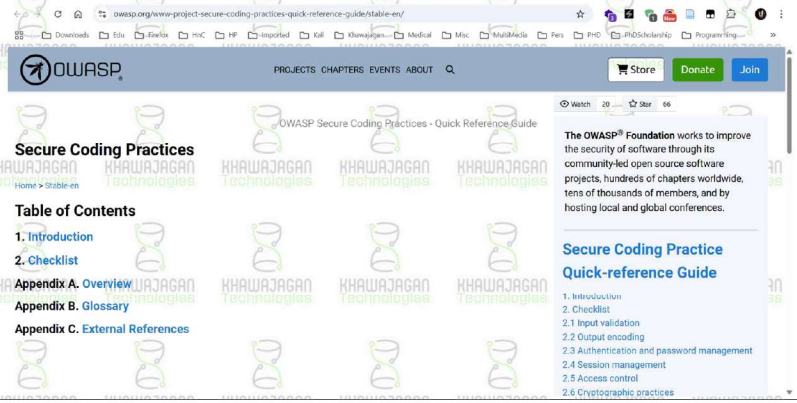
OWASP's Secure Coding Practices

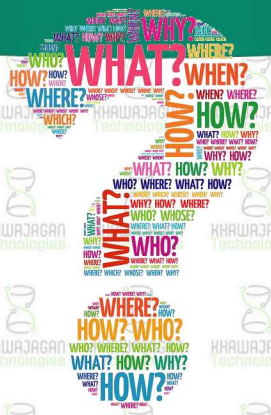
- A handy Secure Coding Practices (SCP) Quick Reference Guide from OWASP is a technology-agnostic set of general software security coding practices in a comprehensive checklist format that can be integrated into the development lifecycle.
- It contains 214 entries on all aspects of programming, communications, and server configurations related to application security.
- The focus is on secure coding requirements, rather than on vulnerabilities and exploits.
- It includes an introduction to software security principles and a glossary of key terms.
- The SCP is designed to serve as a secure coding kick-start and easy reference to help development teams quickly understand secure coding practices.

OWASP's Secure Coding Practices

- https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/
- The SCP is organized as follows:
  - Table of contents
  - Introduction
  - Software Security Principles Overview
  - Secure Coding Practices Checklist
  - Links to useful resources
  - Glossary of important terminology

OWASP's Secure Coding Practices





KHAWAJAGAN
Technologies