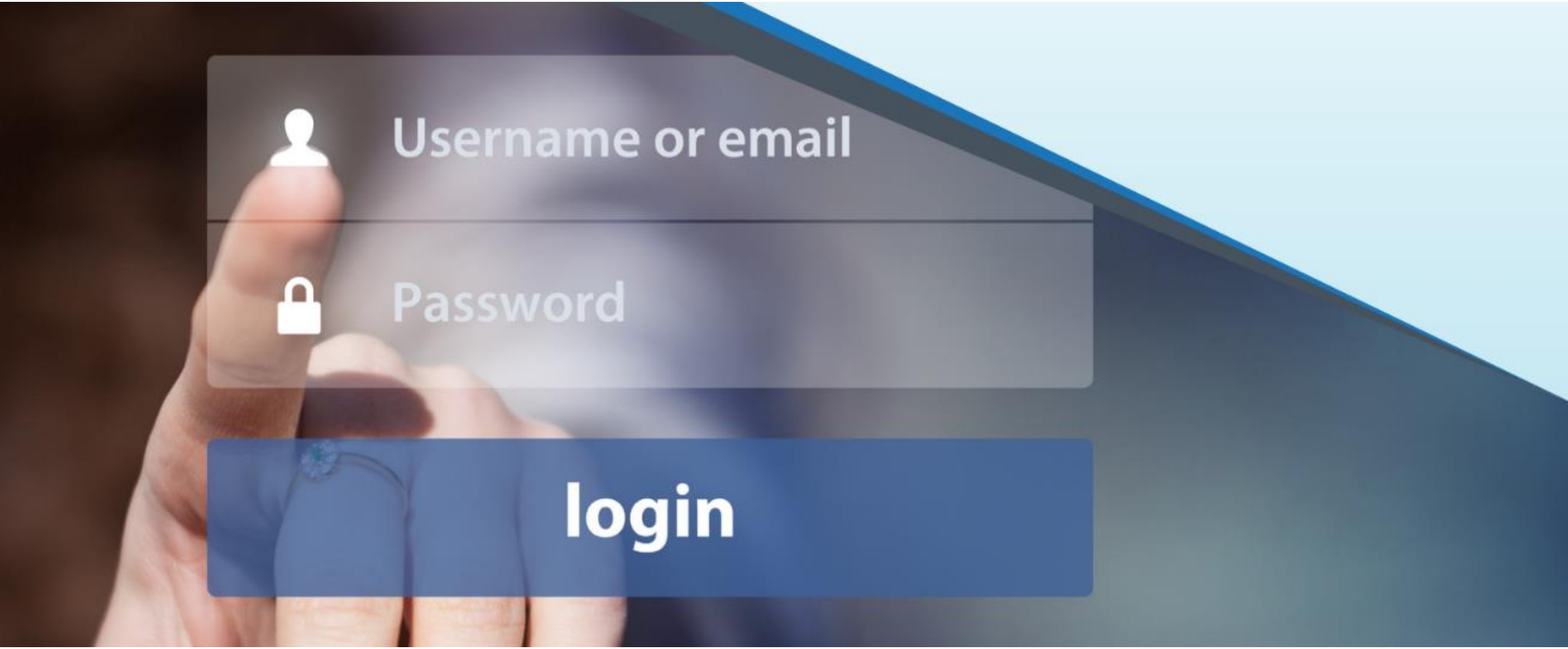

Username or email
Password
login

Secure Coding Practices for Input Validation

Module Objectives

1. Understand the Importance of Robust Input Validation
2. Discuss Secure Input Validation Techniques in Web Forms, ASP.NET Core, and MVC
3. Learn Defensive Coding Techniques against SQL Injection Attacks
4. Learn Defensive Coding Techniques against XSS Attacks
5. Learn Defensive Coding Techniques against Parameter Tampering Attacks
6. Learn Defensive Coding Techniques against Directory Traversing Attacks
7. Learn Defensive Coding Techniques against Open Redirect Vulnerabilities

Does your code perform as expected for the entire range of possible inputs?

Input Validation

1. Input validation is the process of **verifying and testing user inputs** of the application that come from untrusted data sources

2. It is the simplest defensive technique used to secure web applications from **injection attacks**

3. Proper input validation techniques are used to eliminate the **vulnerabilities in web applications**

Why Input Validation?

- Improper validation of input may provide the path for the attackers to perform **injection attacks** such as **cross site scripting attacks** and **SQL injection attacks** on the application

- Firewalls cannot prevent the attacks caused by **malicious or invalid inputs** and processing of these inputs without validation can make the application **vulnerable to attacks**

- Attackers can exploit improper input validation vulnerabilities by **supplying malicious data** to crash the application, manipulate or corrupt databases, etc.

Input Validation Specification

- **The input should be validated against:**
  - Data type (string, integer, real, etc.)
  - Allowed character set
  - Minimum and maximum length
  - Whether null is allowed
  - Numeric range
  - Whether duplicates are allowed
  - Whether the parameter is required or not
  - Specific legal values (enumeration)
  - Specific patterns (regular expressions)

Input Validation Approaches

- **The developer can take two approaches to perform input validation**

Client-side Input Validation

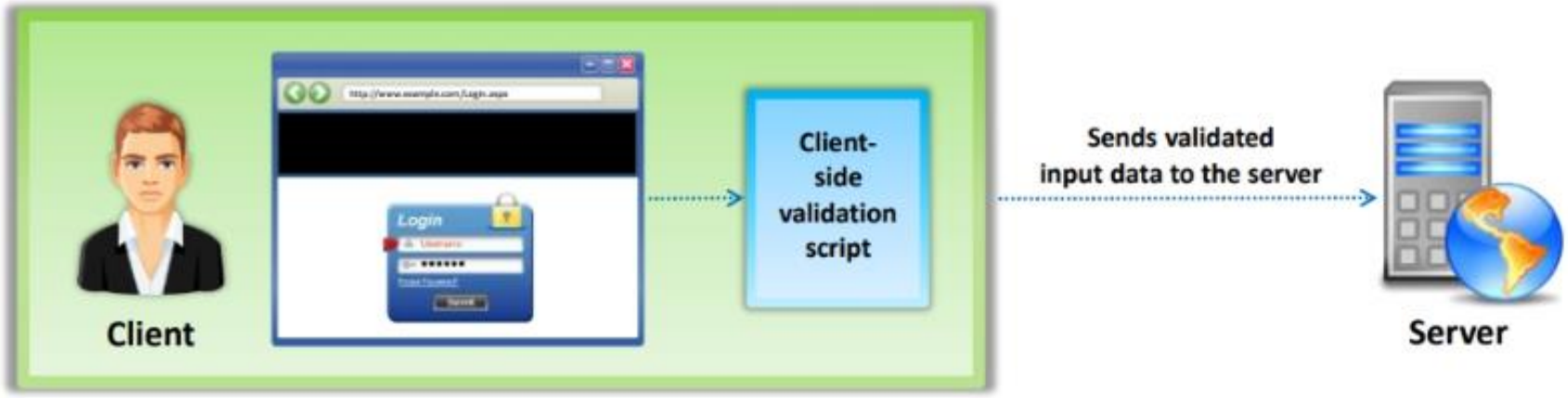
- A client-side language is used to perform client side validation that includes languages such as JavaScript, VBScript, etc.

Server-side Input Validation

- A server-side language is used to perform server-side validation that includes languages such as ASP, PHP, JSP, etc.

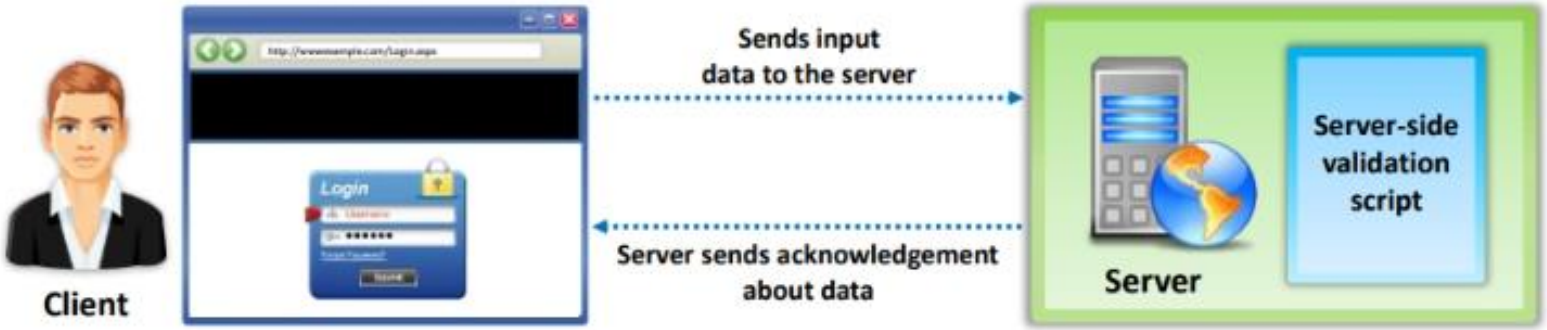
Client-side Input Validation

1. Client-side script for input validation executes at the client side **validating the input data received** from the user and sends the validated data to the server for further processing
2. This approach takes **less bandwidth** and **time** to validate the input data
3. It displays the errors one by one



Server-side Input Validation

- The server-side script for input validation executes on the server and **validates the input coming from the client**
  - Client sends the data to server and waits for its response
  - Server validates input data and sends acknowledgement to client about wrong input data
  - The client again sends the corrected input data to the server
  - This process continues until valid data is entered
- Server-side input validation **consumes extra time** and **bandwidth**
- It increases server **load** and **network traffic**



Client-server Input Validation Reliability

- Client-side validation is **not reliable**, as the attacker can easily **bypass** client-side input validation script by disabling it
- Server-side input validation is the **most reliable form of input validation**
- Both client-side and server-side input validation should be implemented to **secure the application**

Client-side vs Server-side Input Validation

Non-trusted Code (Client-side Script)

```
<script type="text/javascript">
    function EmpIDClientValidate(ctl,args)
    {
        // The value is a multiple of 5 if the mod by 5 is 0
        args.IsValid=(args.Value % 5 == 0);
    }
</script>
</head>
```

- Client-side validation mechanism is **not reliable** as it can be easily bypassed by disabling it

Trusted Code (Server-side Validation Mechanism)

```
protected void ValidateEmpID2_ServerValidate(object source, ServerValidateEventArgs args)
{
    try
    {
        args.IsValid = (int.Parse(args.Value) % 5 == 0);
    }
    catch
    {
        args.IsValid = false;
    }
}
```

- Server-side validation mechanism is **reliable** as it resides on the server and cannot be bypassed

Input Filtering

- Input filtering is a process of rejecting or accepting user inputs as per predefined criteria
- It prevents the application from **unrecognized** or **malicious inputs**
- The user input matches or compares with the **predefined set of input characters** to determine acceptability
- Acceptable input is passed for further processing and the **unwanted** inputs are **blocked**
- There are two techniques to filter inputs
  - Black Listing
  - White Listing

Input Filtering Technique: Black Listing

C | ASE .NET
Certified Application Security Engineer

I. A black list is prepared to include the known **bad characters** such as **<, /, >,** etc.

II. User inputs are checked against these **black listed characters** and filtered out if they are found in the input stream

III. It is not possible to define all bad characters, which may limit the protection against known **bad characters** only

IV. It is an easy to implement

V. It is still possible for an attacker to **craft an attack** by avoiding these specific black listed characters

Input Filtering Technique: White Listing

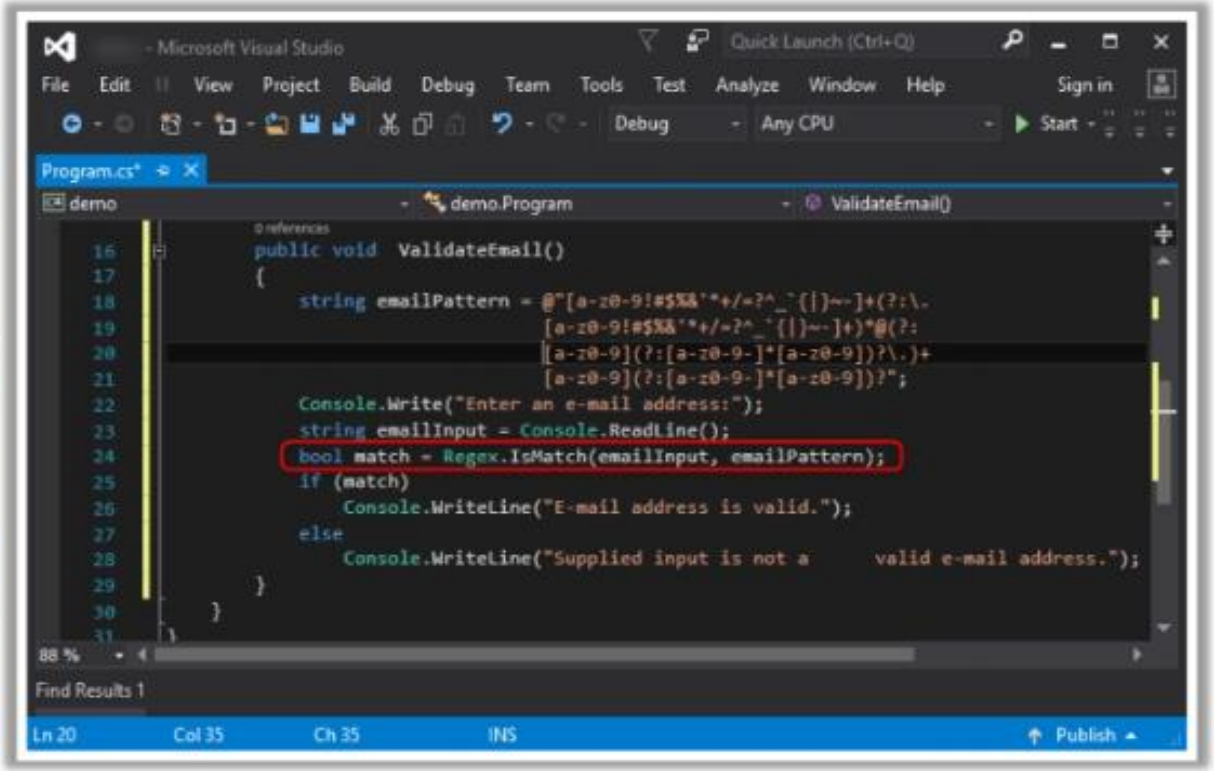
CASE .NET
Certified Application Security Engineer

1. All good characters are listed in the **white list**
2. These characters include **a-z**, **A-Z**, **0-9**, etc.
3. Input is checked against these **good characters**
4. If any characters other than the white listed characters are found, then it filters them out and treats them as malicious input
5. It is recommended to use the white listing technique for **input filtering**
6. It is quite difficult to **implement** and **compile** the white listing technique

Input Filtering using a Regular Expression

- Regular expression offers a concise and flexible way to **identify the patterns** in the given input
- It helps in **validating text** against **defined patterns**
- It also helps in extracting data from the **input text** that matches the defined pattern
- For example, **Regex** class is used to deal with **regular expression** in the .NET framework
- The method **IsMatch()** of **Regex** class is used to match defined patterns from the input text

Example: Use of common regular expression to test for valid email addresses



C|ASE .NET
Certified Application Security Engineer

Secure Coding Practices for Input Validation: Web Forms

ASP.NET Validation Controls

- Validation controls are used to validate user inputs **on the server side**
- ASP.NET frameworks provide a set of validation controls that are used to **validate the user inputs for errors**
- It allows the display of **custom messages for errors**
- These validation controls are added while creating the web form, which are then bound to the specific server control
- It reduces the use of JavaScript written for each type of validation
- It assists the browser in detecting the errors on the **client side** when an **invalid** input is entered and displays the error message without **requesting** the server

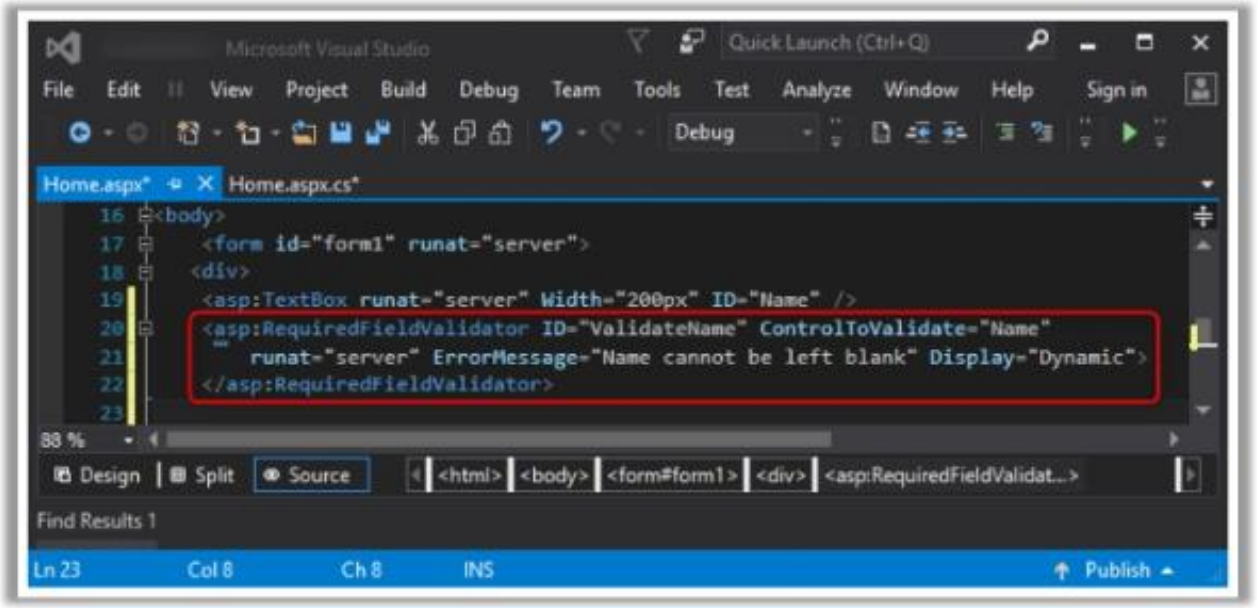
Set of ASP.NET Validation Controls

1. RequiredField Validation Control
2. Range Validation Control
3. Comparison Validation Control
4. RegularExpression Validation Control
5. Custom Validation Control
6. Validation Summary Control

RequiredField Validation Control

1. The **RequiredField** validation control is used to ensure that designated input fields are not left blank
2. It is also used to check if anyone has left a designated input field with its **default value**
3. The server tag `<asp:RequiredFieldValidator>` is used to add the RequiredField validation control in the web form for a specific input field

Sample Code to add RequiredField Validation Control



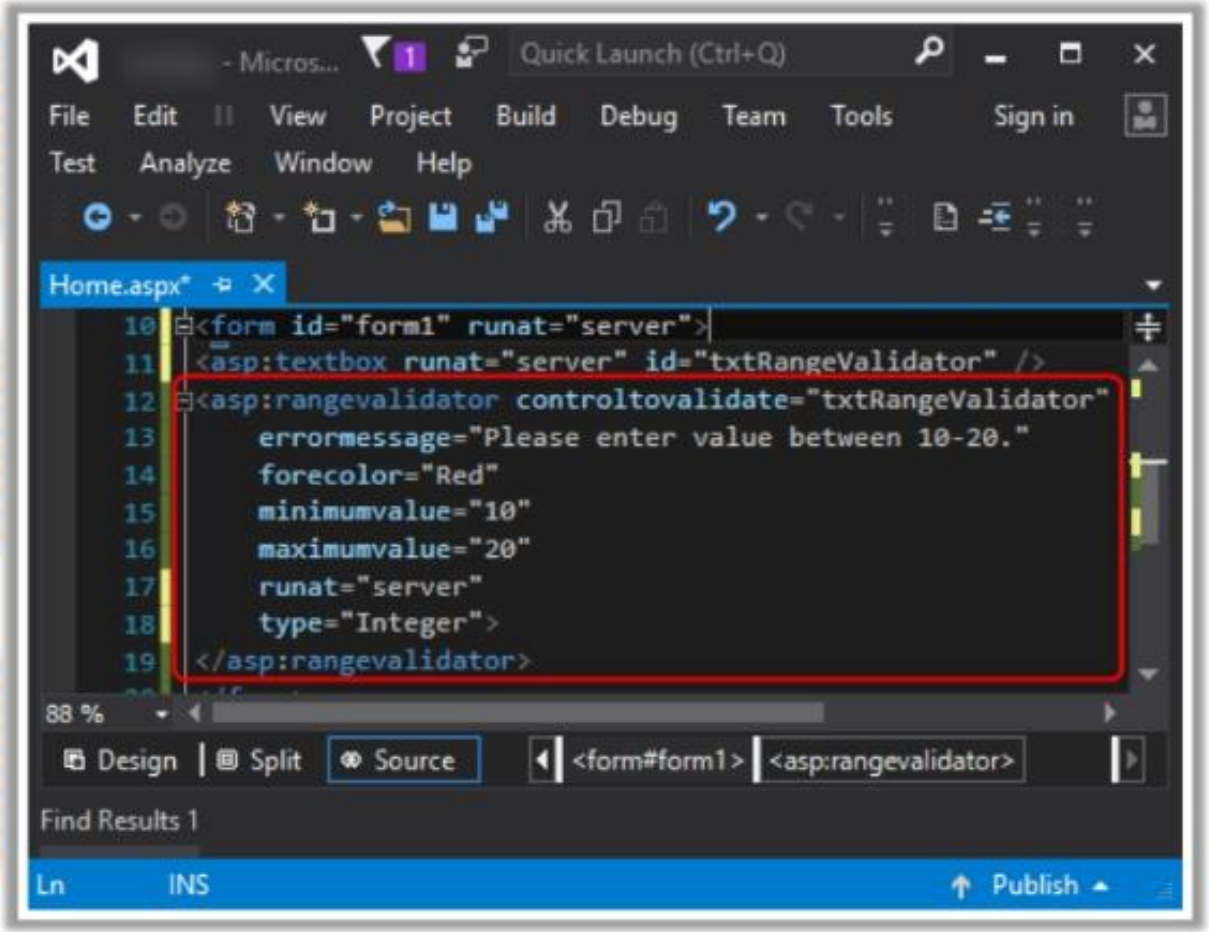
Range Validation Control

- Range validation control ensures that the value entered in the input field is within specified range

- This range is specified by the **maximum** and **minimum properties** of the validation control

- The server tag `<asp:RangeValidator>` is used to add the range validation control in the web form for a specific input field

Sample Code to add Range Validation Control



Comparison Validation Control

- Comparison validation control is used to compare the input value with specified comparisons such as **less than**, **greater than**, and so on

- The server tag `<asp:CompareValidator>` is used to add a comparison validation control in the web form for the specific input field

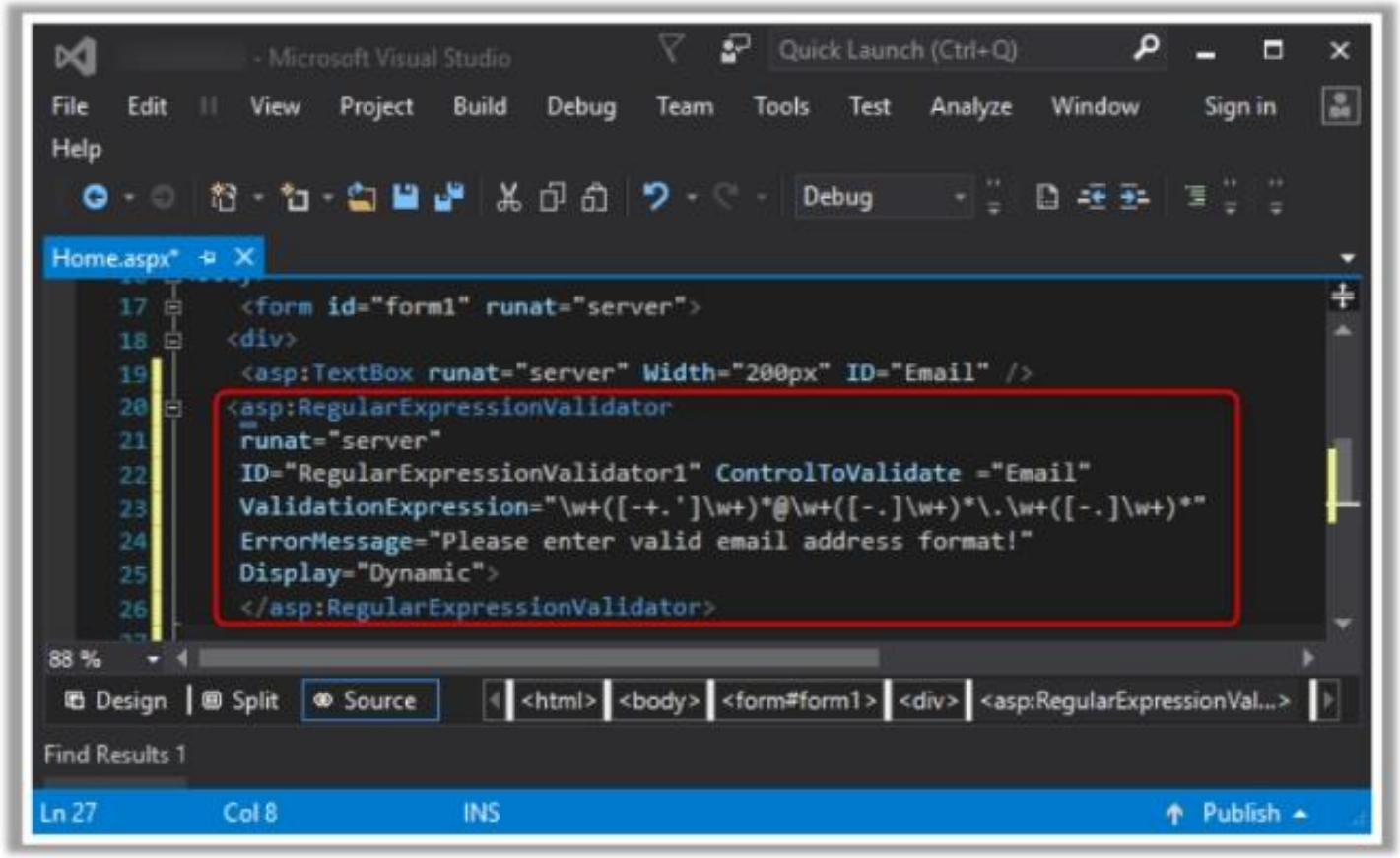
Sample Code to add Comparison Validation Control

```
<body>
    <form id="form1" runat="server">
    <div>
     <asp:TextBox runat="server" Width="200px" ID="Age" />
     <asp:CompareValidator ID="CompareValidator2" runat="server"
         ControlToValidate="Age"
         ValueToCompare="19"
         Operator="GreaterThanEqual"
         ErrorMessage="Sorry! You are under 19 year old"
         Display="Dynamic"
         Type="Integer">
     </asp:CompareValidator>
     </div>
```

RegularExpression Validation Control

- RegularExpression validation control checks whether the format of **input entered is correct or not** according to the specification
- These commonly used formats may include **email address**, **phone number**, **social security number**, **postal code**, etc.
- The server tag `<asp:RegularExpressionValidator>` is used to add the RegularExpression validation control in the web form for a specific input field

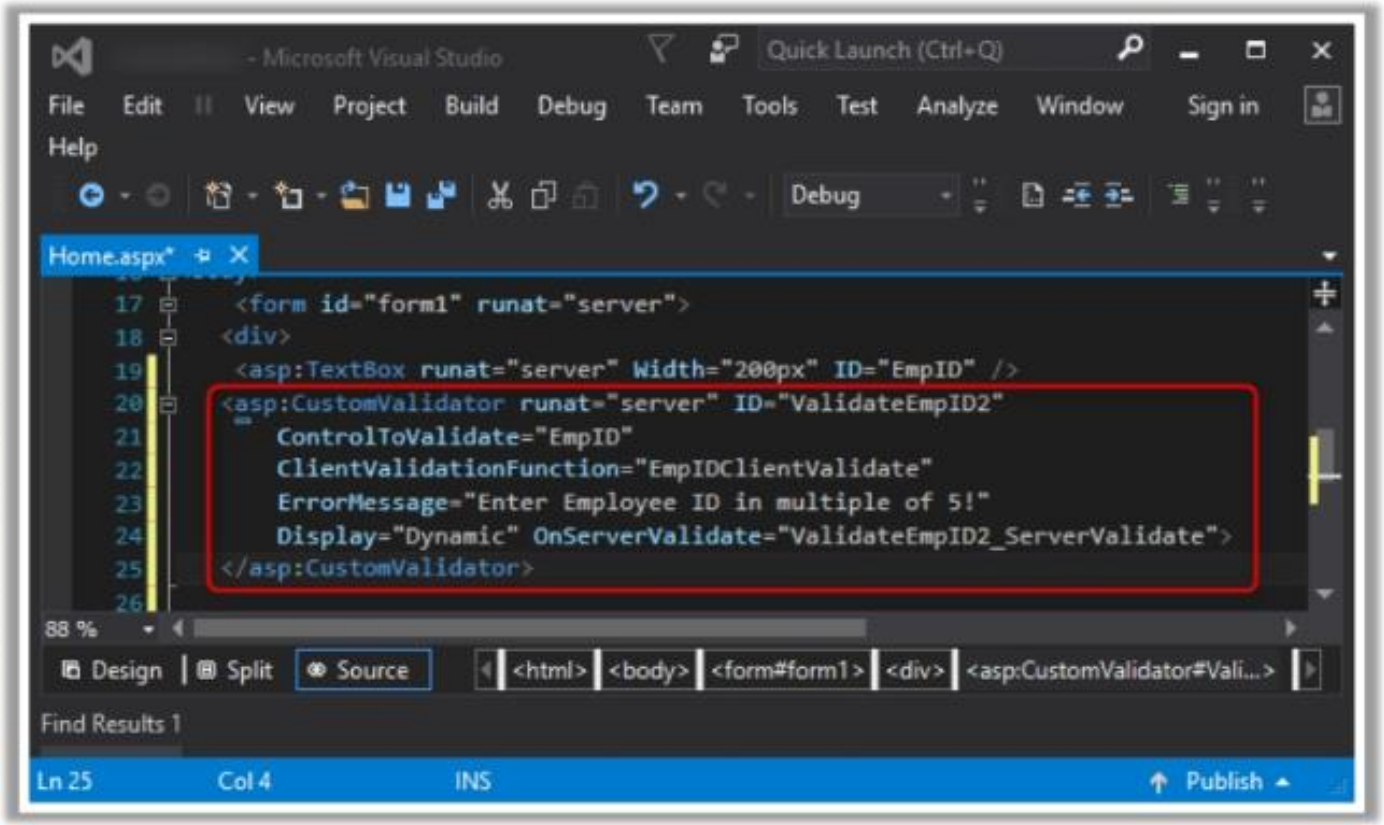
Sample Code to add RegularExpression Validation Control



Custom Validation Control

- The custom validation control allows users to define their own conditions for validating data on the input field
- The **input validation logic** can be defined on both the **client and the server side**
- The client-side script is used to define logic on the client side and server-side language is used to define logic on the server side
- Server tag `<asp:CustomValidator>` is used to add the Custom Validation control in the web form for specific input field

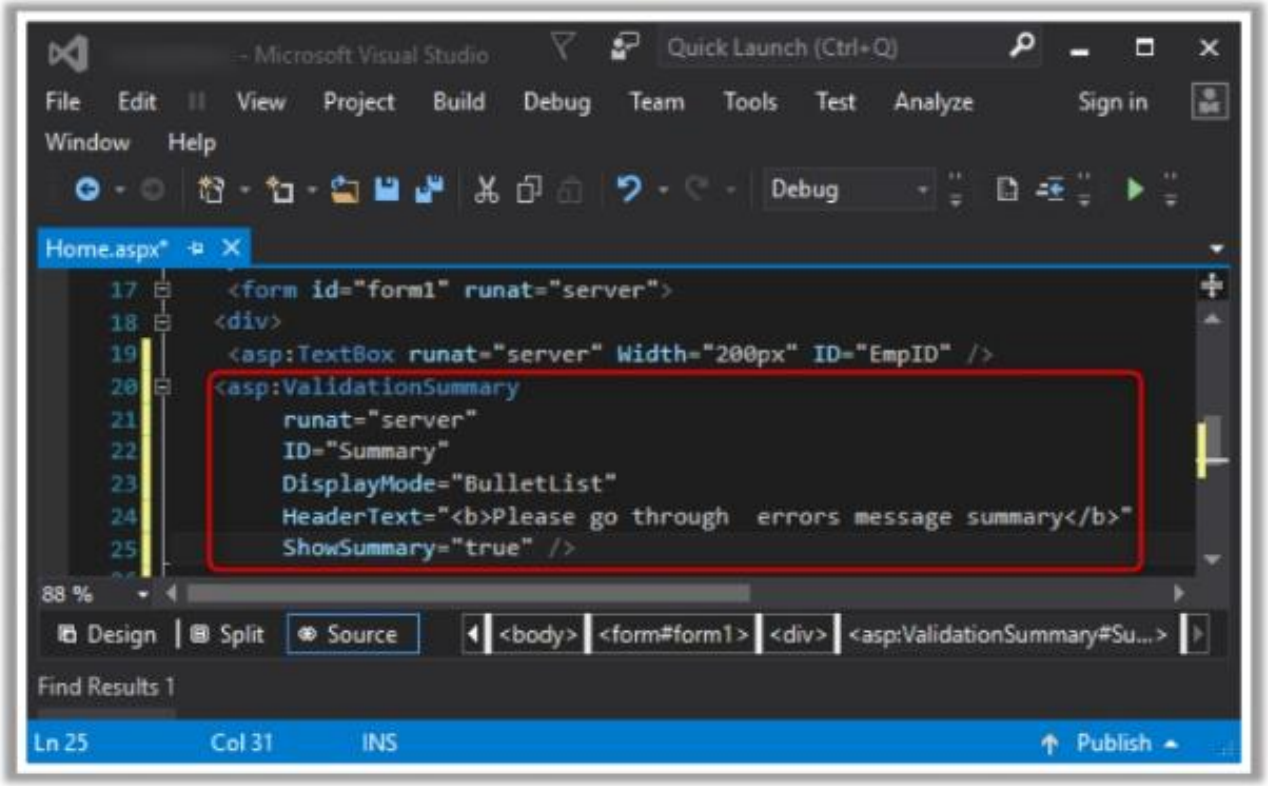
Sample Code to add Custom Validation Control



Validation Summary Control

- The validation summary control displays an **error message summary** for all validation controls on **one page**
- It does **not perform any validation**
- The server tag `<asp:ValidationSummary>` is used to add validation summary control in the web form for specific input field

Sample Code to add Validation Summary Control



SQL Injection Attack Defensive Techniques

- **There are five major defensive techniques used to prevent applications from SQL injection attacks**

1. Use of parameterized queries
2. Use of parameterized stored procedures
3. Handle the special input characters using escape routines
4. Least privilege
5. Constraining input

Using Parameterized Queries

- In parameterized queries, **SQL query** is written without embedding **parameters** in it; instead, each parameter of **query** is supplied dynamically later
- This technique helps in distinguishing between **code** and **data irrespective** of user input
- Parameterized queries do not allow attackers to change the **intent of the query**

Vulnerable Code (Non-parameterized Query)

```
30      public void Login(string user, string password)
31      {
32          string strcon = ConfigurationManager.ConnectionStrings["accounts"].ConnectionString;
33          SqlDataAdapter my_query = new SqlDataAdapter("SELECT * FROM accounts" +
34                "WHERE acc_user='" + user + "' AND acc_password='" + password + "'", strcon);
35
36      }
```

- This non-parameterized query may be **vulnerable to attack** since the attacker may change the intent of the query

Using Parameterized Queries (Cont'd)

Secure Code (Parameterized Query)

```
31      public void Login(string user, string password)
32      {
33          string strcon = ConfigurationManager.ConnectionStrings["accounts"].ConnectionString;
34          SqlDataAdapter my_query = new SqlDataAdapter("SELECT * FROM accounts WHERE acc_user="+
35              " @user AND acc_password=@pass", strcon);
36          SqlParameter userParam = my_query.SelectCommand.Parameters.Add("@user", SqlDbType.VarChar, 20);
37          userParam.Value = user;
38          SqlParameter passwordParam = my_query.SelectCommand.Parameters.Add("@pass", SqlDbType.VarChar, 20);
39          passwordParam.Value = password;
40      }
```

- This parameterized approach of the query helps in preventing the change in the intent of the query, thus preventing an attack

Using Parameterized Stored Procedures

- The parameterized stored procedure also **allows the developer to write SQL code** first and then pass parameters to it
- The only difference is that non-parameterized stored procedures are stored in the **database** with values supplied to them and then they are called the **application**

Vulnerable Code (Non-Parameterized Stored Procedure)

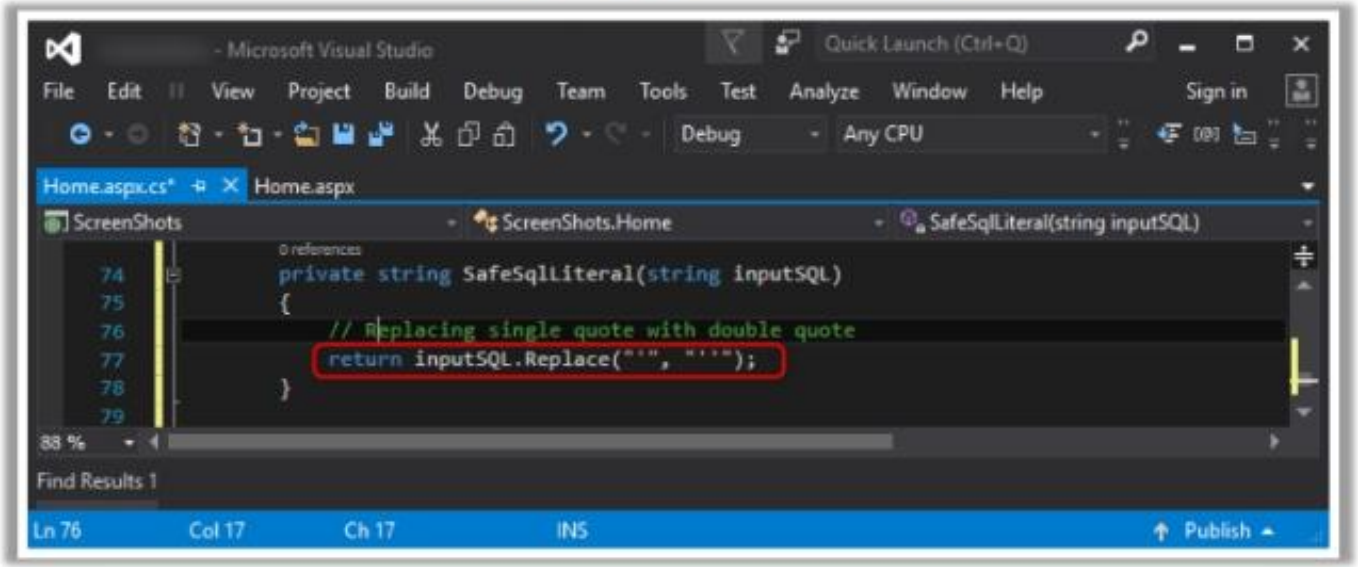
```
1 reference
public void GetAccountDetails(string CustomerName)
    {
        var HTML = string.Empty;
        string strcon = ConfigurationManager.ConnectionStrings["accounts"].ConnectionString;
        SqlConnection con = new SqlConnection(strcon);
        con.Open();
        SqlCommand cmd = new SqlCommand("sp_getAccountBalance '" + CustomerName +"'", con);
        SqlDataReader rdr = cmd.ExecuteReader();
        while (rdr.Read())
```

- This non-parameterized stored procedure approach may be **vulnerable to tampering attack**

Using Escape Routines to Handle Special Input Characters

- This technique is used to escape special characters from **user input** before supplying them to **query**
- It is used when **parameterized queries** or **stored procedures** cannot be used and have no other option besides using **dynamic SQL query**
- In such a situation, it is necessary to safeguard against **special user input characters** supplied that have special meaning to **SQL Server**; if not handled, a character such as (') may cause **SQL injection**
- Escape routines are defined to replace the escape characters with characters having special meaning to **SQL Server** thereby avoiding **harmful characters** from being supplied to the **query**

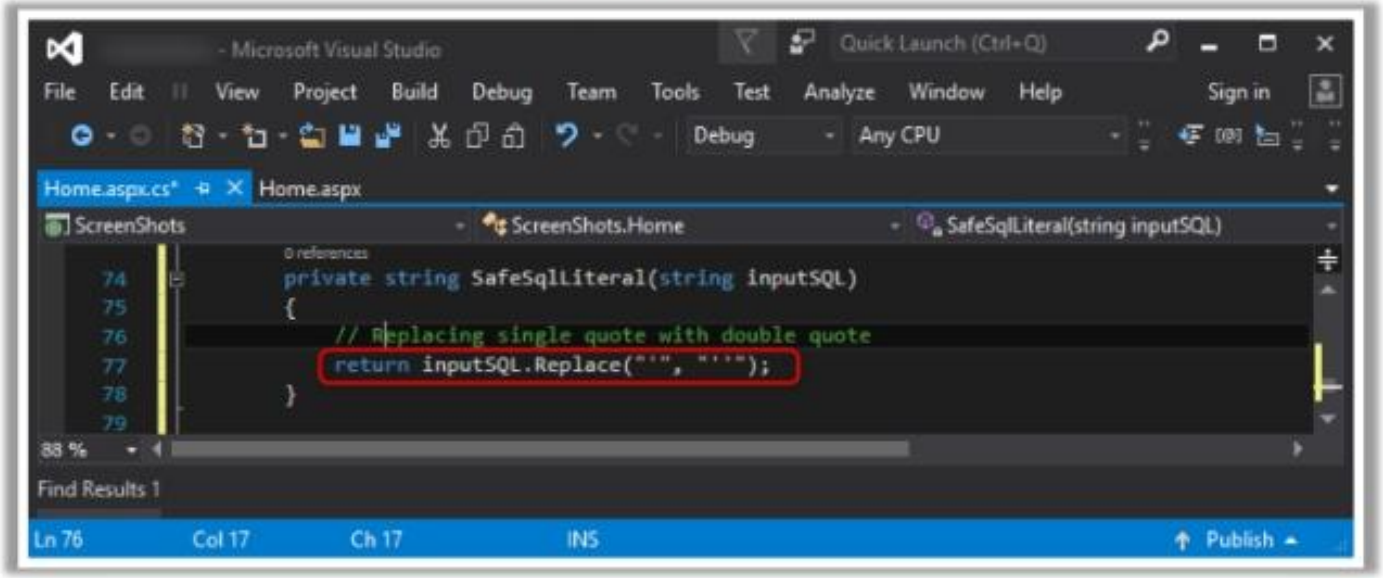
Code showing Sample Escape Routine



Using Escape Routines to Handle Special Input Characters

- This technique is used to escape special characters from **user input** before supplying them to **query**
- It is used when **parameterized queries** or **stored procedures** cannot be used and have no other option besides using **dynamic SQL query**
- In such a situation, it is necessary to safeguard against **special user input characters** supplied that have special meaning to **SQL Server**; if not handled, a character such as (') may cause **SQL injection**
- Escape routines are defined to replace the escape characters with characters having special meaning to **SQL Server** thereby avoiding **harmful characters** from being supplied to the **query**

Code showing Sample Escape Routine



Using a Least-privileged Database Account

This is another technique to prevent **SQL injection** by using **least-privileges** to connect the **database** to the application

The application should never be allowed to acquire privileges as a **DBA** or **Administrator**

The accounts that require only **read access** should be granted access to particular data in the **database**

For accounts that need access to only particular portions of the table, a view should be created on that **table** to **limit the access** of such accounts

Constraining Input

- All query input should be validated against **type**, **length**, **format**, and **range of input**

- Use the white list approach and **regular expressions** for **input validation**

- Use server-side validation approach and its validation controls such as **RegularExpressionValidator** and **RangeValidator** to constraint the input

XSS Attack Defensive Techniques

- **The defensive techniques of XSS attacks are categorized based on:**

Types of user inputs

- These user inputs can be HTML, string, uploaded files, etc.

Place of user inputs

- These are the places where user inputs can be displayed in the HTML document
- These places may include HTML body, HTML attributes, etc.

- **Two major defensive techniques for preventing injection attacks are:**

- Input Validation
- Output Encoding

Output Encoding

- Output encoding is a technique in which characters are treated as data instead of characters by themselves
- It allows the unsafe characters and renders them as harmless text
- It converts the input characters into their equivalent encoded values, which are then sent to web pages
- It informs the relevant interpreter that data is not intended to be executed

Encoding Unsafe Output using HtmlEncode

- Html encoding is done when the data reads from user **input**, **database**, or **local file**
- The attacker uses unsafe characters in the input field to perform injection attacks
- The **HtmlEncode** method is used to convert the **unsafe input characters** to their **HTML-encoded** equivalent
- HtmlEncode converts **unsafe characters** as follows:
  - **<** is converted to **<**
  - **>** is converted to **>**
  - **&** is converted to **&**
  - **(")** is converted to **"**

Encoding Unsafe Output using HtmlEncode (Cont'd)

Illustration of `HtmlEncode` Method

```
Page Language="C#" AutoEventWireup="true" CodeBehind="Home.aspx.cs" Inherits="ScreenShots.Home" ValidateRequest="false"

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script runat="server">
        void submitBtn_Click(object sender, EventArgs e){
            Response.Write(HttpUtility.HtmlEncode(inputTxt.Text)); }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:TextBox ID="inputTxt" Runat="server" TextMode="MultiLine" Width="382px" Height="152px"></asp:TextBox>
            <asp:Button ID="submitBtn" Runat="server"  Text="Submit" OnClick="submitBtn_Click" />
        </div>
    </form>
</body>
</html>
```

- If we run this page and enter some **HTML code** in the input text box, it will produce **safe output**
- For example, `<script>say hello;</script>` input is given to the textbox, it will produce safe output as `<script> say hello;</script>` only instead of running **"say hello"** script

Encoding Unsafe Output using HtmlEncode (Cont'd)

Vulnerable Code

```
<div>
    Response.Write(Request.QueryString["fname"]);
</div>
```

- If HtmlEncode method is not used, then malicious input may **harm the application**

Secure Code

```
<div>
    Response.Write(HttpUtility.HtmlEncode(Request.QueryString["fname"]));
</div>
```

- This method helps to **encode the unsafe output** so that malicious input will not cause any harm to the application

Encoding Unsafe Output using UrlEncode

- The attacker targets the **URL** and **manipulates it to launch an XSS attack** on a particular website
- The victim clicks on the following link, which looks like a legitimate link:
  - **http://www.example.com**
- But the actual link would be:
  - **http://www.example.com/WelcomePage.Aspx?UserName=<script>Attack;</script>**
- The **UrlEncode** method is used to convert the unsafe URL as per **URL encoding rules**
- URLEncode converts characters to an **unsafe URL** as follows:
  - **Spaces** are replaced with **+ sign**
  - **Non-alphanumeric characters** are replaced with their **hexadecimal equivalent**

Encoding Unsafe Output using UrlEncode (Cont'd)

Vulnerable Code

```
public void login_click()
{
    Response.Redirect(ResolveUrl("WelcomePage."+
        "aspx?UserName=" + txtUserName.Text));
}
```

- If the UrlEncode method is not used, then a malicious URL may **harm the application**

Secure Code

```
public void login_click()
{
    Response.Redirect(ResolveUrl("WelcomePage" +
        ".Aspx?UserName=" + Server.UrlEncode(txtUserName.Text)));

}
```

- This method helps to **encode the unsafe URL** so that the malicious URL will not cause any harm to the application

Anti-XSS Library

CASE .NET

Encoding Library

- Anti-XSS library is an **encoding library** used to prevent XSS attacks on the web application

Numerous Encoding Functions

- It provides numerous **encoding functions** for **user inputs** that maybe HTML, HTML attributes, XML, CSS, JavaScript, etc.

While List Technique

- It uses the **white list technique** for encoding

Encoding Output using Anti-XSS Library

- Anti-XSS library can be added as the default encoder by changing the **httpRuntime** configuration element in Web.config file as follows:

```
<compilation debug="true" targetFramework="4.5"/>
<httpRuntime targetFramework="4.5" encoderType="System.Web.Security.AntiXss.AntiXssEncoder,System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
</system.web>
```

- Anti-XSS library can also be added using System.Web.Security.Anti-XSS namespace
- Once the reference is added to the Anti-Cross Site Scripting Library, it can be directly used to call static encoding methods as follows:

```
public void login_click()
{
    String Name = AntiXss.HtmlEncode(Request.QueryString["Name"]);
```

Encoding Output using Anti-XSS Library (Cont'd)

- If the **HtmlEncode** method is not used, then malicious input may **harm the application**

Vulnerable Code

```
Response.Write(Request.QueryString["fname"]);
```

- This method helps to **encode the unsafe output using Anti-XSS Library** so that malicious input will not harm the application

Secure Code

```
Response.Write(Response.Write(AntiXssEncoder.HtmlEncode(Request.QueryString["fname"],false)))
```

Directory Traversing Defensive Technique

- The files that are stored in **subdirectories** are vulnerable to Directory Traversal attacks

Vulnerable Code

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Clear();
    string filename = Request.QueryString["filename"];
    FileInfo file = new FileInfo(Server.MapPath(filename));
    Response.AddHeader("Content-Length", file.Length.ToString());
    Response.WriteFile(file.FullName);
    Response.End();
}
```

Secure Code

- Extract the filename from the input
- Use `Path.Combine()` method to limit files to a particular directory
- Besides the code can be rewritten in a secure way as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Clear();
    string filename =
    Path.GetFileName(Request.QueryString["filename"]);
    FileInfo file = new FileInfo(Server.MapPath(
    Path.Combine("documents", filename)));
    Response.AddHeader("Content-Length", file.Length.ToString());
    Response.WriteFile(file.FullName);
    Response.End();
}
```

Server.MapPath will **not return any path outside the root of web application**

Additional Techniques to Prevent Directory Traversal

1. Obtain an **absolute path** for the file or directory and convert all characters of path into its normal form in case of URI request
2. Process **URI requests** that do not contain file requests
3. Make sure that the first N characters of requested file path are same as the **'Document Root'**
4. If requested **file path** is same as the **document root** then only the request file is returned
5. If requested file path is not same as the document root, then **returns an error** that request is **out of bounds** from the web server

Secure Coding Practices for Input Validation: ASP.NET Core