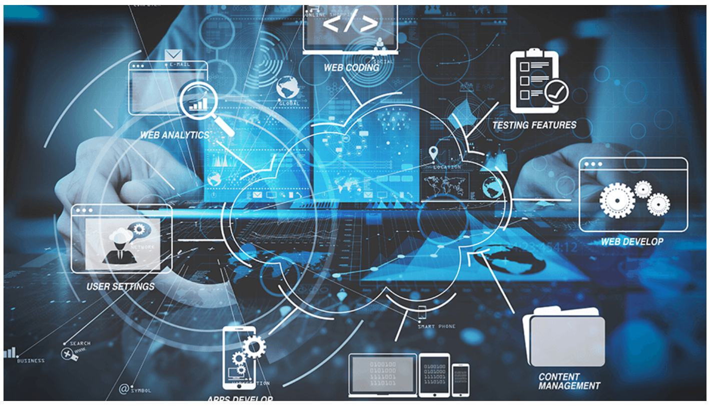


SECURITY FEATURES IN TYPICAL WEB APPLICATIONS

Analysis and Implementation Recommendations for ASP.NET Web Applications

ABSTRACT

This report analyzes two key web application features, Login Page and Account Registration Page and explains common security issues associated with each. It proposes practical mitigation techniques based on OWASP best practices and outlines how these features should be securely implemented in a typical ASP.NET web application.

Loh Xin Wen Melody, 244677K
IT2163-01

I have chosen the following security features for my Assignment:

Statement on Plagiarism and Academic Dishonesty

I certify that this assignment/report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication.

I also certify that this assignment/report has not previously been submitted for assessment in any other unit, and that I have not copied in part or whole or otherwise plagiarised the work of other students and/or persons.

Name	Loh Xin Wen Melody
Signature	Melody
Date	January 5, 2026

1. Account Registration Page

1.1 Assumptions

- Account Registration is the process of creating a new digital identity for a user. This identity will later be used to authenticate the user when accessing the application.
- The Account Registration Page collects essential details such as Name, Email or Phone Number, Address, Occupation, and requires setup of Multi-Factor Authentication (MFA) options (e.g., SMS OTP, Email OTP, or Authenticator App).
- Authentication refers to verifying that an individual, entity, or website is who or what it claims to be by validating one or more authenticators (e.g., passwords, fingerprints, or security tokens).
- All communication between client and server during registration is conducted over TLS (HTTPS) to prevent interception or tampering.
- The application is assumed to be built on ASP.NET Core, using ASP.NET Identity for user management.

1.2 Security Issues 1

- Weak Password Policies

1.2.1 Description

Allowing users to set weak passwords (short, simple, or common) makes accounts vulnerable to brute force, dictionary, and credential stuffing attacks. This can lead to account compromise and session hijacking.

1.2.2 Mitigation Techniques

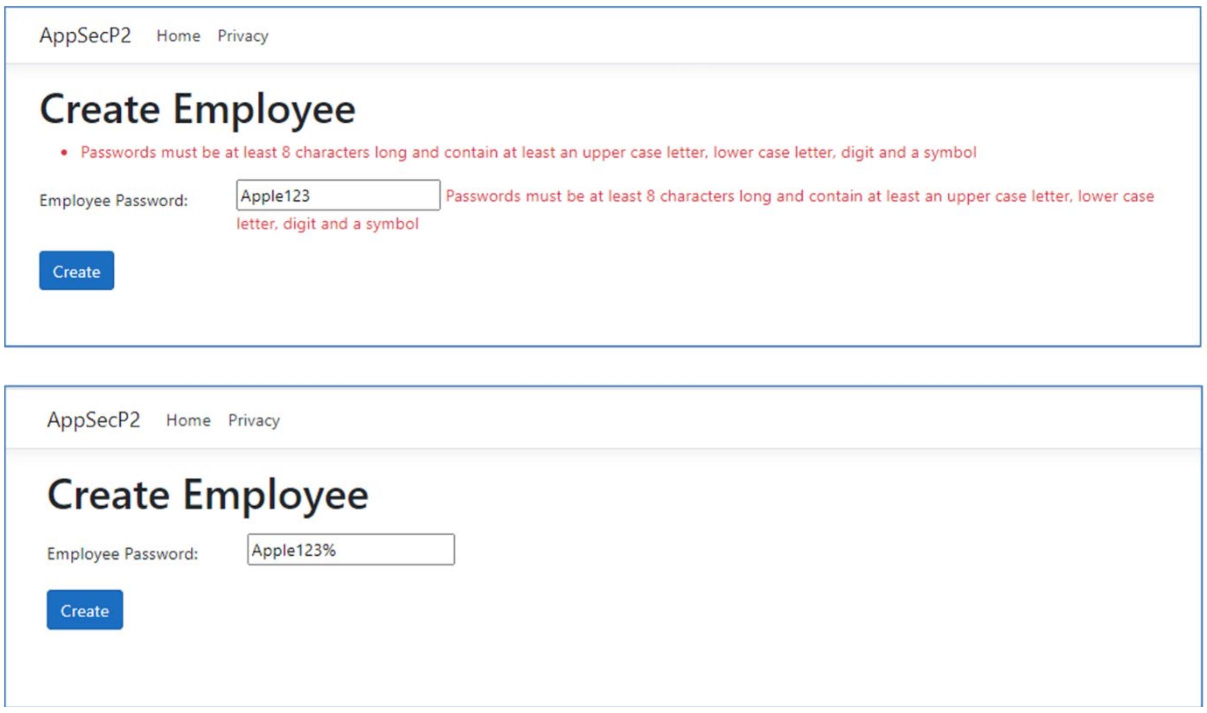
- Enforce Minimum Password Length:
  - Require at least 15 characters if MFA is not enabled.
  - Require at least 8 characters if MFA is enabled.
- Allow all characters including Unicode, whitespace, and special characters to support passphrases and international users.
- Use regex validation to enforce length and printable character rules, while avoiding outdated composition rules.
- Integrate HaveIBeenPwned API to block passwords found in known breaches.
- Provide a password strength meter (e.g., zxcvbn library) to guide users.

1.2.3 Implementation Techniques

- Use ASP.NET Core data annotations with regex validation to enforce password rules. When a weak password is entered, the system generates an error message to guide the user.

```
[RegularExpression(@"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[$@$!%*?&])[A-Za-z\d$@$!%*?&]{8,}$",
   ErrorMessage = "Passwords must be at least 8 characters long and contain at least an upper
      case letter, lower case letter, digit and a symbol")]

        public string Password { get; set; }
```

Figure 1: Example of regex validation for enforcing strong password policies



Create Employee

- Passwords must be at least 8 characters long and contain at least an upper case letter, lower case letter, digit and a symbol

Employee Password: Apple123 Passwords must be at least 8 characters long and contain at least an upper case letter, lower case letter, digit and a symbol

Create Employee

Employee Password: Apple123%

Figure 2: Example of creating an employee password with ASP.NET Core annotations

This ensures that passwords meet minimum length and complexity requirements, and the error message provides immediate feedback to the user.

1.2.4 Supporting Evidence/Justification

This solution aligns with the OWASP Authentication Cheat Sheet and NIST SP800-63B Digital Identity Guidelines, which recommend enforcing strong password policies and providing user feedback. It is effective because it prevents trivial passwords, increases entropy, and reduces the risk of brute force or credential stuffing attacks.

1.3 Security Issues 2

- Input validation

1.3.1 Description

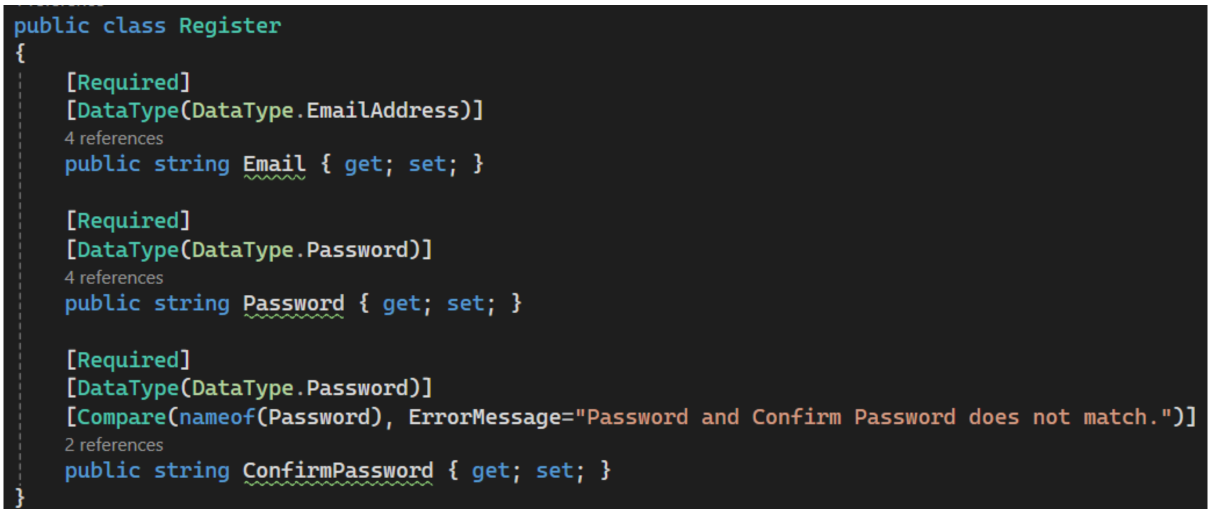
Improper input validation allows attackers to inject malicious data (SQL injection, XSS) or submit malformed values that corrupt the database or bypass business logic.

1.3.2 Mitigation Techniques

- Validate all inputs using allowlist patterns.
- Use regex for structured fields like email and phone number.
- Enforce length limits and required fields for name, address, and occupation.
- Normalize Unicode input to prevent homoglyph attacks.
- Reject or sanitize unexpected characters to prevent injection attacks.

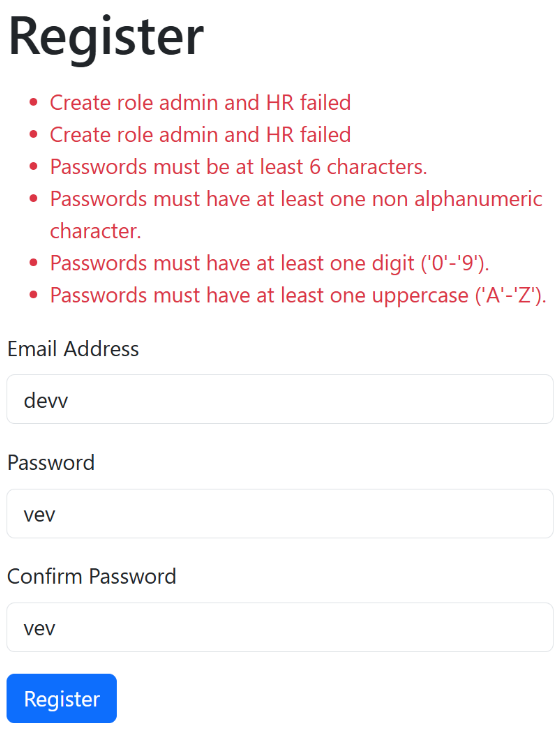
1.3.3 Implementation Techniques

- Use ASP.NET Core data annotations ([Required], [EmailAddress], [StringLength]) to validate inputs server-side. Apply regex for email format. Enforce maximum length for text fields. Normalize Unicode input.



```
public class Register
{
    [Required]
    [DataType(DataType.EmailAddress)]
    4 references
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    4 references
    public string Password { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Compare(nameof(Password), ErrorMessage="Password and Confirm Password does not match.")]
    2 references
    public string ConfirmPassword { get; set; }
}
```

Figure 3: - Example of input validation using data annotations ([Required], [EmailAddress])



Register

- Create role admin and HR failed
- Create role admin and HR failed
- Passwords must be at least 6 characters.
- Passwords must have at least one non alphanumeric character.
- Passwords must have at least one digit ('0'-'9').
- Passwords must have at least one uppercase ('A'-'Z').

Email Address

devv

Password

vev

Confirm Password

vev

Figure 4: Example of registering an account with enforced validation rules

1.3.4 Supporting Evidence/Justification

This approach follows the **OWASP Input Validation Cheat Sheet** and **OWASP Top 10 (A03:2021 Injection)**. Validating inputs early reduces the attack surface and ensures data integrity.

1.4 Security Issues 3

- Captcha/ Bot Prevention

1.4.1 Description

Attackers may use automated scripts to create fake accounts, overwhelming the system and enabling fraud, spam, or denial-of-service conditions.

1.4.2 Mitigation Techniques

- Implement CAPTCHA or reCAPTCHA to distinguish humans from bots.
- Apply rate limiting to restrict the number of registration attempts per IP or session.
- Log and monitor suspicious registration patterns for anomaly detection.

1.4.3 Implementations Techniques

- Integrate Google reCAPTCHA widget in the registration form. Verify CAPTCHA token server-side. Apply rate limiting per IP. Log suspicious attempts.

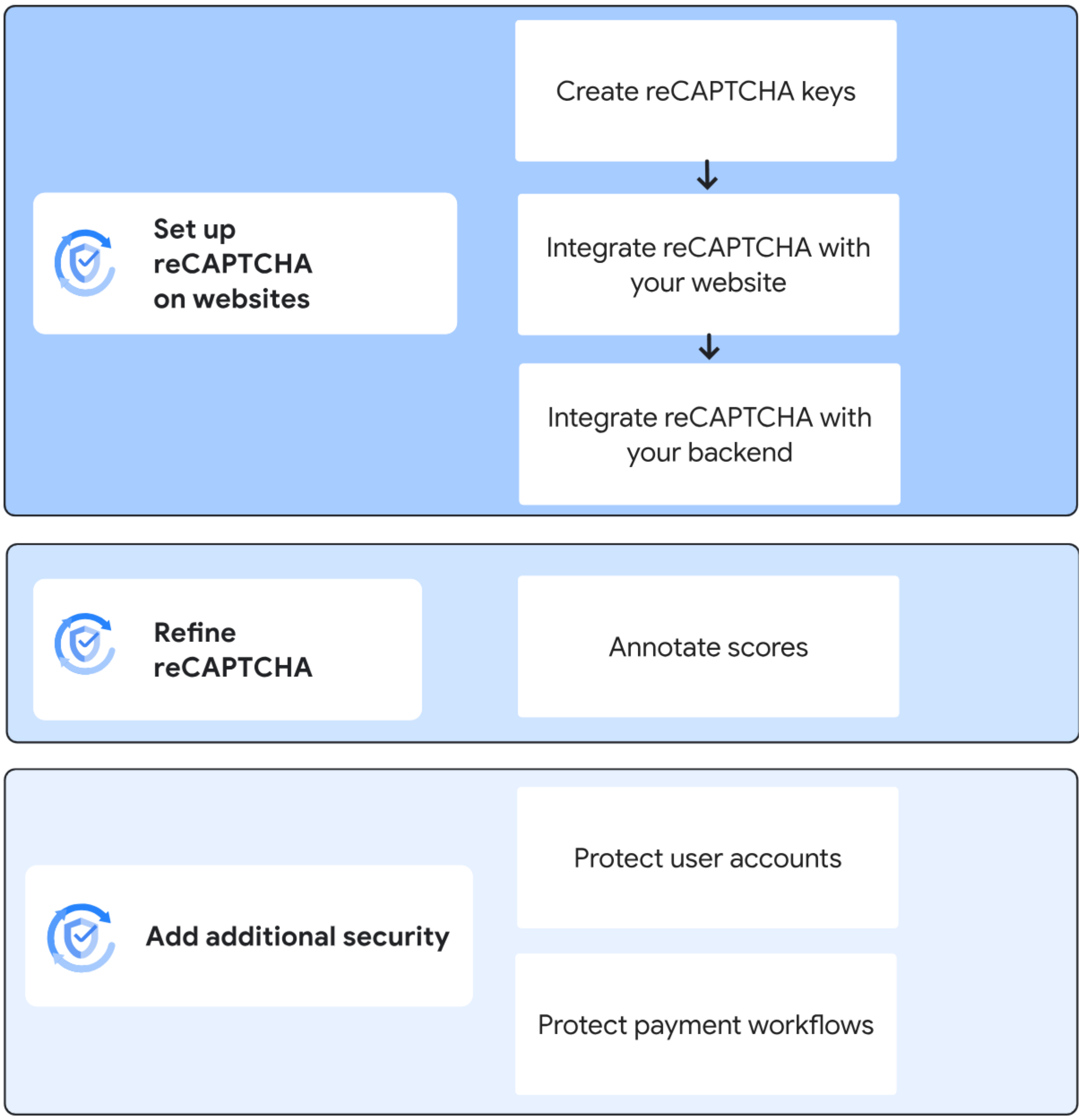
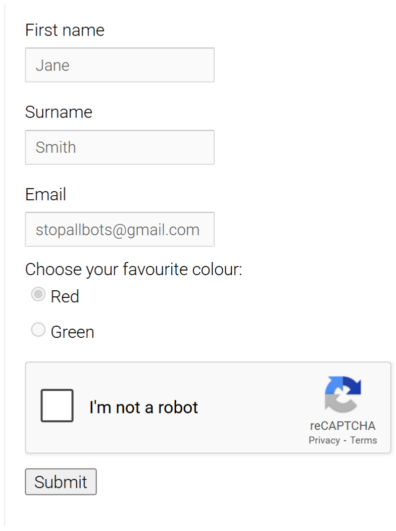

Set up reCAPTCHA on websites
Create reCAPTCHA keys
Integrate reCAPTCHA with your website
Integrate reCAPTCHA with your backend
Refine reCAPTCHA
Annotate scores
Add additional security
Protect user accounts
Protect payment workflows

Figure 5: Flowchart showing CAPTCHA implementation process


First name

Jane

Surname

Smith

Email

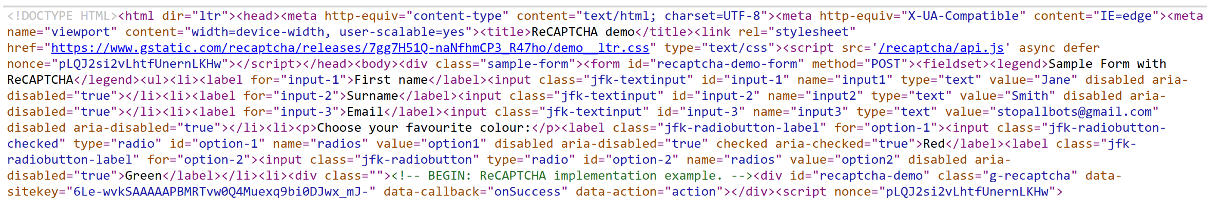
stopallbots@gmail.com

Choose your favourite colour:

● Red

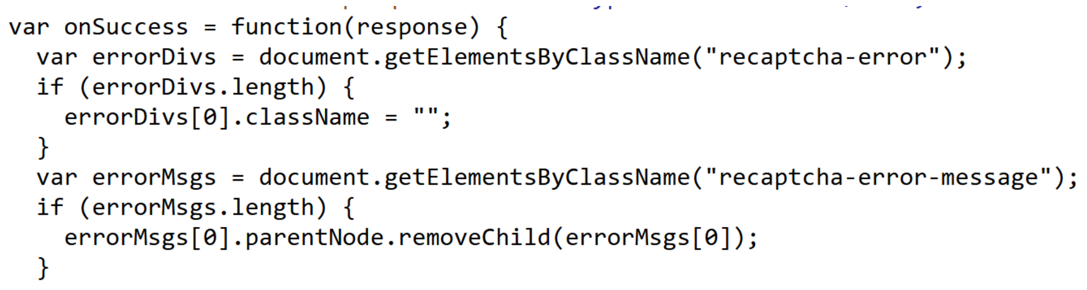
○ Green

Figure 6: Example of CAPTCHA displayed on a website



```
<!DOCTYPE HTML><html dir="ltr"><head><meta http-equiv="content-type" content="text/html; charset=UTF-8"><meta http-equiv="X-UA-Compatible" content="IE=edge"><meta
name="viewport" content="width=device-width, user-scalable=yes"><title>ReCAPTCHA demo</title><link rel="stylesheet"
href="https://www.gstatic.com/recaptcha/releases/7gg7H51Q-naNfhmCP3_R47ho/demo__ltr.css" type="text/css"><script src='/recaptcha/api.js' async defer
nonce="pLQJ2si2vLhtfUnernLKHw"></script></head><body><div class="sample-form"><form id="recaptcha-demo-form" method="POST"><fieldset><legend>Sample Form with
ReCAPTCHA</legend><ul><li><label for="input-1">First name</label><input class="jfk-textinput" id="input-1" name="input1" type="text" value="Jane" disabled aria-
disabled="true"></li><li><label for="input-2">Surname</label><input class="jfk-textinput" id="input-2" name="input2" type="text" value="Smith" disabled aria-
disabled="true"></li><li><label for="input-3">Email</label><input class="jfk-textinput" id="input-3" name="input3" type="text" value="stopallbots@gmail.com"
disabled aria-disabled="true"></li><li><p>Choose your favourite colour:</p><label class="jfk-radiobutton-label" for="option-1"><input class="jfk-radiobutton-
checked" type="radio" id="option-1" name="radios" value="option1" disabled aria-disabled="true" checked aria-checked="true">Red</label><label class="jfk-
radiobutton-label" for="option-2"><input class="jfk-radiobutton" type="radio" id="option-2" name="radios" value="option2" disabled aria-
disabled="true">Green</label></li><li><div class=""><!-- BEGIN: ReCAPTCHA implementation example. --><div id="recaptcha-demo" class="g-recaptcha" data-
sitekey="6Le-wvkSAAAAAPBMRTvw0Q4Muexq9bi0DJwx_mJ-" data-callback="onSuccess" data-action="action"></div><script nonce="pLQJ2si2vLhtfUnernLKHw">
```

Figure 7: Frontend CAPTCHA integration snippet



```
var onSuccess = function(response) {
  var errorDivs = document.getElementsByClassName("recaptcha-error");
  if (errorDivs.length) {
    errorDivs[0].className = "";
  }
  var errorMsgs = document.getElementsByClassName("recaptcha-error-message");
  if (errorMsgs.length) {
    errorMsgs[0].parentNode.removeChild(errorMsgs[0]);
  }
```

Figure 8: JavaScript Callback for CAPTCHA validation

1.4.4 Supporting Evidence/Justification

This solution aligns with the **OWASP Automated Threats to Web Applications** and **OWASP Authentication Cheat Sheet**. CAPTCHA makes automated account creation more difficult and costly for attackers, while rate limiting and logging provide layered defense.

2. Login Page

2.1 Assumptions

- Login refers to the process of verifying an entity's identity.
- A user who has been authenticated is often not authorized to access every resource and perform every action that is technically possible through a system.
- User will need to provide their username and password in order to login to the system
- The application is built on ASP.NET Core with ASP.NET Identity for authentication.
- Login requires username/email and password; MFA is optional but encouraged.
- All login requests are transmitted securely over TLS (HTTPS).
- Session management uses secure cookies with HttpOnly, Secure, and SameSite attributes.

2.2 Security Issues 1

- Session Hijacking

2.2.1 Description

Session hijacking occurs when attackers steal or predict session identifiers, allowing them to impersonate legitimate users. This can happen through XSS stealing cookies, session fixation, or insecure session handling.

2.2.2 Mitigation Techniques

- Use secure cookie attributes: HttpOnly, Secure, SameSite.
- Regenerate session ID after login or privilege changes.
- Ensure session IDs have ≥128 bits entropy and are randomly generated.
- Implement claims-based authentication and authorization policies to tightly control access.

2.2.3 Implementation Techniques

In ASP.NET Core, configure cookie authentication with secure options and regenerate session IDs after login. The following example shows how to implement cookie authentication, enforce claims, and configure login paths.

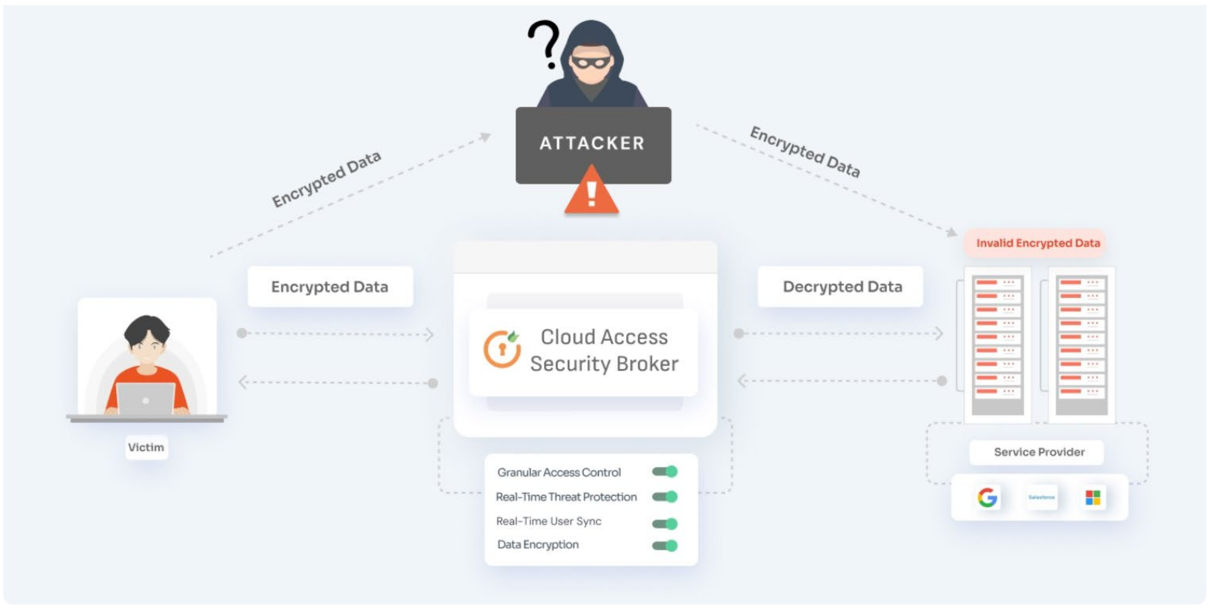
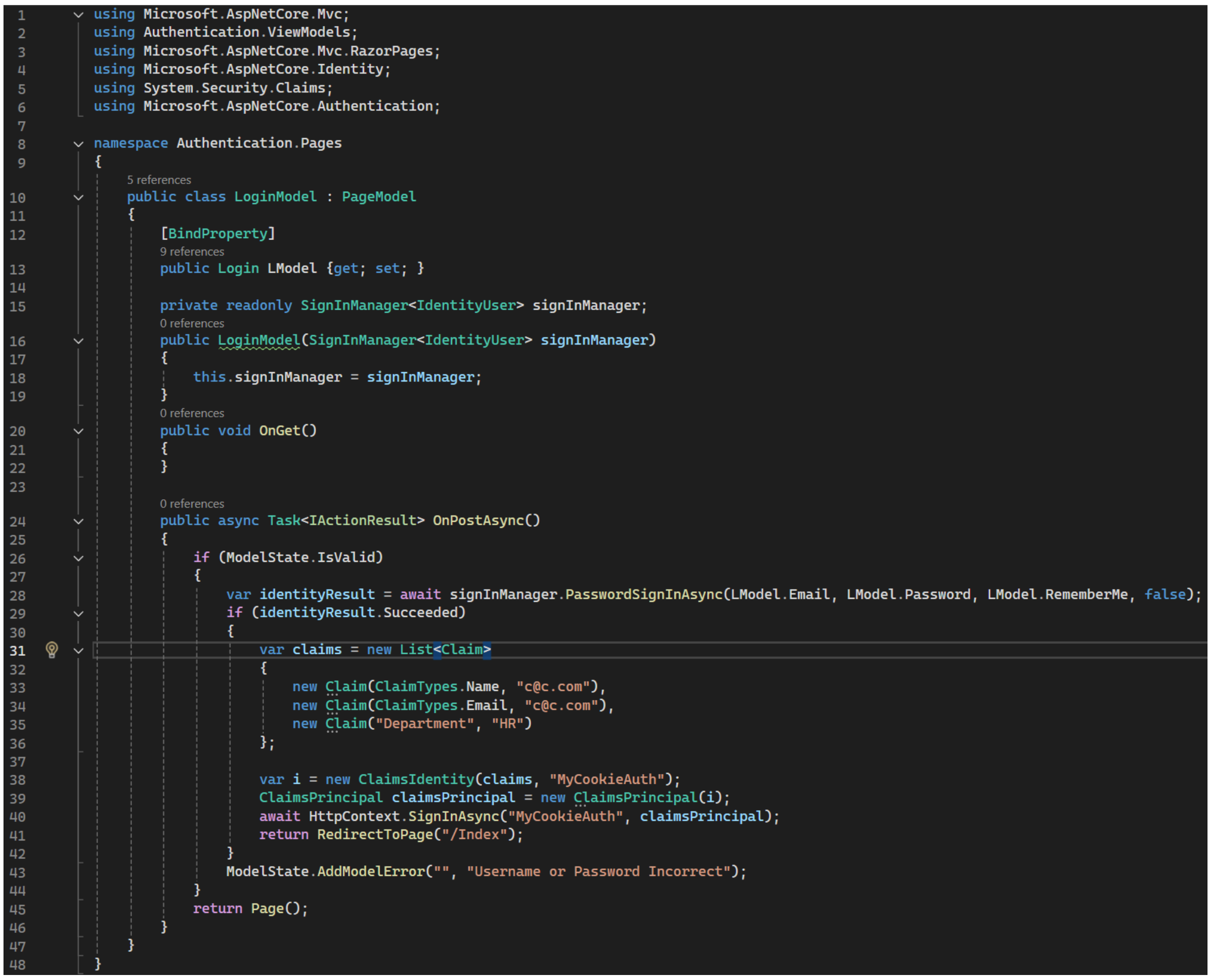
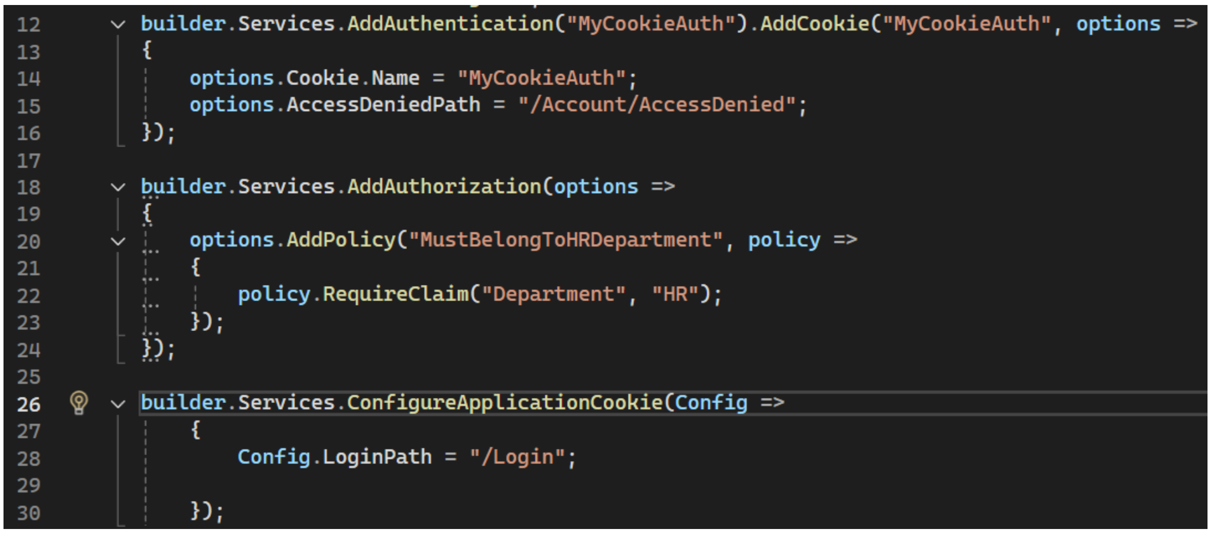


Figure 9: Example of how session hijacking can be executed



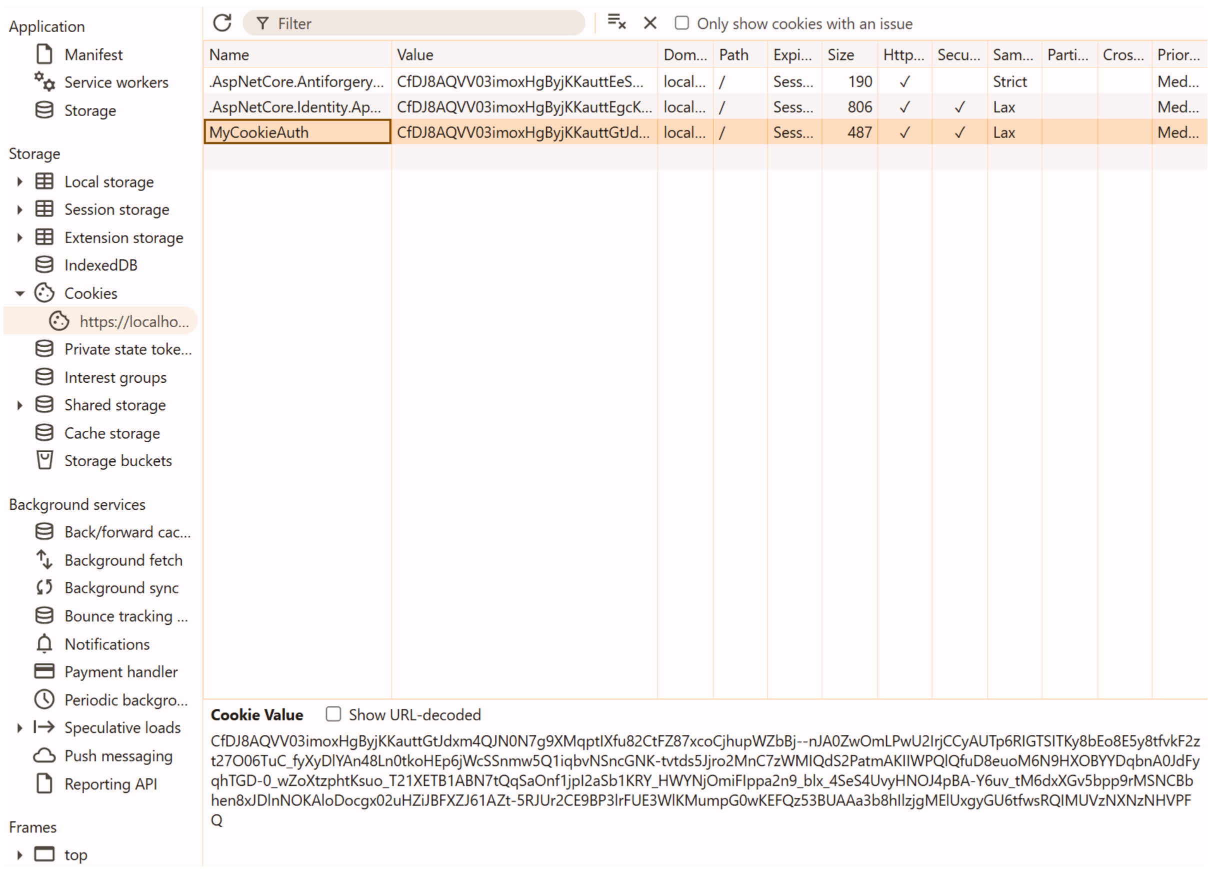
```
using Microsoft.AspNetCore.Mvc;
using Authentication.ViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Identity;
using System.Security.Claims;
using Microsoft.AspNetCore.Authentication;

namespace Authentication.Pages
{
    5 references
    public class LoginModel : PageModel
    {
        [BindProperty]
        9 references
        public Login LModel {get; set; }

        private readonly SignInManager<IdentityUser> signInManager;
        0 references
        public LoginModel(SignInManager<IdentityUser> signInManager)
        {
            this.signInManager = signInManager;
        }
        0 references
        public void OnGet()
        {
        }

        0 references
        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {
                var identityResult = await signInManager.PasswordSignInAsync(LModel.Email, LModel.Password, LModel.RememberMe, false);
                if (identityResult.Succeeded)
                {
                    var claims = new List<Claim>
                    {
                        new Claim(ClaimTypes.Name, "c@c.com"),
                        new Claim(ClaimTypes.Email, "c@c.com"),
                        new Claim("Department", "HR")
                    };

                    var i = new ClaimsIdentity(claims, "MyCookieAuth");
                    ClaimsPrincipal claimsPrincipal = new ClaimsPrincipal(i);
                    await HttpContext.SignInAsync("MyCookieAuth", claimsPrincipal);
                    return RedirectToPage("/Index");
                }
                ModelState.AddModelError("", "Username or Password Incorrect");
            }
            return Page();
        }
    }
}
```

Figure 10: Configuration in Login.cshtml.cs for cookie authentication



```
builder.Services.AddAuthentication("MyCookieAuth").AddCookie("MyCookieAuth", options =>
{
    options.Cookie.Name = "MyCookieAuth";
    options.AccessDeniedPath = "/Account/AccessDenied";
});

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("MustBelongToHRDepartment", policy =>
    {
        policy.RequireClaim("Department", "HR");
    });
});

builder.Services.ConfigureApplicationCookie(Config =>
    {
        Config.LoginPath = "/Login";

    });
```

Figure 11: Configuration in Program.cs for cookie/session setup



Name	Value	Dom...	Path	Expi...	Size	Http...	Secu...	Sam...	Parti...	Cros...	Prior...
.AspNetCore.Antiforgery...	CfDJ8AQVV03imoxHgByjKKauttEeS...	local...	/	Sess...	190	✓		Strict			Med...
.AspNetCore.Identity.Ap...	CfDJ8AQVV03imoxHgByjKKauttEgcK...	local...	/	Sess...	806	✓	✓	Lax			Med...
MyCookieAuth	CfDJ8AQVV03imoxHgByjKKauttGtJd...	local...	/	Sess...	487	✓	✓	Lax			Med...

Cookie Value ☐ Show URL-decoded

```
CfDJ8AQVV03imoxHgByjKKauttGtJdxm4QJN0N7g9XMqptIXfu82CtFZ87xcoCjhupWZbBj--nJA0ZwOmLPwU2IrjCCyAUTp6RIGTSITKy8bEo8E5y8tfvkF2z
t27O06TuC_fyXyDlYAn48Ln0tkoHEp6jWcSSnmw5Q1iqbvNSncGNK-tvtds5Jjro2MnC7zWMIQdS2PatmAKIIWPQlQfuD8euoM6N9HXOBYYDqbnA0JdFy
qhTGD-0_wZoXtzphtKsuo_T21XETB1ABN7tQqSaOnf1jpl2aSb1KRY_HWYNjOmiFlppa2n9_blx_4SeS4UvyHNOJ4pBA-Y6uv_tM6dxXGv5bpp9rMSNCBb
hen8xJDlnNOKAloDocgx02uHZiJBFXZJ61AZt-5RJUr2CE9BP3lrFUE3WlKMumpG0wKEFQz53BUAAa3b8hIlzjgMElUxgyGU6tfwsRQIMUVzNXNzNHVPF
Q
```

Figure 12: Example of user login with claims-based authentication

This ensures cookies are properly configured, sessions are tied to claims, and unauthorized access is redirected securely.

2.2.4 Supporting Evidence/Justification

- This solution aligns with the **OWASP Session Management Cheat Sheet**, which requires secure cookie flags, session ID regeneration, and strong entropy for identifiers. Using claims-based authentication further enforces authorization policies, reducing the risk of hijacked sessions being misused.

2.3 Security Issues 2

- Credential Storage

2.3.1 Description

If login systems store or handle credentials insecurely (e.g., plaintext passwords), attackers can steal and reuse them. Replay attacks may also occur if tokens or credentials are reused without validation.

2.3.2 Mitigation Techniques

- Hash passwords using strong algorithms (e.g., bcrypt, Argon2, PBKDF2).
- Use salted hashes to prevent rainbow table attacks.
- Implement replay protection by invalidating old tokens and enforcing expiration.

2.3.3 Implementation Techniques

- ASP.NET Identity uses PBKDF2 by default but can be configured to use stronger algorithms like Argon2. Ensure passwords are hashed with a unique salt per user. Tokens should be single-use and expire quickly.

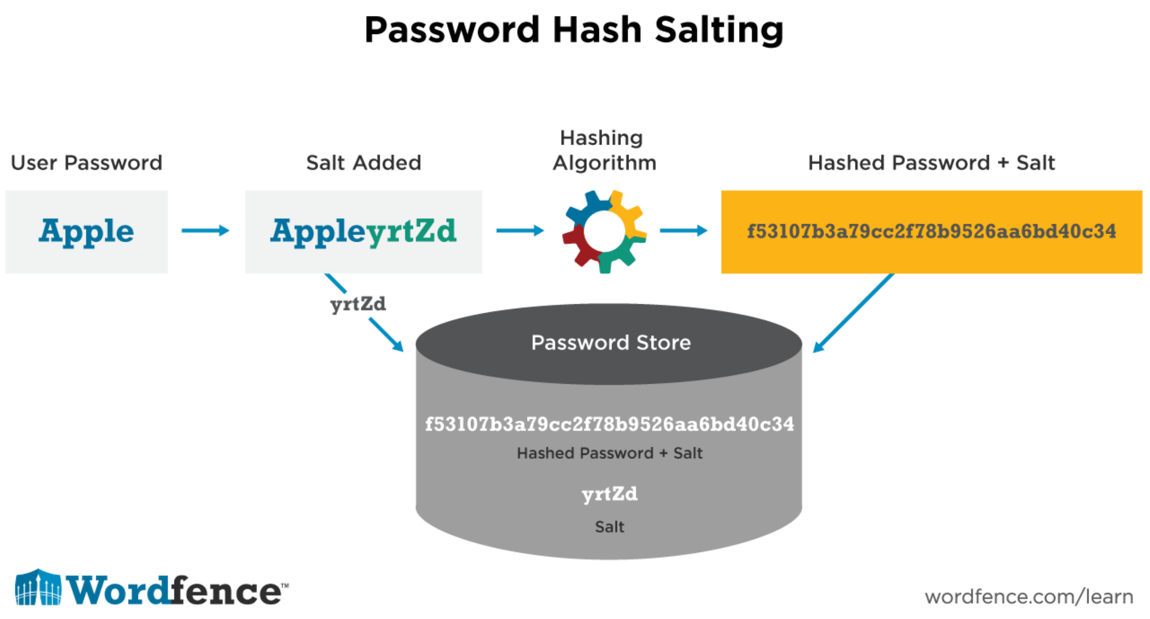
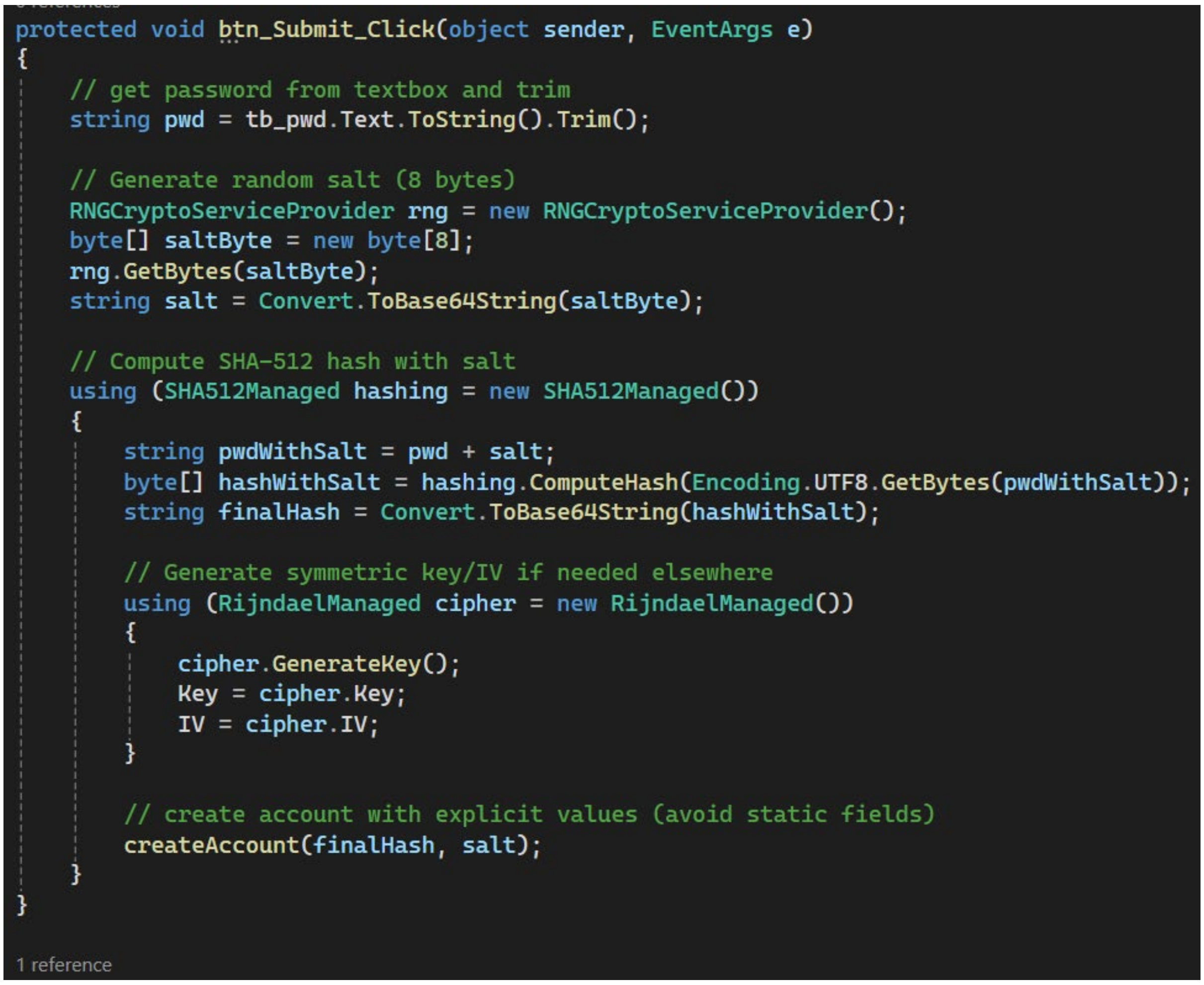
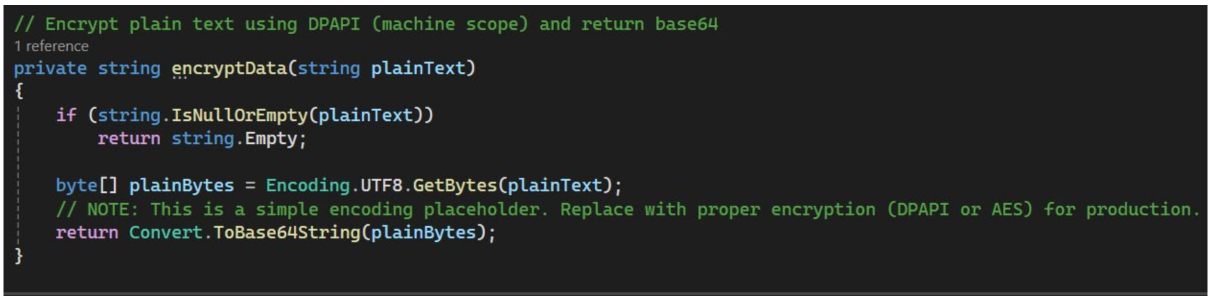


Figure 13: Flowchart of password hashing and salting process



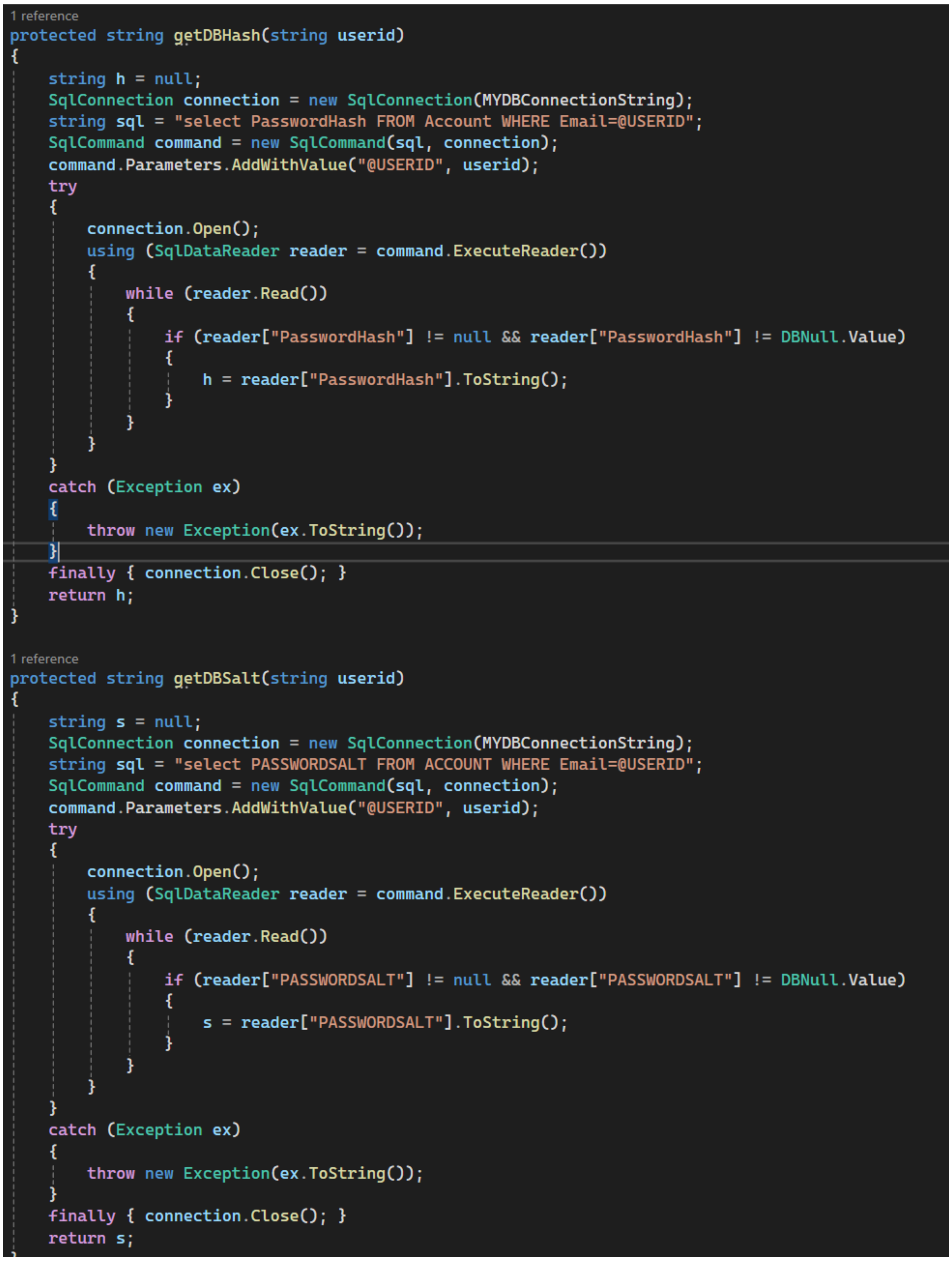
```
protected void btn_Submit_Click(object sender, EventArgs e)
{
    // get password from textbox and trim
    string pwd = tb_pwd.Text.ToString().Trim();

    // Generate random salt (8 bytes)
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
    byte[] saltByte = new byte[8];
    rng.GetBytes(saltByte);
    string salt = Convert.ToBase64String(saltByte);

    // Compute SHA-512 hash with salt
    using (SHA512Managed hashing = new SHA512Managed())
    {
        string pwdWithSalt = pwd + salt;
        byte[] hashWithSalt = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwdWithSalt));
        string finalHash = Convert.ToBase64String(hashWithSalt);

        // Generate symmetric key/IV if needed elsewhere
        using (RijndaelManaged cipher = new RijndaelManaged())
        {
            cipher.GenerateKey();
            Key = cipher.Key;
            IV = cipher.IV;
        }

        // create account with explicit values (avoid static fields)
        createAccount(finalHash, salt);
    }
}

1 reference
```

Figure 14: Example of password salting and hashing with SHA-512



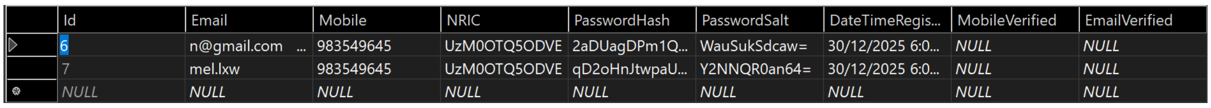
```
// Encrypt plain text using DPAPI (machine scope) and return base64
1 reference
private string encryptData(string plainText)
{
    if (string.IsNullOrEmpty(plainText))
        return string.Empty;

    byte[] plainBytes = Encoding.UTF8.GetBytes(plainText);
    // NOTE: This is a simple encoding placeholder. Replace with proper encryption (DPAPI or AES) for production.
    return Convert.ToBase64String(plainBytes);
}
```

Figure 15: Example of encrypting sensitive data (e.g., NRIC, mobile numbers)



```
1 reference
protected string getDBHash(string userid)
{
    string h = null;
    SqlConnection connection = new SqlConnection(MYDBConnectionString);
    string sql = "select PasswordHash FROM Account WHERE Email=@USERID";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.AddWithValue("@USERID", userid);
    try
    {
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                if (reader["PasswordHash"] != null && reader["PasswordHash"] != DBNull.Value)
                {
                    h = reader["PasswordHash"].ToString();
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
    finally { connection.Close(); }
    return h;
}

1 reference
protected string getDBSalt(string userid)
{
    string s = null;
    SqlConnection connection = new SqlConnection(MYDBConnectionString);
    string sql = "select PASSWORDSALT FROM ACCOUNT WHERE Email=@USERID";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.AddWithValue("@USERID", userid);
    try
    {
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                if (reader["PASSWORDSALT"] != null && reader["PASSWORDSALT"] != DBNull.Value)
                {
                    s = reader["PASSWORDSALT"].ToString();
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
    finally { connection.Close(); }
    return s;
```

Figure 16: Example of retrieving stored hash and salt securely from database



	Id	Email	Mobile	NRIC	PasswordHash	PasswordSalt	DateTimeRegis...	MobileVerified	EmailVerified
▷	6	n@gmail.com ...	983549645	UzM0OTQ5ODVE	2aDUagDPm1Q...	WauSukSdcaw=	30/12/2025 6:0...	NULL	NULL
	7	mel.lxw	983549645	UzM0OTQ5ODVE	qD2oHnJtwpaU...	Y2NNQR0an64=	30/12/2025 6:0...	NULL	NULL
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 17: Example showing password hashed and salted securely in database

2.3.4 Supporting Evidence/Justification

- This solution aligns with the **OWASP Password Storage Cheat Sheet**, which recommends salted, adaptive hashing algorithms and secure credential retrieval. It also follows **OWASP Cryptographic Storage Guidelines**, which emphasize strong encryption and secure key management. These practices prevent credential theft, replay attacks, and database compromise.

2.4 Security Issues 3

- Cross-Site Request Forgery

2.4.1 Description

CSRF attacks trick authenticated users into unknowingly submitting requests on their behalf, such as login or state-changing actions. Without protection, attackers can exploit the trust between the browser and the server to hijack sessions or perform unauthorized actions.

2.4.2 Mitigation Techniques

- Implement anti-CSRF tokens in all forms that perform state-changing actions.
- Validate tokens server-side for every POST request.
- Use SameSite cookies to prevent cross-site requests.
- Configure session services with strict timeouts to reduce exposure.

2.4.3 Implementations Techniques

- In ASP.NET Core, you can apply [ValidateAntiForgeryToken] to controller actions to enforce CSRF protection. The framework automatically generates and validates tokens. Session services are configured with strict idle timeouts to limit exposure.

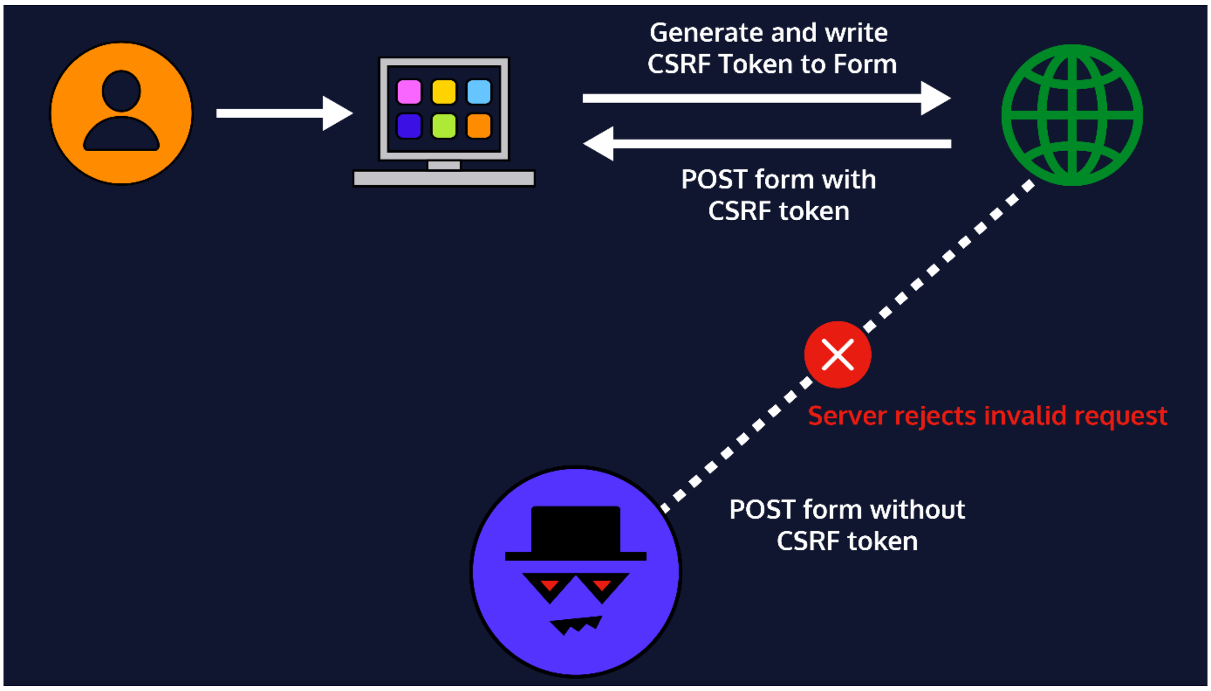
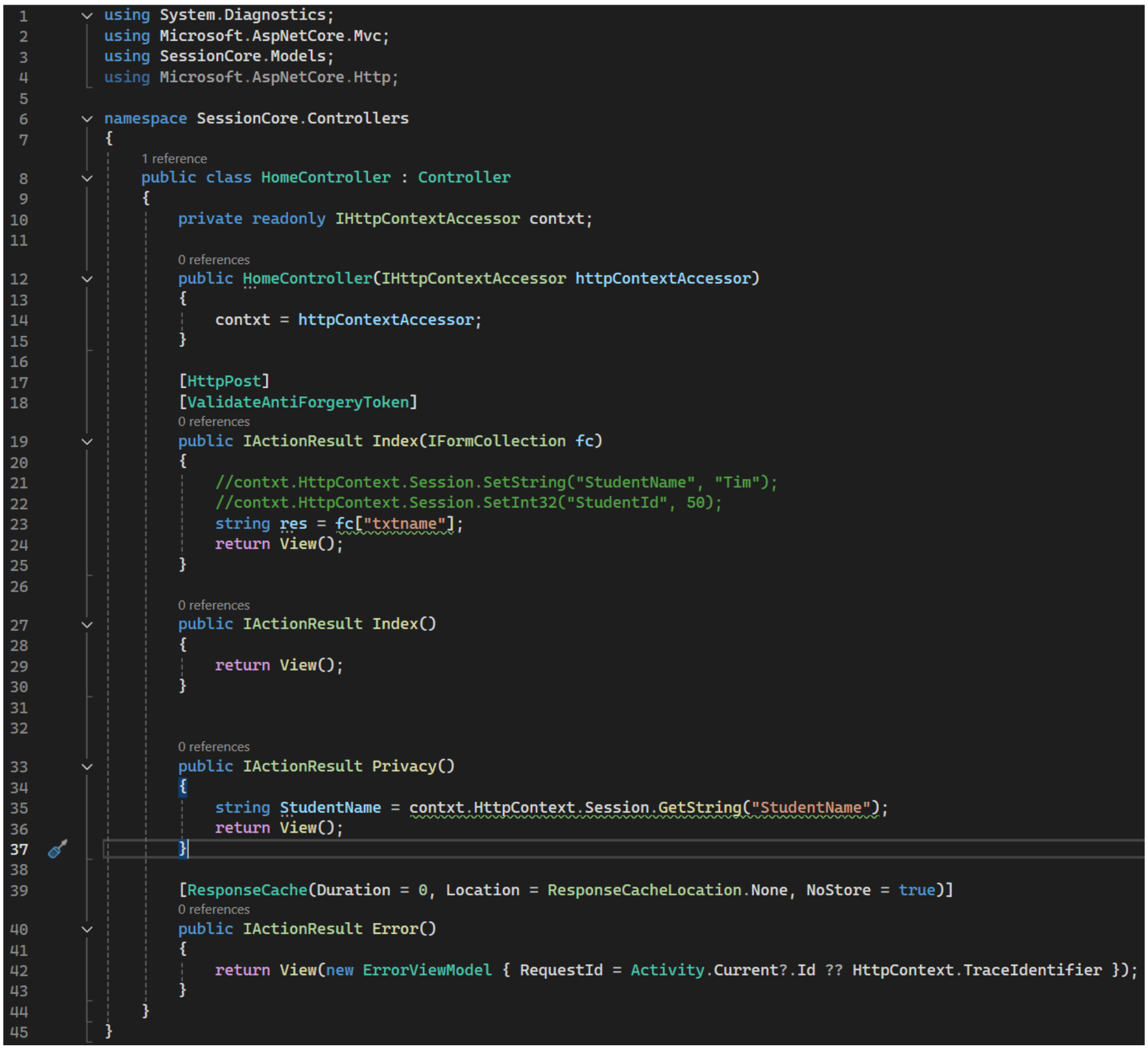
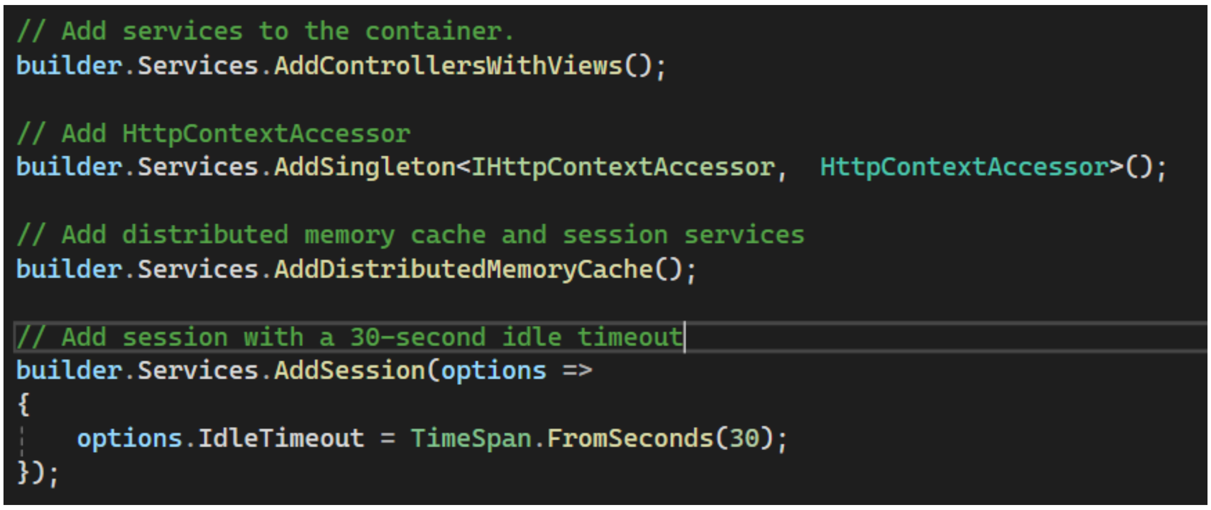


Figure 18: Flowchart showing CSRF token workflow



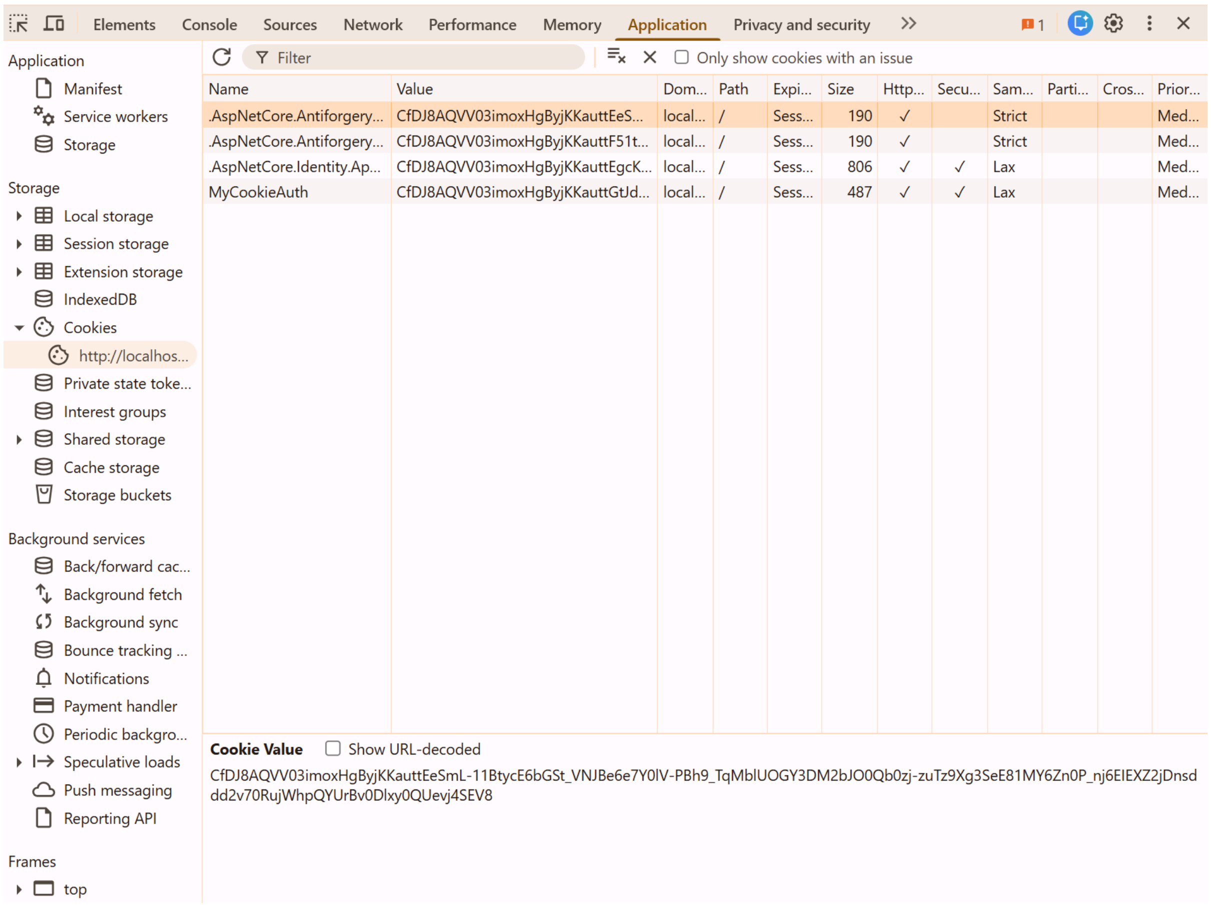
```
using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using SessionCore.Models;
using Microsoft.AspNetCore.Http;

namespace SessionCore.Controllers
{
    1 reference
    public class HomeController : Controller
    {
        private readonly IHttpContextAccessor contxt;

        0 references
        public HomeController(IHttpContextAccessor httpContextAccessor)
        {
            contxt = httpContextAccessor;
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        0 references
        public IActionResult Index(IFormCollection fc)
        {
            //contxt.HttpContext.Session.SetString("StudentName", "Tim");
            //contxt.HttpContext.Session.SetInt32("StudentId", 50);
            string res = fc["txtname"];
            return View();
        }

        0 references
        public IActionResult Index()
        {
            return View();
        }


        0 references
        public IActionResult Privacy()
        {
            string StudentName = contxt.HttpContext.Session.GetString("StudentName");
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        0 references
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Figure 19: Example of controller configuration with [ValidateAntiForgeryToken]



```
// Add services to the container.
builder.Services.AddControllersWithViews();

// Add HttpContextAccessor
builder.Services.AddSingleton<IHttpContextAccessor,  HttpContextAccessor>();

// Add distributed memory cache and session services
builder.Services.AddDistributedMemoryCache();

// Add session with a 30-second idle timeout
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(30);
});
```

Figure 20: Example of Program.cs configuration for CSRF/session services



Name	Value	Dom...	Path	Expi...	Size	Http...	Secu...	Sam...	Parti...	Cros...	Prior...
.AspNetCore.Antiforgery...	CfDJ8AQVV03imoxHgByjKKauttEeS...	local...	/	Sess...	190	✓		Strict			Med...
.AspNetCore.Antiforgery...	CfDJ8AQVV03imoxHgByjKKauttF51t...	local...	/	Sess...	190	✓		Strict			Med...
.AspNetCore.Identity.Ap...	CfDJ8AQVV03imoxHgByjKKauttEgcK...	local...	/	Sess...	806	✓	✓	Lax			Med...
MyCookieAuth	CfDJ8AQVV03imoxHgByjKKauttGtJd...	local...	/	Sess...	487	✓	✓	Lax			Med...

Cookie Value ☐ Show URL-decoded

CfDJ8AQVV03imoxHgByjKKauttEeSmL-11BtycE6bGSt_VNJBe6e7Y0lV-PBh9_TqMblUOGY3DM2bJO0Qb0zj-zuTz9Xg3SeE81MY6Zn0P_nj6EIEXZ2jDnsd
dd2v70RujWhpQYUrBv0Dlxy0QUevj4SEV8

Figure 21: Example of AntiForgeryToken in Razor view

2.4.4 Supporting Evidence/Justification

- This solution aligns with the **OWASP CSRF Prevention Cheat Sheet**, which recommends unique anti-CSRF tokens validated server-side. By enforcing token validation and strict session handling, forged requests are blocked, ensuring that login and other sensitive actions are intentional and user-initiated.

3. References

3.1 *Authentication cheat sheet*¶. Authentication - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

3.2 *Input validation cheat sheet*¶. Input Validation - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

3.3 *Owasp automated threats to web applications*. OWASP Foundation. (n.d.). https://owasp.org/www-project-automated-threats-to-web-applications/

3.4 *Session management cheat sheet*¶. Session Management - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

3.5 *Password storage cheat sheet*¶. Password Storage - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

3.6 *Cross-site request forgery prevention cheat sheet*¶. Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

3.7 *Transport Layer Security Cheat sheet*¶. Transport Layer Security - OWASP Cheat Sheet Series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_Sheet.html