Application Security Assignment Report

By: Trumen Lim (203771H)

https://github.com/trumenl/ApplicationSecurityAssgn_203771h

Client-side password complexity checker

A function validate() is created, and it checks all the basic requirements of the password such as the length of password being more than 12 characters, having a minimum of 1 number, 1 upper case character, 1 lower case character and 1 special case character respectively, by the use of “if” statements. If any of the “if” statements is passed through, the label “lbl_pwdchecker” will be changed to the string that is returned from the “if” statement, which shows the basic requirement that is not fulfilled.

```
<!-- client side validation -->
<script type="text/javascript">
    function validate() {
        var str = document.getElementById('<%=tb_password.ClientID %>').value;

        if (str.length < 12) {
            document.getElementById("lbl_pwdchecker").innerHTML = "Password length must be at least 12 Characters";
            document.getElementById("lbl_pwdchecker").style.color = "Red";
            return ("too short");
        }
        else if (str.search(/[0-9]/) == -1) {
            document.getElementById("lbl_pwdchecker").innerHTML = "Password requires at least 1 number";
            document.getElementById("lbl_pwdchecker").style.color = "Red";
            return ("no_number");
        }
        else if (str.search(/[A-Z]/) == -1) {
            document.getElementById("lbl_pwdchecker").innerHTML = "Password requires at least 1 upper case character";
            document.getElementById("lbl_pwdchecker").style.color = "Red";
            return ("no_uppercase");
        }
        else if (str.search(/[a-z]/) == -1) {
            document.getElementById("lbl_pwdchecker").innerHTML = "Password requires at least 1 lower case character";
            document.getElementById("lbl_pwdchecker").style.color = "Red";
            return ("no_lowercase");
        }
        else if (str.search(/[!@#$%^&*]/) == -1) {
            document.getElementById("lbl_pwdchecker").innerHTML = "Password requires at least 1 special character";
            document.getElementById("lbl_pwdchecker").style.color = "Red";
            return ("no_special");
        }

        document.getElementById("lbl_pwdchecker").innerHTML = "Excellent!";
        document.getElementById("lbl_pwdchecker").style.color = "Green";


    }
</script>
```

This validate() function is called using the OnKeyUp method, which is triggered whenever the user releases a key on the keyboard in the password field, thus constantly validating the password that the user is typing in.

```
Password:    <asp:TextBox ID="tb_password" runat="server"  OnKeyUp="validate()" Width="209px" ></asp:TextBox>
```

Server-side password complexity checker

```
1 reference
private int checkPassword(string password)
{
    int score = 0;

    if (password.Length < 12)
    {
        return 1;
    }
    else
    {
        score = 1;
    }

    if (Regex.IsMatch(password, "[a-z]"))
    {
        score++;
    }

    if (Regex.IsMatch(password, "[A-Z]"))
    {
        score++;
    }

    if (Regex.IsMatch(password, "[0-9]"))
    {
        score++;
    }



    if (Regex.IsMatch(password, "[@#$%&?]"))
    {
        score++;
    }




    return score;
}
```

For the server-side password complexity checker, it is implemented using a password scoring system, where the checkPassword() function is called whenever the check password button is clicked. This function takes in the password entered, and uses “if” statements to check the basic requirements. For each requirement passed, 1 is added to the score, and the score is eventually returned.

```
protected void btn_checkPassword_Click(object sender, EventArgs e)
{
    // implement codes for the button event
    // Extract data from textbox
    int scores = checkPassword(tb_password.Text);
    string status = "";
    switch (scores)
    {
        case 1:
            status = "Very Weak";
            break;
        case 2:
            status = "Weak";
            break;
        case 3:
            status = "Medium";
            break;
        case 4:
            status = "Strong";
            break;
        case 5:
            status = "Very Strong";
            break;
        default:
            break;
    }
    lbl_pwdchecker.Text = "Status : " + status;
    if (scores < 4)
    {
        lbl_pwdchecker.ForeColor = Color.Red;
        return;
    }
    lbl_pwdchecker.ForeColor = Color.Green;
}
```

Based on the scores, a switch case is implemented to return the status accordingly, which is displayed through the “lbl_pwdchecker” label.

Implementation of password protection

To protect the password, I will firstly generate a long random salt using a CSPRNG, and prepend the salt to the password and hash it with a standard cryptographic hash function such as SHA256. This can be done by the following below.

```
0 references
protected void btn_Submit_Click(object sender, EventArgs e)
{
    //string pwd = get value from your Textbox
    //httptutility.html encode for encoding, prevent xss
    string pwd = HttpUtility.HtmlEncode(tb_password.Text.ToString().Trim()); ;
    //Generate random "salt"
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
    byte[] saltByte = new byte[8];
    //Fills array of bytes with a cryptographically strong sequence of random values.
    rng.GetBytes(saltByte);
    salt = Convert.ToBase64String(saltByte);
    SHA512Managed hashing = new SHA512Managed();
    string pwdWithSalt = pwd + salt;
    byte[] plainHash = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwd));
    byte[] hashWithSalt = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwdWithSalt));
    finalHash = Convert.ToBase64String(hashWithSalt);
    RijndaelManaged cipher = new RijndaelManaged();
    cipher.GenerateKey();
    Key = cipher.Key;
    IV = cipher.IV;
    createAccount();
    Response.Redirect("Login.aspx", false);
}
```

```
static string finalHash;
static string salt;
byte[] Key;
byte[] IV;
```

The salt and hash is then stored in the database upon account creation.

```
cmd.Parameters.AddWithValue("@PasswordHash", finalHash);
cmd.Parameters.AddWithValue("@PasswordSalt", salt);
```

Encryption of credit card details

To encrypt the credit card details, I've used the rijndael algorithm and created a key, and generated the IV. Then, I convert the clear the text data to an array of bytes, and encrypt the clear text byte array.This can be seen from below.

```
// Use to encrypt credit card details (number, expiry date, cvv)
3 references
protected byte[] encryptData(string data)
{
    byte[] cipherText = null;
    try
    {
        RijndaelManaged cipher = new RijndaelManaged();
        cipher.IV = IV;
        cipher.Key = Key;
        ICryptoTransform encryptTransform = cipher.CreateEncryptor();
        byte[] plainText = Encoding.UTF8.GetBytes(data);
        cipherText = encryptTransform.TransformFinalBlock(plainText, 0, plainText.Length);
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
    finally { }
    return cipherText;
}
```

This function is called on the credentials as they are added into the database, to store the encrypted data and the IV.

```
cmd.Parameters.AddWithValue("@CC_Number", HttpUtility.HtmlEncode(Convert.ToBase64String(encryptData(cc_number.Text.Trim()))));
cmd.Parameters.AddWithValue("@CC_Date", HttpUtility.HtmlEncode(Convert.ToBase64String(encryptData(cc_date.Text.Trim()))));
cmd.Parameters.AddWithValue("@CC_CVV", HttpUtility.HtmlEncode(Convert.ToBase64String(encryptData(cc_cvv.Text.Trim()))));
```

Secure session (Fixed session issues) & Clean logout

To do this, after setting the normal Session variable, a GUID (a unique value and almost impossible to guess) can be created and saved as a new Session variable called AuthToken. The same GUID is then saved into a cookie named AuthToken.

```
Session["LoggedIn"] = HttpUtility.HtmlEncode(tb_userid.Text.Trim());
    //Response.Redirect("HomePage.aspx", false);



// Creates a new GUID ( a unique value & almost impossible to guess) and save it as a new seesion variable
//called AuthToken. This same GUID is then saved into a cookie named Authtoken
string guid = Guid.NewGuid().ToString();
    Session["AuthToken"] = guid;

    Response.Cookies.Add(new HttpCookie("AuthToken", guid));
```

Thereafter, on every page load, this cookie value will be matched with the session value. If both matches, the user will be allowed to enter the application, otherwise the user will be redirected to the login page. In short, the normal “LoggedIn” session, the new session variable called AuthToken and the new Cookie AuthToken are checked, and if all three are not null, the new session variable AuthToken and new Cookie AuthToken values are compared. If it matches, the users will be directed to the homepage, otherwise they will be redirected to the login page.

```
// Checks if user is logged in, if they are not, redirect to login page
0 references
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["LoggedIn"] != null && Session["AuthToken"] != null && Request.Cookies["AuthToken"] != null)
    {
        if (!Session["AuthToken"].ToString().Equals(Request.Cookies["AuthToken"].Value))
        {
            Response.Redirect("Login.aspx", false);
        }
        else
        {
            lblMessage.Text = "You have successfully logged in";
            lblMessage.ForeColor = System.Drawing.Color.Green;
            btnLogout.Visible = true;
        }
    }
    else
    {
        Response.Redirect("Login.aspx", false);
    }
```

Upon logout, apart from clearing the session, the ASP.NET_SessionId cookie is explicitly expired to make sure that this cookie is removed from the browser when the user clicks on the Logout button, and thereafter, the AuthToken cookie is explicitly expired as well.

Because of this, even if the ASP.NET_SessionId cookie value is known to the hijacker, he will not be able to login to the application as we are checking for the new Session value with the new cookie that is created by us and their GUID value is created by us. As such, a hijacker can know the Cookie value but he can't know the Session value that is stored in the web server level, and since this AuthToken value changes every time the user logs in, the older value will not work and the hijacker will not be able to even guess this value.

```
//Clear session and cookies when logging out
0 references
protected void LogoutMe(object sender, EventArgs e)
{

    Action = "Has logged out successfully";
    createLog();
    //Clear() removes all variables stored in session and
    //if user try to browse the site, same sessionID which was previously assigned to him will be used.
    Session.Clear();
    //Abandon() removes all variables stored in session, fire session_end event, and
    //if  user try to browse the site, a new sessionID will be assigned to it.
    Session.Abandon();
    Session.RemoveAll();
    Action = "Has logged out successfully";


    Response.Redirect("Login.aspx", false);

    if (Request.Cookies["ASP.NET_SessionId"] != null)
    {
        Response.Cookies["ASP.NET_SessionId"].Value = string.Empty;
        Response.Cookies["ASP.NET_SessionId"].Expires = DateTime.Now.AddMonths(-20);

    }

    if (Request.Cookies["AuthToken"] != null)
    {
        Response.Cookies["AuthToken"].Value = string.Empty;
        Response.Cookies["AuthToken"].Expires = DateTime.Now.AddMonths(-20);
    }
}
```

Session timeout

To do this, the following <sessionState mode = “InProc” timeout=”1”/> code is implemented in the web.config file. Here, the timeout is set to “1”, which means that there will be a session timeout when a minute has passed after the user has logged in. Once it is timeout, the user will be redirected to the login page again once he refreshes the page.

```
<configuration>
  <system.web>

      <!--Set session timeout to 1 min -->
      <sessionState mode = "InProc" timeout="1"/>

    <compilation debug="true" targetFramework="4.7.2" />
    <httpRuntime targetFramework="4.7.2" />
  </system.web>
```

Able to login after registration

Upon login, there will be decryption, in which the same algorithm that was used to encrypt the data is chosen. The key and IV which were used are retrieved, as well as the encrypted data. The data is then decrypted and converted back to its original format. If the hashes match, the user will be logged in.

```
protected void LoginMe(object sender, EventArgs e)
{


    string userid = HttpUtility.HtmlEncode(tb_userid.Text.ToString().Trim());
    string pwd = HttpUtility.HtmlEncode(tb_pwd.Text.ToString().Trim());
    SHA512Managed hashing = new SHA512Managed();
    string dbHash = getDBHash(userid);
    string dbSalt = getDBSalt(userid);

    try
    {
        //if (ValidateCaptcha())
        //{


            if (dbSalt != null && dbSalt.Length > 0 && dbHash != null && dbHash.Length > 0)
            {


            string pwdWithSalt = pwd + dbSalt;
                byte[] hashWithSalt = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwdWithSalt));
                string userHash = Convert.ToBase64String(hashWithSalt);

                if (userHash.Equals(dbHash))
                {

                Action = " Has logged in successfully";
                createLog();
                Session["LoggedIn"] = HttpUtility.HtmlEncode(tb_userid.Text.Trim());
                    //Response.Redirect("HomePage.aspx", false);
```

```
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)
                {
                    if (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)
                {
                    if (reader["PASSWORDSALT"] != DBNull.Value)
                    {
                        s = reader["PASSWORDSALT"].ToString();
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
    finally { connection.Close(); }
    return s;
}
```

Above are the getDBHash and getDBSalt methods which are used to retrieve the hash and salt from the database for comparison, based on the email entered by the user upon login.

Audit Log

Audit logs are compiled whenever a user has logged in, and logged out. In the case of the audit log of login, when the user has logged in, this createlog() function is called.

```
//audit log
1 reference
protected void createLog()
{
    try
    {
        using (SqlConnection con = new SqlConnection(MYDBConnectionString))
        {
            using (SqlCommand cmd = new SqlCommand("INSERT INTO AuditLog VALUES (@ActionLog, @Email, @DateTime)"))
            {
                using (SqlDataAdapter sda = new SqlDataAdapter())
                {
                    cmd.CommandType = CommandType.Text;
                    cmd.Parameters.AddWithValue("@Email", HttpUtility.HtmlEncode(tb_userid.Text.Trim()));
                    cmd.Parameters.AddWithValue("@DateTime", DateTime.Now);
                    cmd.Parameters.AddWithValue("@ActionLog", Action);
                    cmd.Connection = con;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    con.Close();
                }
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
}
```

This function goes inside the AuditLog table in the database and creates a log using the email of the user, as well as the datetime and action, in which case is “user has successfully logged in”.

The same can be done accordingly, for other logs such as account logout.

Implementation of reCaptcha V3 service

While frontend codes including the site and secret keys are being implemented, the backend features this ValidateCaptcha() function which is called during the login to verify is the user is indeed human, based on their actions.

```
//captcha
0 references
public bool ValidateCaptcha()
{
    bool result = true;

    string captchaResponse = Request.Form["g-recaptcha-response"];
    HttpWebRequest req = (HttpWebRequest)WebRequest.Create
        ("https://www.google.com/recaptcha/api/siteverify?secret=6Lf6a2UeAAAAAAXT-U8zE3af5llG8UcsWeB4zsZe &response" + captchaResponse);

    try
    {
        using (WebResponse wResponse = req.GetResponse())
        {
            using (StreamReader readStream = new StreamReader(wResponse.GetResponseStream()))
            {
                string jsonResponse = readStream.ReadToEnd();
                lbl_gScore.Text = jsonResponse.ToString();
                JavaScriptSerializer js = new JavaScriptSerializer();
                MyObject jsonObject = js.Deserialize<MyObject>(jsonResponse);
                result = Convert.ToBoolean(jsonObject.success);

            }
        }
        return result;
    }
    catch (WebException ex)
    {
        throw ex;
    }

}
```

Prevention of SQLI and XSS

To prevent SQLI, parameterized queries are implemented for each value that is added into the database, as can be seen below.

```
cmd.Parameters.AddWithValue("@Email", HttpUtility.HtmlEncode(tb_userid.Text.Trim()));
```

To prevent XSS, encoding is also implemented whenever value from input fields are being read, as can be seen below.

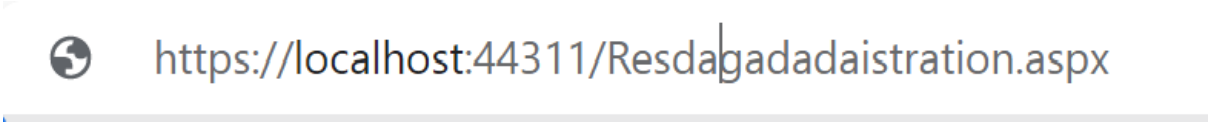
```
string userid = HttpUtility.HtmlEncode(tb_userid.Text.ToString().Trim());
```

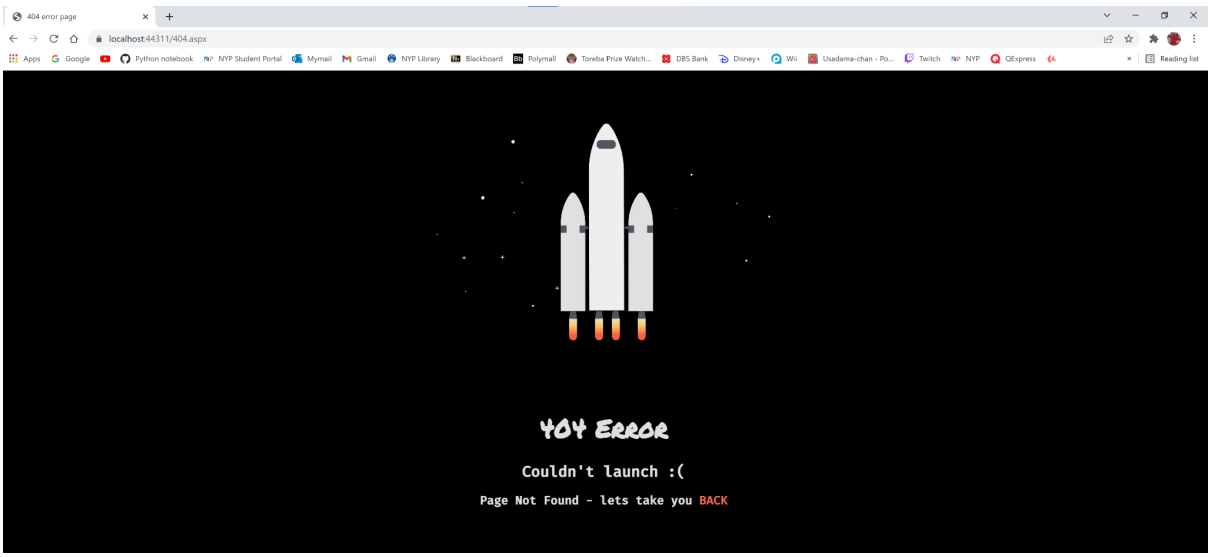
Proper error handling

To do this, simply remove the error status codes and add in redirects which links to the custom error pages accordingly, in the web.config file.

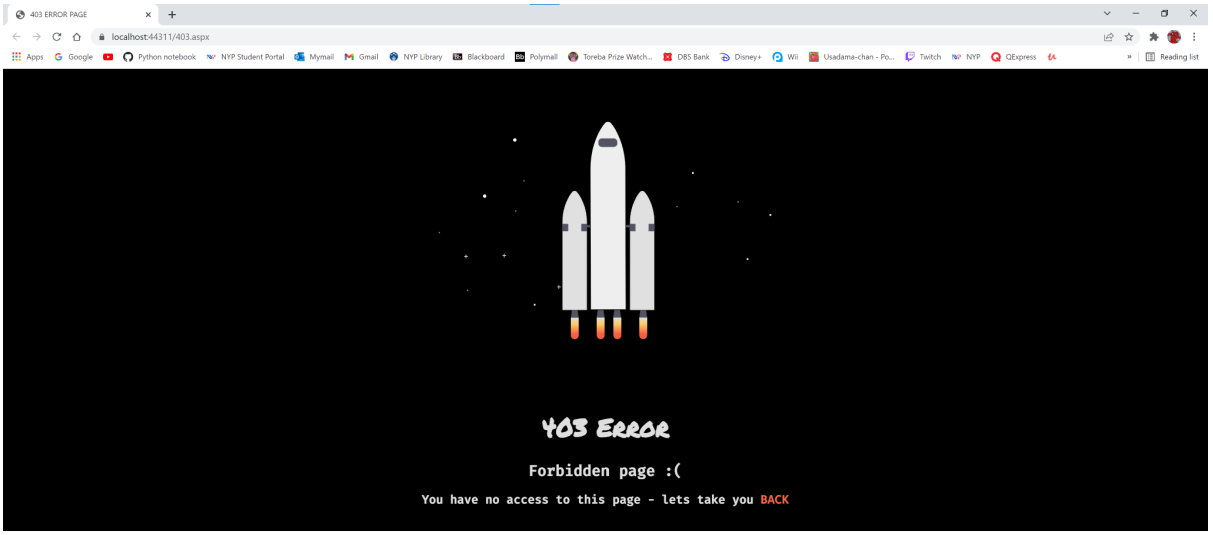
```
<system.webServer>

    <httpErrors errorMode="Custom" existingResponse="Replace">
        <remove statusCode="404"/>
        <error statusCode="404" path="/404.aspx" responseMode="Redirect"/>
        <remove statusCode="403"/>
        <error statusCode="403" path="/403.aspx" responseMode="Redirect"/>
    </httpErrors>


</system.webServer>
```

Test case for error.404: Change the address to make it invalid.





Test case for error.403: Run the web.config file itself. Since it is forbidden to enter the file in web browser, error.403 will occur.



2 Factor authentication for login

For 2 factor authentication in login, after entering the userid and password, the user will also have to enter a randomly generated code sent to their emails for further authentication before they are logged in. To do this, during the login, if the userid and password matches the one in the database, a random number code is created. This random code is then added into the database using this CreateOTP() function. An email with the verification code is then sent to the user's email, using this verificationcode() function, while the user is redirected to the verification page.

```
Random random = new Random();
RandomNumber = random.Next(000000, 999999).ToString();

CreateOTP(userid, RandomNumber);

verificationcode(RandomNumber);


Response.Redirect("Verification.aspx", false);
```

```
//adds rng otp to database
1 reference
protected string CreateOTP (string userid, string RandomNumber)
{
    string otp = null;
    SqlConnection con = new SqlConnection(MYDBConnectionString);
    string sql = "update Account set Verification = @Verification where Email=@Email";
    SqlCommand cmd = new SqlCommand(sql, con);
    cmd.Parameters.AddWithValue("@Verification", RandomNumber);
    cmd.Parameters.AddWithValue("@Email", userid);

    try
    {
        con.Open();
        using(SqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                if (reader["Verification"]!= null)
                {
                    otp = reader["Verification"].ToString();
                }
            }
        }
    }

    catch (Exception ex)
    {
        throw new Exception(ex.ToString());

    }

    finally { con.Close(); }

    return otp;
}



//Uses smtp mail server, sends verification code to user's email
1 reference
protected string verificationcode(string verifycode)
{
    string senderAddress = "Trumen <limtrum1@gmail.com>";

    string str = null;



   var smtpClient = new SmtpClient("smtp.gmail.com")
    {
        Port = 587,
        Credentials = new NetworkCredential("limtrum1@gmail.com", "Rebelt7i@123"),
     EnableSsl = true,
    };

    var messageSend = new MailMessage
    {
        Subject = "Verification",
        Body = "This is your verification code " + verifycode + " for SITCONNECT, Thank you!"
    };

    messageSend.To.Add(tb_userid.Text.ToString());
    messageSend.From = new MailAddress(senderAddress);

    try
    {
        smtpClient.Send(messageSend);
        return str;
    }

    catch
    {
        throw;
    }


}
```

In the verification page, this VerifyCode() function is used to retrieve the verification code in the database.

```
//retrieves the verification code from database
1 reference
protected string vCodeOTP(string email)
{
    string otp = null;
    SqlConnection con = new SqlConnection(MYDBConnectionString);
    string sql = "select Verification from Account where Email = @EMAIL";
    SqlCommand cmd = new SqlCommand(sql, con);
    cmd.Parameters.AddWithValue("@EMAIL", email);
    try
    {
        con.Open();
        using (SqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                otp = reader["Verification"].ToString();
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.ToString());
    }
    finally
    {
        con.Close();
    }
    return otp;
}
```

The VerifyCode() function is then used upon clicking the verify button, to check if the verification code entered matches the one in the database. If it matches, the user will be directed to the home page.

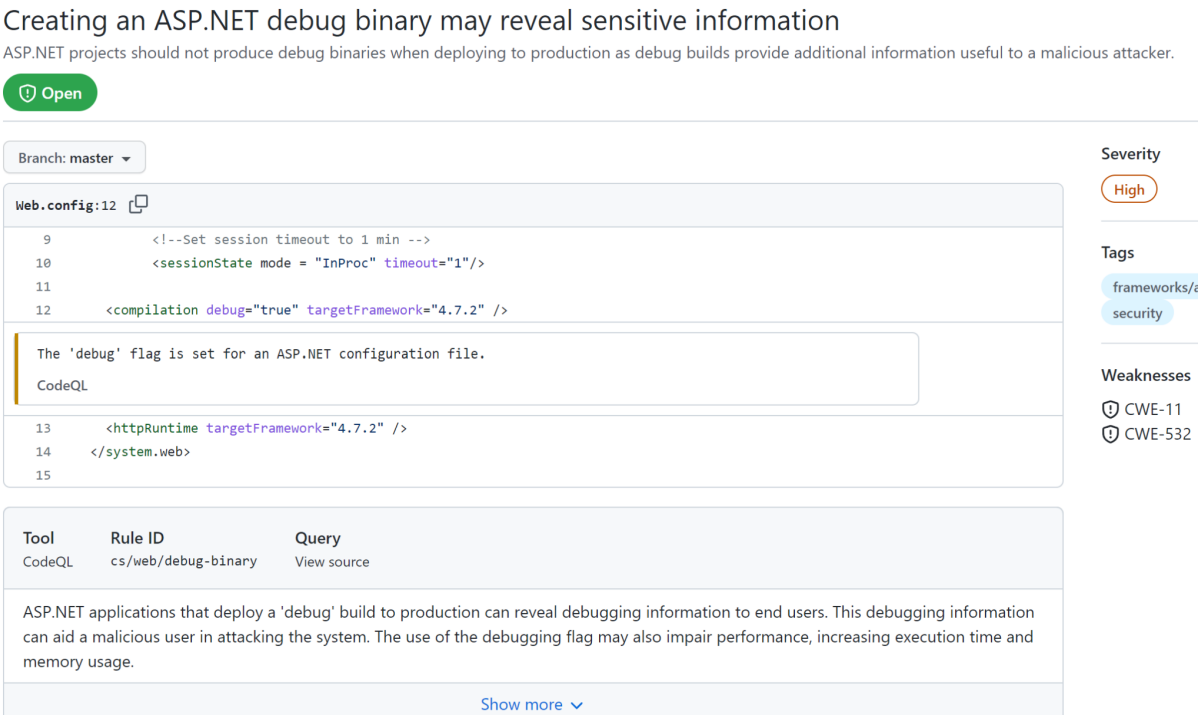
```
//checks if verification code entered is the same as one in database, if so, pass
0 references
protected void VerifyCode(object sender, EventArgs e)
{
    if (HttpUtility.HtmlEncode(verification_code.Text.ToString()) == vCodeOTP(Session["LoggedIn"].ToString()))
    {


        Response.Redirect("HomePage.aspx", false);

    }
    else
    {
        lbl_message.Text = "Incorrect verification code!";
    }
}
```

Change password

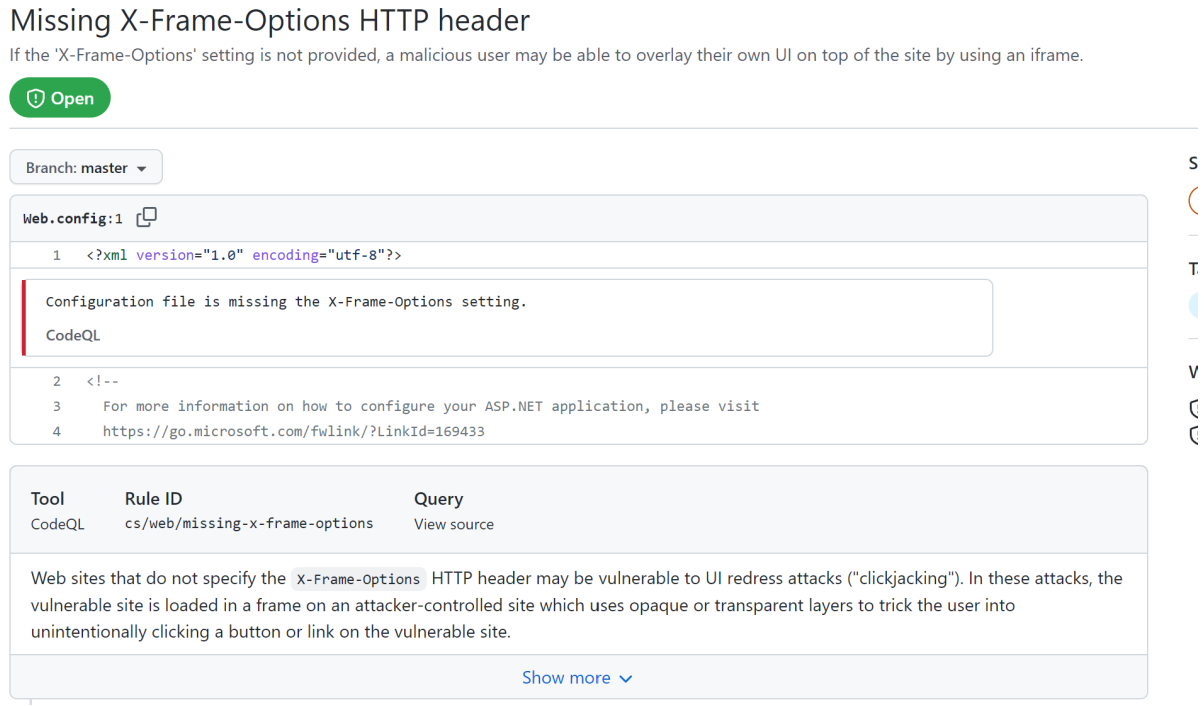
To do this, the new password entered by the user will undergo the same process as the one in implementation of password protection, and the resulting salt and hash will be updated the replace the existing one in the database.

```
//hashes new password entered and updates it in database
0 references
protected void ChangePassword(object sender, EventArgs e)
{


    //string MYDBConnectionString = System.Configuration.ConfigurationManager.ConnectionStrings["MYDBConnection"].ConnectionString;

    //SqlConnection connection = new SqlConnection(MYDBConnectionString);

    string sql = "UPDATE [Account] SET [PasswordHash] = @PasswordHash, [PasswordSalt] = @PasswordSalt WHERE Email = @USERID";
    //SqlCommand command = new SqlCommand(sql, connection);

    using (SqlConnection con = new SqlConnection(MYDBConnectionString))
    {

        using (SqlCommand cmd = new SqlCommand(sql))
        {
            using (SqlDataAdapter sda = new SqlDataAdapter())
            {

                //string pwd = get value from your Textbox
                string pwd = ChangePasswd.Text.ToString().Trim();
                //Generate random "salt"
                RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
                byte[] saltByte = new byte[8];
                //Fills array of bytes with a cryptographically strong sequence of random values.
                rng.GetBytes(saltByte);
                salt = Convert.ToBase64String(saltByte);
                SHA512Managed hashing = new SHA512Managed();
                string pwdWithSalt = pwd + salt;
                byte[] plainHash = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwd));
                byte[] hashWithSalt = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwdWithSalt));
                finalHash = Convert.ToBase64String(hashWithSalt);
                RijndaelManaged cipher = new RijndaelManaged();
                cipher.GenerateKey();
                Key = cipher.Key;
                IV = cipher.IV;

                cmd.CommandType = CommandType.Text;
                cmd.Parameters.AddWithValue("@PasswordHash", finalHash);
                cmd.Parameters.AddWithValue("@PasswordSalt", salt);
                cmd.Parameters.AddWithValue("@USERID", Session["LoggedIn"].ToString());

                cmd.Connection = con;
                con.Open();
                cmd.ExecuteNonQuery();
                con.Close();

            }
        }
    }

    password_message.Text = "Your password have been changed.";
}
```

Source code testing issues and mitigation techniques



ASP.NET projects should not produce debug binaries when deploying to production as debug builds provide additional information useful to a malicious attacker. To resolve this, the code ‘debug=”true”’ has been removed.



Web sites that do not specify the `X-Frame-Options` HTTP header may be vulnerable to UI redress attacks ("clickjacking"). In these attacks, the vulnerable site is loaded in a frame on

an attacker-controlled site which uses opaque or transparent layers to trick the user into unintentionally clicking a button or link on the vulnerable site.

As such, to resolve this, the `X-Frame-Options` HTTP header has been set to `DENY`, to instruct web browsers to block attempts to load the site in a frame.