UDC: 004.65:004.772.056.5
Professional paper

SECURING MVC-BASED LMS PLATFORMS: ADDRESSING AUTHENTICATION, XSS, AND INJECTION VULNERABILITIES

Asri NUHI, Neshat AJRULI, Florim IDRIZI, Florinda IMERI, Agon MEMETI*

Department of Computer Sciences, Faculty of Natural Sciences and Mathematics University of Tetova, Tetova, North Macedonia
** Corresponding Author, e-mail: agon.memeti@unite.edu.mk*

Abstract

This article presents the most critical security weaknesses of Learning Management Systems (LMS) based on Model-View-Controller (MVC) architecture and a study case from the LMS at University of Tetova. Right down to the weaknesses in authentication and authorization systems (for example, weak passwords or lack of access controls). Furthermore, the study addresses the issue of Cross-Site Scripting (XSS) based, focal and reflected XSS - and how SQL injection threats also impact database security. What makes our work original is of course the case based and pragmatic approach where we dissect real world vulnerabilities in gap analysis and then recommend particular countermeasures (e.g., role base authorization, parameterized query execution). Based on the results, recommendations for LMS security and the confidentiality of educational data are provided.

Keywords: IP addresses, GlassWire Firewall, LMS, Error Logs, Cybersecurity.

1. Introduction

Learning Management Systems (LMS) play the role of distributing and managing educational content. Initially created with a focus on instruction, these systems have since expanded to feature the capability of monitoring student progress and administrative purposes. Nevertheless, in spite of being learning institutions' basic structures, LMSs has not remained in a vacuum immune to the dynamics running on this world of cyber-threat as we witness it being slashed (most often without success) every other day by threats that agitate its users private data and educational process [1]. Consequently, their security is the highest priority.

This work examines the security problems of LMS upon a Model-View-Controller (MVC) architecture model referencing Tetova University that has been given as an example. The aim is to present an extensive analysis on the major security problems, focusing deeply on processes of authentication&authority, Cross-Site Scripting (XSS) attacks as well as Code Injection vulnerabilities.

The paper is organized as follows: Authentication and Authorization of LMS Platform: discusses how an authenticated state can be established for a user inside an LMS, including how users are identified and how resource access is managed; Common Authentication and Authorization Vulnerabilities: we list vulnerabilities often associated with authentication e.g. weak password policy, poor session management, flaws in the use of access control system; Cross-Site Scripting (XSS) Attacks: In this section variety types of XSS are considered i.e., stored XSS attacksqwerty Applications Protocols; Injection Attacks: We cover Injection attacks with the main focus being on SQL Injection as a case study and how this can be used to break into database security and a Case Study. To apply this issue, this section introduces a case study to show how you can implement these mechanisms suggested in their mitigations and present both vulnerability management as well as common mitigation techniques.

We outline the significant areas of LMSs that need to be known and understood in lieu of security and the actions system administrators or developers can undertake to ensure their LMS are protected from common web application attacks. It also contributes to literature by

providing an LMS security vulnerability assessment of top-to-bottom practical testing experience on the MVC technology architecture. The article also offers pragmatic resolutions, such as role-based authorization, parameterized queries use and other risk reducers.
Here we present the LMS at the University of Tetova and offer real examples in terms of security issues and management solutions for an academic system. It fills an important gap in the literature by bridging security theory with practical recommendations specific to LMS settings.

2. Related works

Based on [2] authors show that preventing sql injection protects your databases from the most prevalent type of attack. The guide also mentions that Parameterized queries and stored procedures are the first line of defense as they "separate" user input from the SQL statement, making SQL injection attacks impossible. It also highlights the need for input validation and error handling, reminding that these are not utilities that mitigate (but somewhere allows) injection attacks. This is an important source in the practical perspective of securing database operations in software development.
Next based on [3] and [4] authors explain the procedure for configuring roles and policies to limit user permissions accordingly which can also serve as an ASP. NET security in general - or as a refresher: It includes form-based and federated security, claims-based identity and authorization; Covers the important features of ASP. NET Core’s security framework. [4] shows that there is a list of dynamic and pragmatic access control policies that give you robust to tailored rules in your application. It explains key concepts, like least privilege, separation of duties, mandatory access controls. The cheat sheet recommends designing for secure operation, i.e., avoiding hard-coded permissions/control as well as taking inconsistency of policy enforcement between apps into account. This paper serves as a clear prescription for developers and security professionals looking to make the authorization of their application more secure.
Complete list of organizational and information systems security and privacy controls is given at NIST Special Publication 800-53. The document addresses a broad and thorough set of controls that encompass security categories such as access control, incident response, system and communications protection. It offers guidelines for developing a security posture that is aligned with federal policy best practices. The document is authored for use by enterprises that are endeavoring to develop and execute an effective security program that addresses compliance needs and [5].

3. Implementing authentication and authorization in an mvc-based lms

3.1. Authentication and Authorization: In this LMS login we have the authentication process starting at OnPostAsync which is activated on the user clicking the submit button of the form. The approach begins by validating the form data. If the model state for the model is valid, then we use SignInManager PasswordSignInAsync function to check for user's sign in based on Email and Password supplied. This function checks the credentials against users data and returns whether the login attempt has completed successfully or not. The RememberMe option is also respected, enabling users to persist logged in between sessions when they check that option.
Importantly, the implementation includes an option to trigger account lockout on multiple failed login attempts, though it is currently disabled (lockoutOnFailure: false).
Authorization and Role-Based Redirection: Upon successful authentication, the code proceeds to the authorization phase, where it determines the appropriate role-based redirection for the logged-in user. The FindByEmailAsync method of the UserManager retrieves the user's details

based on their email address. The user's roles are then checked using IsInRoleAsync, and the user is redirected to different sections of the LMS based on their assigned role. For instance, users assigned to the "Zyra për Arsim" role are redirected to the /za page, while those in the "Profesor" role are redirected to the /professor page. This role-based redirection ensures that users are directed to the appropriate area of the application based on their responsibilities and permissions.

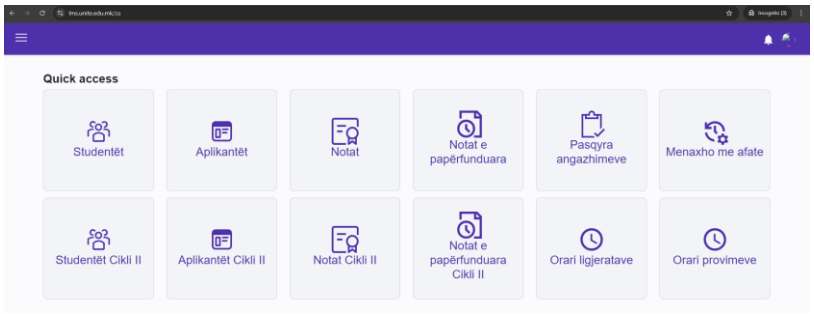

Figure. 1. Page of role, office for education profile

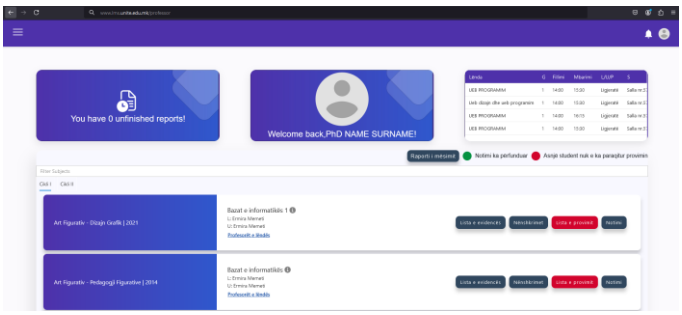

Figure. 2. Page of role, professor profile

Handling Failed Logins and Logging: A failed login is captured in a model state error message after which the log-in page is redisplayed with an appropriate error message to enable users to fix their credentials. The code also has logging (using ILogger) which logs an entry that the user X successfully logged in. This logging is used to audit an account and for looking at login activity. If the login fails, they redisplay the form with an error message that says something like "Invalid sign-in" that lets the user know the combination of username and password was not valid (but we can't say it outright) so they can try again. The system increases the security while making the user happy by reading both failed and successful login attempts.

Authentication within the LMS is done using an MVC pattern by leveraging ASP.NET Core Identity. The LoginModel class manages user login by declaring dependencies like UserManager, SignInManager, and logging services, which are injected through the constructor. The user input is received using a strongly-typed InputModel with fields for email, password, and a "Remember Me" checkbox. The OnPostAsync method, the heart of the class, validates the form data submitted and calls the PasswordSignInAsync method to authenticate the user. Upon successful authentication, the system conducts role-based redirection to the relevant sections of the LMS - for instance, redirecting professors to the "/professor" page or students to their profile page. Unsuccessful login attempts result in model errors, providing feedback to the users. Logging functionality records successful logins for auditing. This method enhances usability and security by effectively handling roles and session states.

The PasswordSignInAsync method authenticates a user asynchronously by validating their password and email (or username). Its asynchronous nature makes it non-blocking, hence very effective in handling sign-in requests. The method receives a number of parameters: the email or username of the user, password, a RememberMe boolean to decide whether the login should

be remembered using a cookie, and lockoutOnFailure, which specifies whether the account should be locked out in case of repeated failures.
This method returns a SignInResult object that indicates the result of the login operation. Upon success, an authentication cookie is issued, logging the user in. When the login fails—because of invalid credentials or an invalid account—the method returns a failure result that can be used by the application to show an error message. If lockoutOnFailure is true, the system will lock out the account after too many invalid attempts, increasing security. Inadequate password policies create significant security vulnerabilities by permitting easily guessable or crackable passwords. Short passwords, or those that are not complex or do not contain a mix of character types, are susceptible to automated and social engineering attacks. For instance, passwords that are fewer than eight characters long or consist only of lowercase letters are far less secure than those that include a mix of uppercase letters, numbers, and special characters. Strong password policies need to be enforced to block unauthorized access.
ASP.NET Core Identity also supports the setup of strong password policies for enhanced security, as can be seen in the next code snippet from an ASP.NET Core application:

```
builder.Services.Configure<IdentityOptions>(options =>
{
  options.Password.RequiredLength = 6;
  options.Password.RequireLowercase = true;
  options.Password.RequireUppercase = true;
  options.Password.RequireNonAlphanumeric = false;
  options.Password.RequireDigit = true;
  options.Password.RequiredUniqueChars = 6;
})
```

a) Cross-Site Scripting (XSS) Attacks

Cross-site scripting (XSS) is a type of weakness that enables attackers to inject malicious scripts into web pages which are of interest by other users, thus creating various harmful effects, including theft of data, session hijacking, or even web content corruption. XSS attacks exploit a user's trust in a web page or application. The main types of XSS attacks are [7]:

- Stored XSS: This occurs when an attacker’s malicious script is permanently stored on the target server and then it is served to users who visit the affected pages e.g., an attacker might insert a script into a user profile that gets executed whenever users view that profile [8].
- Reflected XSS: This attack occurs when malicious scripts embedded in a server response are executed by the victim's browser, reflected by a web server via a URL or request parameter, e.g. an attacker could send a URL that includes malicious code, which is executed when the victim clicks the link. [9].
- DOM-based XSS: In this variant, the attack occurs entirely on the client side, where the malicious script is executed because of modifying the document object model (DOM) of the page, usually through JavaScript. This type of XSS does not require server-side involvement, and the scripts are executed on the client side by manipulating the DOM in an insecure way [10], which is illustrated below:

```
<!-- Vulnerable to XSS -->
 <div class="container-fluid d-flex align-items-center">
    <div class="row">
       <div class="col-lg-7 col-md-10">
          <h1 class="display-2 text-white">Hello @StudentComplete.emri @StudentComplete.mbiemri</h1> <!-- User input is not encoded -->
       </div> </div> </div>
```

A very effective strategy for preventing cross-site scripting (XSS) attacks is to use HTML encoding, which transforms special characters in user input into their corresponding HTML entities, thereby preventing the browser from interpreting these characters as executable code. By encoding user input, you ensure that any potentially malicious code entered by users is treated as plain text rather than as part of the HTML or JavaScript, thereby neutralizing the risk of XSS attacks [11].

b) Injection Attacks

Injection attacks, specifically SQL Injection, happen when an attacker is able to input or modify queries directly into fields or parameters resulting in data manipulation/ deletion and coming from the database. Such attacks leverage the weaknesses in query forming and processing, which leads to unauthorized access or data mutilation. Adequate treatment of SQL Injection is essential to ensuring database security and integrity [12].
Parameterized queries are a strong safeguard against SQL injection. Below is a code example where I'm using parameterized queries to safely integrate input provided by a user-provided in an SQL command so that I don't end up vulnerable to an injection attack:

```
public Task<List<ProfessorProfileData>> GetProfessorData(string staffId)
{
   var parameters = new { StaffId = staffId };
   string sql = @"SELECT *, staf.staff_id, staff_translation.first_name, staff_translation.last_name, staf.image, staf.email, staf.personal_number, staf.date_of_birth, staf.gender, staf.status, ts.scientific_call, staf.staff_type, nationality.nationality, countries.country
        FROM staff
        INNER JOIN staff_translation ON staff.staff_id = staff_translation.staff_id
        INNER JOIN nationality ON staff.nationality_code = nationality.nationality_code
        INNER JOIN countries ON staff.country_code = countries.country_code
        INNER JOIN active_contracts k ON k.staff_id = staff.staff_id AND k.status = 'Active'
        INNER JOIN scientific_calls ts ON ts.scientific_call_code = k.scientific_call_code AND ts.language = 'en'
```

```
        WHERE staff.staff_id = @StaffId AND nationality.language =
'en' AND countries.language = 'en' AND staff_translation.language =
'en'";

   return _db.LoadData<ProfessorProfileData, dynamic>(sql,
parameters);
}
```

4. Case study: security best practices in learning management systems

Email verification during user registration is an important security practice to verify the legitimacy of user accounts. With a second step of confirming the email address is even verified that the user not only controls an e-mail account, but also his legitimate owner of this account. This makes sure that the users can be replied to by email, and also brings a prevention to fake accounts and spams. It also offers additional security by allowing access only to the authentic users to LMS [13].
Here I am using most of asp. YYSTACK.NET (C#) and MS Access whereas you are using pure ASP. NET Core Identity:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
   returnUrl = returnUrl ?? Url.Content("~/");
   ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
   if (ModelState.IsValid)
   {
      var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
      var result = await _userManager.CreateAsync(user, Input.Password);
      if (result.Succeeded)
      {
         _logger.LogInformation("User created a new account with password.");
         var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
         code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
         var callbackUrl = Url.Page(
            "/Account/ConfirmEmail",
            pageHandler: null,
            values: new { area = "Identity", userId = user.Id, code = code, returnUrl =
returnUrl },
            protocol: Request.Scheme);
         using (MailMessage mail = new MailMessage())
         {
            mail.IsBodyHtml = true;
            mail.From = new MailAddress("email@unite.edu.mk", "UT E-System");
            mail.To.Add(Input.Email);
            mail.Subject = $"Confirm your email";
            mail.Body = $"Please confirm your account by <a href =
'{HtmlEncoder.Default.Encode(callbackUrl)}'> clicking here </a>.";
```

```
            using (SmtpClient smtp = new SmtpClient("smtp.gmail.com", 587))
            {
                smtp.UseDefaultCredentials = false;
                smtp.Credentials = new
System.Net.NetworkCredential("email@unite.edu.mk", "Password");
                smtp.EnableSsl = true;
                smtp.Send(mail);
            }
        }
        if (_userManager.Options.SignIn.RequireConfirmedAccount)
        {
            return RedirectToPage("RegisterConfirmation", new { email =
Input.Email, returnUrl = returnUrl });
        }
        else
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
    }
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError(string.Empty, error.Description);
    }
  }
  // If we got this far, something failed, redisplay form
  return Page();
}
```

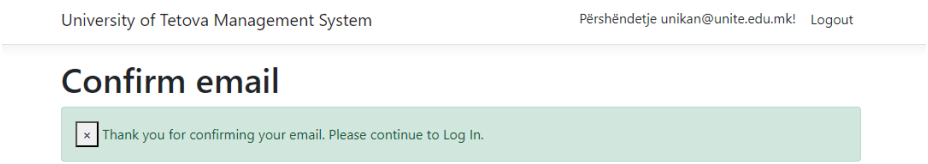

Figure 3. Account confirmation

4.1 Implementing Role-Based Authorization: User role is an essential security concept in web application design & development and Role-Based Access Control (RBAC) can be widely used to mitigate the risks. You can allow or deny different users access to various parts of your application and what they can do with it by attributing roles to the users. This not only eases permission management but also increases security by preventing users from accessing any feature or data that are irrelevant to their role [14].

Implementation Example:

The following code snippet demonstrates how to implement role-based authorization in an ASP.NET Core MVC application:

```
class HomeController inherits Controller

    // Action for Professors
    method ProfesorOnly() with authorization for role "Professor"
        return View()

    // Action for Students
    method StudentDashboard() with authorization for role "Student"
        return View()

    // Action for Referents
    method ReferentDashboard() with authorization for role "Referent"
        return View()
```

Class Definition: HomeController inherits from the base Controller class.

- Action Methods:
    - ProfesorOnly: Accessible only to users in the "Professor" role. It returns a view specific to professors.
    - StudentDashboard: Accessible only to users in the "Student" role. It returns the student dashboard view.
    - ReferentDashboard: Accessible only to users in the "Referent" role. It returns the referent dashboard view.

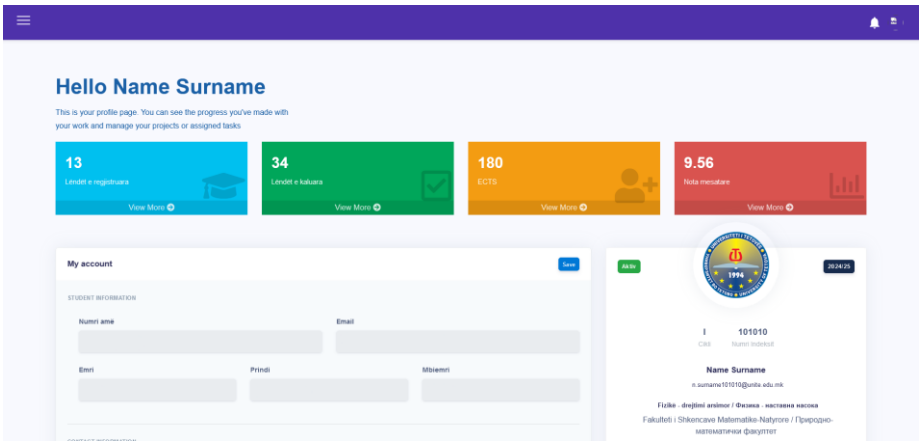

Figure 4. Student dashboard

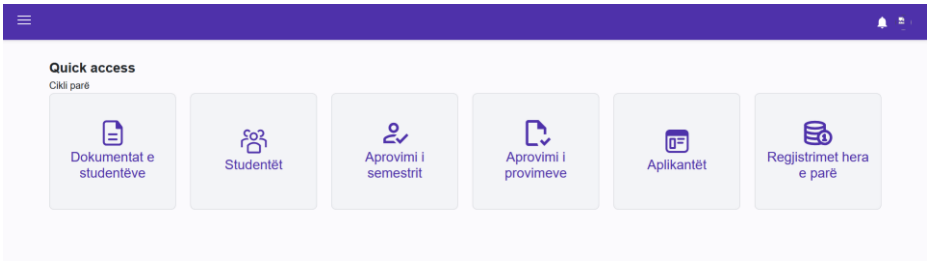

Figure 5. Admin dashboard

Every method is protected with an authorization attribute that limits the access based on user roles. Every method also returns a view based on the particular user role, keeping the implementation simple and obvious.

5. Conclusion

Securing the MVC model of the Learning Management Systems is critical for maintaining integrity, confidentiality, and availability for educational data. The results from the analysis in this paper of University of Tetova LMS revealed that authentication vulnerabilities, cross site scripting and SQL injection are still the most serious security vulnerabilities. With hands-on examples, like enforcing strong password policies using ASP. NET Core Identity implemented for role-based authorization, HTML encoding was applied to avoid XSS and parameterized queries used to prevent SQL injection have on the other hand shown as demonstrated in this article how these threats can be minimized.
The case study also showcased how such practical controls as email verification and role-based access controls improve user trust within the system. Finally, the paper stresses that LMS security must be taken care of at every possible level and not as an after-thought. Adhering to the best practices provided here will prepare academic communities against ever fluctuating cyber threats and secure their cyber infrastructure to create a trusted learning environment for learners as well as educators.

References

[1] Cavus, N. (2015). Distance Learning and Learning Management Systems. Procedia - Social and Behavioral Sciences, 191, 872–877. https://doi.org/10.1016/j.sbspro.2015.04.611

[2] Johny, J., Nordin, W., Lahapi, N. M., & Leau, Y. B. (2021). SQL Injection Prevention in Web Application: A Review. In Advances in Intelligent Systems and Computing (Vol. 1234, pp. 357–368). Springer. https://doi.org/10.1007/978-981-16-8059-5_35

[3] Saseendran, S. (2023). Authentication and Authorization in ASP.NET Core Web API with JSON Web Tokens. International Journal of Computer Applications, 182(6), 15–22.

[4] Midhun, T., Kumar, M., & Anoop, M. (2015). A Survey on Authorization Systems for Web Applications. IOSR Journal of Computer Engineering (IOSR-JCE), 17(3), 1–5. https://doi.org/10.9790/0661-17310105

[5] National Institute of Standards and Technology (NIST). (2020). Security and privacy controls for information systems and organizations: NIST special publication 800-53, revision 5. Gaithersburg, MD: NIST.

[6] Almehmadi, T., & Alsolami, F. (2019). Password Security in Organizations: User Attitudes and Behaviors Regarding Password Strength. In Advances in Intelligent Systems and Computing (Vol. 940, pp. 15–28). Springer. https://doi.org/10.1007/978-3-030-14070-0_2

[7] Nagarjun, P. M. D., & Shakeel, S. (2020). Cross-site Scripting Research: A Review. International Journal of Advanced Computer Science and Applications, 11(10), 481–487. https://doi.org/10.14569/IJACSA.2020.0110481

[8] Hydara, I., Md Sultan, A. B., Zulzalil, H., & Admodisastro, N. (2014). Current State of Research on Cross-Site Scripting (XSS) – A Systematic Literature Review. Information and Software Technology, 58, 162–178. https://doi.org/10.1016/j.infsof.2014.07.010

[9] Tan, X., Xu, Y., Wu, T., & Li, B. (2023). Detection of Reflected XSS Vulnerabilities Based on Paths-Attention Method. Applied Sciences, 13(13), 7895. https://doi.org/10.3390/app13137895

[10] Cross-Site-Scripting Attacks and Their Prevention during Development. (2017). International Journal of Engineering Development and Research, 5(3), 153–159. https://rjwave.org/IJEDR/papers/IJEDR1703023.pdf

[11] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., & Pretschner, A. (2016). Security Testing. Elsevier.

[12] Hu, J., Zhao, W., & Cui, Y. (2020). A Survey on SQL Injection Attacks, Detection and Prevention. In Proceedings of the 2020 International Conference on Computer Engineering and Application (pp. 483–488). ACM. https://doi.org/10.1145/3383972.3384028

[13] Luo, M., Zhang, J., He, D., & Shen, J. (2016). Security Analysis of a User Registration Approach. The Journal of Supercomputing, 72, 4099–4122. https://doi.org/10.1007/s11227-015-1619-1
[14] Park, J., & Sandhu, R. (2001). Role-Based Access Control on the Web. ACM Transactions on Information and System Security (TISSEC), 4(1), 37–71. https://doi.org/10.1145/383775.383777