Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2018, 5(12):1149-1151

Research Article ISSN: 2394 - 658X

Building Secure ASP.NET MVC Web Applications: Solutions for Developers

Vishnupriya S Devarajulu

Vishnupriyasupriya@gmail.com

ABSTRACT

Security is a critical factor for ASP.NET applications, particularly in an insecure web environment. This article explores various layers of security and emphasizes the importance of integrating security measures while building ASP.NET applications. It discusses common vulnerabilities and provides practical solutions for developing secure ASP.NET applications.

Key words: ASP.NET, MVC, Security, Authentication, Authorization, Session Management, SQL Injection, Cross-Site Scripting, CSRF, Secure Coding Practices

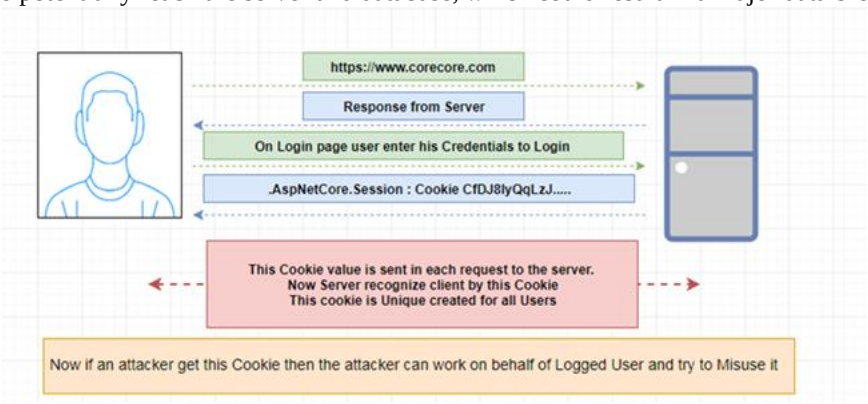
---

INTRODUCTION

Security is crucial factor for any application, and this is especially true for ASP.NET applications. Security should be implemented from the beginning of a SDLC of new application. Many developers often focus on building their application first and try to secure the application later. This often leads to problems, sometimes even requiring a complete rewrite the code, which can lead to new bugs. By integrating security from the start, you avoid having to comb through the entire project later for vulnerabilities that attackers could exploit. In this article, we'll explore the solutions for developing a secure ASP.NET applications.

AUTHENTICATION AND SESSION MANAGEMENT

Authentication involves getting a username and password from a user and checking them against a trusted source to allow access to certain parts of the application. If authentication isn't managed properly, attackers could steal user credentials, like session tokens or cookies. This could give them full access to the application, allowing them to potentially reach the server and database, which could result in a major data breach.



Ways in which an attacker can steal data: Not Secure connection (Not Using SSL), Predictable login credentials, Not storing credentials in Encrypted form, Improper Application logs out.

Solution:

1. Remove [".AspNetCore.Session"] after logout and use SSL for Securing Cookies, Session.

2. Secure Cookies by Setting HTTP and Implement two factors authentication.
3. Make sure the user password is complex and hashed before storing it in the users table
4. When you want to authenticate a user who is trying to login, make sure you hash the password he provided in the input field and compare the value with the hashed value stored in the Database.
5. Modify the Session ID after logout and on login generate new Session ID by using "System.Web.SessionState.SessionIDManager"
6. Make sure that none of the admin and regular users have "db_owner" access to Database

Authorization: At some point in developing most websites, we will need to restrict access to certain parts from accessing. Authorization determines whether an identity should be granted access to a specific resource.
Solution: In ASP.NET, there are two ways to authorize access to a given resource: URL authorization and File authorization.
Cookie Stealing: Cookies are essential for making the web user-friendly since most sites use them to identify users after they log in. Without cookies, we will have to log in repeatedly on every page. But, if attackers steal our cookies, they can impersonate us.
Solution: Use SSL certificate, only allow HTTPS requests, Apply Secure and HttpOnly flags in the web.config. This will ensure that they are only sent over an SSL connection.
Over Posting: ASP.NET MVC Model Binding is a powerful feature that greatly simplifies the process handling user input by automatically mapping the input to your model properties based on naming conventions. However, this presents another attack vector, which can allow your attacker.
Solution: Mark the property as [ReadOnly]. More commonly, we can use a BindAttribute on the method parameters and just include (whitelist) the properties you want to allow for binding.
Sensitive Data Exposure: The data which is traveling (sending and receiving) and data which is at one place should have protections. As we are into web development, we store Users personal information like (Password, Passport details, Credit Card Numbers, Health records, financial records, business secrets).
Solution: Always Send Sensitive Data to Server in an Encrypted format using Strong Hashing with Seed (Random Hash), Always apply SSL to the Web application, Do not store Sensitive data if want to store using Strong Hashing techniques.
Audit Trail
In a production application, we have millions of transactions that occur in that Client or Users Create data, Update data, Delete Data and anyone who can access can make any transactions which will be hard to monitor.
Solution: Keep Audit Trail of all User activity on Web Application and always Monitor it.
Cross-Site Scripting (XSS) Attacks
Cross-site Scripting (XSS) injects malicious scripts through input fields, posing a major security threat. It enables attackers to steal credentials and sensitive data, making it the top vulnerability on the Web. XSS exploits various points like inputs, query strings, HTTP headers, and db content.
Solution: To protect against XSS attacks:

1. **Regular Expression Validation:** Use regular expressions to validate input, preventing malicious inputs from being accepted.
2. **HTML Encoding:** Razor in MVC automatically encodes output from variables, providing basic protection against XSS.
3. **URL Encoding:** Ensure URLs transferring data via query strings are encoded to prevent XSS attacks. Libraries for HTML and URL encoding are available via NuGet for robust protection.

Blocking Brute Force Attacks: A brute-force attack is an attempt to discover a password by systematically trying every possible combination of letters, numbers, and symbols until you discover the one correct combination that works.
Solution: Lock user account after specific number of login attempts, Enable Google reCAPTCHA on Login page.

VALIDATING FILE UPLOADS ON INDEX HTTPPOST METHOD

When we validate file upload we check if a file extension is proper or not, then we consider the file as a valid file. But in a real scenario, an attacker can upload a malicious file which may cause a security issue. The attacker can change file extension [tuto.exe to tuto.jpeg] and the malicious script can be uploaded as an image file. Most developers just look on the file extension of the file and save in folder or database, but file extension is valid not file, it may have a malicious script.
Solution: The first thing we need to do is validate file uploads, allow only access to files extension which are required, Check the file header, Validating File Uploads on Index HttpPost Method In this method, first we are going to check file upload count if the count is zero, then User has not uploaded any file. If file count is greater than zero, then it has a file, and we are going to read Filename of File Along with Content Type and File Bytes.

CSRF (Cross Site Request Forgery): Cross-Site Request Forgery (CSRF) tricks authenticated users into unknowingly executing actions on a web application. Attackers use social engineering tactics to manipulate users into performing actions like fund transfers or profile changes without their consent. This attack targets state-changing requests, posing a significant risk, especially for administrative accounts.

Solution: To prevent Cross-Site Request Forgery (CSRF) attacks in ASP.NET MVC, follow these steps:

1. Add asp-antiforgery="true" attribute to the form HTML tag to generate an anti-forgery token.
2. In the [HttpPost] Action Method that handles the form submission, add [ValidateAntiForgeryToken] attribute.
3. Setting asp-antiforgery="true" generates a hidden field with a unique token and adds a session cookie to the browser.
4. When the form is submitted, ASP.NET MVC verifies the __RequestVerificationToken hidden field and session cookie.
5. If these values are missing or don't match, ASP.NET MVC prevents the action from processing, effectively thwarting CSRF attacks.

SQL Injection Attack: SQL injection remains a prominent threat in recent years. Exploiting vulnerabilities in database-driven applications, attackers can execute malicious SQL statements to access sensitive data or compromise the entire database server. This poses significant risks to data security and system integrity.

Solution: It's crucial to validate input on both the client and server sides to prevent special characters that could be exploited in SQL injection attacks. The default DB Owner role grants extensive database permissions, posing a security risk if misused. To mitigate this risk, create new user accounts with minimal privileges tailored to specific needs. Additionally, employing stored procedures with type-safe SQL parameters helps safeguard against SQL injection vulnerabilities.

CONCLUSION

Securing ASP.NET web applications involve integrating robust security measures to remove vulnerabilities and protect sensitive data. Solutions include implementing secure authentication, authorization, and rigorous input validation. Protecting against SQL injection and XSS attacks, adopting strict file upload validation, maintaining audit trails, and applying best practices like two-factor authentication and anti-CSRF measures are essential. By adhering to these principles and staying updated on emerging threats, developers can safeguard user trust and ensure application resilience against cyber threats.

REFERENCES

[1]. Microsoft Security Developer Center, http://msdn.microsoft.com/en-us/security/default.aspx
[2]. Code Project Article, https://www.codeproject.com/Articles/1259066/10-Points-to-Secure-Your-ASP-NET-Core-MVC-Applic-2
[3]. LinkedIn Article, https://www.linkedin.com/pulse/building-secure-aspnet-mvc-web-applications-rajnish-kumar-sharma/
[4]. ASP.NET Security Book, Barry Dorrans, http://www.wrox.com/WileyCDA/WroxTitle/Beginning-ASP-NETSecurity.productCd-0470743654.html
[5]. OWASP Top 10 for .NET Developers, http://www.troyhunt.com/2010/05/owasp-top-10-for-net-developers-part-1.html
[6]. AntiXSS Library, http://antixss.codeplex.com/