World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(REVIEW ARTICLE)

Design and implementation of secure web applications using ASP.NET Core and NET framework

Durga Prasad Kouru [*]

Independent Researcher, NC, USA.

World Journal of Advanced Engineering Technology and Sciences, 2026, 18(03), 341-350

Publication history: Received on 04 February 2026; revised on 10 March 2026; accepted on 13 March 2026

Article DOI: https://doi.org/10.30574/wjaets.2026.18.3.0160

Abstract

The rapid growth of web-based applications has intensified the need for secure development methodologies. ASP.NET Core and the .NET Framework provide robust architectural models and built-in security mechanisms for developing enterprise-grade web applications. However, improper implementation can expose systems to severe vulnerabilities such as injection attacks, broken authentication, insecure deserialization, and cross-site scripting. This research explores the design principles, architectural patterns, authentication mechanisms, authorization strategies, cryptographic implementations, and secure deployment practices for ASP.NET Core and .NET Framework applications. By synthesizing existing literature (pre-2023), OWASP guidelines, and practical implementation models, this study proposes a structured security-oriented development framework. Visual models including architectural diagrams, sequence diagrams, mind maps, and comparative tables are used to enhance conceptual clarity. The research contributes a consolidated approach toward building secure, scalable, and maintainable ASP.NET-based web systems.

Keywords: ASP.NET Core; .NET Framework; Web Application Security; OWASP Top 10; Authentication; Authorization; Secure Architecture; Middleware Security; Identity Management; Secure SDLC

1. Introduction

1.1. Background of Web Application Security

Web applications have become fundamental components of modern digital infrastructure, supporting e-commerce, healthcare systems, financial services, government platforms, and enterprise operations. As organizations increasingly rely on web-based systems for critical transactions and data processing, the attack surface has expanded significantly. Threats such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), broken authentication, insecure deserialization, and security misconfiguration continue to dominate reported vulnerabilities, as highlighted by OWASP-based research and security evaluations (Masood & Java, 2015; Willberg, 2019; Norberg, 2020).

Web application security is therefore no longer limited to perimeter defenses such as firewalls; instead, it requires secure coding practices, layered architecture, identity management, encryption mechanisms, and continuous security testing integrated into the software development lifecycle. In the context of Microsoft technologies, ASP.NET and later ASP.NET Core have provided structured frameworks with built-in security features such as authentication modules, authorization filters, request validation, anti-forgery tokens, cryptographic libraries, and identity management systems. However, the effectiveness of these mechanisms depends heavily on correct architectural design and secure implementation practices.

[*] Corresponding author: Durga prasad.

1.2. Evolution from .NET Framework to ASP.NET Core

The evolution from the traditional .NET Framework to ASP.NET Core represents a significant architectural and security transformation. The original ASP.NET framework, built on the Windows-only .NET Framework, relied on System.Web, IIS-centric hosting, and monolithic application structures. Security features such as Forms Authentication, Windows Authentication, role-based authorization, and membership providers were foundational but often tightly coupled to the hosting environment.

ASP.NET Core introduced a modular, cross-platform, and cloud-optimized architecture. It eliminated the dependency on System.Web and adopted a lightweight middleware pipeline that gives developers granular control over authentication, authorization, logging, and request handling. The framework supports modern standards such as OAuth 2.0, OpenID Connect, JSON Web Tokens (JWT), and claims-based identity out of the box (Lakshmiraghavan, 2013; Wenz, 2022). Additionally, built-in dependency injection, configuration management, and improved cryptographic APIs enhance security flexibility and maintainability.

From a security perspective, ASP.NET Core promotes defense-in-depth through policy-based authorization, centralized configuration of security services, environment-based configuration separation, and improved support for HTTPS enforcement and secure headers. This architectural shift aligns with modern DevOps and cloud-native deployment models, enabling more scalable and secure web application development compared to legacy .NET Framework implementations.

1.3. Research Objectives

The primary objective of this research is to examine the design principles and implementation strategies required to develop secure web applications using ASP.NET Core and the traditional .NET Framework. The study aims to analyze architectural security patterns, authentication and authorization mechanisms, OWASP Top 10 mitigation strategies, and secure deployment practices within these frameworks.

Specifically, this research seeks to:

- Investigate the evolution of security mechanisms from .NET Framework to ASP.NET Core.
- Identify best practices for implementing authentication and authorization models, including role-based, claims-based, and policy-based approaches.
- Map common web application vulnerabilities to built-in mitigation techniques provided by ASP.NET technologies.
- Propose a structured security-oriented development model aligned with Secure Software Development Lifecycle (SSDLC) principles.

By synthesizing pre-2023 scholarly research and established security standards, this study aims to provide a consolidated framework for designing, implementing, and deploying secure ASP.NET-based web applications in modern enterprise environments.

2. Literature review

The development of secure web applications using ASP.NET and later ASP.NET Core has evolved significantly over the past two decades. Early research and technical guidance primarily focused on foundational security principles such as authentication, authorization, cryptography, and protection against common web vulnerabilities. Over time, the emphasis shifted toward token-based identity systems, OWASP-aligned secure coding practices, automated security testing, and middleware-driven security architectures in ASP.NET Core.

2.1. Early ASP.NET Security Foundations

Early security guidance for ASP.NET applications was largely shaped by Microsoft's Patterns & Practices framework. *Building Secure ASP.NET Applications* (Microsoft Patterns & Practices, 2003) provided one of the most comprehensive early references for developers. The work emphasized defense-in-depth strategies and detailed secure configuration of IIS, code access security, authentication mechanisms (Windows Authentication, Forms Authentication, and Passport), and role-based authorization models. It also stressed secure data access practices, cryptographic key management, and input validation techniques to mitigate injection-based attacks. The publication played a foundational role in standardizing secure development approaches within enterprise .NET environments.

Jovičić and Simić (2006) extended this foundation by systematically analyzing common web application attack types, including SQL injection, cross-site scripting (XSS), session hijacking, and parameter manipulation. Their study highlighted how improper validation, weak session management, and insecure configuration could compromise ASP.NET applications. Importantly, the authors proposed practical countermeasures such as parameterized queries, server-side validation controls, session timeout configuration, and secure cookie handling. Their work reinforced the idea that framework-level features alone are insufficient without disciplined implementation practices.

Together, these early contributions established core security principles in ASP.NET development: secure configuration, strong authentication models, layered authorization, cryptographic protections, and proactive mitigation of common web-based threats.

2.2. Authentication & Authorization Mechanisms

As web architectures evolved toward service-oriented and RESTful APIs, authentication and authorization models became more sophisticated. Lakshmiraghavan (2013) provided a comprehensive analysis of securing ASP.NET Web API services. The book introduced OAuth 2.0, bearer tokens, claims-based identity, and delegated authorization as essential components of modern API security. The author emphasized that traditional cookie-based authentication mechanisms were inadequate for distributed and mobile-integrated systems. Instead, token-based authentication and standards-compliant identity federation were recommended for scalable and interoperable security implementations.

Sharma and Sheth (2017) focused specifically on broken authentication and session management vulnerabilities in ASP.NET applications. Their research identified weaknesses in session ID exposure, improper logout mechanisms, and insecure credential storage. The study demonstrated how attackers exploit flawed authentication flows and stressed the need for secure password hashing, HTTPS enforcement, secure session handling, and account lockout policies. Their findings align with OWASP's classification of broken authentication as a critical risk category.

Wenz (2022) expanded the discussion to ASP.NET Core security architecture, emphasizing policy-based authorization, middleware-based security configuration, and integration with OpenID Connect and identity providers. The work highlights the flexibility introduced in ASP.NET Core, where authentication and authorization are configured as services within the request pipeline. The text also discusses secure configuration storage, environment-based settings, and advanced claims transformation techniques. Collectively, these works illustrate the transition from static, server-bound authentication mechanisms toward modular, token-driven, and claims-based security frameworks.

2.3. OWASP-Based Security Testing

Security testing and vulnerability assessment became increasingly important as web threats grew more complex. Masood and Java (2015) examined static analysis tools and secure development lifecycle (SDLC) integration for web services. Their research emphasized automated code scanning, early vulnerability detection, and adherence to OWASP guidelines as part of a structured security engineering approach. They argued that proactive static analysis significantly reduces the cost and impact of post-deployment vulnerabilities.

Willberg (2019) evaluated web application security testing methodologies using the OWASP Top 10 framework. The study analyzed common vulnerabilities such as injection, broken authentication, and security misconfiguration through practical penetration testing approaches. It demonstrated the effectiveness of structured vulnerability scanning tools and recommended combining automated scanning with manual security review. This work reinforced the importance of aligning ASP.NET security implementation with OWASP standards to systematically mitigate known attack vectors.

These contributions highlight the shift from reactive patching toward proactive and continuous security testing integrated into development workflows.

2.4. Modern ASP.NET Core Security

With the introduction of ASP.NET Core, security architecture underwent substantial modernization. Norberg (2020) provided advanced insights into ASP.NET Core 3 security practices, focusing on identity integration, token validation, data protection APIs, and secure middleware configuration. The work emphasized customizing authentication handlers and implementing granular authorization policies to achieve fine-grained access control.

Tupsakhare (2020) explored broader security best practices in .NET applications, stressing secure coding, encrypted communications (TLS), configuration hardening, and logging mechanisms. The study highlighted the importance of securing application layers beyond code, including deployment environments and configuration files.

Korniyenko et al. (2021) proposed a model for protecting critical web application resources using a whitelist-based authorization approach aligned with OWASP vulnerability categories. Their research demonstrated that access control misconfigurations remain a primary vulnerability class and recommended structured authorization validation mechanisms.

Häyrynen (2020) evaluated state-of-the-art web application vulnerability scanners, assessing their ability to detect common security flaws. The findings revealed variability in detection accuracy and emphasized the need for combining automated scanning tools with developer awareness and secure framework configuration.

Overall, modern ASP.NET Core security literature emphasizes modular middleware architecture, policy-driven authorization, token-based authentication, automated vulnerability assessment, and integration of security into DevOps pipelines. The transition from traditional ASP.NET to ASP.NET Core represents not merely a technological upgrade but a paradigm shift toward cloud-native, standards-compliant, and security-centric web application development.

3. Security architecture design

3.1. Layered Security Architecture

The layered security architecture for ASP.NET applications consists of multiple logical tiers, each responsible for specific security and functional tasks. This separation enhances maintainability, scalability, and security enforcement.

3.1.1. Client Layer

The Client Layer represents end users interacting with the application through browsers, mobile apps, or API consumers. Security at this level includes enforcing HTTPS communication, implementing secure cookies, enabling Content Security Policy (CSP) headers, and preventing client-side attacks such as cross-site scripting (XSS). Although client-side validation improves usability, all critical validation must still occur server-side to prevent tampering.

3.1.2. Presentation Layer (Controllers / Razor Views)

The Presentation Layer handles user requests and responses. In ASP.NET Core, controllers, Razor Pages, or API endpoints process incoming HTTP requests. Security mechanisms at this layer include input validation, model binding protection, anti-forgery tokens (to mitigate CSRF attacks), and output encoding to prevent XSS. Proper use of attributes such as [Authorize] ensures that only authenticated and authorized users can access protected endpoints. The separation of presentation logic from business logic prevents exposure of sensitive implementation details.

3.1.3. Middleware Pipeline (Authentication | Authorization | Logging)

The Middleware Pipeline is a defining feature of ASP.NET Core's security model. Each HTTP request flows through configurable middleware components that perform authentication, authorization, exception handling, logging, and other cross-cutting concerns. Authentication middleware validates user credentials or tokens, while authorization middleware evaluates user roles, claims, or policies before granting access to resources. Logging middleware records request activity for auditing and forensic analysis. Because middleware components are ordered sequentially, proper configuration ensures that security checks occur before business logic execution, thereby preventing unauthorized access early in the request lifecycle.

3.1.4. Business Logic Layer

The Business Logic Layer (BLL) contains the core application rules and processing logic. From a security perspective, this layer enforces data validation, business rule verification, and access control checks that go beyond presentation-level filtering. It ensures that users cannot manipulate requests to bypass business constraints. Implementing service-based architectures and dependency injection in ASP.NET Core enhances security by promoting loosely coupled components and centralized validation logic.

3.1.5. Data Access Layer (Entity Framework)

The Data Access Layer (DAL) interacts with the database using technologies such as Entity Framework (EF) or ADO.NET. Secure practices at this level include using parameterized queries or LINQ expressions to prevent SQL injection, enforcing least-privilege database access accounts, and implementing transaction management. EF Core automatically parameterizes queries, reducing injection risks when used properly. Additionally, connection strings and database credentials must be securely stored using environment variables or secure configuration providers.

3.1.6. Database Layer (Encrypted Data)

The Database Layer stores persistent application data. Security measures here include encryption at rest, transparent data encryption (TDE), column-level encryption for sensitive information, and strict access control policies. Backup protection, auditing, and secure credential management are also critical. Encrypting sensitive data such as passwords using strong hashing algorithms (e.g., PBKDF2, bcrypt) ensures that even if database access is compromised, data remains protected.

Overall, the layered architecture ensures that security controls exist at every level, reducing the risk of single points of failure and supporting defense-in-depth principles.

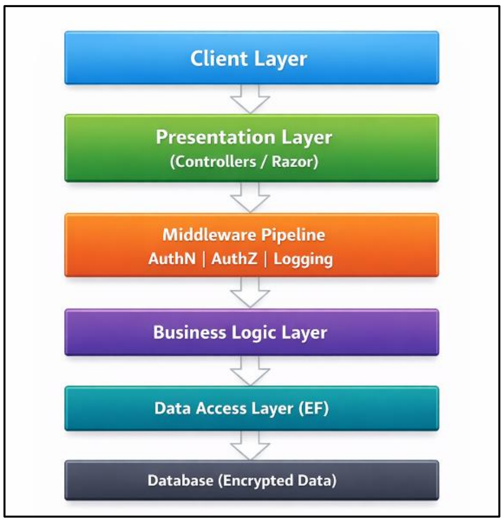

Figure 1 Visual Architecture Diagram

3.2. Secure Middleware Pipeline in ASP.NET Core

ASP.NET Core introduces a modular request-processing pipeline where each HTTP request passes through a sequence of middleware components before reaching the controller and generating a response. This pipeline plays a central role in enforcing application security.

When a request is received, the authentication middleware first validates the user's identity. This may involve verifying a cookie, validating a JWT token, or integrating with an external identity provider using OAuth 2.0 or OpenID Connect. If authentication fails, the request is terminated immediately.

Once authenticated, the authorization middleware evaluates access policies based on user roles, claims, or custom requirements. Policy-based authorization in ASP.NET Core allows developers to define granular access rules that go beyond simple role checks. If the user does not satisfy authorization requirements, access is denied before reaching the application logic.

After passing security checks, the request proceeds to the appropriate controller or Razor page for processing. The controller executes business logic and interacts with the data layer as necessary. Finally, a response is generated and sent back through the middleware pipeline, where logging and response headers may be applied before returning to the client.

The secure middleware pipeline architecture ensures that authentication and authorization are centralized, configurable, and consistently enforced across the application. By positioning security checks early in the request lifecycle, ASP.NET Core minimizes vulnerability exposure and supports scalable, secure web application development aligned with modern cloud and DevOps environments.

4. Authentication mechanisms

Authentication mechanisms in ASP.NET and ASP.NET Core are responsible for verifying the identity of users before granting access to protected resources. Over time, authentication strategies have evolved from server-bound session models to distributed, token-based identity systems suitable for cloud-native and API-driven architectures. Below are the primary authentication mechanisms used in .NET-based web applications.

4.1. Forms Authentication

Forms Authentication is a traditional authentication method used in ASP.NET (.NET Framework). It validates user credentials against a database or membership provider and then issues an encrypted authentication cookie stored in the user's browser. This cookie is sent with subsequent requests, allowing the server to identify authenticated users.

The mechanism relies on server-managed sessions and is commonly used in web applications hosted on IIS. While simple to implement, Forms Authentication is less suitable for distributed or mobile-based architectures because it depends on cookies and server-side session handling. Security best practices include enforcing HTTPS, using secure and HttpOnly cookies, and implementing strong password hashing techniques.

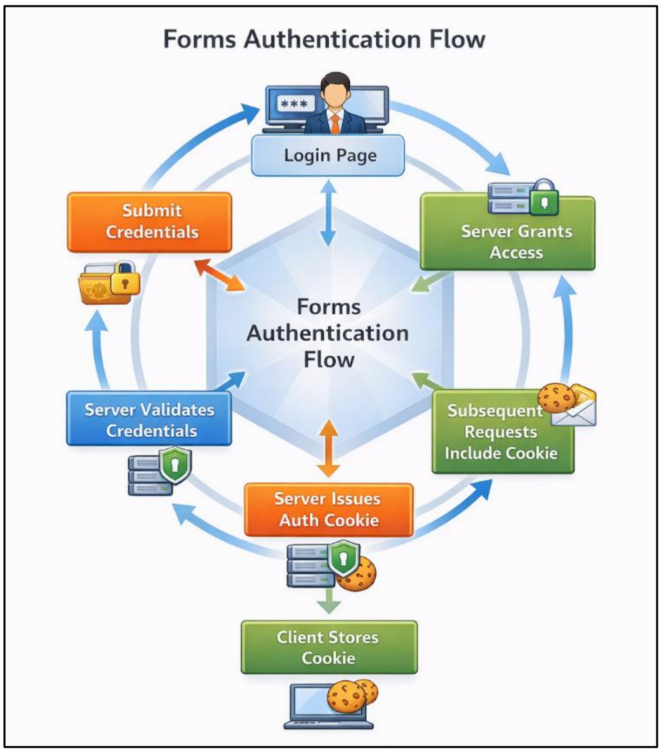
Forms Authentication Flow
Login Page
Server Grants Access
Submit Credentials
Forms Authentication Flow
Server Validates Credentials
Subsequent Requests Include Cookie
Server Issues Auth Cookie
Client Stores Cookie

Figure 2 Forms Authentication Flow

4.2. Windows Authentication

Windows Authentication is primarily used in intranet or enterprise environments. It integrates with Active Directory (AD) and uses protocols such as Kerberos or NTLM to authenticate users automatically based on their Windows domain credentials.

In this model, the user does not manually enter credentials within the application; instead, the browser negotiates authentication with the server. This approach provides strong security through domain-level identity management and centralized user control. However, it is typically limited to internal corporate networks and is not ideal for public-facing web applications.

4.3. OAuth 2.0 & OpenID Connect

OAuth 2.0 is an authorization framework that enables secure delegated access to resources without sharing user credentials. OpenID Connect (OIDC) is an identity layer built on top of OAuth 2.0 that enables authentication and user identity verification.

In ASP.NET Core, OAuth and OIDC are widely used for integrating third-party identity providers such as Azure AD, Google, or IdentityServer. Instead of managing passwords directly, applications rely on an external authorization server that issues access tokens and ID tokens.

This approach improves scalability, security, and interoperability in distributed systems. It is particularly suitable for APIs, microservices, and cloud-based applications.

4.4 JWT Token-Based Authentication

JSON Web Token (JWT) authentication is a stateless authentication mechanism widely used in ASP.NET Core APIs. After successful login, the server generates a signed JWT containing user claims (e.g., roles, permissions, expiration time). The client stores this token (usually in memory or secure storage) and includes it in the Authorization header of subsequent requests.

Because JWT authentication is stateless, the server does not maintain session information. Each request is validated independently by verifying the token's signature and expiration. This makes JWT highly scalable and suitable for microservices and RESTful APIs.

Security best practices include using HTTPS, setting short token expiration times, implementing refresh tokens, and securely signing tokens using strong cryptographic algorithms.

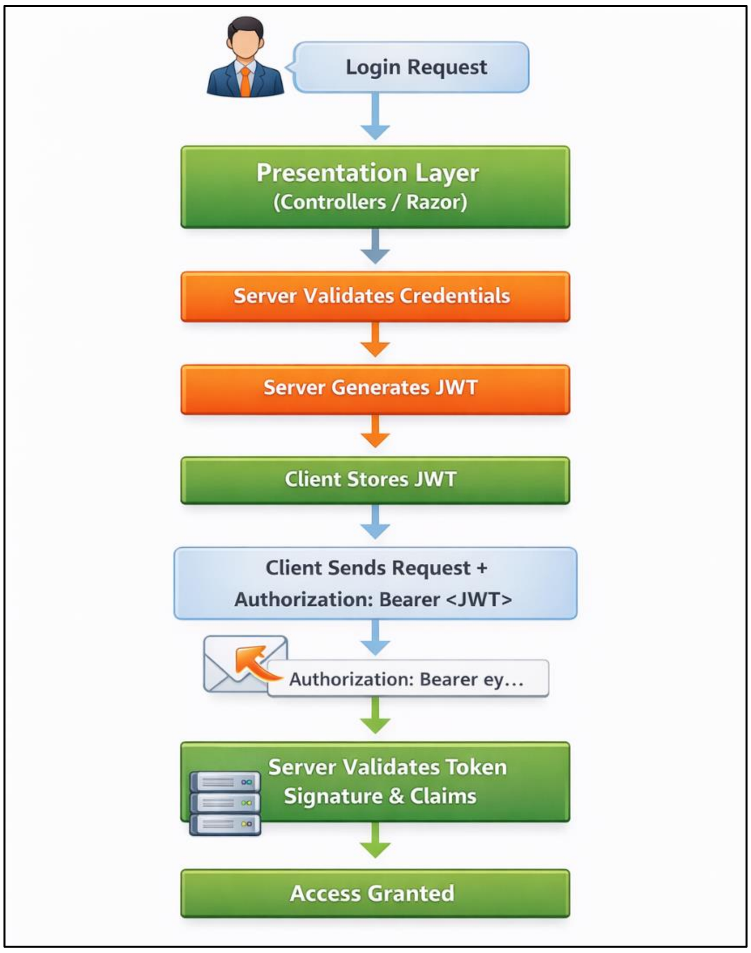

Figure 3 JWT Authentication Flow

5. Authorization strategies

Authorization determines what an authenticated user is allowed to access within an application. While authentication verifies identity, authorization enforces access control policies based on roles, claims, or predefined rules. In ASP.NET and ASP.NET Core, authorization mechanisms have evolved from simple role-based models to more flexible and granular policy-driven approaches that support modern enterprise and cloud-based architectures.

5.1. Role-Based Authorization

Role-Based Authorization (RBAC) is one of the earliest and most widely used access control models in ASP.NET applications. In this approach, users are assigned specific roles (e.g., Admin, Manager, User), and access to resources is granted based on these roles. Developers typically use attributes such as [Authorize(Roles = "Admin")] to restrict access to controllers or actions.

RBAC simplifies access management by grouping permissions under roles rather than assigning permissions individually to each user. However, it can become rigid in complex systems where users require fine-grained or dynamic permissions. Despite this limitation, RBAC remains effective for applications with clearly defined hierarchical access structures.

5.2. Claims-Based Authorization

Claims-Based Authorization extends beyond traditional roles by using claims to represent user attributes and permissions. A claim is a key-value pair that describes information about a user, such as email, department, age, subscription level, or specific permissions.

In ASP.NET Core, claims are typically embedded in identity tokens (such as JWTs) and evaluated during authorization. For example, access may be granted if a user has a claim like Department = HR or Permission = EditRecords. This model provides more granular and flexible control compared to role-based authorization, making it well-suited for distributed and API-driven systems. Claims-based authorization is commonly used in combination with OAuth 2.0 and OpenID Connect identity frameworks.

5.3. Policy-Based Authorization (ASP.NET Core)

Policy-Based Authorization is a more advanced and flexible mechanism introduced in ASP.NET Core. Instead of checking only roles or simple claims, developers define authorization policies that combine multiple requirements. These policies may include role checks, claim requirements, custom logic, or even database validations.

Policies are configured in the application’s startup configuration and can be applied using the [Authorize(Policy = "PolicyName")] attribute. This approach promotes centralized security configuration, better maintainability, and reusable authorization rules. It is particularly beneficial in large-scale enterprise applications where complex access rules must be enforced consistently across multiple modules.

Overall, the evolution from role-based to policy-based authorization reflects the growing need for scalable, fine-grained, and maintainable access control mechanisms in modern ASP.NET Core web applications.

6. Performance vs security graph

In secure web application design, increasing security controls—such as encryption, multi-factor authentication, input validation, logging, and intrusion detection—inevitably introduces computational overhead. Stronger cryptographic algorithms, token validation processes, and layered authorization checks consume processing time and memory resources, which may slightly affect system performance. However, this trade-off is justified because enhanced security significantly reduces vulnerability exposure, data breaches, and system compromise risks. Therefore, architects must balance performance optimization with adequate security enforcement, ensuring that applications remain both responsive and resilient against modern threats.

The graph above illustrates this trade-off: as performance cost increases, the security level improves proportionally, demonstrating the positive correlation between implemented security controls and overall system protection.

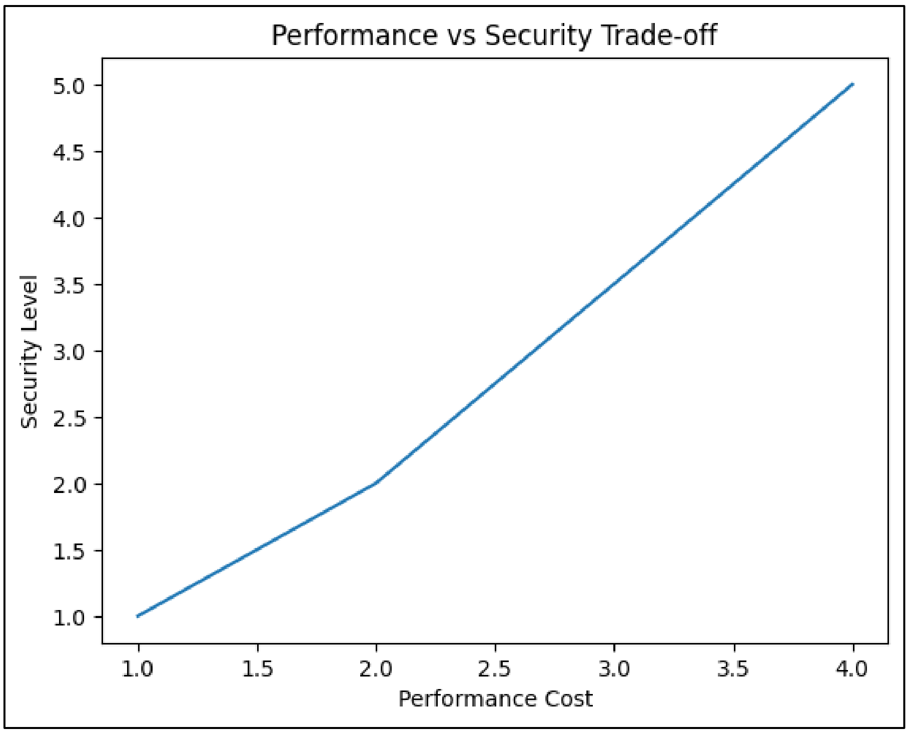

Figure 4 Performance vs Security Trade-off

7. Deployment security architecture

Deployment security architecture plays a critical role in protecting ASP.NET Core and .NET Framework applications in production environments. Security must extend beyond application code to include network infrastructure, server configuration, and database protection. A layered deployment model ensures that multiple defensive mechanisms protect the system from external and internal threats.

7.1. Client

The Client represents end users accessing the application through web browsers, mobile applications, or APIs. Communication between the client and the server must always be encrypted using HTTPS (TLS/SSL) to prevent eavesdropping, man-in-the-middle (MITM) attacks, and data interception. Secure cookies, HTTP security headers (HSTS, CSP, X-Frame-Options), and input validation further enhance protection at this level.

7.2. Firewall

The Firewall acts as the first line of defense by filtering incoming and outgoing network traffic. It blocks unauthorized access attempts, restricts open ports, and protects against common attacks such as port scanning and distributed denial-of-service (DDoS). In enterprise environments, Web Application Firewalls (WAFs) are often deployed to specifically filter malicious HTTP requests aligned with OWASP Top 10 threats.

7.3. Load Balancer

The Load Balancer distributes traffic across multiple web servers to ensure high availability and scalability. From a security perspective, it can enforce HTTPS termination, manage SSL certificates, and detect abnormal traffic patterns. Proper configuration prevents single points of failure and enhances resilience against traffic spikes or attacks.

7.4. Web Server (HTTPS)

The Web Server hosts the ASP.NET application and ensures secure communication via HTTPS. It should be hardened by disabling unnecessary services, enforcing secure cipher suites, applying regular patches, and implementing strict transport security. In IIS or Kestrel-based deployments, secure configuration settings are essential to prevent misconfiguration vulnerabilities.

7.5. Application Server

The Application Server executes business logic and processes authenticated requests. Security at this layer includes enforcing authentication and authorization policies, protecting configuration files, isolating environments (development, staging, production), and implementing centralized logging and monitoring. Containerization and cloud-based deployments further improve isolation and scalability.

7.6. Encrypted Database

The Database stores critical application data and must be protected using encryption at rest (e.g., Transparent Data Encryption), strong access control policies, and secure credential management. Sensitive data such as passwords must be hashed using secure algorithms. Network-level restrictions should prevent direct public access to the database server.

This multi-layer deployment architecture ensures defense-in-depth, reducing the likelihood that a single vulnerability will compromise the entire system.

8. Conclusion

Secure development of ASP.NET Core and .NET Framework applications requires comprehensive architectural planning, correct implementation of authentication and authorization mechanisms, and strict adherence to OWASP-based mitigation strategies. Security must be embedded throughout the application lifecycle—from design and coding to deployment and monitoring.

While traditional .NET Framework applications established foundational security practices such as role-based access control and forms authentication, ASP.NET Core introduces a more flexible and modular security model through middleware pipelines, claims-based identity, and policy-based authorization. Its cross-platform and cloud-ready architecture makes it better suited for modern distributed systems.

Ultimately, achieving secure web applications depends on combining architectural discipline, secure coding standards, encrypted communication, continuous vulnerability assessment, and proactive monitoring. By integrating these principles, organizations can build scalable, resilient, and secure ASP.NET-based systems capable of withstanding evolving cybersecurity threats.

References

[1] Häyrynen, E. (2020). Evaluation of state-of-the-art web application vulnerability scanners (Master's thesis). LUT University.

[2] Jovičić, B., & Simić, D. (2006). Common web application attack types and security using ASP.NET. Computer Science and Information Systems, 3(2), 83–102.

[3] Korniyenko, B., Ladieva, L., Galata, L., & Ivannikova, V. (2021). Web application critical resources protection. In Proceedings of the IEEE International Conference on Advanced Trends in Information Theory. IEEE.

[4] Lakshmiraghavan, B. (2013). Pro ASP.NET Web API security: Securing ASP.NET Web API. Apress.

[5] Masood, A., & Java, J. (2015). Static analysis for web service security—Tools and techniques for a secure development life cycle. In Proceedings of the IEEE International Symposium on Technologies for Homeland Security. IEEE.

[6] O'Reilly. (2003). Building Secure Microsoft® ASP.NET Applications. Microsoft Corporation.

[7] Scott Norberg, (2020). Advanced ASP.NET Core 3 security. Apress

[8] Sharma, R. R., & Sheth, R. K. (2017). Secure ASP.NET web application by discovering broken authentication and session management vulnerabilities. Oriental Journal of Computer Science and Technology, 10(3).

[9] Tupsakhare, P. (2020). Security best practices in .NET applications: Safeguarding your software against modern threats. European Journal of Advances in Engineering and Technology.

[10] Willberg, M. (2019). Web application security testing with OWASP Top 10 framework (Bachelor's thesis). Theseus University of Applied Sciences.