International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group | Volume 7 Issue 1 PP 287-296, 2026
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V7I1P135

Original Article

Review of Secure API Development and Authentication Mechanisms in ASP.NET Core Applications

George Zacharia
Software Engineer, Independent Researcher.

Received On: 27/01/2026 ***Revised On: 28/02/2026*** ***Accepted On: 03/03/2026*** ***Published On: 08/03/2026***

***Abstract** - ASP.NET Core applications API development and authentication. It discusses the security design of ASP.NET Core, which is structured as a modular design, an HTTP request pipeline built on middleware, and is integrated with CoreCLR, CoreFX, and Kestrel web server. Some of the key security features discussed in the study include authentication, authorization, data protection, enforcement of the HTTPS protocol, and security headers. The different authentication systems, such as claims-based authentication, token authentication using JWT, OAuth 2.0, OpenID Connect, etc., are examined in terms of the working principles, tokens, and delegated authorization. Additionally, the paper covers the common API security concerns, including injection attacks, broken authentication, insecure data transmission, and denial-of-service attacks. It also recommends best practices, such as rate restriction, safe code, API gateway, and continual monitoring, to mitigate these threats. The traditional and modern security methods are compared and contrasted to demonstrate the superiority of the token-based security model and the zero-trust model for scalable, cloud-native ASP.NET Core frameworks.*

***Keywords** - ASP.NET Core, Secure API Development, Authentication, Authorization, SON Web Token (JWT), OAuth 2.0.*

1. Introduction

Application Programming Interfaces (APIs) are now an essential part of current web and distributed systems, which allow secure communication and exchange of data between heterogeneous applications. In microservices architectures, cloud-native, and distributed environments, APIs are a structured interface that enables independent services to communicate without losing abstraction [1]. RESTful APIs have continued to be popular because they are easy, scalable, and compatible with the HTTP-based communication even as other paradigms like Graph are coming up to perform data queries with optimality [2][3]. With the increasing usage of APIs as access points to important systems, the security of APIs nowadays is a key concern in modern software-engineering.

The move to microservices and clouds has increased the level of dependence on API gateways and central access layers that control the routing of requests, load balancing, and authentication. As dies are intermediaries between client applications and back-end services, they are of high value to cyber threats [4][5]. Risks that are common in distributed API environment include unauthorized access, misuse of tokens, data breaches and disruptions to services [6]. Secure API endpoints and the availability, integrity, and confidentiality of system resources necessitate robust authentication and authorization procedures.

These considerations have contributed to ASP.NET Core's rise to prominence as a leading framework for the secure and performant building of online APIs. It has an inbuilt system of authentication schemes, authorization policies, data protection and HTTPS enforcement [7]. Web Tokens (JWT), OAuth 2.0, cookies-based authentication, and Windows authentication are some of the authentication technologies that developers can employ, depending on the deployment requirements. These features notwithstanding, APIs still remain vulnerable due to incorrect configuration, inconsistent practices in implementation, and changing attack vectors [8]. The act of securing APIs is becoming more complicated because developers have to deal not only with the threats that are inherent to the web but also with the threats that are related to APIs. Use of RESTful services has not been readily adopted with standardized or strictly implemented security practices [9][10]. Trade-offs between performance and security in most instances also add to the exploitable vulnerabilities. The challenges highlight why a systematic study of the practice of secure API development, especially in the ASP.NET Core ecosystem, is necessary.

1.1. Structure of the Paper

The paper is structured as follows. Section II presents secure API development features and built-in security architecture in ASP.NET Core. Section III discusses authentication mechanisms, including claims-based, token-based (JWT), OAuth 2.0, and OpenID Connect models. Section IV examines common API security threats and best practices for mitigation in ASP.NET Core applications. Section V summarizes the comparative analysis and key observations from the review, and Section VI concludes the study with major findings and future research directions.

2. Secure API Development in ASP.NET Core

API development in ASP.NET Core is discussed by focusing on the security architecture and the security mechanisms built into the platform. Data security, authorisation, and authentication in the new web API can be effectively implemented with the help of the framework, as is explained in the document. The open-source, cross-platform ASP. NET Core from Microsoft is an excellent option for creating modern, high-performance web applications and RESTful APIs. Its modular and middleware design enables the creation of cloud-ready apps for Windows, Linux, and MacOS that are scalable.

Authentication, authorisation, data protection, and security headers are just a few of the security capabilities offered by ASP.NET Core. The previous research emphasizes that it provides in-built identity management, access control based on policy and cryptographic service to ensure the security of application data [11]. It is also compatible with the HTTPS enforcement with configurable security header to address the typical web vulnerabilities. These security capabilities though considered comprehensive and strong require proper configuration and secure practices of implementation.

2.1. Architecture of ASP.NET Core

A streamlined version of the .NET Framework, ASP.NET Core, was released in 2016. It is cross-platform, modular, and built on a lightweight middleware pipeline that manages HTTP requests and makes use of flexible components. Its design prioritises both performance and modularity. Based on the principle of a pull mechanism, the .NET Core framework allows applications to begin with a minimal infrastructure and then load the necessary modules from NuGet packages according to the application's needs. This framework consists of CoreCLR, an optimised runtime, and CoreFX, a subset of the.NET Framework class libraries. Figure 1 shows the main components of the system:

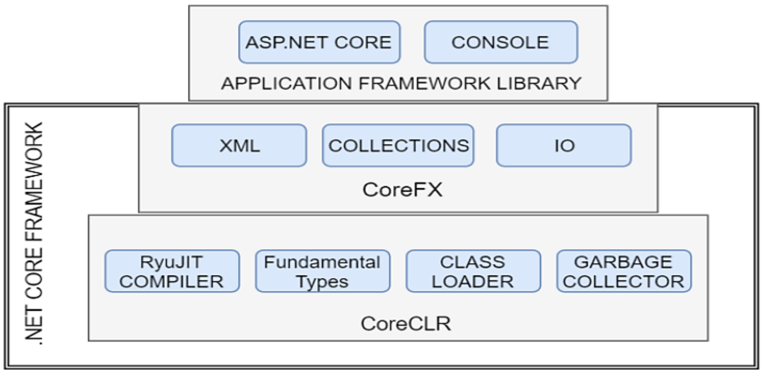


Fig 1: ASP.NET Core in .NET Core Platform

- Core CLR: The JIT compiler, type system, and class loader are all part of the.NET Core runtime, which is the most fundamental part of any runtime environment. Different configurations are developed for each OS platform since the Core CLR is built for each platform. App developed with the.NET framework This feature allows Core to run on any platform.
- CoreFX: A subset of the class libraries in the complete .NET Framework, .NET CoreFX includes essential libraries such as IO, collections, file systems, and XML. Because it specifies a shared foundation of essential libraries, it is also known as Unified BCL.
- Web server (Kestrel): The cross-platform .NET Core runtime makes use of the Kestrel web server. Thanks to the platform's strong separation of OS and web server functionality, the.NET Core platform's web server is no longer limited to Internet Information Services (IIS). For all of I/O needs, Kestrel has covered with its asynchronous I/O libraries. Standard fully-fledged web servers are used for all non-IO operations, including HTTP parsing and framing.

2.2. ASP.NET Core Framework is

Web Application Framework Core is the name of the latest version of the .NET Framework. Constructing state-of-the-art web apps in the cloud has never been easier than with this cross-platform framework. It uses modular components, which allow for more leeway when developing software solutions, and thereby promotes an application's decoupled structure. The ASP.NET Core framework allows for the creation of truly cross-platform applications, which means they are compatible with a wide range of platforms and operating systems [12]. A fully-functional web framework based on the new .NET Core architecture, it employs modern software design principles. As for software best practices, it employs the Model-View-Controller (MVC) design, a modular HTTP request pipeline (a method for efficiently processing requests using small discrete modules), and dependency injection.

Features

- Its foundation in NuGet packages makes it entirely modular, allowing users to include only the components that are really required by the application. Thanks to this method, it becomes a lightweight framework.
- It allows for a smooth migration from on-premises to cloud-based deployments.

- The ability to build portable programs and its support for multiple platforms are two of its main features.
- Separation of concerns and the ability to construct scalable, tested applications are benefits of its basic web architecture, which is based on the MVC pattern.

The HTTP request pipeline details the steps that a web app must take in order to reply to HTTP queries. At startup, the application solely launches the server-side components that are absolutely necessary. By calling their appropriate middleware, additional functionality is initialized. Loaded as middleware services are features such as authentication, logging, security, MVC, entity framework, etc., that are necessary throughout a request's lifespan. Figure 2 shows the fundamental flow of an HTTP request and the server's response.

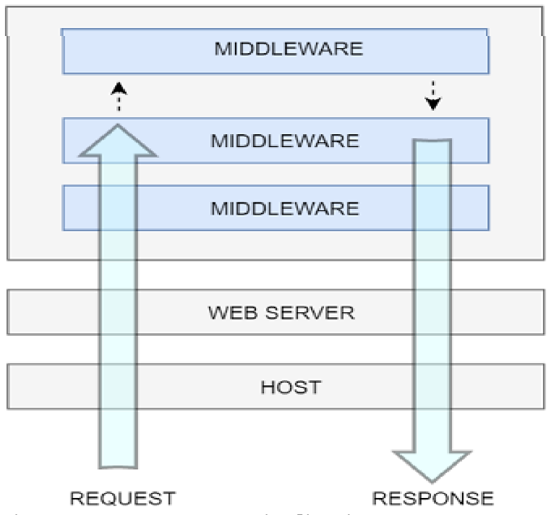


Fig 2: HTTP Request Pipeline in ASP.NET Core

The basic idea behind a request pipeline is to adhere to a clean request-response pattern, as seen in Figure 2. A web server passes a request from a browser on to an application, which in turn receives the request from a stack of middleware. Application pipelines rely on middleware, which is software that processes requests and replies. The various pieces of middleware can either process requests and send them on to the next layer of processing, or they can communicate directly without executing the remainder of the pipeline. The Model-View-Controller pattern is implemented by the framework in the form of ASP.NET Core MVC. Separating the program into its three main componentsthe model, the view, and the controlleris the goal of the MVC paradigm in software design. As a result, roles and their interdependencies can be better understood, which is crucial for the application's success.

- Model: It stands for the data in the domain. This is the only one of the three parts that can communicate with the database.
- View: The responsibility of presenting information to the user falls on View, as the name suggests. View templates, which are basically built into the components, make it easier to include code in the HTML file. Its primary function is to manage the visual aspects of user interfaces.
- Controller: The driver is the part of the system that talks to the model and view. The user is sent to the correct view once it establishes a connection with the model in response to a server request.

MVC allows for the separation of concerns, which in turn makes web application development more flexible. Therefore, it is possible to manage, test, and scale each component independently.

2.3. Key Features and Benefits of ASP.NET Core

A cutting-edge, open-source framework for building scalable, dynamic web applications is Microsoft's ASP.NET Core MVC [13][14]. As an integral aspect of the broader ASP.NET Core ecosystem, it offers a consistent foundation for building web and cloud applications. The performance-enhancing features of ASP.NET Core MVC set it apart from other frameworks:

- Modular Architecture: ASP.NET Core MVC is designed to be modular, allowing developers to easily add or remove components as needed. This improves the application's overall performance while reducing overhead. Because of its modular design, it is also very straightforward to update and maintain.
- Dependency Injection: The application is more testable and loosely coupled thanks to the framework's built-in dependency injection. Dependency injection improves application performance and usability by facilitating effective management of application elements.
- Asynchronous Programming: Asynchronous programming, which does not necessarily block, is encouraged by ASP.NET Core MVC. Improved scalability and responsiveness of web apps is achieved by this, which is particularly useful for I/O-bound operations such as database access and web service requests.
- Performance Improvements: There are a number of performance improvements in the ASP.NET Core MVC compared to earlier versions. Streamlined request processing, less memory footprint, and lightweight runtime all contribute to the application's enhanced performance.

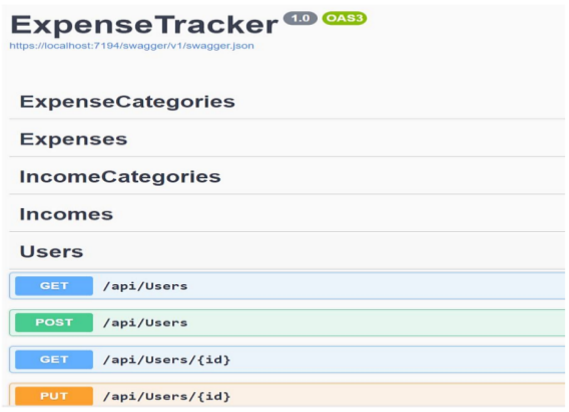
2.4. Authentication and Authorization

Authentication is a procedure that verifies the identity of a user or application and then the API is granted access. Authentication can be applied in ASP.NET Core by means of several schemes: There are several types of authentication, including Basic, Token-Based (e.g., JWT), OAuth 2.0, and Windows Authentication. The purpose of these safeguards is to ensure that the protected resources can only be accessed by authorized users or services. Authentication is usually done in either a login stage or token validation, whereby the user credentials or tokens get verified with configured identity providers or databases.

On the other side, a valid user's ability to access resources or execute methods is determined by the authorisation procedure. As a service, ASP offers authorisation based on roles and policies. Using the network core, and can authorise controllers, operations, or endpoints. Access control may be implemented either according to the role (e.g., Admin, User), claims, or according to the policies, which means that a user can only be able to execute the operations that are allowed by their assigned privileges. Both authentication and authorization should be properly applied in order to keep the API access secure and managed.

2.5. API Development

The ASP.NET Core Web API is used in the expenditure tracker software to create RESTful endpoints for managing the categories, income data, spending, and user authentication. To ensure that the API endpoints are functioning properly and that requests and answers behave as expected, Postman is utilised for testing purposes [15]. The API queries are integrated on the client side with the help of Angular services through the HTTP Client module, which depends on organised communication between the frontend and the backend. To enable the full CRUD operations, every functional area has conventional HTTP operations (GET, POST, PUT, and DELETE) [16]. These breakpoints aid in separating the presentation layer from the business logic layer and provide for secure data manipulation and access. The cost tracker application's API endpoints are illustrated in Figure. 3, which also displays the organization and structure of comparable services in ASP.NET Core.


ExpenseTracker 1.0 OAS3
https://localhost:7194/swagger/v1/swagger.json

ExpenseCategories
Expenses
IncomeCategories
Incomes
Users
GET /api/Users
POST /api/Users
GET /api/Users/{id}
PUT /api/Users/{id}

Fig 3: API Endpoints for Expense Tracker

2.6. Dependency Injection Pattern

Software design principles such as the Dependency Injection Pattern are extremely important in application programming languages like ASP.The architecture of the .NET Core MVC. To avoid requiring one object to be cognizant of the other objects on which it depends in order to carry out its function, dependency injection emphasises object isolation [17]. This style has a number of strengths, including allowing an object to be reconfigured by replacing the implementations of its dependencies without changing the components, making the dependencies of a particular object explicit by giving them formal arguments in the form of the object constructors and initializations, and easy testing by replacing the mock implementations of the dependent objects.

A crucial part of the ASP.NET Core MVC framework, the Dependency Injection Pattern streamlines both development and maintenance. It should be mentioned that ASP.NET Core follows the MVC design pattern. In this pattern, the Model is responsible for providing the View with data, the View for creating the interface and the user experience, and the Controller for processing user requests and making sure the correct Model and View are being used. Such a pattern not only facilitates reuse of code and separation of layers in applications, but it is also helpful in reducing the complexity and the workload on the developers and it is an attractive asset to develop generic software applications.

3. Authentication Mechanisms in Asp.Net Core

The authentication mechanisms are developed based on ASP.NET Core, such as claims-based authentication, token-based authentication (JWT), OAuth 2.0, and OpenID Connect. It also shows their working processes, security attributes, and involvement in the securing of RESTful APIs with the help of token management, delegated authorization, and encrypted communications via HTTPS/SSL.

3.1. Claim-Based Authentication

Clients and users can login with claim-based authentication, a more generic authentication mechanism, by requesting the system to verify credentials using claims about the user from an external source. To put it simply, a claim is any piece of information that identifies a person in some way [18]. For the purpose of clarity, it is comparable to an envelope that holds user data for a claim; claims are accepted as a name-value key pair made up of an authentication token that may contain a signature. External authentication services, such as Facebook, Twitter, Yahoo!, etc., rely on the rich framework provided by OAuth. It has recently gone mainstream, and find it in many different programs, including SharePoint 2013. Web API 2 and MVC5 both offer authentication based on claims.

3.2. Token-Based Authentication

The creation of detection rules, request logs, and the overall system architecture facilitates the execution and assessment of the suggested token-based monitoring system. A system that keeps tabs on token consumption and instantly flags any suspicious activity based on predetermined detection rules. The system identifies the use of tokens and shows the presence of active refresh tokens so that the administrator can revoke tokens of compromised accounts in a timely manner [19]. The system sends alerts when something is suspicious and the administrator is in a position to take direct action and investigate further. Using a web API that is based on JWT authentication, this is a representation of the suggested monitoring system (see Figure 4).

Client authentication through the authentication API is the first step when the client is accessing a protected API. A refresh token, which is a random string, is stored in a database and an access token, which stores user details, is issued to the user after they successfully log in by the authentication API [20]. Web storage (local/session storage) is where the access token is kept by the client, while cookies are where the refresh token is kept. Future calls to APIs that require protection are made with the access token, whereas the data of the corresponding request is also transmitted to the monitoring system API [21]. The client notifies the refresh token API of both the current and expired tokens when the access token expires. The API then makes new tokens, updates the database to show that the old token has been revoked, and sends the new request data to the system that is keeping an eye on things. In order to render the refresh token invalid, it must be sent to the revoke token API explicitly after logging out. At the same time, a risk assessment function is initiated and data collected from requests is added to a database by the monitoring system.

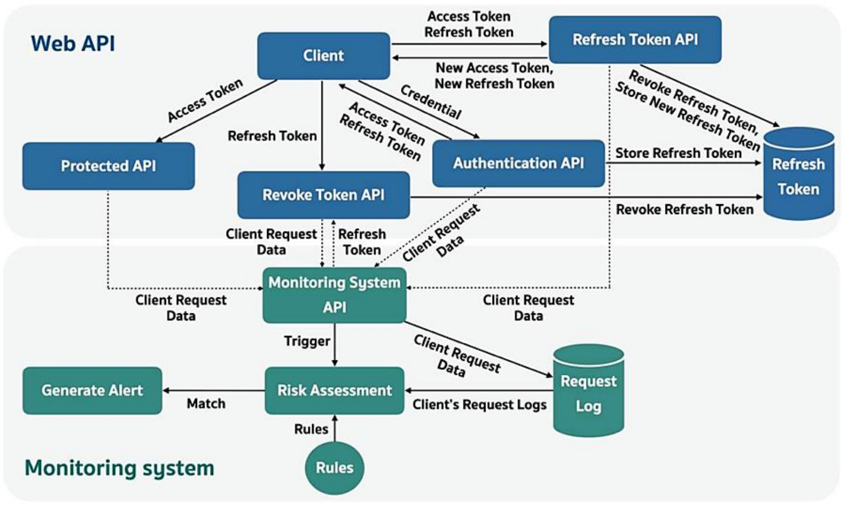


Fig 4: The Architecture of A Web API Using JWT Authentication

JWT and other tokens are stored on the client side instead of the server. As a result, the server is unable to explicitly revoke a token after it has been issued, when a user logs out, or during a token compromise [22]. Because members of the bad community can still access protected resources until the token automatically expires, this restriction presents a significant security risk. Several security studies have shown that this kind of technique is competitive with current methods for authenticating devices and users in IoT systems, and the goal of the analysis is to make authentication more resilient [23]. A server-generated token is a string that the client can receive in an HTTP request or another protocol's plaintext transmission.

3.3. OAuth2.0 Authentication

An OAuth framework grants restricted access to an HTTP service to an application developed by a third party. Either the resource owner can do it directly or have the third-party app negotiate and approve the interface between the resource owner and the HTTP service [24]. As an open standard, it enables clients to safely assign access to servers and resources. A resource server is a type of server that stores protected resources and uses access tokens to access them. This server has permission to use the protected resources and can react to them. After successful authentication, the client receives access tokens from the authentication server. It is crucial for the security of APIs and web services. The ASP.NET Web API frequently uses OAuth 2.0 to issue authentication tokens to clients so they can make queries. A protocol for managing authentication, OpenID Connect (OIDC) is based on OAuth2, the de facto standard in authorisation.

Flow and Security Mechanisms are:

Web apps and APIs can now securely share resources thanks to OAuth 2.0, a widely used security standard (Figure 5).

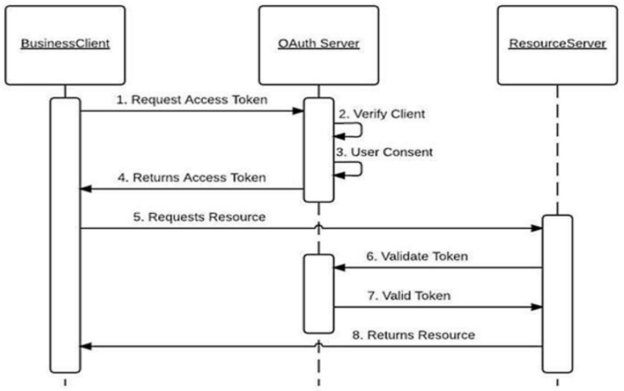


Fig 5: OAuth 2.0 Authorization Use Case Flow

Users are able to securely provide external parties access to their data while preserving their credentials thanks to OAuth 2.0, an authorization framework defined by the IETF through RFC 6749. With OAuth 2.0's token-based protocols, API security systems are created, which are more secure and scalable than old authentication techniques. Internet giants like Google, Microsoft, Facebook, and Amazon rely heavily on OAuth 2.0 to secure their APIs and authenticate users. OAuth 2.0 effectively safeguards computer systems from attacks that rely on credentials, such as phishing and session hijacking. Because OAuth 2.0 replaces direct credential sharing with access tokens, it reduces attack options and safeguards systems from unauthorized access.

ASP.NET Core programs are made more secure with the addition of OAuth 2.0. This version adds support for token expiration rules, refresh token authentication, and scope authorisation constraints. Strict usage limitations and brief token durations significantly reduce the number of attack vectors in a well-configured OAuth 2.0 implementation.

3.4. OpenID Connect

The OAuth Authorisation Protocol is extended by this protocol, which may also be used as an authentication protocol to safely sign users into online applications. ID tokens, introduced by OpenID Connect, are security tokens that enable user verification and identification. Make use of it to acquire refresh tokens, bearer tokens, and access tokens, all of which are security tokens. Small and secure, the Bearer token provides its owner with access to a restricted resource. Since bearer tokens do not have an intrinsic mechanism to prohibit unauthorized parties from utilising them, they must be transmitted over a secure channel like Transport Layer Security [25]. Refresh tokens represent persistent authorisation for a particular client to access resources on the owners' behalf. Clients and authorisation servers swap these tokens.

3.5. Secure with SSL

SSL is a top method for protecting RESTful web services. One way to ensure the security of data transfer with HTTPS is by utilizing encryption techniques. However, OAuth solutions that enable third-party application authentication like Facebook, Google, and Twitter are becoming increasingly popular for authorizing applications. Because of this, the user of the program can keep the credential in their possession and use it anytime they need access to the server's resources [26]. Restful web services sometimes get developed on top of ad hoc enhancements to older applications, which means that security measures are optimised for security management implementation but often forgotten. The success of an enterprise system is dependent on the security measures put in place. Therefore, appropriate protections must be in place to stop material from being altered, destroyed, lost, or disclosed without authorisation when it is created and submitted as a final judgement. Despite its seeming simplicity, the rest-based method is more susceptible to security concerns than standard web services since it does not conduct systematic security research and design.

4. API Security Threats and Practices in ASP.NET Core Applications

API security risks in an ASP.NET environment Injection attacks, faulty authentication, insecure data transmission, misconfiguration, and denial-of-service threats are all core application concerns. The article goes on to detail the best security techniques for dealing with these vulnerabilities and making APIs more resilient, including measures like rate limitation, safe code, an API gateway, and constant monitoring.

4.1. Common API Security Threats

Threats to API security could compromise the safety of data and services. It is critical to be aware of these dangers and take precautions. A few typical ones are these:

4.1.1. Injection Attacks

Injection attacks are threats that involve coding malicious code into API requests to control the backend systems. Examples are SQL injection and command injection [27], in which malefactors can use improper input validation to access or modify sensitive information. The use includes such preventive control measures as stringent input validation, parameterized queries, and secure coding.

4.1.2. Broken Authentication

Broken authentication vulnerability arises as a result of improper usage of identity verification tools. Weak password policies, exposed tokens or weak session management can provide attackers with access to information that is used to impersonate legitimate users. There is a necessity to perform effective authentication schemes such as multi-factor authentication (MFA) and secure management of tokens.

4.1.3. Insecure Data Transmission

The attacks of man-in-the-middle (MITM) able to capture the sensitive information when APIs leave the data untouched and unencrypted. Data in transit should be subjected to strict communication platforms such as HTTPS using TLS encryption.

4.1.4. Broken Access Controls

Broken access control occurs in cases where users gain unauthorized access to assets they are not meant to possess. This may lead to data exposure, data manipulation or system disruption. These risks are mitigated through the assistance of role-based access control (RBAC), least-privilege principles and relevant authorization checks.

4.1.5. Unrestricted Resource Consumption (Denial-of-Service Attacks):

Unlimited or unlimited use and rate-limiting APIs are susceptible to unnecessary requests that overload system resources to cause service disruption. Such attacks can be prevented by implementing rate limiting, throttling, monitoring and automated systems of attack detection.

4.1.6. Security Misconfiguration

API configuration is an exploitable vulnerability caused by default credentials, outdated software, debugging endpoints that are left open, or weak encryption configuration. This needs to be minimized by regular security audits, patch management and practices of secure configuration [28].

4.1.7. Improper Inventory Management

Their inability to have the right documentation and control of the API versions and endpoints may be a source of leaving the degraded or unsecured interfaces. Good API lifecycle, version management, and decommissioning of outdated endpoints are critical in ensuring security.

4.1.8. Unsafe Consumption of APIs

APIs are frequently based on third-party services. External APIs are prone to attacks, and no validation of their risks should be done blindly [29]. The third-party integrations can be tested with input validation and response verification, dependency monitoring, and security testing.

The awareness of these typical threats can help organizations to deploy powerful security architecture and deploy proactive defense measures that guarantee the reliability and security of API ecosystems.

Table I presents a comparative analysis of various authentication mechanisms used in ASP.NET Core applications [30]. It highlights their working principles, security levels, advantages, limitations, and suitable use cases. The comparison helps identify the most appropriate authentication strategy based on application requirements, scalability needs, and security considerations.

Table 1: Comparing Traditional vs. Modern Security

Feature	Traditional Security Approach	Modern Security Approach (OAuth, JWT, Zero-Trust)
Authentication Model	Session-based authentication using cookies and server-side session storage	Token-based authentication using JWT and OAuth 2.0 (stateless validation)
Access Control Mechanism	RBAC	Policy-based access control, ZTA, adaptive IAM
Threat Mitigation Strategy	Perimeter security (firewalls), static access rules	AI-driven anomaly detection, behavioral analytics, continuous monitoring
Scalability	Limited scalability due to server-side session dependency	High scalability with stateless tokens and cloud-native architecture
Infrastructure Model	Monolithic or centralized systems	Microservices and distributed cloud environments
Common Vulnerabilities	Session hijacking, credential theft, password-based attacks	Token leakage, improper token expiration, weak signing key management
Security Enforcement	Network-layer protection focused	Identity-centric security with continuous verification

4.2. API Security Best Practices

The following are some of the best practices when securing APIs:

- Strong Authentication and Authorization: This is one area that should make use of some of the best practices through the use of standards like OAuth to control access and API keys to make simple yet effective transactions [31][32].
- Broad Testing of APIs: APIs are also to be frequently tested to confirm the security controls and vulnerabilities, and fixed as soon as possible.
- Constant Surveillance: Be on constant surveillance of the API use and take measures in case of any suspicious activity in real-time.
- Rate Limiting: Rate limits must be added to curb abuse and threats relating to denial-of-service attacks.
- Secure Coding Practices: The APIs created with the consideration of the aspect of security and one can adhere to the steps suggested by the application of the secure coding rules in eliminating the prevalent vulnerabilities.
- Apply API Gateway: Uses API gateway to control, track, and secure API traffic with the aim of ensuring that APIs have another level of protection.
- Frequent Updates and Patch Management: Software may be updated on a regular basis with the latest available patches and updates, as this is one of the methods of minimizing the attack surface of software and, therefore, the attack vectors.

5. Literature Review

The comparative overview of the recent papers about secure API development, authentication schemes, and structure built on the ASP.NET Core, as depicted in Table II summarizes the research focus, research approach, core findings, constraints and issues, and frames the current developments and gaps in API security and enterprise integration models.

A. G. Bhartariya, S. K. Singh, and A. K. Bharti (2025) propose a systematic approach to API adoption by business organizations, based upon a comprehensive survey of current literature on API adoption strategies. Dwelling upon the increasing popularity of RESTful APIs in the modern connected world, the paper demonstrates how APIs are becoming one of the key facilitators of the process of uniting heterogeneous technologies, systems, and platforms. Through APIs, organizations able to access new business opportunities, scale operational efficiency, and provide innovative services to customers. The primary idea behind the given study is to develop a framework for data integration based on the RESTful APIs and especially security. In the paper, the author emphasizes the critical role of security in API-based integration, including authentication and authorization, as well as implementing the best practices in order to protect both data privacy and integrity [33].

A. C and D. Lino Abraham Varghese (2025) analyze existing protocols, such as those for identity, SMS, and federated authentication; delve into the fundamental principles of OAuth 2.0; and compare its security and performance features to those of traditional authentication models. Also provide an example of a real-world deployment with Spring Security, which shows the whole authorisation process, system architecture, and response behaviour. This demonstrates that OAuth 2.0 is a reliable choice for secure API interactions in multi-domain systems, as it enhances distributed resource access security while being user-friendly and scalable [34].

E. Hofmann (2025) offers a comprehensive study of these changes by placing the development of ASP.NET in the context of the evolving web service semantics, quality of service (QoS) issues and the multi-agent models. Based on the modern research and industrial practices, the analysis uses tools, strategies and implementation approaches to the web applications, which increase the efficiency and maintainability of the web applications critically. The methodology of this study involves a qualitative review of secondary and primary literature and focuses on the semantic web service discovery, context-awareness, and event-driven system patterns. It has been shown that the implementation of ASP.NET Core is not only optimized to ensure high performance but also enables the free flow of integration with the hybrid enterprise ecosystem, improves interoperability, and uses advanced data governance policies [35].

C. McCabe, A. I. C. Mohideen, and R. Singh (2025) discuss the historical use of username and password combinations is not sufficient protection and has led to the adoption of technologies like two-factor authentication (2FA). Although the 2FA provides some extra security through extra verification, the practices are not immune to attacks. Despite the introduction of 2FA, the relentless activities of the hackers are still formidable challenges to the security of digital space. This is an attempt to apply blockchain technology as a 2FA. The results of the given work indicate that blockchain-based 2FA systems might be more effective in enhancing the security of the digital world than traditional 2FA systems [36].

M. A. Talekattu, D. R. Katiyar, and D. S. B (2024) present the architecture and development of two separate ASP.NET Core web applications: one for booking resorts and another for tracking expenses. An ASP.NET Web API with Angular component can handle all of the user's money, while an ASP.NET MVC component would handle the resort's activities throughout the booking app. Additionally, in order to eliminate performance bottlenecks and provide optimal performance, the applications are safe, user-friendly, and scalable [37].

S. Phanireddy (2023) proposes a threat model that zeroes in on RESTful APIs in microservice settings, where they are vulnerable to insider threats, injections, and failed authentication. Multiple layers of protection, including robust authentication and authorisation, input validation, rate limitation, and an API gateway in the middle, are suggested by this threat model. The bulk of security concerns and system vulnerabilities can be effectively addressed by implementing these procedures, which have been proven in real-time circumstances. Additionally, the study delves into industry trends, highlighting the ever-evolving security landscape and the necessity of embracing AI security advancements like quantum cryptography [38].

Table 2: Comparative Analysis of Secure API Development, Authentication Mechanisms, and ASP.NET Core-Based Frameworks

Authors (Year)	Study On	Approach	Key Findings	Challenges	Limitations
A. G. Bhartariya, S. K. Singh, & A. K. Bharti (2025)	API adoption framework in business environments	Structured framework development through extensive literature review focusing on RESTful API integration and security mechanisms	APIs enable heterogeneous system integration, business innovation, and operational efficiency; strong emphasis on authentication, authorization, and data protection	Ensuring secure API integration across diverse enterprise systems	Conceptual framework; lacks large-scale empirical validation
A. C. C & D. Lino Abraham Varghese (2025)	OAuth 2.0 for secure API authentication	Comparative analysis of OAuth 2.0 with traditional authentication; real-world implementation using Spring Security	OAuth 2.0 improves security, scalability, and usability in distributed multi-domain systems	Managing authorization flow complexity and token management	Focused mainly on OAuth 2.0; limited evaluation against emerging authentication standards
E. Hofmann (2025)	The ASP.NET Framework and Its Evolution is foundational to online	Qualitative synthesis of literature; analysis of semantic web services, QoS, and multi-agent frameworks	ASP.NET Core enhances performance, interoperability, hybrid integration, and data governance	Integrating legacy systems with modern ASP.NET Core frameworks	Primarily literature-based; limited quantitative benchmarking

C. McCabe, A. I. C. Mohideen, & R. Singh (2025)	Blockchain-based Two-Factor Authentication (2FA)	Implementation of blockchain as enhanced 2FA security mechanism	Blockchain-based 2FA strengthens security beyond traditional 2FA methods	Complexity, cost, and scalability of blockchain implementation	Practical deployment challenges and lack of long-term performance evaluation
M. A. Talekattu, D. R. Katiyar, & D. S. B (2024)	Web application development using ASP.NET Core	Development of resort booking (MVC) and expense tracker (Web API + Angular) applications	Demonstrates scalability, security, and performance optimization using ASP.NET Core	Addressing performance bottlenecks and maintaining security	Application-specific case study; limited generalizability
S. Phanireddy (2023)	Security threat model for RESTful APIs in microservices	Development of layered security model including authentication, encryption, rate limiting, and API gateway	Multi-layered defense effectively mitigates authentication flaws, injection attacks, and insider threats	Evolving cyber threats and need for AI-driven and quantum-resistant security	Focuses mainly on RESTful APIs; limited empirical performance validation

6. Conclusion and Future Work

In ASP.NET Core, API implementation plays a significant role in protecting the trending web applications and cloud systems against various security threats. It provides a robust and scalable architecture using the frameworks of CoreCLR, CoreFX, Kestrel, middleware pipelines and embedded dependency injection to create scalable and high-performance APIs. The fact that it fully provides security, authentication, and authorization, enforces HTTPS, security headers, and protection mechanisms on data gives it a strong foundation for securing application resources. Secure access control systems include token-based and policy-based authentication methods; newer authentication systems include JWT, OAuth 2.0, OpenID Connect, and claims-based authentication. Additionally, the enumeration of common API threats, e.g., injection attacks, broken authentication and denial-of-service, points to the significance of active security countermeasures. Strong authentication, rate limiting, secure code, API gateways, and round-the-clock monitoring can all be implemented in an ASP.NET Core-based application to make it suitable in both an enterprise and a cloud-native environment.

The issues that may be discussed in the prospective study are the implementation of AI-based threat detection, zero-trust architecture prototypes, and automated security testing into the ASP.NET Core APIs. The resilience of APIs to the new cyber threats in the distributed and microservice-based environment can also be improved by investigating the concept of identity management using blockchains and adaptive access control systems.

References

[1] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, “Web API evolution patterns: A usage-driven approach,” *J. Syst. Softw.*, vol. 198, p. 111609, Apr. 2023, doi: 10.1016/j.jss.2023.111609.

[2] S. Prakash, “Web APIs -Uses , Roll , Challenges , Design points , in applications,” *Int. J. Creat. Res. Thoughts*, vol. 13, no. 3, pp. 352–363, 2025.

[3] J. W. Sajja, “Building Secure AI Agents for Autonomous Data Access in Compliance/Regulatory-Critical Environments,” *Comput. Fraud Secur.*, pp. 363–373, Sep. 2024, doi: 10.52710/cfs. 746.

[4] T. P. Gbenle, A. A. Abayomi, A. C. Uzoka, J. C. Ogeawuchi, O. S. Adanigbo, and O. T. Odofin, “Applying OAuth2 and JWT Protocols in Securing Distributed API Gateways: Best Practices and Case Review,” *Int. J. Multidiscip. Res. Growth Eval.*, vol. 3, no. 5, pp. 628–634, 2022, doi: 10.54660/.IJMRGE.2022.3.5.628-634.

[5] N. Prajapati, “The Role of Machine Learning in Big Data Analytics: Tools, Techniques, and Applications,” *ESP J. Eng. Technol. Adv.*, vol. 5, no. 2, 2025, doi: 10.56472/25832646/JETA-V5I2P103.

[6] S. Koukuntla, “Secure API Design and Authentication Strategies for Distributed Microservices Systems,” *Int. J. Contemp. Res. Multidiscip.*, vol. 3, no. 5, pp. 274–282, 2024.

[7] M. A. Rizvi and N. Jain, “Securing RESTful APIs with Middleware-based Threat Mitigation,” *Int. J. Comput. Appl.*, vol. 27, no. 5, pp. 52–60, 2025, doi: 10.5120/ijca2026926220.

[8] G. Modalavalasa, “Exploring Big Data Role in Modern Business Strategies: A Survey with Techniques and Tools,” *Int. J. Adv. Res. Sci. Commun. Technol.*, pp. 431–441, 2023, doi: 10.48175/ijarsct-11900b.

[9] F. Tanveer, F. Iradat, W. Iqbal, and A. Ahmad, “Towards Secure APIs: A Survey on RESTful API Vulnerability Detection,” *Comput. Mater. Contin.*, vol. 84, no. 3, pp. 4223–4257, 2025, doi: 10.32604/cmc. 2025.067536.

[10] H. P. Kapadia, “Voice and Conversational Interfaces in Banking Web Apps,” *J. Emerg. Technol. Innov. Res.*, vol. 8, no. 6, pp. g817–g823, 2021.

[11] H. P. Kapadia, “AI Enhanced Web Accessibility Features,” vol. 8, no. 4, pp. 476–483, 2021.

[12] D. Patel, “The Role of Amazon Web Services in Modern Cloud Architecture: Key Strategies for Scalable Deployment and Integration,” *Asian J. Comput. Sci.*

Eng., vol. 9, no. 4, pp. 1–9, 2024.

[13] P. R. Rao, S. Jain, and D. P. Tyagi, "Enhancing Web Application Performance : ASP . NET Core MVC And Azure Solutions," *J. Emerg. Trends Nov. Res.*, vol. 2, no. 5, pp. 309–326, 2024.

[14] S. K. Chintagunta and S. Amrale, "A Deep Learning Framework for Adaptive E- Learning : Integrating Learning Style Detection in Web-Based Platforms," *Int. J. Adv. Res. Sci. Commun. Technol.*, pp. 716–727, 2024, doi: 10.48175/IJARSCT-19397.

[15] A. Talekattu, R. Katiyar, and S. B, "Design of Web Applications in ASP.NET," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 12, no. 6, pp. 536–541, Jun. 2024, doi: 10.22214/ijraset.2024.63128.

[16] A. Warrier, "Securing and Scaling API Gateways in Hybrid Environments," *J. Artif. Intell. Mach. Learn. Data Sci.*, vol. 3, no. 3, pp. 2914–2920, Sep. 2025, doi: 10.51219/JAIMLD/Arjun-warrier/607.

[17] P. R. Marapatla, "Building a Comprehensive API Ecosystem for Non-profit Digital Analytics," *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol.*, vol. 11, no. 1, pp. 1167–1172, Jan. 2025, doi: 10.32628/CSEIT251112121.

[18] A. R. Toorpu, S. K. Vududala, A. Nerella, and B. P. Madupati, "Hybrid AI Models for Privacy-Preserving Big Data Analytics in Distributed Environments," in *2025 Global Conference in Emerging Technology (GINOTECH)*, IEEE, May 2025, pp. 1–8. doi: 10.1109/GINOTECH63460.2025.11076666.

[19] B. B. Rao and A. A. Waoo, "A Token-Based Authentication System That Identifies Users And Device In An Iot Application/Ecosystem," *Journal Emerg. Technol. Innov. Res.*, vol. 7, no. 10, pp. 3066–3069, 2020.

[20] P. R. Marapatla, "Intelligent APIs: AI-Powered Ecosystem for Nonprofit Digital Transformation," *J. Inf. Syst. Eng. Manag.*, vol. 10, no. 60s, pp. 605–618, Sep. 2025, doi: 10.52783/jisem.v10i60s.13174.

[21] P. Rujichaikul and I. Rassameeroj, "Token-Based Authentication Monitoring System," *J. Cyber Secur. Mobil.*, vol. 14, no. 4, pp. 777–798, oct. 2025, doi: 10.13052/jcsm2245-1439.1441.

[22] A. Nerella and J. W. Sajja, "Responsible AI in Enterprise Applications: Balancing Innovation and Compliance," *Comput. Fraud Secur.*, vol. 2023, no. 7, Jul. 2023, doi: 10.52710/cfs. 744.

[23] A. Syed, "Securing IoT-Driven Supply Chains," in *Supply Chain Software Security*, Berkeley, CA: Apress, 2024, pp. 289–342. doi: 10.1007/979-8-8688-0799-2_7.

[24] V. Chandra, N. Challa, and S. K. Pasupuletti, "Authentication and Authorization Mechanism for Cloud Security," *Int. J. Eng. Adv. Technol.*, vol. 8, no. 6, pp. 2072–2078, Aug. 2019, doi: 10.35940/ijeat.F8473.088619.

[25] M. R. R. Deva, "A review of application programming interface management systems and their role in seamless integration between software applications," *Asian J. Comput. Sci. Eng.*, vol. 9, no. 1, pp. 1–9, 2025.

[26] M. I. Hussain and N. Dilber, "RESTful Web Services Security By Using ASP.NET Web API MVC-Based," *J. Indep. Stud. Res. Comput.*, vol. 12, no. 1, 2014, doi: 10.31645/2014.12.1.1.

[27] A. H. Yadav, N. J. Yadav, and O. B. Singh, "Enhancing API Security: Strategies, Challenges, and Best Practices," *Int. J. Res. Publ. Rev.*, vol. 5, no. 6, pp. 2900–2906, 2024, doi: 10.2139/ssrn.4909110.

[28] A. Warrier, "Enterprise Healthcare API Management: Authentication, Authorization, and Rate Limiting for Regulated Environments," *J. Adv. Dev. Res.*, vol. 10, no. 1, jun. 2019, doi: 10.71097/IJAIDR.v10.i1.1572.

[29] S. Thangavel, S. Srinivasan, S. B. V. Naga, and K. Narukulla, "Distributed Machine Learning for Big Data Analytics: Challenges, Architectures, and Optimizations," *Int. J. Artif. Intell. Data Sci. Mach. Learn.*, vol. 4, no. 3, pp. 18–30, Oct. 2023, doi: 10.63282/3050-9262.IJAIDSML-V4I3P103.

[30] S. S. Chinthalapudi, "Enhancing Security in ASP.NET Core Applications: Implementing Oauth, JWT, and Zero-Trust Models," *Int. J. Innov. Sci. Res. Technol.*, vol. 10, no. 3, pp. 2561–2575, Apr. 2025, doi: 10.38124/ijisrt/25mar1677.

[31] V. Borate, A. Adsul, and S. Bhusari, "Securing APIs: Strategies, Standards, and Best Practices," *Int. J. Adv. Res. Sci. Commun. Technol.*, vol. 5, no. 3, pp. 208–217, Nov. 2025, doi: 10.48175/IJARSCT-29828.

[32] J. E. Kofi, "Monitoring Cloud Performance Metrics Utilizing AI to Estimate the Efficiency of Cloud Operations," in *2025 7th International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, IEEE, Dec. 2025, pp. 1–6. doi: 10.1109/ISAECT68904.2025.11318827.

[33] A. G. Bhartariya, S. K. Singh, and A. K. Bharti, "Framework for Data Integration in Cross-Domain Pervasive Environments: Lightweight and Secure Restful Api," *J. Theor. Appl. Inf. Technol.*, vol. 103, no. 18, pp. 7362–7375, 2025.

[34] A. C C and L. A. Varghese, "A Scalable OAuth 2.0-Based Authorization Framework for Secure Resource Access in Distributed Systems," *Int. J. Creat. Res. THOUGHTS*, vol. 13, no. 5, pp. 381–391, 2025, doi: 10.56975/ijcrt.v13i5.286865.

[35] E. Hofmann, "Advancing Web Application Architectures: Evolution from ASP.NET to ASP.NET Core and the Integration of Semantic and Event-Driven Frameworks," *Int. J. Mod. Med.*, vol. 04, no. 10, pp. 131–139, 2025.

[36] C. McCabe, A. I. C. Mohideen, and R. Singh, "A Blockchain-Based Authentication Mechanism for Enhanced Security," *Sensors*, vol. 24, no. 17, p. 5830, Sep. 2024, doi: 10.3390/s24175830.

[37] M. A. Talekattu, "Design of Web Applications in ASP.NET," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 12, no. 6, pp. 536–541, Jun. 2024, doi: 10.22214/ijraset.2024.63128.

[38] S. Phanireddy, "Securing RESTful APIs in Microservices Architectures: A Comprehensive Threat Model and Mitigation Framework," *Int. J. Emerg. Res. Eng. Technol.*, vol. 4, no. 2, pp. 64–73, 2023, doi: 10.63282/3050-922X.IJERET-V4I2P107.