Università di Catania
Dipartimento di Matematica e Informatica

Project - Internet Security

Broken Access Control

Paolo Maria Scarlata - 1000025346

Academic year 2023-2024

Abstract

This document delves into the crucial role of access control mechanisms in safeguarding web applications and highlights the vulnerabilities arising from improper implementation, with a particular focus on Broken Access Control vulnerabilities. Access control is fundamental to modern web security, ensuring that users interact with resources in a regulated manner. However, weaknesses such as Insecure Direct Object References (IDOR), privilege escalation, and improper authentication mechanisms expose systems to significant threats, including unauthorized access to sensitive data and administrative functions.

The report provides a detailed analysis of access control models, including Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC), and explores their application in real-world scenarios. Using a deliberately vulnerable web application as a case study, the report demonstrates how flaws in JSON Web Token (JWT) implementation—such as the use of an empty secret key and insufficient token validation—can be exploited to bypass authentication and authorization protocols. Additionally, it showcases the exploitation of IDOR vulnerabilities to access unauthorized user data through URL manipulation.

To address these risks, the document proposes a multi-layered security strategy, emphasizing defense-in-depth, encryption of sensitive data, server-side validation, and the implementation of robust authentication and session management practices. Furthermore, the report underscores the importance of adopting security frameworks, such as OWASP guidelines, and conducting regular vulnerability assessments to adapt to the evolving threat landscape.

The findings of this study aim to raise awareness about the critical nature of access control vulnerabilities and provide actionable recommendations for developers and security professionals to enhance the resilience of their web applications.

Contents

1 Foundations of Access Control

In modern web applications, access control serves as the cornerstone of security architecture, acting as the primary mechanism for regulating user interactions with system resources. This fundamental security service operates by mediating resource access based on user identity, supported by complementary security measures such as authentication and authorization protocols.

Access control's significance extends beyond simple permission management, playing a crucial role in maintaining data confidentiality. While encryption often takes center stage in computer security discussions, it primarily functions as an implementation tool for enforcing access control policies. This relationship highlights the interconnected nature of security measures in protecting sensitive information.

The cybersecurity landscape presents numerous challenges related to access control implementation. Web applications face constant threats from malicious actors who exploit vulnerabilities in access control systems. These exploitations can lead to severe consequences, including unauthorized data exposure, system manipulation, and service disruption. Of particular concern are scenarios where attackers bypass authorization mechanisms, potentially gaining administrative privileges and compromising entire systems.

To mitigate these risks, organizations must implement precise access restrictions that align with the principle of least privilege. This approach requires careful consideration of user roles and their corresponding access needs, ensuring that individuals can access only the resources necessary for their specific functions. However, the implementation of such controls presents significant challenges, as evidenced by Broken Access Control's position as the top security risk in recent security assessments.

1.1 Implementing Access Control Systems

In today's rapidly evolving technological landscape, businesses have fundamentally transformed their service delivery methods to meet growing customer expectations and changing behavioral patterns. Web applications have become the primary tool across industries for managing critical business processes, including supply chain management, customer relationship management, and staff management. These web applications leverage session management functionality to respond swiftly and securely to service requests from authorized users. A typical application session begins with user verification through specific authentication factors, such as username and password. This authentication process enables the application to create a personalized environment where users can securely access their authorized resources. Access control features serve as crucial security mechanisms that regulate access to web resources, including web pages and database tables, while preventing unauthorized access attempts. However, the widespread adoption of web applications for business operations, while revolutionary, has introduced heightened risks when vulnerabilities persist due to inadequate design and coding practices during development. The architecture of web applications' access control systems may contain various design flaws, including insufficient input validation, sensitive data exposure, session misconfiguration, and directory accessibility issues. These vulnerabilities can potentially grant elevated privileges to both regular users and malicious actors. When exploited, these Broken Access Control vulnerabilities can lead to severe consequences, from unauthorized administrative access to complete system compromise. The significant impact of authentication and access control vulnerabilities has been demonstrated through their discovery in widely-used software applications like IIS and WordPress. These weaknesses often manifest through various attack vectors, including SQL injection, cross-site request forgery, cross-site scripting, and file inclusion vulnerabilities, largely stemming from inadequate user authentication and session management practices.

1.1.1 Key Models of Access Control

- **Discretionary Access Control (DAC**) systems place control in the hands of the resource owner or authorized individuals. This system allows owners to manage other users' permissions directly, determining who can read, write, execute, or access specific resources. While highly flexible and widely used in both business and government sectors, DAC's inherent flexibility can also be its weakness. It proves particularly effective for homes or small businesses requiring security for limited access points, allowing business owners or homeowners complete authority over entry and exit management.

- **Mandatory Access Control (MAC)** systems provide enhanced security by removing end-user control over access permissions. Instead, system administrators maintain centralized control over access rights, classifying end users and determining their access to specific areas. This approach contributes to improved security and data protection through strict hierarchical control. While excellent for complex security requirements, MAC systems are best suited for small to medium organizations due to the manual effort required for permission management.
- **Role-Based Access Control (RBAC)** systems assign permissions based on organizational roles rather than individual users. This approach automatically grants necessary permissions to users based on their designated roles within the organization. RBAC proves particularly effective for large organizations with numerous employees, though it can also benefit smaller companies seeking to enhance their security structure. The system's ability to automatically assign permissions based on roles significantly reduces administrative overhead and improves security consistency.

Beyond these systems, access controls are categorized into three fundamental types based on user interaction: Vertical access controls manage access to sensitive functionality between different user levels, ensuring various user categories can access appropriate application functions based on their role. Horizontal access controls prevent users from accessing resources belonging to other users at the same privilege level, maintaining individual user privacy and data security. Context-dependent access controls determine resource accessibility based on the application's current state or user interaction sequence, preventing out-of-order operations that could compromise system integrity. The implementation of these systems and control types creates a comprehensive security framework essential for modern web applications. Their effectiveness depends on proper configuration and regular maintenance to address evolving security challenges.

1.2 Evolution of Broken Access Control Risks

The evolution of Broken Access Control within the OWASP Top 10 security risks framework presents a fascinating trajectory of how this vulnerability has gained increasing prominence in web security. This chapter examines the historical progression of access control vulnerabilities through various OWASP releases, highlighting the shifting understanding and categorization of these security risks.

In 2003, when OWASP released its first Top 10 list, access control-related vulnerabilities appeared under the designation "Remote Administration Flaws," positioned as the second most critical security risk (A2). The following year, 2004, marked an important shift in terminology, with the vulnerability being renamed to "Broken Access Control" while maintaining its A2 position. This change reflected a growing understanding of the broader implications of access control vulnerabilities beyond remote administration issues.

A significant transformation occurred in 2007 when OWASP split the concept into two distinct categories: "Insecure Direct Object Reference" (A4) and "Failure to Restrict URL Access" (A10). This division acknowledged the different manifestations of access control vulnerabilities and their unique security implications. The 2010 update maintained "Insecure Direct Object Reference" at A4 while elevating "Failure to Restrict URL Access" from A10 to A8, indicating its increasing importance.

The 2013 release brought further refinement to the categorization. While "Insecure Direct Object Reference" remained unchanged at A4, "Failure to Restrict URL Access" evolved into "Missing Function Level Access Control" and moved to position A7. This modification reflected a more nuanced understanding of function-level security concerns in web applications.

A pivotal moment came in 2017 when OWASP consolidated these separate categories back into a single entry. "Insecure Direct Object Reference" (A4) and "Missing Function Level Access Control" (A7) were merged under the unified heading of "Broken Access Control," positioned at A5. This consolidation recognized the interconnected nature of these vulnerabilities and the need for a comprehensive approach to access control security.

The most recent development occurred in 2021, when "Broken Access Control" rose to the top position (A1) in the OWASP Top 10. This elevation to the highest risk category underscores the critical nature of access control vulnerabilities in modern web applications and their potential impact on system security.

This historical progression reveals several key insights. First, it demonstrates the security community's evolving understanding of access control vulnerabilities. Second, it highlights the dynamic nature of web

security threats and the need for continuous adaptation in security frameworks. Finally, it emphasizes the growing importance of access control in the overall security landscape of web applications.

The following chapters will build upon this historical foundation to examine current vulnerabilities, detection methods, and protection strategies in detail. Understanding this historical context is crucial for appreciating the complexity of current access control challenges and developing effective security solutions.

1.3 Common Access Control Vulnerabilities

A Broken Access Control vulnerability occurs when unauthorized users gain access to restricted resources, circumventing security processes. Exploiting these flaws can lead to unauthorized access to private information or systems, often due to weak authentication and permission protocols.

Common access control vulnerabilities manifest in various forms. One example involves completely unprotected functionality, where sensitive features or data can be accessed directly through specific URLs due to the lack of server-side enforcement. In many cases, applications rely on user interface elements to control access rather than implementing robust backend validation.

Another example is identifier-based functions, where resource access depends on identifiers passed as URL parameters or POST requests. If these identifiers are predictable, unauthorized users can manipulate them to access restricted resources without authentication.

Multistage functions, often implemented across multiple steps, present another vulnerability. When applications fail to validate access at each step, attackers can exploit these assumptions, bypassing restrictions by targeting specific stages in the process.

Static files hosted on a server can also be a weak point. These resources are sometimes excluded from dynamic access control mechanisms, allowing users to access restricted files by directly navigating to their URLs.

Platform misconfigurations, such as incorrect rules for HTTP methods or URL paths, create further risks. Inadequate setup at the platform level can result in unauthorized access, despite apparent safeguards. For instance, POST requests to administrative paths might inadvertently bypass authentication checks if misconfigured.

Insecure methods of implementing access control exacerbate these vulnerabilities. Parameter-based access control, for example, uses hidden fields, cookies, or query strings to determine user permissions. If these mechanisms are manipulated, attackers can bypass restrictions and gain elevated access.

Reliance on the HTTP Referer header for access control is similarly flawed. This approach assumes the validity of Referer values in requests, which attackers can forge to gain unauthorized access to sensitive resources.

Location-based access control, which restricts access based on geographical location, can also be easily circumvented. Techniques such as using proxies, VPNs, or altering geolocation settings allow attackers to disguise their true locations and bypass these restrictions.

Eliminating these vulnerabilities requires rigorous implementation of server-side access controls, periodic auditing of configurations, and comprehensive testing for weak points. A robust approach to access control ensures the security of sensitive systems and data while mitigating the risks posed by unauthorized access.

Broken Access Control vulnerabilities open numerous pathways for attackers to exploit weaknesses in application security. Several common methods stand out in terms of their prevalence and impact.

Injection flaws arise when untrusted input is injected into an application, causing unexpected behavior. This can allow attackers to modify application data or access sensitive information without authorization. Such flaws often result from inadequate input validation and pose significant risks to data integrity and confidentiality.

Cross-Site Scripting (XSS) occurs when untrusted input is integrated into a web page's output, enabling attackers to execute malicious scripts in a user's browser. This can lead to consequences such as cookie theft, session hijacking, and unauthorized actions within the context of a user's session.

Broken item authentication and session management present another critical risk. These vulnerabilities occur when applications fail to validate or protect authentication and session-related data adequately. Attackers can exploit this to gain unauthorized access to sensitive resources, manipulate session data, or impersonate legitimate users.

1.4 Strategies for Securing Access Control

Securing web applications involves several concerns, effective access control should be built. To help developers in building effective control over access, all vulnerabilities that may give attackers some room to exploit should be kept in mind. Besides, the following points should be prioritized by the developers:

- The knowledge of the user: Do not presume that the users' access may be unaware of approachable URLs, account numbers, and document IDs.
- Assume that not all user-submitted parameters are trusted.
- Be mindful that the user may access pages in an inappropriate order.

Additionally, developers can follow some good practices, such as basing all access control actions on the user's session and checking access restrictions using a central application component. Furthermore, if the functionality is sensitive, such as administrative pages, developers can further restrict access by utilizing IP addresses, which act as a form of authentication to restrict functionality access to users within a specific network range.

While there are numerous good practices that can be adopted, the most effective method is implementing multiple layers of security through a defense-in-depth approach. This involves using a multilayered privilege model where access controls are enforced at each layer. A matrix can be used to display user roles in the application and the privileges granted to each role at each tier.

1.5 Methods for Identifying Access Control Flaws

Static and dynamic analyses are the primary methodologies available for detecting Broken Access Control vulnerabilities. Dynamic analysis is utilized to identify bugs in programs that are actively running. Conversely, static analysis operates in a non-runtime context, employing a different strategy for precision analysis.

One technique within static analysis is **force browsing**, which involves bypassing links presented on a webpage and using brute force techniques to gain direct access to specific resources. This method often mimics the actions of lower-privileged users who attempt to access resources intended for higher-privileged users.

Context-based detection identifies security-sensitive operations within a program. This approach employs methodologies such as grammar without context to analyze the program's structure. Researchers argue that these operations should exhibit consistent contextual elements and redirects. Consistency between these factors enables the identification of potential access control vulnerabilities within the software.

Model-based detection begins with an analysis of the architecture of an access control implementation. Researchers construct a comprehensive framework to accurately model the implementation. This model includes components for access control mechanisms, which are then assessed to verify the correctness of their implementation in web applications. The objective is to identify potential weaknesses in the access control system.

Graph-based detection provides a flexible approach to analyzing web applications. This method uses representations such as control flow graphs, site maps, and permission verification graphs. Nodes within these graphs typically represent web pages or resources, and researchers use algorithms, flow analysis, and restrictions to evaluate the reachability between nodes. If a non-privileged user can access a privileged node, access control flaws are identified. This technique is highly scalable, allowing researchers to propose suitable solutions based on the specific characteristics of the web application once vulnerabilities are detected.

Most of the research depends on static detection. Static detection has some limitations, such as depending on source codes, making it difficult for the third party to detect Broken Access Control because no one will give source codes to the third party to examine it. Also, static detection cannot detect the new hacking techniques.

Dynamic Application Security Testing (DAST) involves analyzing the entire running application to detect vulnerabilities, including those related to broken access control. It simulates attack scenarios by interacting with the application in real time, focusing on how its various components, such as front-end, back-end, and APIs, behave under potentially malicious input. DAST does not rely on prior knowledge or

documentation of the application; it dynamically explores the application during runtime, identifying issues like unauthorized access to resources or privilege escalation. Its broader scope makes it a versatile tool for testing live applications comprehensively.

On the other hand, **automated detection through API** specification processing focuses specifically on APIs, analyzing their design and structure as documented in specifications like OpenAPI or Swagger files. This approach often occurs earlier in the development lifecycle, allowing developers to identify potential vulnerabilities before the API is live. By examining the parameters and access control mechanisms outlined in the documentation, it targets issues unique to APIs, such as overly permissive endpoints or insecure direct object references (IDOR). This method is particularly effective in pre-runtime phases or for assessing API-specific vulnerabilities.

The key difference lies in their scope and timing. DAST is a broad, runtime-focused approach that dynamically explores the entire application without needing prior knowledge of its structure, making it ideal for testing live systems. In contrast, automated API detection is narrower in focus, relying on pre-existing API documentation to predict vulnerabilities, particularly during the development or pre-deployment phases, and is tailored specifically to API-centric issues. These distinctions highlight their complementary roles in a comprehensive security testing strategy.

2 Analysis of Vulnerable Implementations

The application consists of four main files that together create a web-based user management system. Each file serves a specific purpose in the application's architecture, forming a complete client-server implementation.

Access control vulnerabilities represent some of the most critical security flaws in modern web applications, often leading to unauthorized access to sensitive data and system functions. This technical demonstration explores a purposely vulnerable web application designed to illustrate common manifestations of broken access control mechanisms. The application serves as an educational platform for understanding how these vulnerabilities can be identified, exploited, and ultimately prevented in real-world systems.

The demonstration system implements a user management platform built using Node.js and Express.js, incorporating JSON Web Tokens (JWT) for authentication. The application manages sensitive user information, including banking details and fiscal data, making it an ideal candidate to demonstrate the severe implications of access control failures. The system architecture distinguishes between administrative and standard user roles, with distinct permission sets for accessing and managing user data.

Figure 1: JWT Structure

The core functionality includes a comprehensive user management system with an administrative panel, personal data access APIs, and role-based access controls. The application maintains user profiles containing sensitive information, such as credit card numbers, fiscal codes, and IBAN numbers, protected by a JWT-based authentication system. The system's access control matrix theoretically restricts standard users to accessing only their personal data, while administrators have broader permissions including user management capabilities.

For demonstration purposes, the system includes two deliberately implemented vulnerabilities: Insecure Direct Object Reference (IDOR) and Privilege Escalation through JWT manipulations. These vulnerabilities represent common security oversights in real-world applications, though this implementation intentionally amplifies their exploitability for educational purposes.

The application provides multiple user accounts for testing these vulnerabilities, including an administrative account (username: giampaolo, password: adminpass) and standard user accounts (paolo/password1 and sergio/password2). These accounts facilitate the demonstration of both horizontal privilege bypass, where users can access data of others at their same privilege level, and vertical privilege escalation, where standard users can gain administrative capabilities.

It is crucial to emphasize that this application has been developed exclusively for educational and research purposes. The intentional security vulnerabilities embedded within the system make it fundamentally unsuitable for production environments or real-world deployment. Instead, it serves as a controlled environment for security professionals, developers, and students to understand the mechanisms of broken access control and the importance of proper security implementations.

2.1 Breakdown of Vulnerable System Design

Serving as the backbone of the application, the server.js file implements an Express.js server to manage user authentication and data processing. This server relies on an in-memory database represented by an array of user objects, storing sensitive personal information such as credit card numbers, fiscal codes, and IBAN numbers. Key functionalities include a login route for generating JWT tokens, a user data retrieval endpoint, and administrative endpoints for user management. Despite its use of JSON Web Tokens, the authentication system employs an empty secret key, raising security concerns. Middleware for token verification is also present but relies on a simple decode operation instead of proper validation.

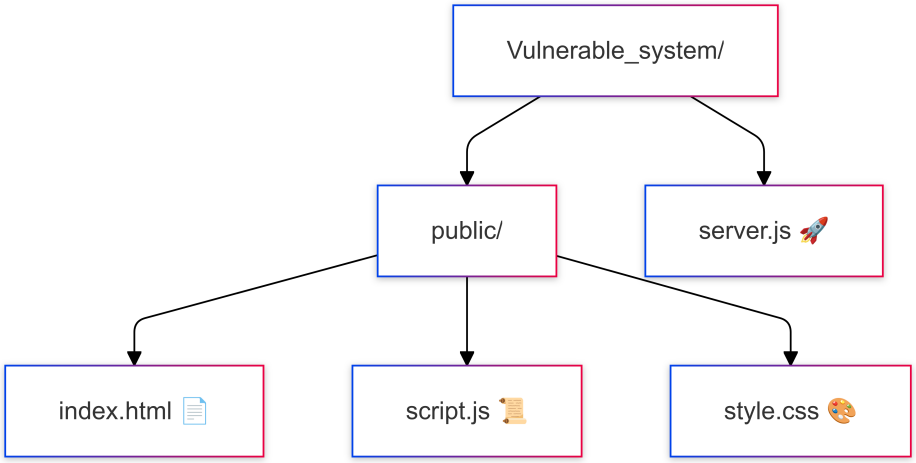

Figure 2: Directory's structure

On the frontend, the implementation spans three files: index.html, style.css, and script.js. The index.html file offers the basic structure of the application interface, organizing it into two main sections—a login form and a dashboard. Depending on user permissions, the dashboard dynamically displays either user data or administrative controls. Its minimalistic structure prioritizes functionality over complexity, ensuring ease of use.

Visual styling is handled by the style.css file, which crafts a clean, professional appearance with a consistent color scheme. A light background contrasts with green accents on interactive elements. Responsive design principles are employed, ensuring usability across various screen sizes. Features like card-style containers for data, subtle shadows for depth, and hover effects for interaction enhance the modern, user-friendly interface.

Client-side functionality resides in the script.js file, where JavaScript code manages user interactions and communicates with the server. Core functions include handling user authentication, fetching and displaying user data, and managing the administrative panel. Server communication is facilitated by the fetch API, using proper headers for authentication. The file also supports administrative actions, such as user deletion, ensuring dynamic management of the application state.

This application's architecture adopts a traditional client-server model with a clear division of responsibilities. While the server handles authentication and data management, the client focuses on user interface and interaction. RESTful API endpoints facilitate communication, using JSON for data exchange. Despite its functionality, the architecture requires significant security enhancements to meet production standards.

Sensitive personal information is a critical aspect of the data management system. User records include typical account details, such as usernames and passwords, alongside highly sensitive data like credit card numbers, fiscal codes, and IBANs. Role-based access control distinguishes between standard users and administrators, but the current implementation suffers from vulnerabilities that could compromise security.

To maintain application state, client-side JavaScript employs global variables for the authentication token and user information. Functions are implemented to handle both user-specific and administrative tasks, with conditional rendering based on user roles. Error handling is present for server communication,

offering user feedback through the interface, though best practices for secure client-side data handling are not fully implemented.

Consistency in visual design is achieved through the styling implementation, which uses a container-based layout system. Proper spacing, alignment, and clear visual hierarchy contribute to a professional and organized appearance. Form elements, data displays, and administrative tools are styled to ensure clarity and usability, with interactive states enhancing the user experience.

These components together form a functional web application that showcases common patterns in web development, such as user authentication, role-based access control, and data management. However, the implementation highlights several critical security flaws that must be addressed before deployment in a production environment.

2.2 Code-Level Security Vulnerabilities

The provided server implementation exhibits numerous significant security vulnerabilities that could severely compromise user data and system integrity. This analysis examines the various security flaws present in the system and their potential implications.

2.2.1 Specific Broken Access Control Weaknesses

Privilege-Escalation : The most immediate concern lies in the implementation of JSON Web Tokens (JWT) for authentication, the server utilizes an empty secret key for token signing, which fundamentally undermines the entire security model. This critical oversight allows malicious actors to forge valid tokens with ease, effectively bypassing the authentication system entirely. Moreover, the authenticate middleware employs jwt.decode() instead of jwt.verify(), meaning the system never validates token signatures. This implementation flaw compounds the empty secret key issue, as the server blindly trusts any well-formed JWT without verifying its authenticity.

IDOR : Another severe vulnerability present in the system is the implementation of Insecure Direct Object References (IDOR) at the user data endpoint. The /api/users/:userId/data endpoint accepts a user ID parameter without performing any ownership validation, allowing any authenticated user to access any other user's personal data simply by modifying the userId parameter in their requests. This vulnerability is particularly concerning given the sensitive nature of the data being exposed, which includes credit card numbers, fiscal codes, and IBAN numbers.

2.2.2 Additional Security Flaws

The authentication system itself contains fundamental security flaws that extend beyond the JWT implementation. User passwords are stored in plaintext within the users array, making them immediately accessible if the system is compromised. The absence of password hashing and salting mechanisms means that user credentials are perpetually at risk (if a server.js.bak is located in the public directory, then is readable to the attackers, and the username and password are hardcoded in plaintext). Furthermore, the login endpoint lacks rate limiting, making it susceptible to brute force attacks. The system also lacks proper session management and token expiration mechanisms, meaning that once a token is issued, it remains valid indefinitely (vulnerable frome replay attacks).

The handling of sensitive personal data throughout the application raises serious privacy concerns. Credit card numbers, IBAN numbers, and fiscal codes are stored and transmitted in plaintext, without any encryption or data masking(whitout fixing the plaintext stored data, even the signed token becomes vulnerable). This exposure of sensitive financial information violates numerous data protection regulations and puts users at significant risk of financial fraud. The system should implement proper encryption for sensitive data both at rest and in transit.

These security flaws represent significant risks to user privacy and system integrity. The combination of improper JWT handling, IDOR vulnerabilities, plaintext password storage, and unencrypted sensitive data creates a perfect storm of security vulnerabilities. This implementation serves as a compelling example of

how seemingly functional code can harbor serious security flaws that could lead to devastating breaches in a production environment.

2.3 Understanding the Attacker Profile

The attacker model considered in this analysis corresponds to the profile of a **General Attacker**, characterized by the use of accessible tools and a basic understanding of common exploitation techniques. This attacker is assumed to possess valid credentials for logging into the system as a standard user, allowing them to initiate attacks from within the application.

The tools employed by this attacker include widely available resources such as Burp Suite, which enable the interception and manipulation of HTTP requests. The attacker's technical knowledge is moderate, focusing on fundamental web application principles such as session management, URL structures, and parameter manipulation, rather than advanced cryptographic or protocol-level expertise.

The primary objective of this attacker is to exploit vulnerabilities in the application to gain unauthorized access to sensitive data or restricted functionalities. This includes exploiting weaknesses such as Insecure Direct Object Reference (IDOR) and manipulating JWT tokens to escalate privileges or bypass access controls.

This model is suitable for the context of the application, where the vulnerabilities described can be exploited with basic tools and moderate skills. It is assumed that the attacker operates within the constraints of the application and does not have full control over the communication channel, such as intercepting or altering network packets. Instead, the focus is on manipulating the client-server interactions using commonly available techniques.

3 Demonstrating Exploits on Vulnerable Systems

This demonstration outlines the process of exploiting broken access control vulnerabilities in the web application using Burp Suite as the primary interception tool. While the demonstration utilizes two computers for clarity, the entire process can be replicated on a single machine. All the necessary resources and files required to replicate the demonstration can be found in the corresponding Github repository.

Initial setup and Enviroment the demonstration environment consists of two computers: one acting as the attacker's machine running Burp Suite, and another serving as the target system hosting our vulnerable application. The attacker's machine is configured with Burp Suite Professional or Community Edition, with the proxy properly configured to see HTTP requests and send edited HTTP requests.

3.1 Exploiting JWT Token Vulnerabilities

The first attack vector exploits the improper implementation of JWT token validation. The process begins with a legitimate login to the application using standard user credentials. When logging in as a standard user (e.g., username: "paolo", password: "password1"), the application generates a JWT token. To connect, i had to enter the IP address of the hosting machine along with port 3000. If using a single machine, it is http://localhost:3000.

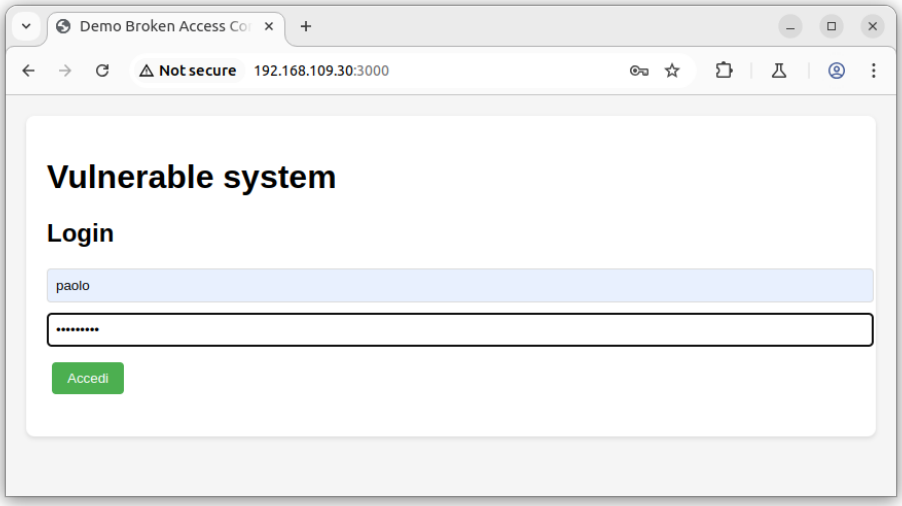

Figure 3: User Dashboard Overview – Interface and Functional Components

Using Burp Suite's proxy, we intercept this token and examine its structure (fig.4). The vulnerability becomes apparent when we analyze the token's implementation. The server uses an empty secret key and implements jwt.decode() instead of jwt.verify(), making it susceptible to manipulation.

Using Burp Suite's decoder, we can modify the JWT payload to elevate our privileges (fig. 5), JWT use arm64 coding. By changing the "role" claim from "standard" to "admin" and the "id" value if desired, we can create a new token that the server will accept without validation (fig. 6), it is recommended to use jwt.io for decoding due to its user-friendly interface.;

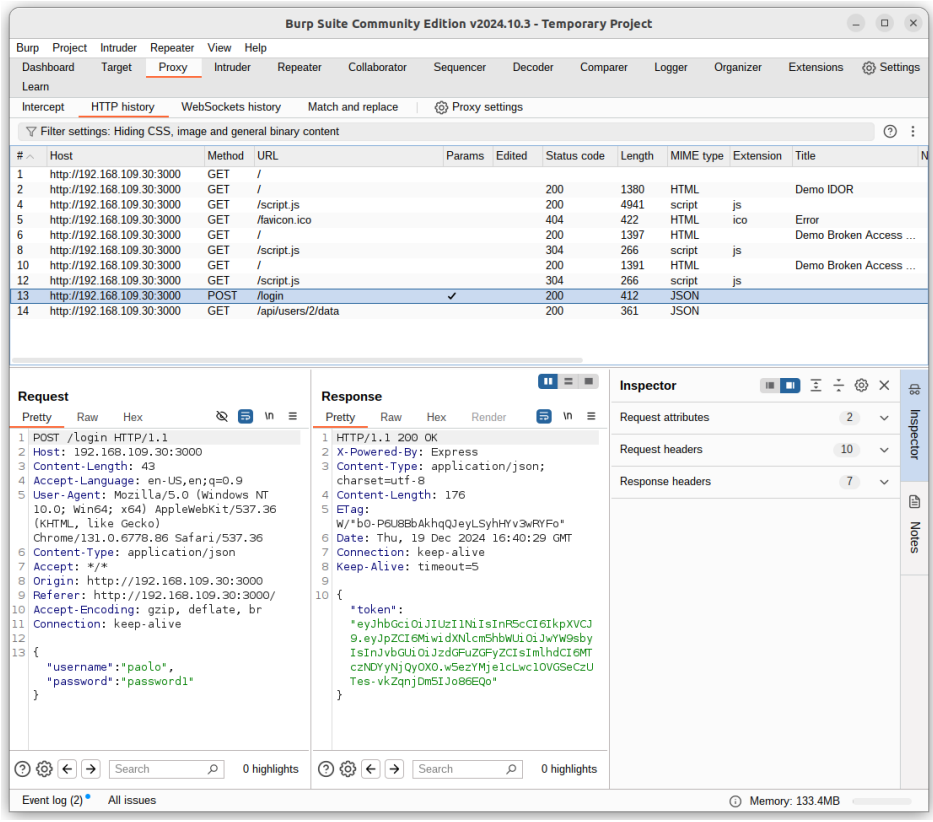

Figure 4: burpsuite screenshot

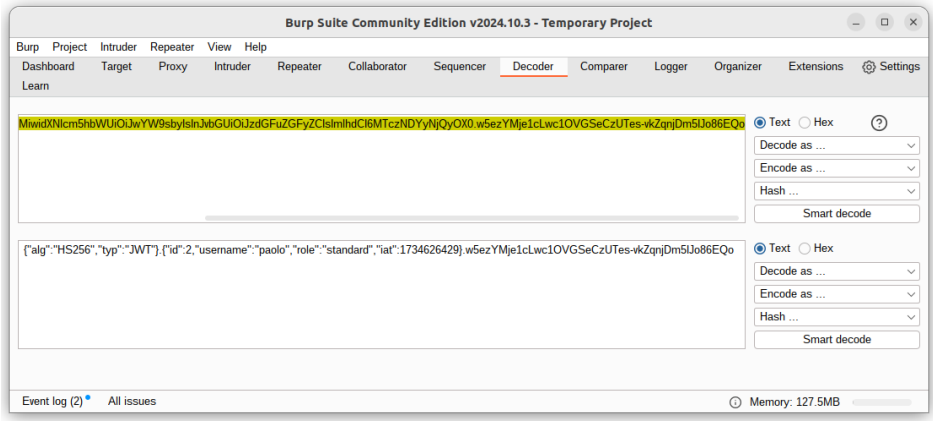

Figure 5: Decoding the token

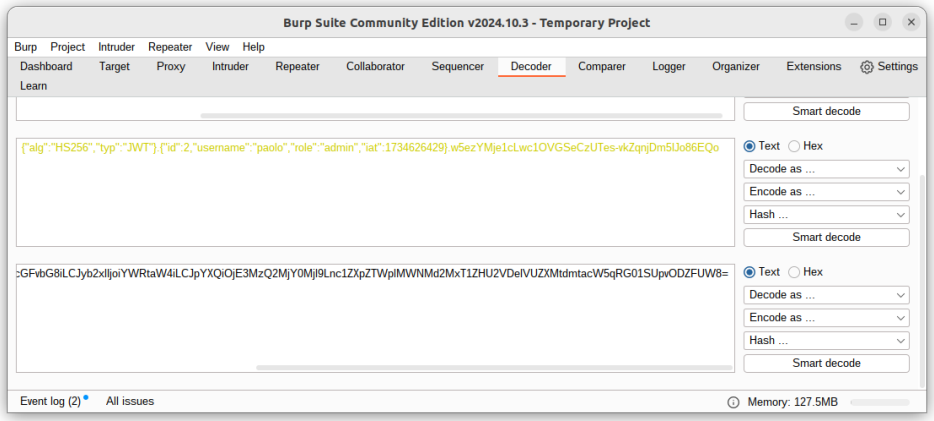

Figure 6: Encoding the modified token

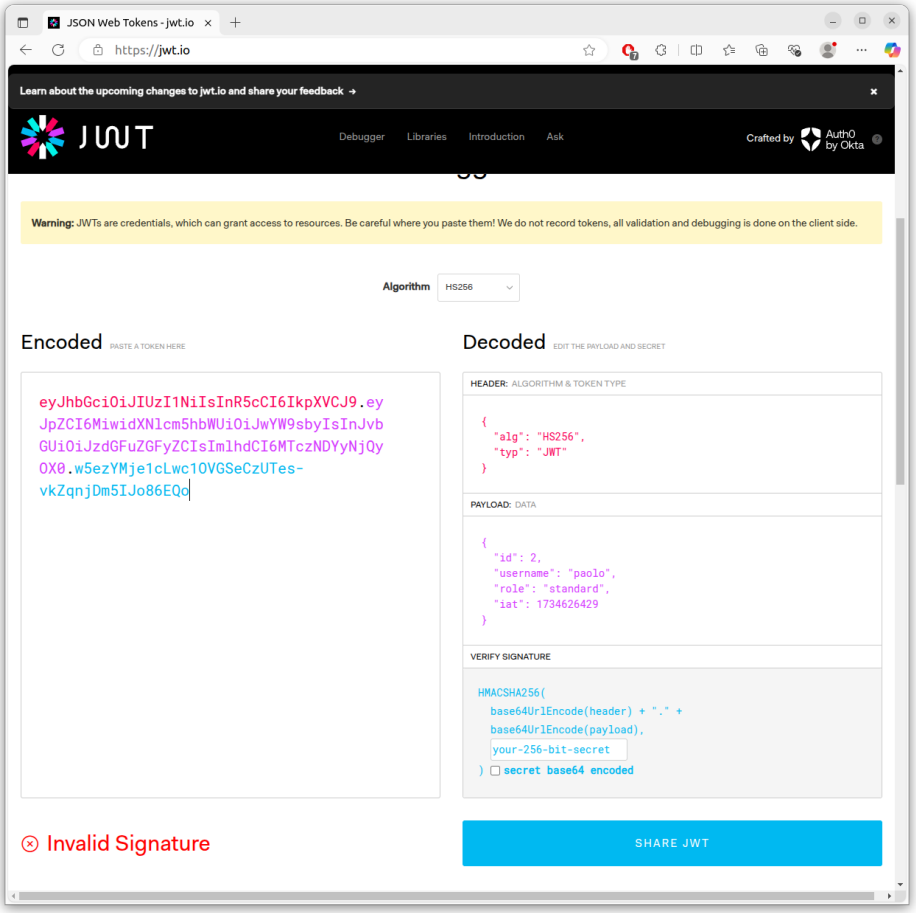

Figure 7: Orignial Token

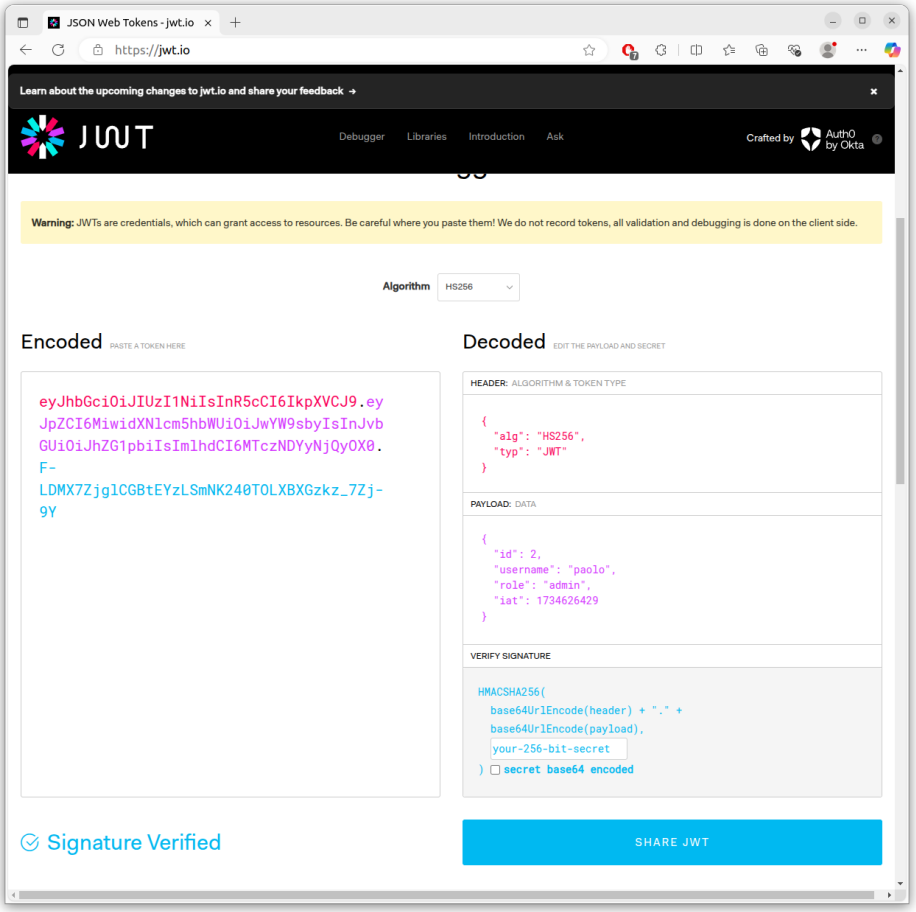

Figure 8: Modified Token

When initially accessing the Admin Panel as the standard user "paolo," the system correctly enforces access control by displaying an authorization error (fig. 9). This demonstrates the intended behavior of the access control system for unauthorized users.

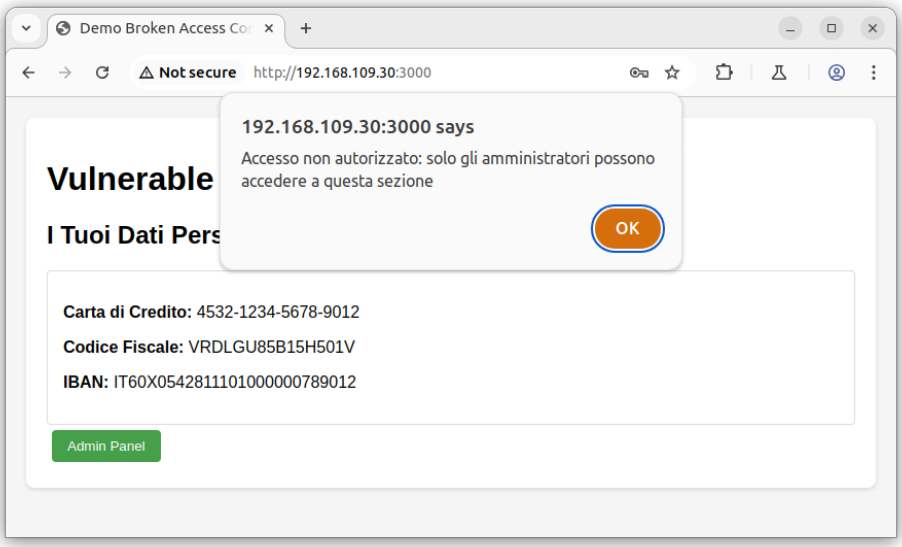

Figure 9: trying to access Admin Panel on user

However, to bypass this restriction, we can intercept and modify the HTTP request for the admin panel access. Using Burp Suite's proxy functionality, we capture the original request and replace the authorization token in the header with our modified JWT token that contains elevated admin privileges. This manipulation effectively bypasses the system's access controls, as the application fails to properly validate the token's authenticity and accepts our modified credentials.

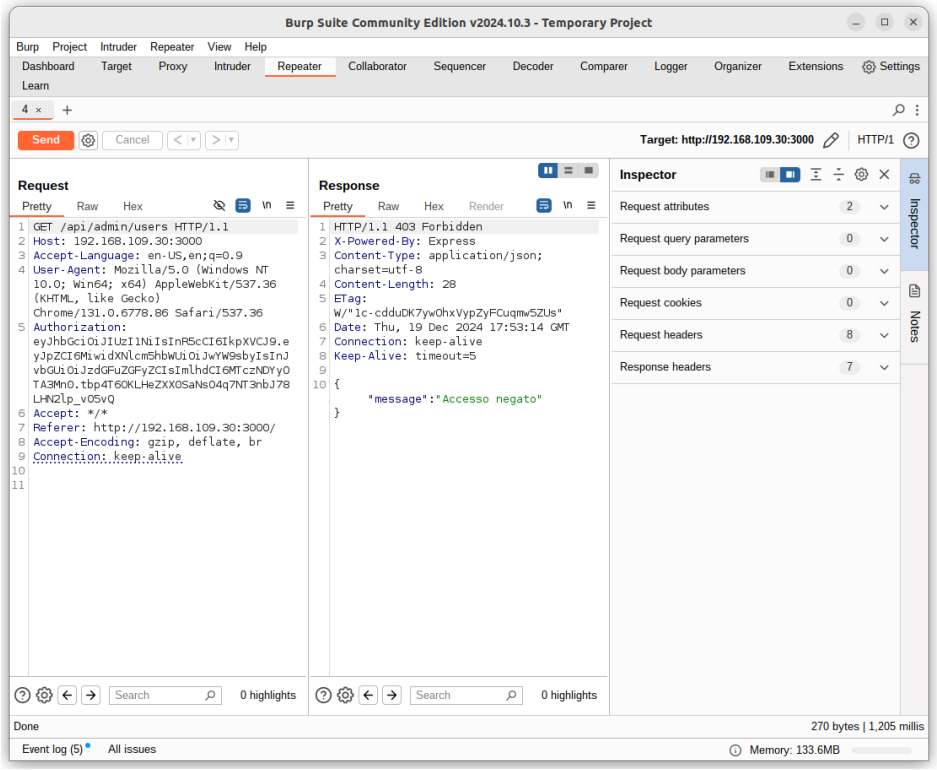

Figure 10: request whit user token

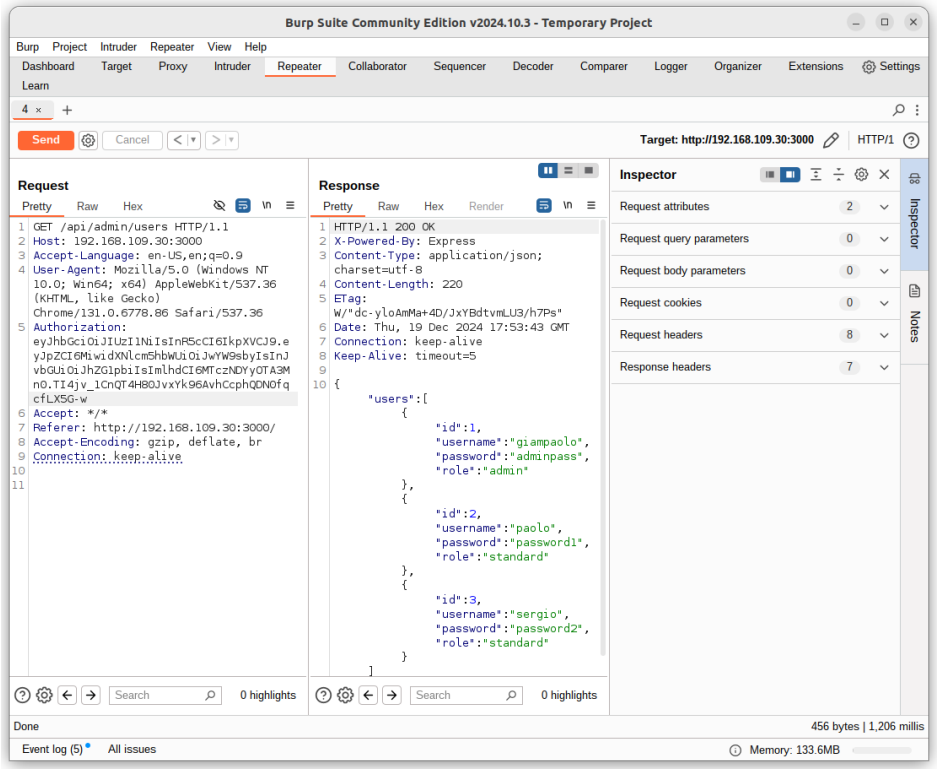

Figure 11: request whit modified token

The successful manipulation of the JWT token demonstrates a critical security vulnerability in the application's authentication and authorization mechanisms. By exploiting the system's improper token validation, specifically the use of jwt.decode() instead of jwt.verify() and an empty secret key, we were able to escalate privileges from a standard user to an administrator role without requiring legitimate administrative credentials.

This security breach has severe implications for the application's integrity. An attacker with knowledge of JWT structure can easily modify their authorization token to gain unauthorized access to administrative functions, potentially compromising sensitive user data and system controls. The vulnerability is particularly concerning because it bypasses the application's intended access control mechanisms entirely, rendering role-based access control ineffective.

The root cause of this vulnerability lies in two critical implementation flaws: the absence of a secure secret key for token signing and the failure to properly verify token signatures. These oversights allow attackers to modify token payloads at will, with the system accepting these modifications as legitimate. The attack's success highlights the importance of proper JWT implementation, including secure token signing, verification, and the use of strong secret keys in production environments.

3.2 Exploiting IDOR Weaknesses

The second critical vulnerability in our application demonstrates a classic case of Insecure Direct Object Reference (IDOR). This security flaw arises from the application's failure to implement proper authorization checks when accessing user-specific resources. The vulnerability exists in the endpoint responsible for retrieving user personal data, allowing any authenticated user to access sensitive information belonging to other users simply by manipulating the user ID parameter in the URL.

To demonstrate this vulnerability, we begin by authenticating to the application using standard user credentials. For this demonstration, we'll use the account "paolo" with the password "password1". Upon successful authentication, we can access our own user data through the endpoint `/api/users/2/data`, where "2" represents paolo's user ID in the system. The application correctly serves this data, as we are the legitimate owner of this resource (fig 12).

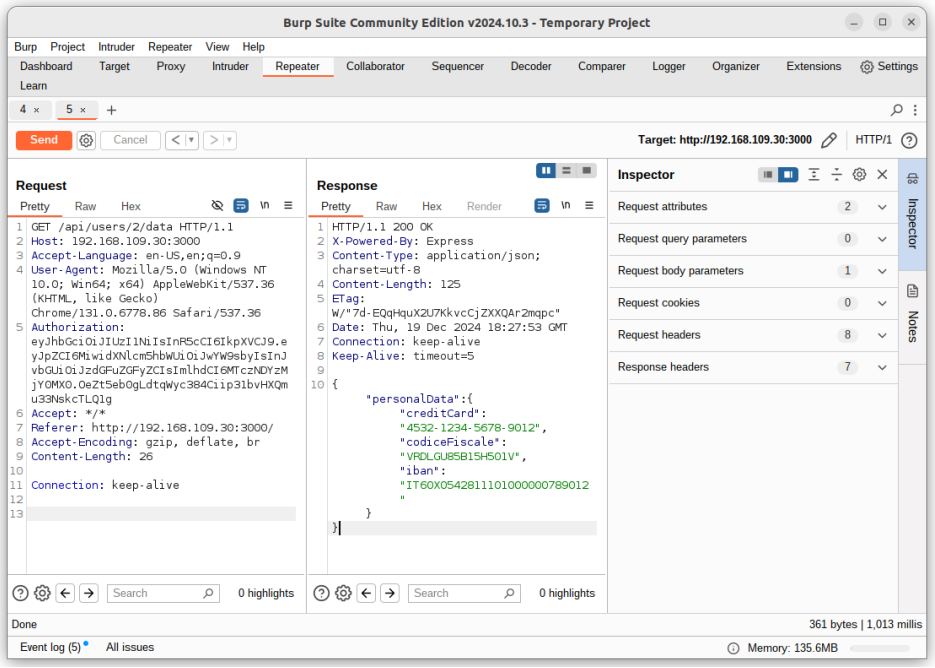

Figure 12: Request for user 2 data

The vulnerability becomes apparent when we examine the application's behavior with modified user IDs. Using Burp Suite's proxy to intercept and modify requests, we can change the user ID parameter in the URL. For instance, modifying the endpoint to `/api/users/1/data` attempts to access the administrative user's data. Instead of implementing proper access controls and validating whether the requesting user has permission to access this resource, the server blindly retrieves and returns the requested data (fig 13).

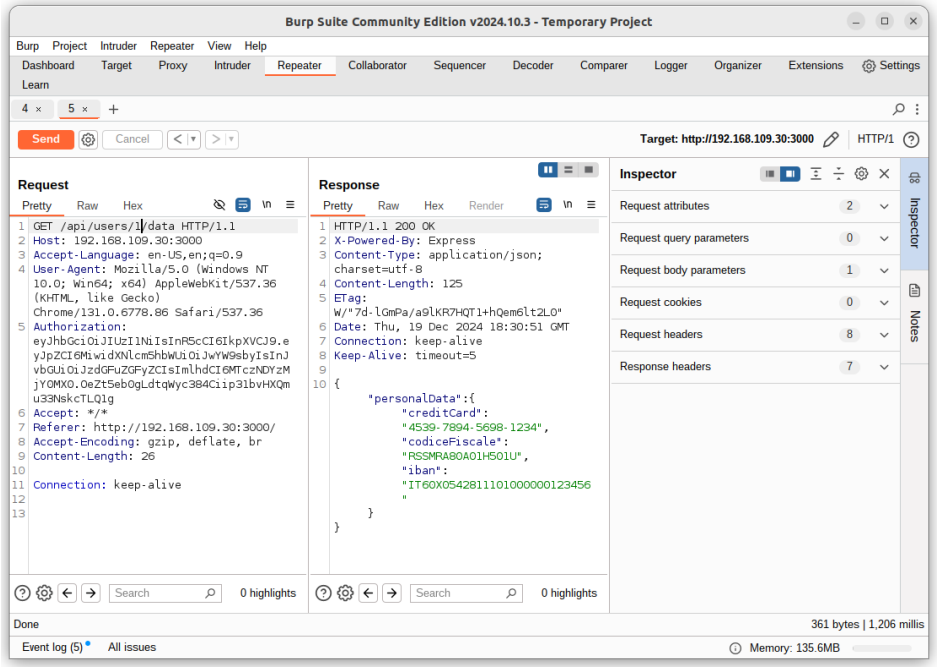

Figure 13: Request for user 1 data

This vulnerability is particularly severe because the exposed endpoint returns sensitive personal information, including credit card numbers, fiscal codes (codice fiscale), and IBAN numbers. The server's response contains this sensitive data in plaintext, making it immediately accessible to any authenticated user who understands how to manipulate the request parameters.

The exploitation process is straightforward and can be automated using Burp Suite's Intruder feature. By setting up a simple numerical iteration of the user ID parameter, an attacker can systematically harvest sensitive information from all users in the system. The lack of rate limiting on this endpoint further compounds the vulnerability, allowing rapid enumeration of user data. The root cause of this vulnerability lies in the server's implementation of the data retrieval endpoint.

The successful exploitation of this IDOR vulnerability demonstrates the importance of implementing proper authorization checks at every level of the application. Each request to access sensitive data should verify not only the authentication status of the user but also their authorization to access the specific resource being requested.

Normal User

```
B → S : {Username_B, Password_B}
S → B : {A_ID=1, Token_B}
B → S : GET /api/user/1 (Token_B)
S → B : {Data2}
```

Attacker user

```
T → S : {Username_T, Password_T}
S → T : {A_ID=2, Token_T}
T → S : GET /api/user/2 (Token_T)
S → T : {Data2}
T → S : GET /api/user/1 (Token_T)
S → T : {Data1}
```

4 Strengthening System Security

As we delve deeper into the system's vulnerabilities and their implications, it is essential to establish a clear understanding of the foundational challenges faced by contemporary security frameworks. Addressing these issues not only contextualizes the technical deficiencies but also provides a rationale for the necessity of robust countermeasures. By examining both systemic shortcomings and potential solutions, we can better appreciate the dynamic interplay between architectural flaws and practical defenses. This broader perspective sets the stage for a detailed exploration of specific vulnerabilities and their associated risks.

4.1 Enhancing Security Against JWT and IDOR Threats

4.1.1 Securing JWT Tokens

- **Prior State:** The original JWT implementation exhibited severe security deficiencies that fundamentally compromised the application's authentication mechanism. At its core, the system utilized an empty secret key for token signing, essentially nullifying the security benefits of JWT signatures. This critical oversight allowed malicious users to forge valid tokens without detection. The authentication middleware further compounded this vulnerability by using jwt.decode() instead of proper verification, meaning the system never validated the authenticity of incoming tokens. The token payload contained sensitive role information that could be freely modified, as the system lacked any mechanism to detect such alterations.

- **Current Enhanced State:** The revamped JWT implementation now incorporates multiple layers of security controls. At the foundation, the system generates a cryptographically secure random secret key during server initialization, stored securely and used consistently for all token operations. This key ensures that tokens cannot be forged without access to the secret, as the signature verification will fail for any modified tokens.

  The authentication process has been completely restructured to implement comprehensive token validation. Each token now includes additional security-critical fields such as issuance time, expiration time, and a unique token identifier. The system enforces strict expiration times, automatically rejecting any tokens that have exceeded their validity period. The middleware now properly verifies token signatures using jwt.verify(), ensuring that any attempt to modify the token payload will be detected and rejected. The role-based access control has been strengthened by implementing additional verification steps. When a token is presented, the system not only verifies its cryptographic integrity but also validates the claimed role against the user's actual role stored in the database. This prevents privilege escalation attempts even if an attacker manages to modify the token payload, as the system will detect the discrepancy between the claimed and actual user roles.

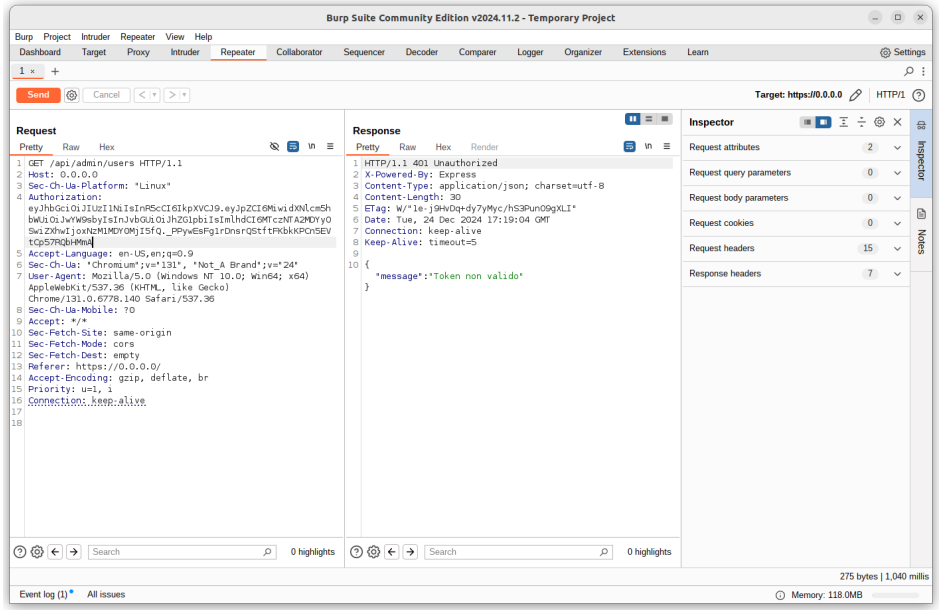

Figure 14: JWT manipulation whit enforced system

Attack

```
T → S : {Username_T, Password_T}
S → T : {Token_T: {Role=User, A_ID=2}}
T → T : Modify {Token_T: {Role=User}} → {Token_T: {Role=Admin}}
T → S : GET /admin (Token_T: {Role=Admin, A_ID=2})
S → T : {Admin_Data}
```

Defense

```
T → S : {Username_T, Password_T}
S → T : {Token_T: {Role=User, A_ID=2, Signature=S}}
T → T : Modify {Token_T: {Role=User}} → {Token_T: {Role=Admin}}
T → S : GET /admin (Token_T: {Role=Admin, A_ID=2, Signature=S})
S : Verify Signature (Token_T)
S → T : Access Denied (Invalid Signature)
```

4.1.2 Mitigating IDOR Vulnerabilities

- **Prior State:** The original implementation of user data access endpoints demonstrated a classic Insecure Direct Object Reference vulnerability. The system's only security check was verifying the existence of the requested user ID, with no validation of whether the requesting user had permission to access that data. This meant that any authenticated user could access any other user's personal information simply by modifying the user ID parameter in their requests. The vulnerability was particularly severe given the sensitive nature of the exposed data, including financial and personal information.

- **Current Enhanced State:** The improved system implements a comprehensive authorization framework for all data access requests. When a user attempts to access personal data, the system performs a multi-step verification process to ensure proper authorization. First, it validates the requesting user's identity through the enhanced JWT verification process. Then, it implements strict ownership validation, verifying whether the requesting user either owns the requested resource or has explicit administrative privileges to access it. The authorization checks are now handled entirely server-side, preventing any client-side manipulation attempts. The system maintains a clear separation between authentication and authorization, ensuring that simply being authenticated is not sufficient to access protected resources. Each request is validated against the defined access control matrix, with explicit checks for each operation type.

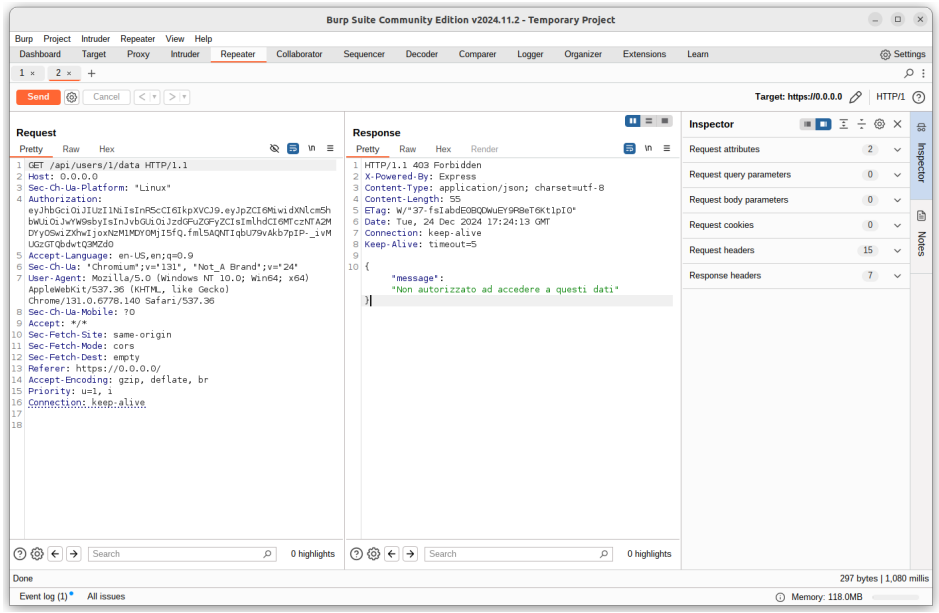

Figure 15: IDOR in enforced system

4.1.3 Additional Security Enhancements

The web application has undergone significant security enhancements through architectural improvements and the introduction of modern security tools. This analysis examines the key changes and their impact on system security.

The core security architecture now implements AES-256-CBC encryption for all sensitive data, utilizing the crypto module for secure key generation and initialization vectors. This ensures that sensitive information, including credit card numbers, fiscal codes, and IBAN numbers, remains encrypted both in storage and during processing. The system generates cryptographically secure random keys through crypto.randomBytes(), establishing a strong foundation for all security operations. Each encryption operation utilizes a unique initialization vector, preventing pattern analysis and ensuring that identical data produces different ciphertext across multiple encryptions.

Authentication security has been strengthened through the implementation of a sophisticated rate-limiting system. The application now tracks login attempts using both username and IP address combinations, implementing a 15-minute lockout period after five failed attempts. This defense mechanism effectively prevents brute force attacks while maintaining system accessibility for legitimate users. The rate limiting system also includes an automatic reset feature after one hour of inactivity, ensuring that temporarily locked accounts can be recovered without administrative intervention.

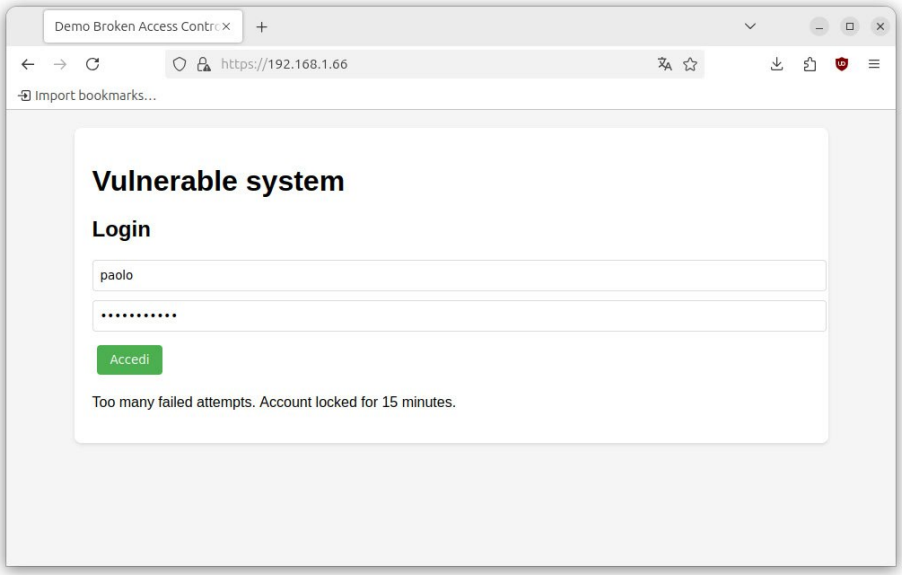

Figure 16: login requests limit

Communication security has been enhanced through mandatory HTTPS implementation with proper SSL/TLS configuration. The system now manages certificates appropriately and automatically redirects HTTP traffic to HTTPS, ensuring encrypted communication channels between client and server. The HTTPS implementation ensures that all data transmission, including authentication tokens and sensitive user information, remains protected against interception and manipulation.

New security tools have been integrated to support these improvements. Bcrypt provides secure password hashing, replacing previous password storage methods. . The JWT implementation has been enhanced with proper token signing and verification, using cryptographically secure keys generated through the crypto module.

These changes represent a shift from basic security measures to an enterprise-grade implementation that protects data through multiple layers of security while maintaining system functionality. Each security layer works in concert with the others: encrypted data storage protects sensitive information at rest, HTTPS ensures secure transmission, rate limiting prevents automated attacks, and proper authentication mechanisms control access to protected resources.

4.2 Structure differences

A well-organized system structure is a cornerstone of maintaining robust security measures. Structural organization involves not only the arrangement of directories and files but also the strategic placement of critical components to minimize vulnerabilities and maximize efficiency. In the context of secure web applications, the structural design serves as a blueprint for enforcing access controls, protecting sensitive data, and facilitating efficient development practices.

This section provides a overview of the structural organization in two distinct system implementations: one showcasing a vulnerable design and the other representing an enforced secure model.

```
Enforced_System/
|-- certificates/
|   |-- certificate.pem
|   |-- private.key
|-- node_modules/
|-- public/
|   |-- index.html
|   |-- script.js
|   |-- server.js.bak
|   |-- style.css
|-- server.js
```

```
Vulnerable_system/
|-- node_modules/
|-- public/
|   |-- index.html
|   |-- script.js
|   |-- style.css
|   |-- server.js.bak
|-- server.js
```

Policy Suite

1. Authentication and Authorization Policy

- All users must authenticate using valid credentials via the `/login` endpoint.
- Authentication is implemented using JWT tokens with a 1-hour expiration period.
- Roles are defined as:
  - **Admin**: Full access to all resources and user management features.
  - **Standard User**: Restricted to accessing their own data only.
- Unauthorized access attempts are logged and rate-limited.

2. Rate Limiting and Account Lock Policy

- Login attempts are capped at 5 within a 15-minute window per username/IP.
- Exceeding the limit results in a 15-minute lockout.
- Accounts are automatically unlocked after the lockout period or after one hour of no further failed attempts.

3. Data Encryption and Confidentiality Policy

- Sensitive data such as credit card numbers, tax codes, and IBANs are encrypted using AES-256-CBC with unique IVs.
- Encryption keys and initialization vectors are securely generated and stored.
- Decryption is performed only for authenticated and authorized users.

4. Access Control Policy

- **Admin Panel**: Accessible only to admins. Includes features to list, delete, and view user data.
- **User Data**: Admins can access all user data, while standard users are limited to viewing their own.
- **Protected Endpoints**: All sensitive API endpoints require a valid JWT token in the `Authorization` header.

5. Secure Communication Policy

- All communication is enforced over HTTPS using SSL/TLS.
- HTTP requests are automatically redirected to HTTPS.

6. Frontend Security Policy

- The frontend differentiates user roles:
  - Admin users can access and manage all panels.
  - Standard users see only personal data options.
- Admin features (e.g., buttons to delete users) are hidden from standard users in the UI.

7. User Role and Privilege Policy

- Users are assigned roles (admin or standard) at the time of creation.
- Admin privileges are strictly controlled, and admin accounts cannot be deleted by other admins.
- Standard users cannot perform actions outside their scope (e.g., accessing admin endpoints).

8. Token Management Policy

- Tokens must be included in the `Authorization` header for all requests to protected routes.
- Expired or invalid tokens are rejected, prompting users to reauthenticate.

9. User Management Policy

- Admins can:
  - View all users.
  - Delete standard users but cannot delete other admin accounts.
- Standard users cannot access or manipulate other users' data or accounts.

10. Error Handling Policy

- All unauthorized access attempts return appropriate HTTP status codes (401 for unauthorized, 403 for forbidden).
- Descriptive error messages guide users to resolve issues (e.g., "Invalid credentials" or "Account locked").

Resource	**Admin**	**Standard User**	**Unauthenticated User**
`/login`	Authenticate (POST)	Authenticate (POST)	Authenticate (POST)
`/api/users/:userId/data`	Read all user data (GET)	Read own data (GET, only if `userId` matches)	No Access
`/api/admin/users`	View all users (GET)	No Access	No Access
`/api/admin/users/:userId`	Delete non-admin users (DELETE)	No Access	No Access
Dashboard (Frontend)	Full Access (View all panels and data)	Limited Access (View only their data)	No Access
Admin Panel (Frontend)	Full Access (View and manage all users)	No Access	No Access

Table 1: Role-Based Access Control Matrix

Tools and Frameworks for Security Implementation

- **Express.js**
  Express.js is a minimalist web framework for Node.js, designed for building web applications and APIs. It provides a robust set of features such as routing, middleware integration, and template rendering, making HTTP request and response handling straightforward.

- **JWT (JSON Web Token)**
  JWT is a compact, self-contained mechanism for securely transmitting information as a JSON object. It is primarily used for authentication and session management. The server generates a token containing encoded user data, which is sent with subsequent requests for user verification.

- **Bcrypt**
  bcrypt is a password-hashing library that provides strong encryption to securely store passwords. It hashes passwords before storing them and compares hashed passwords during login to validate credentials.

- **Crypto Module**
  The `crypto` module in Node.js provides cryptographic functionalities, such as encryption, decryption, and random data generation. It encrypts and decrypts sensitive data (e.g., credit card numbers, IBANs) for secure storage and retrieval.

- **HTTPS Module**
  The `https` module in Node.js enables secure communication over HTTPS. It sets up an HTTPS server using SSL/TLS certificates, ensuring data encryption during transit.

- **Static Files**
  Static files include HTML, CSS, and JavaScript files served to the client. They provide the front-end interface and manage user interactions with the server.

- **Rate Limiting (Custom Implementation)**
  A mechanism to prevent brute-force attacks by limiting login attempts per user or IP address. It locks user accounts after 5 failed attempts within a 15-minute window.

- **Middleware**
  Middleware functions intercept requests before they reach route handlers, such as parsing JSON request bodies (`express.json()`) or validating JWT tokens (`authenticate`).

5 Conclusion

The exploration of access control vulnerabilities in this project sheds light on a persistent and disheartening truth: the security of web applications often hinges on fragile systems riddled with preventable flaws. Despite decades of research, best practices, and an entire industry dedicated to cybersecurity, many systems continue to fall prey to the same fundamental mistakes. Broken Access Control, Insecure Direct Object References (IDOR), and poorly implemented JWT tokens are not new threats; they are relics of an industry that often prioritizes speed over security and functionality over resilience.

The demonstration of vulnerabilities in this project should serve as a wake-up call to developers and security professionals alike. Allowing attackers to manipulate URLs to access sensitive data or forging admin-level tokens due to the absence of proper verification is, frankly, unacceptable in today's digital landscape. These are not cutting-edge zero-day exploits; they are the security equivalent of leaving your front door unlocked in a neighborhood full of burglars.

Even more concerning is how such vulnerabilities persist in environments where stakes are high, whether involving sensitive personal data, financial records, or administrative privileges. Developers, driven by tight deadlines and resource constraints, often sacrifice security at the altar of efficiency, leaving end-users exposed to avoidable risks. Worse yet, many organizations view security as an afterthought, something to be patched together once the system is already live, a reactive rather than proactive approach that virtually guarantees vulnerabilities will be exploited.

This project not only highlights these issues but also offers a roadmap for addressing them. The implementation of secure token handling, robust encryption protocols, and strict authorization checks are steps in the right direction, but they are not groundbreaking. They are, at their core, fundamental best practices that should have been implemented from the start. Organizations must embrace a defense-in-depth approach, layering their security measures and testing them rigorously, not because it is innovative, but because it is necessary. The OWASP guidelines, though not a panacea, offer a robust foundation for organizations that are serious about security.

However, even the best technical solutions are only as good as the culture that sustains them. The cybersecurity industry must confront a harsh reality: many vulnerabilities exist not because of a lack of knowledge or tools but because of complacency and negligence. Regular vulnerability assessments, education, and a commitment to security must be ingrained into the development lifecycle, not treated as optional extras. Until then, developers will continue to leave the door ajar, and attackers will keep walking in.

In the end, this project underscores a simple but often ignored truth: security is not a luxury, it is a necessity. It is not an add-on or a nice-to-have; it is the foundation upon which trust is built in a digital world. And yet, as the case studies here demonstrate, it is a foundation that is all too often riddled with cracks. Until the industry as a whole chooses to prioritize security with the seriousness it deserves, we will continue to see history repeat itself in the form of breaches, losses, and damage, most of which could have been avoided. Perhaps this cynicism will serve as motivation, because if nothing else, it is better to be cynical and secure than optimistic and breached.

Glossary

Access Control: A system that limits access to resources based on user identity and permissions.

Authentication: The process of verifying a user's identity using credentials such as a username and password.

Authorization: The process of determining what actions a user is allowed to perform after successful authentication.

Broken Access Control: A vulnerability that allows attackers to bypass access restrictions.

Discretionary Access Control (DAC): An access control model where resource owners control permissions.

Mandatory Access Control (MAC): A model where permissions are managed by system administrators according to strict policies.

Role-Based Access Control (RBAC): Permissions are assigned to roles, and users inherit permissions based on their roles.

Vertical Access Control: Prevents users from accessing resources reserved for higher privilege levels.

Horizontal Access Control: Prevents users from accessing resources of other users at the same privilege level.

Context-Based Access Control: Grants or denies access based on the context, such as application state or user interaction sequence.

Insecure Direct Object Reference (IDOR): A vulnerability where attackers manipulate object references (e.g., IDs) to access unauthorized data.

Cross-Site Scripting (XSS): A vulnerability that allows attackers to inject malicious scripts into web pages viewed by others.

Privilege Escalation: A vulnerability that enables attackers to gain unauthorized access to higher privilege levels.

Replay Attack: An attack where valid data transmission is maliciously or fraudulently repeated.

JSON Web Token (JWT): A token format for securely transmitting information between parties.

JWT Secret Key: A key used for signing and verifying JWTs to ensure their authenticity.

Token Expiration: A mechanism that invalidates tokens after a specific time period to improve security.

Static Analysis: Examines source code without executing it to find vulnerabilities.

Dynamic Analysis: Tests an application while it is running to identify vulnerabilities in real-time.

Dynamic Application Security Testing (DAST): A testing method that simulates attacks on a live application to find vulnerabilities.

Force Browsing: A technique to access unauthorized resources by manipulating URLs or paths.

Encryption: The process of converting plaintext into ciphertext to protect data from unauthorized access.

HTTPS: A secure version of HTTP that uses encryption to protect data in transit.

Defense-in-Depth: A strategy that uses multiple layers of security to protect a system.

Rate Limiting: Restricting the number of requests a user can make in a given period to prevent abuse.

OWASP (Open Web Application Security Project): A nonprofit organization providing resources for improving web application security.

OWASP Top 10: A list of the ten most critical web application security risks.