Cairo University
Faculty of Engineering
Electronics and Electrical Communications Department

Graduation Project

Secure Healthcare Access and Monitoring System Based on Zero Trust Architecture

Submitted by:

Engy Raafat Melek Abd El-Nour

Shady Bassem Mokhtar Kamel

Sylvia Youssef Fouad Youssef

Maryam Ahmed Kamal Mamdouh

Yasmin Hany Momtaz Abo El-Magd

Supervised by:

Prof. Dr. Magdy Elsoudani

Acknowledgment

We want to express our deepest gratitude to our professor, Dr. Magdy Elsoudani, for his guidance, patience, mentorship, and support throughout his exceptional supervision of our project.

Dr. Magdy Elsoudani's suggestion to explore modern security principles provided great opportunity for us along with his commitment to ensuring the highest learning outcomes. Although his guidance was fundamental to balance between implementing robust security models and aligning them with real-world industry practices. His guidance not only enhanced the project's impact but also deepened our understanding of modern security principles due to emphasizing the importance of bridging theoretical and practical gaps, it has been an invaluable opportunity to learn and grow under his expertise.

Table of Contents

List of Figures

List of Tables

Table of Abbreviations

5G	Fifth Generation
6G	Sixth Generation
AES-256	Advanced Encryption Standard - 256
API	Application Programming Interface
BCORE	Black Core
BYOD	Bring Your Own Device
CBC	Cipher Block Chaining
CORS	Cross Origin Resource Sharing
CSS	Cascading Style Sheets
DISA	Defense Information Systems Agency
DDoS	Distributed Denial of Service
DERP	Detour Encrypted Routing for Packets
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of Service
ECDSA	Elliptic Curve Digital Signature Algorithm
FastAPI	Fast Application Programming Interface
GB	Gigabyte
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAM	Identity and Access Management
ID	Identity Document
IDSs	Intrusion Detection Systems
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
IV	Initialization Vector
JS	JavaScript
JWS	JSON Web Signature

JWT	JSON Web Token
LANs	Local Area Networks
MAC	Message Authentication Code
MCF	Modular Crypt Format
MFA	Multi-Factor Authentication
MySQL	My Structured Query Language
NIC	Network Interface Card
ORM	Object Relational Mapping
OS	Operating System
OTP	One-Time Password
P2P	Peer-to-Peer
PA	Policy Admin
PE	Policy Engine
PEP	Policy Enforcement Point
PIN	Personal Identification Number
RAM	Random Access Memory
RBAC	Role-Based Access Control
RSA	Rivest – Shamir – Adleman
SAML	Security Assertion Markup Language
SFA	Single-Factor Authentication
SIEM	Security Information and Event Management
SHA	Secure Hashing Algorithm
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SPE	Security Policy Engine
TCP	Transmission Control Protocol
TEE	Trust Evaluation Engine
UDP	User Datagram Protocol
UE	User Equipment
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
UTF-8	Unicode Transformation Format - 8
VLANs	Virtual Local Area Networks
VPNs	Virtual Private Networks

VSC	Visual Studio Code
XML	Extensible Markup Language
ZT	Zero Trust
ZTA	Zero Trust Architecture

Abstract

This project presents a secure hospital web platform built using Zero Trust Architecture (ZTA) to protect sensitive medical data and services. As cyber threats become more advanced, older security methods that rely on protecting the outer network are no longer enough. ZTA follows a "*Never Trust, Always Verify*" approach by checking each user's identity, device security, and access context before allowing entry. The platform is designed for doctors, nurses, and patients, with security integrated at every level.

It uses Multi-Factor Authentication (MFA) and JSON Web Tokens (JWT) to ensure only the right users can access specific parts of the system. JWTs securely carry user information like roles and expiry times in a verified and tamper-proof way. All communication is encrypted to maintain data privacy and integrity. Role-based access control (RBAC) ensures that each user only sees what they are permitted to. The system also includes segmentation of services by tailscale and dynamic access enforcement based on behavioral signals. The platform is hosted on a secure cloud environment to support scalability and availability. To monitor activity, the platform uses the ELK stack—Elasticsearch, Logstash, and Kibana—which collects and analyzes logs in real time. These logs track logins, API usage, role-based actions, and unusual access to private data, helping detect suspicious behavior while staying compliant with data privacy regulations.

CHAPTER ONE: INTRODUCTION

1.1. Introduction [1] [2]

The digital age has reshaped healthcare. Instead of keeping patient records in drawers or physical files, healthcare data is now part of a dynamic, interconnected digital network. Mobile devices, the Internet of Things (IoT), and cloud storage have allowed healthcare providers to share and access data quickly from anywhere, improving patient care and operational efficiency. However, there are disadvantages of this digital shift like more exposed to security risks, making sensitive information more vulnerable to cyberattacks.

Many organizations, including those in the healthcare sector, have long depended on perimeter-based security models. In these traditional systems, the focus was on creating a strong, clear boundary, like a fortified wall or castle fence, that separated the trusted internal network from potentially hostile external networks. The idea was simple: as long as an attacker could not breach that outer layer (using firewalls, VPNs, intrusion detection systems, and other such defenses), everything inside could be assumed safe. This approach worked well in environments where systems were relatively isolated and threats were predominantly external. However, modern healthcare environments are much more complex. Hospitals and clinics now use multiple internal networks, remote offices, and cloud services. This complexity blurs the clear boundary, allowing attackers to easily move laterally after breaching the initial defenses. Consequently, the traditional perimeter-based model is no longer sufficient to protect today's sensitive healthcare data, so we need to move to ZTA.

Unlike the traditional perimeter-based approach, ZTA is built on the principle of "*Never Trust, Always Verify*". This principle assumes that threats can exist from users in both outside and inside the network, and therefore there are continuous authentication, validation, and authorization of every request to access data or any resource inside the network.
Zero Trust Architecture enforces only the availability of the needed access controls not more than this, so that each user or device gets only the exact permissions they need. Even if an intruder breaks in, they cannot easily move around the network because every access request is individually verified. This constant check helps protect modern healthcare systems against a wide range of complex threats.

The concept of zero trust has been suggested in cybersecurity long before the term "Zero Trust" was introduced. Zero Trust Architecture (ZTA) is becoming progressively more accepted by governments, business, and academic institutions since it evolved from a security concept to a crucial network security solution. The Defense Information Systems Agency (DISA) and the Department of Defense published their work on a more secure enterprise strategy Black Core [BCORE], which involved a transition from a perimeter-based security model to one that focused on the security of individual transactions [1].

Zero trust then has been a modern security term used to describe various cybersecurity solutions that changed security away from implied trust philosophy of the zero-trust concept was fundamentally introduced by Kindervag et al of Forrester Research in his report titled: "*No More Chewy Centers: Introducing The Zero Trust Model Of Information Security*", which put the outlines and description of critical flaws of traditional security models that trust internal networks automatically, which cause leaving them vulnerable to insider threats and advanced attacks [3].

Aiming to mitigate those security issues; the idea of "*Zero Trust*" based on three fundamental security concepts was introduced which are: "*Ensure That All Resources Are Accessed Securely Regardless of Location*", "*Adopt A Least Privilege Strategy and Strictly Enforce Access Control*", and "*Inspect and Log All Traffic*" [2].

The following *Figure 1* shows the timeline of ZTA development and investments:

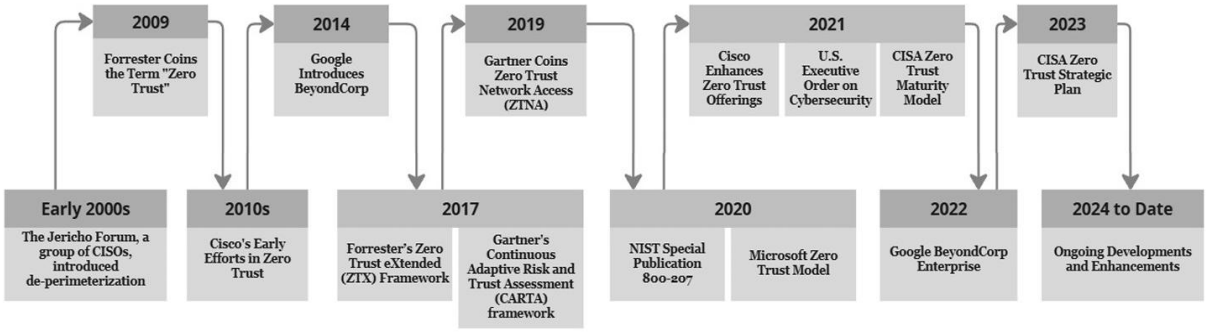

Figure 1 **ZTA historical timeline** [3]

Healthcare systems including healthcare information have been through a fast transition to digitalization instead of paper documents due to the need for using medical devices remotely, servers connected globally, and the Internet of Things which made data sharing essential. Digitalization of healthcare systems can lead to a higher possibility of being affected by an information system breach that leads to the theft of patient data. That's why numerous professional and academic attempts to create and propose alternative security frameworks for protecting patient data worldwide [2].

Zero trust then has been a modern security term used to describe various cybersecurity solutions that changed security away from implied trust based on network location and instead focused on evaluating trust on a per-request basis. Recently, private industry and higher education have also contributed to the evolution from perimeter-based security to a security strategy based on ZT principles [1].

1.2. Motivation [1] [2]

The limitations of the traditional security measures appeared when cyberattacks happened in 2021. In a single year, 56 healthcare data breaches were documented, compromising over 45 million patient records. These breaches not only damaged patient confidence but also had sensitive data being offered on the dark web for record-breaking amounts. The causes of such weaknesses ranged from external hacking attacks to insider threats, poor network defense, and poor mobile security policies. These types of breaches necessitate the need for a better adaptive and resilient security system that can both counter outside and inside threats.

The motivation to develop this project was the need to protect the healthcare data which is a fundamental resource to patient care and medical intervention.
As healthcare IT systems grow more complex and cyberattacks become more danger and grow quickly as reported in 2021—it's clear that traditional, perimeter-based security methods aren’t enough. Instead of trusting anyone or anything inside the network, a Zero trust checks every access request, whether it comes from inside or outside the organization. It follows the rule of "*Never Trust, Always Verify*". This method not only fixes the weaknesses of old security models but also creates a flexible and strong framework tailored to meet the specific challenges of the healthcare sector.

CHAPTER TWO: INTRODUCTION TO ZERO TRUST

Zero Trust Architecture is a cybersecurity framework founded on the principle of "*Never Trust, Always Verify*" which keeps the users monitored continuously even after being authenticated, authorized, and given the access to the system resources. This approach never trusts the user and always suspects the user's behavior during the session. So, it enforces strict verification, ensuring that access is granted only when necessary and continuously validated instead of relying on traditional security models that grant broad access based on location or role only.

2.1. Principles of Zero Trust [1] [2]

- **Never Trust, Always Verify**
  Every request is treated as untrusted, requiring authentication and authorization before granting access, and continuous identity verification after granting access.

- **Strong Authentication**
  Robust identity verification methods are required; multi-factor authentication (MFA) and cryptographic methods in order to ensure users and devices are securely identified.

- **Least Privilege Access**
  Users and devices are granted the minimum level of permissions necessary to perform their specific tasks reducing the risk of unauthorized actions and limiting the potential impact of a security breach. This is applied through Role-based Access Control (RBAC) which means assigning the access rights to each user based on role.

- **Micro-segmentation**
  The network is divided into isolated zones like different LANs or VLANs. This limits the literal movement attacks.

- **Continuous Monitoring**
  Systems must constantly analyze behavior and detect anomalies during the session applying the security policies to identify potential threats in real time.

2.2. Benefits of Zero Trust [4] [5]

- **Reduction of Attack Surface and Prevention of Lateral Movement**
  By implementing micro-segmentation with strict access controls at each boundary which significantly reduces the available attack surface. An attacker's ability to move laterally across the network to access other sensitive data or systems is severely restricted. Furthermore, ZT introduction of access based on behavior, user risk and device risk posture, the organization can significantly reduce risk by making it more difficult for users who lost trust and considered to be attackers to discover the network or gain access to it.

- **Protection Against Insider Threats**
  Zero trust works well against malicious insiders as it does against unintentional data leaks from sincere staff members. An insider cannot simply exploit their network presence to obtain data they are not authorized to see because every user is continuously checked.

- **Enhanced Data Security**
  Since ZT is concerned in protecting and ensuring the strictly authorized access of network perimeter and data as it's considered to be the most valuable asset of any system. Additionally, a zero-trust system focuses on meeting the security needs of the system while protecting user data and offering precise authentication and verification.

- **Improved Visibility and Analytics**
  To enforce the "*Never Trust, Always Verify*", a ZTA needs to continuously monitor all network traffic and user activity as well. This generates an empowered stream of data that provides deep visibility into what is happening across the entire system environment. Main motivation of an organization to apply ZT is approving asking every user and every device every time access to the network is requested by validating who, why and how. This architecture is capable of least-privilege access which allows the organization to maintain strict verification of all network users and devices activity.

- **Cloud Compatibility**
  As businesses expand their infrastructure to incorporate servers and cloud-based apps and increase the number of endpoints in their network, a ZTA is an essential security solution. Because it applies security principles uniformly to all users and devices, regardless of location, a ZT network is effectively unbounded.

- **Remote Work and BYOD Provisioning**
  Since ZT does not consider device owner or the network it belongs to, but it only considers the user himself and the device being authenticated, this can enable personal device use.

2.3. Architecture of Zero Trust [2]

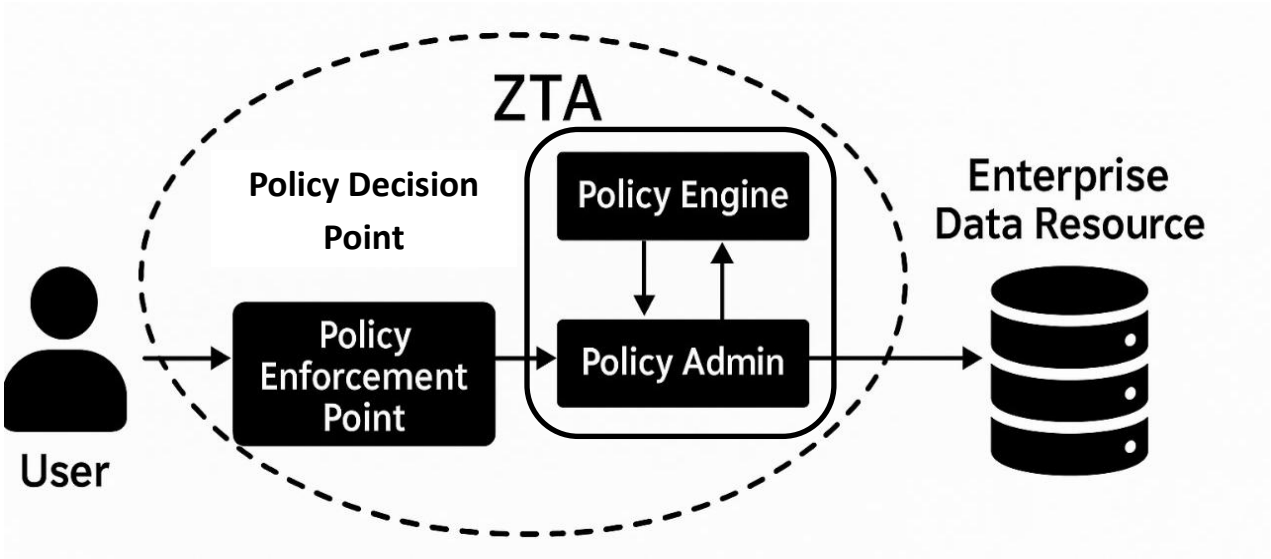

Figure 2 **Zero Trust Architecture** [2]

Zero Trust Architecture (ZTA) is composed of three main logical blocks which introduce the connection between the user and the system resources as shown in *Figure 2*.

The logical blocks are:

- Policy Engine (PE).
- Policy Admin (PA).
- Policy Enforcement Point (PEP).

But some models consider both PE and PA logical blocks as one block called Policy Decision Point (PDP).

The description of the logical components:

2.3.1. Policy Decision Point (PDP)

2.3.2.1. Policy Engine (PE)

This block is the fundamental decision-making unit which is responsible for making the decisions to allow, deny, or revoke access of the user to the enterprise resource based on the enterprise policies defined by the organization and uses contextual information such as user identity, device status, and request attributes which are provided by policy admin component (PA) in order to apply ZT principles on the received requests.

2.3.2.2. Policy Admin (PA)

All user requests from PEP will be forwarded to the policy admin (PA), who will then administer the policies in accordance with the entries in the stated statements. In order for the policy engine to make a decision, the policy administrator will collect the user's data, attributes, and the kind of resources the user is attempting to access. The decision will be implemented at the relevant PEP after it is made.

2.3.2. Policy Enforcement Point (PEP)

The policy enforcement point is the endpoint which users requesting access to data resources from networked devices or applications will go through. Policies defined in the system framework are enforced by the PEP. The policy administrator receives the request from PEP. The policy engine, which includes the security policies, will decide whether to approve the request, and the policy enforcement point implements the decision.

CHAPTER THREE: PROPOSED SYSTEM MODEL

3.1. Tools Used in ZTA Implementation

- Visual Studio Code (VSC).
- MySQL Workbench.
- HyperText Markup Language (HTML) and Cascading Style Sheets (CSS).
- Python.
- JavaScript (JS).
- Node.js
- FastAPI; A python framework for backend design.
- JSON Web Token (JWT).
- Tailscale.
- Wireshark.
- ELK SIEM Tool.
- GitHub.
- Railway; A cloud deployment platform that simplifies hosting and managing web applications and services.

3.2. Zero Trust Architecture Model

Our project proposes a ZTA model, as shown in *figure 3*, implemented on a healthcare environment and simulating the user interface using a website. This model is based on the main logical components of ZTA; Policy Engine (PE), Policy Admin (PA), and Policy Enforcement Point (PEP).

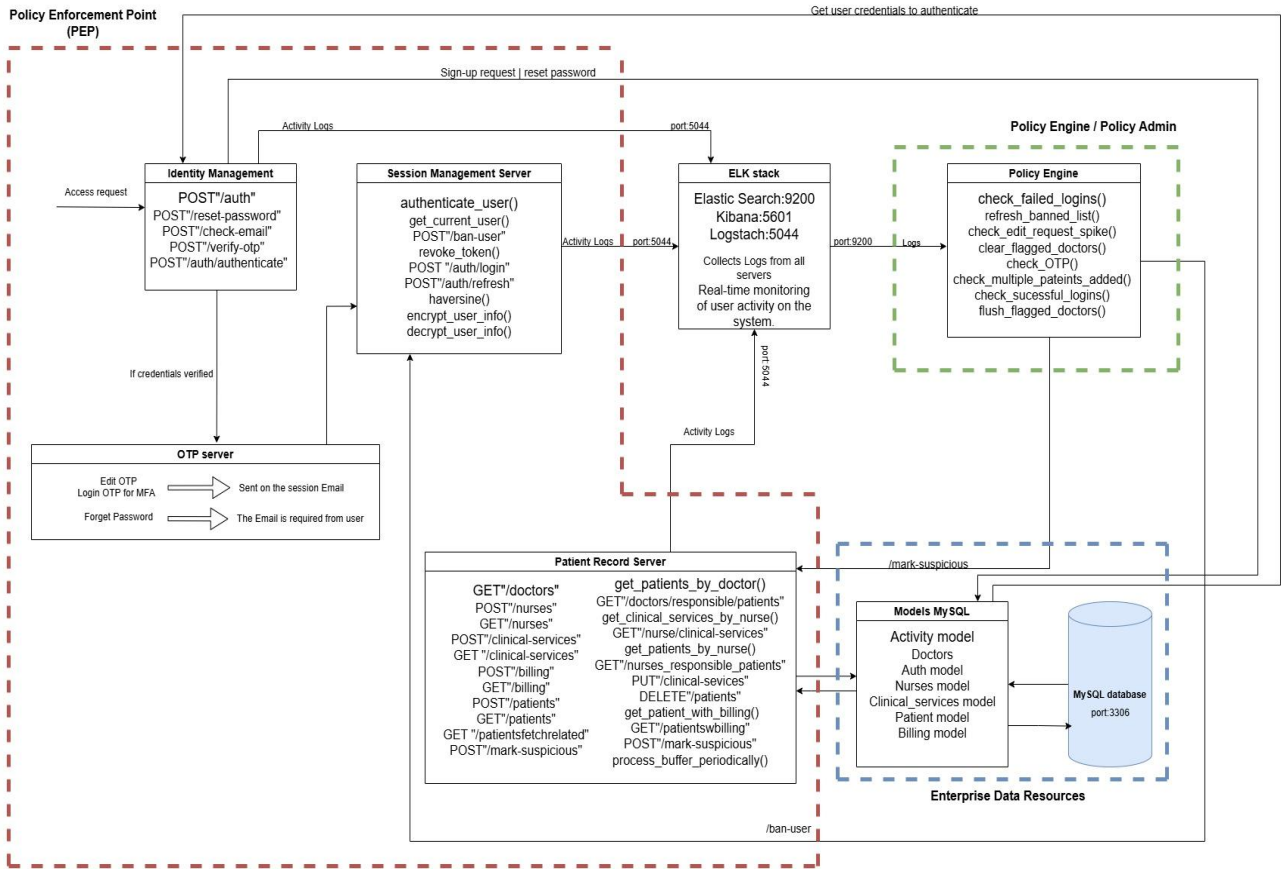

Figure 3 **Proposed Zero Trust Architecture Model**

In this model, both PE and PA are designed together and represented by Policy Engine shown in *figure 3*, while the PEP is introduced by the backend consisting of Identity Management Server and OTP Server, Session Management Server, and Patient Record Server. In ZTA, the enterprise data resource is represented by My SQL database where all data are stored and Models My SQL is the interface between the PEP and the database.

3.2.1. Policy Engine

Policy engine service automated functions continuously watch user activity, detect abnormal behavior, and trigger corrective actions in real time.
Basically, it relies on scheduled Elasticsearch logs to look for suspicious user behavior patterns and spikes throughout his session. When a rule is triggered, the engine sends HTTP calls to a FastAPI security endpoint to take the necessary procedure. This implementation of the policy engine creates a feedback loop that detects, responds to, and audits security events with minimal unusual behavior.

Key Responsibilities:

- Log interrogation.
- Decision logic & rate limiting.
- Automated enforcement to backend API.
- State tracking & expiry.
- Observability & Filtering.

Tools used: Python.

3.2.2. Policy Enforcement Point

The Policy Enforcement Point is broken down into the following modules as shown in *figure 3*:

3.2.2.1. Identity Management Server and OTP Server

The Identity Management server is considered a critical component for managing user identities within the healthcare system. It has an essential role in the identity and access management (IAM) system as this server contains endpoints responsible for managing user identity-related operations the website. It facilitates secure user registration, role-based record initialization, password management, and email-based verification using OTP. The server interfaces with a MySQL database using SQLAlchemy ORM and reports logs for sensitive events as OTP failures for example to the system Logstash server.

Key Responsibilities:

- Signup and registration.
- Reset password.
- OTP Verification.

Tools used: FastAPI – Node.js.

3.2.2.2. Session Management Server

The Session Management Server is the core server that handles users' sessions and ensures a secure and efficient use of system resources. It supports robust authentication through JWT (JSON Web Tokens) access and refresh tokens. This server is carefully logging critical events to a Logstash instance for monitoring.

To enhance security and follow a zero-trust approach system's session management server monitors selected user parameters to achieve continuous monitoring throughout the session as well as token integrity.

Session services include user login, token generation, validation, refreshing, and revocation, as well as intelligent mechanisms such as short-lived token issuance, in-session token revocation, user temporary banning, and continuous session monitoring within refresh requests.

Key Responsibilities:

- User Authentication.
- Access Token Creation.
- Token Refresh Handling and offering security features within session as protection against session hijacking and DoS by overloading the server with refresh requests.
- Session Revocation.
- User Banning.

Tools used: FastAPI – JWT.

3.2.2.3. Patient Record Server

The Patient Record Server is a vital backend server responsible for securing and modularity of our healthcare management system. It basically handles: patients, doctors, nurses, billing, and clinical services efficiently while ensuring security by an endpoint. This server provides well-structured data management, integrity, and traceability, making it an essential component in healthcare data processing and services.

The server interfaces with a MySQL database using SQLAlchemy ORM and reports logs for important events as a user accessing an endpoint or RBAC violation to the system's Logstash server which ensures real-time monitoring and analysis.

Key Responsibilities:

- **For doctors:**
  - View patient records: retrieve / read their patients details and medical history.
  - Update clinical records: modify the treatment plans and prescriptions for their patients.
  - Add a new patient to his patients list.
  - Delete a patient record from his patient list.
- **For nurses:**
  - Can view treatment records related to their department and put checking marks if they finished their tasks.
- **For patients:**
  - Can see their information, diagnoses, and treatment plans.
  - Can see their billing history: retrieve past transactions and check payment status.
- **For unauthorized access attempts:**
  - Unauthorized access attempts are logged and blocked immediately. If an unauthorized action is detected, the user's token is revoked to prevent further system interaction and ending his session.

Tools used: FastAPI.

3.2.3. ELK Stack

ELK is a Security Information and Event Management (SIEM) tool that consists of Elasticsearch, Kibana, and Logstash used for continuous monitoring of logs.
The ELK stack plays an essential role by collecting, storing, and exposing the operational data it needs.

Logstash collects and sends activity logs to Elasticsearch, these logs are collected from: Session management server, Identity management, Patient record server. Elastic search indexes the logs with millisecond-level timestamps and keyword fields which can be searched for later in policy engine. Kibana is the visual interface to handle and manage logs.

Key Responsibilities:

- Ingestion & parsing.
- High performance storage.
- Real time analytics.
- Archiving logs.
- Visualization & reporting.
- Security & compliance.

Tools used: ELK SIEM Tool (Elasticsearch – Kibana – Logstash).

3.2.4. Models MySQL

It is an interface between healthcare management and MySQL database, using SQLAlchemy to handle structured data for users, staff, patients, clinical services, and billing. It manages authentication credentials and interrelated medical and financial records.

Key Responsibilities:

- Authentication credentials.
- Staff Management.
- Patient Records.
- Clinical Services.
- Billing.
- Entity Relationships.

Tools used: FastAPI – MySQL.

3.3. Deployment Architecture and Hosting

As part of our graduation project, we deployed our complete system architecture using Railway, a modern hosting platform. Our application consists of multiple services distributed across separate servers to ensure modularity, scalability, and maintainability. Railway allowed us to manage and host all services under one platform, with GitHub integration and automated deployment pipelines. Without hosting, an app or website has no live server to run on, and users cannot access it online.

3.3.1. System Architecture Overview

The project consists of the following main components:

- **Frontend**
  A static website built using HTML, CSS, and JavaScript.
- **Backend (API Server)**
  A FastAPI application developed in Python, responsible for handling core business logic, database operations, and API endpoints.
- **OTP Services**
  - **Login & Edit Request Service**
    A Node.js server handling OTP validation for login and data modification requests.
  - **Forgot Password Service**
    A separate Node.js server dedicated to OTP handling for password reset requests.
- **Database**
  A MySQL database managed on Railway, used by the FastAPI backend for persistent data storage.

Each of these services was deployed individually on Railway as a separate project, allowing for independent scaling and maintenance.

3.3.2. Deployment Process

3.3.2.1. Frontend Deployment

The frontend was deployed as a static site. We pushed the code to a GitHub repository, then connected this repository to a Railway project. Railway automatically detected the static files and served them directly over HTTPS. The frontend interacts with the backend and OTP services via their respective public URLs.

3.3.2.2. FastAPI Backend Deployment

The FastAPI server was containerized and deployed through a separate Railway project. The source code was hosted on GitHub. Railway built and served the application using Nixpacks. Environment variables were configured through Railway's dashboard, including database credentials and external service URLs. Upon deployment, the backend was assigned a public endpoint, which the frontend and OTP services use for communication.

3.3.2.3. OTP Services Deployment (Node.js)

We split the OTP functionality into two microservices to isolate the logic and improve system clarity:

- **Login & Edit Request OTP Service**
  Deployed as a standalone Node.js service using Railway, connected to the main backend and used during secure login and sensitive data modification operations by doctors.
- **Forgot Password OTP Service**
  Also deployed as a standalone Node.js server on Railway. It handles email-based OTP generation and verification for password reset requests.

Each of these services was deployed from its own GitHub repository and managed as a separate Railway project. Railway provided public endpoints that the backend could call when initiating or verifying OTP requests.

3.3.2.4. MySQL Database Setup

The database was provisioned using Railway's built-in MySQL plugin. Upon creation, Railway provided us with credentials (host, port, username, password, and database name), which we securely stored as environment variables in the FastAPI project. The backend connects to this database using a MySQL client library (MySQL-connector).

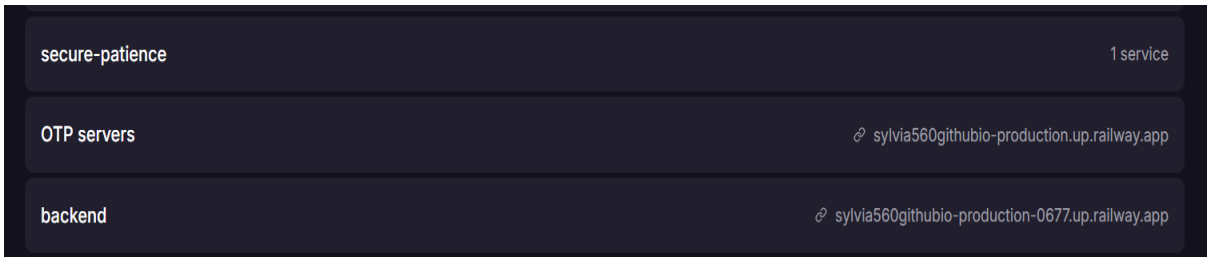

Figure 4 **Servers hosted on Railway**

As shown in *figure* 4, the three servers we have: secure-patience (that is connected to MySQL database), OTP servers with two separate microservices and backend server.

3.3.3. Final Configuration and System Integration

Once all services were deployed and configured, we finalized the integration across the system:

- The frontend communicates with the FastAPI backend and invokes the OTP services when necessary (e.g., during login or password reset).
- All services are deployed on Railway under separate environments, ensuring isolation and clean separation of concerns.
- Public endpoints provided by Railway are used for service-to-service and client-to-server communication.
- Environment variables across all services were carefully configured and encrypted using Railway's secure environment settings.

CHAPTER FOUR: IMPLEMENTATION OF THE LOGICAL BLOCKS OF ZTA

4.1. Encryption and Hashing

4.1.1. Cryptography in our ZTA System

In our ZT-based healthcare system, encryption and hashing play a central role in protecting sensitive data and ensuring secure access. ZTA works on the principle of "*Never Trust, Always Verify*", which requires strong cryptographic techniques to secure both data and identities across the system.

Applied cryptographic methods in our project:

- **AES-256 Encryption**
  We used AES-256 to encrypt sensitive patient and doctor information, including treatment records and national IDs. This ensures that even if data storage is compromised, unauthorized access is prevented.

- ***Bcrypt* Password Hashing**
  Users' passwords are hashed using $bcrypt$ before being stored in the database. This adds salting and computational complexity, protecting against brute-force and rainbow table attacks.

- ***Base*64 Encoding**
  To support compatibility when transmitting encrypted data over services that only accept text, we use $Base64$ encoding for a safe transmission not for security.

These cryptographic controls are embedded at multiple levels, from database protection to user authentication, and are essential in supporting the ZT model. They enforce access control, maintain data integrity, and help comply with privacy and security regulations relevant to healthcare applications.

4.1.2. Chosen Algorithm and Technical Specifications

4.1.2.1. Symmetric Encryption [6]

To protect sensitive data within the system, symmetric encryption is employed using the Fernet protocol from the Python cryptography library. Fernet is a high-level cryptographic scheme that ensures both confidentiality and integrity of data. It is built on top of industry-standard algorithms and is designed to be easy to use while offering strong security guarantees.

Fernet:

- **Symmetric Encryption**
  Fernet uses the same key for both encrypting and decrypting data.
- **Secure**
  It guarantees that a message encrypted with it cannot be manipulated or read without the correct key.
- **Authentication**
  It includes integrity checks to ensure the encrypted data has not been tampered with.
- **Key Rotation**
  Fernet supports key rotation through MultiFernet.
- **Implementation**
  It is built on standard cryptographic primitives and is secure by design.
- **Use Cases**
  It is suitable for encrypting data that fits in memory.

Cryptography.fernet:

- **Library**
  It is a module within the cryptography library, which provides cryptographic recipes and primitives for Python developers.
- **Usage**
  It provides a simple way to encrypt and decrypt messages using symmetric encryption.
- **Key Generation**
  It allows the generation of secure keys for encryption.
- **Data Handling**
  It works with byte strings for cryptographic operations.
- **Limitations**
  It is generally unsuitable for very large files, as the complete message contents must be available in memory.

Let us discuss the encryption algorithm details.

Encryption Details:

- **Encryption Scheme**
  Fernet (from the cryptography library).
- **Underlying Algorithm**
  AES (Advanced Encryption Standard).
- **Key Size**
  128 bits (derived from a 32-byte key).
- **Cipher Mode**
  CBC (Cipher Block Chaining).
- **Padding Scheme**
  PKCS7 (used internally for block alignment).
- **Authentication**
  HMAC (Hash-based Message Authentication Code) using SHA-256 for integrity verification.
- **Key Composition**
  - 16 bytes used for AES-128 encryption.
  - 16 bytes used for HMAC-SHA256 authentication.
- **Output Encoding**
  URL-safe $Base64$ (to ensure safe storage and transmission of encrypted content).
- **Number of Rounds**
  10 (standard for AES-128 as defined by the AES specification).

The encryption key used for securing certain fields in our system appears as a 44-character string when stored in plaintext. However, this is the $Base64$-encoded version of the original key. Since $Base64$ encoding represents every 3 bytes of binary data as 4 characters, a 32-byte (256-bit) AES key results in a 44-character string after encoding. This format ensures safe storage and transmission of the key in systems that require textual data, while maintaining compatibility with the AES-256 encryption standard. [7]

```
# Encryption Key
ENCRYPTION_KEY = "dVis0RwmM8y9jvckrSxFM3WrHOQfvbNN9gstq7CT8S4="
```

Figure 5 **Encryption Key**

This encryption method ensures that encrypted data cannot be decrypted or tampered with without access to the original secret key, and that any unauthorized modifications will be detected and here is an example of the cipher text generated from the plain text in *figure 6* encrypted by the key above in *figure 5*.

Figure 6 **Cipher text of encrypted data**

4.1.2.2. Password Hashing

Passwords are never stored in plaintext. In our project we used $Bcrypt$ in python (Blowfish algorithm), it is a 256-bit symmetric-key password hashing function designed by Bruce in 1993 but is still used today as one of the strongest algorithms in hashing. It is designed to be slow and computationally expensive making it resistant to brute-force attacks. It also uses built-in salting which makes hashing more secure. This approach ensures that even if the password database is compromised, attackers will face significant difficulty in recovering the original passwords. *Figure 7* represents the hashed passwords stored in the database.

Hashing Details:

- **Hashing Algorithm**
  $Bcrypt$.
- **Work Factor (Cost Parameter)**
  12 (adds significant computational cost to each hashing operation, making attacks more difficult).
- **Salting**
  Automatically generated and included with the hash to ensure uniqueness and protect against rainbow table attacks.
- **Hash Output Format**
  Modular Crypt Format (MCF), which embeds versioning, cost, salt, and hash in a single string.

User_ID	Username	Password
503919	alexandrialara	$2b$12$jgLKhiVKwwV4YHXeTqA/rua1g42uTGvkqRIkBnQlMloZfQ3Rxo0Tm
14222	aliciafitzgeralddds	$2b$12$JZbAh4ESnTCL1hRZa5LWDeyNPLZ012n5kouQnZR8EbH8SsjdLzq8O
410880	alyssawilliams	$2b$12$3o51vS7eQdAsTfXqsThGHOYXDqO7z7G8IRWtDF8QK3tgKxh7qfZLC
890647	amywilliams	$2b$12$RwjcDfAA/8RmLWxPLreVuOQ2G2nWfGleTfNe/o8TpgSA9hhWDUaEu
848418	anthonyowens	$2b$12$3a0HiIBYATGL7y11PbMniO0FPyi7/tUetaHWumIA16ueOZ2F.YeMq
727669	ashleyperez	$2b$12$FQm2vE5N4BKiqXXsOL.UG.Xqm7sHPKcP/n2DXT1WJOGmjDNsVZZ7y
170993	ashleyrobertson	$2b$12$aa65f1rxqMVZnCiE47fRsuKP7dtbfzRN.pDrC7wLV139JI17J5sye
610680	barbararuiz	$2b$12$AzdpQ4.5.C5nibXXz0FpuOigywnk/aEtJXArY9dOji2Nn5Y3jtIlK
276375	benjaminwalker	$2b$12$cfSyPHD47SqePcHCReeqkOVzSuqyDDHxiAFpBd2ugkyxsTHNQsZ2W
469354	bethanymartin	$2b$12$4.Ih.RNuu7FRp7FaH5FlOuR7vR2xdo1gsuVb1O/vTfTX2Lectvj9O
562788	brandondurham	$2b$12$jIsfvBDICNN8biu0QIIpzecdqAGqgKJYfU5MTsNcpn/rjdV17cHFm
525354	brettromero	$2b$12$I7EB9XWyhMEVP6T3i3jSb.4ZSmMSq7KfOyt4HpJ7/XsH4F/mVK2fS
12453	brianerickson	$2b$12$J2GnAU3gHaFtWtBQH6xs.uib.vrt7fM0RxhyT1irrCGyVtVnjn5o2

Figure 7 **Hashed passwords in the database**

The following *Table 1* explains the different fields in the cipher text generated. [8]

Table 1 **Hash Output Format**

Section	Example	Description
Prefix	$2b$	Identifies the $bcrypt$ version ($2a$, $2b$, or $2y$).
Cost factor	12	2^{12} $iteration$
Salt	9UjFvOnGyUXRrKbFp/2J3e	A **22-character base64-variant encoding** of a **16-byte salt.**
Hash	Zq9VBykg9/zi9J8cMXlJkgHke3xJ2Xi	The **31-character encoding of the 192-bit (24-byte) $bcrypt$ output.**

4.1.3. Theoretical Brute Force Analysis

AES-256 has a key space of 2^{256} possible combination, making it effectively unbreakable by brute force attacks with current and foreseeable technology. Even if a device could check a trillion keys per second, it would still take an incomprehensible amount of time to try every key.

Let us break this down:

- 1 trillion guesses per second = 10^{12}.
- Total keys = $2^{256} \approx 1.15 \times 10^{77}$.
- Time required = $\frac{1.15 \times 10^{77}}{10^{12}} = 1.15 \times 10^{65}\ seconds$.
- This is far longer than the age of the universe (approximately $10^{17}\ seconds$ which is around $3.170979 \times 10^{9}\ year$).

This analysis shows that brute force attacks are not a realistic threat to our encryption, assuming keys are properly generated and protected.

4.1.4. Database Field Encryption

In our ZTA system, specific fields in the database are encrypted to protect the most sensitive information. This follows the principle of least privilege and ensures that data exposure is minimized even in the event of a database breach.

- **Patients Table**
  The Contact, Allergies, Chronic Conditions, Purpose of Visit columns are encrypted.
- **Clinical services Table**
  The Medication Name, Dosage Instructions, Treatment Details columns are encrypted.

```
# Encrypt fields in Clinical_services_modified table
clinical_services_records = db.query(modelsmysql.Clinical_services).all()
for record in clinical_services_records:
        record.Medication_Name = encrypt_data(record.Medication_Name)
        record.Dosage_Instructions = encrypt_data(record.Dosage_Instructions)
        record.Treatment_Details = encrypt_data(record.Treatment_Details)

# Encrypt fields in Patients table
patients_records = db.query(modelsmysql.Patient).all()
for record in patients_records:
        record.Contact = encrypt_data(record.Contact)
        record.Allergies = encrypt_data(record.Allergies)
        record.Chronic_Conditions = encrypt_data(record.Chronic_Conditions)
        record.Purpose_of_Visit = encrypt_data(record.Purpose_of_Visit)
```

Figure 8 **Encrypted fields in the database**

Each field was selected based on sensitivity and regulatory compliance requirements. Encrypting these fields helps ensure confidentiality while allowing the system to remain functional. *Figure 8* represents the part of the code in which theses sensitive fields in the database are encrypted.

4.2. Multi-Factor Authentication (MFA) [1] [9]

4.2.1. Comparative Analysis of Single-Factor and Multi-Factor Authentication

As the volume of data is growing exponentially every day, the risk of cyber threats increases correspondingly in the real world. This demands the implementation of more advanced security levels in order to ensure that sensitive information remains highly-protected against the evolving cyberattacks, such as brute force attack, and phishing.

In this project, our focus is on a medical environment, which is characterized by its critical and sensitive nature, as it involves handling a vast amount of confidential information, including personal identification, medical histories, diagnostic results, and treatment records. Such environments require the implementation of a comprehensive and very well-structured security system in order to protect such massive and sensitive data from any breaches. Ensuring the protection of this data is not only a technical necessity but also a legal and ethical obligation, particularly in light of stringent data protection regulations governing the healthcare sector.

In order to achieve our objective, we propose the implementation of a secure authentication framework relying on Multi-Factor Authentication (MFA) rather than Single-Factor Authentication (SFA). Single-Factor Authentication is considered an inadequate method for protecting high-risk and very sensitive systems where it typically involves only password for authenticating a user. In contrast, Multi-Factor Authentication introduces a higher level of security by requiring two or more authentication factors from different authentication categories which are:

- **Knowledge**; something the user know, such as password or PIN.
- **Possession;** something the user have, such as One-Time Passwords (OTPs) generated by smartphone applications or sent via email or SMS.
- **Inherence;** something the user is, such as a biometric like fingerprints and facial or voice recognition.

4.2.2. MFA in Zero Trust Architecture

In order to align with the main concept of Zero Trust Authentication, which is "*Never Trust*", an approach with an additional layer of security level is required. Such a layered approach enhances the security of sensitive systems and reduces the probability of unauthorized access. So, using Multi-Factor Authentication is a must in order to keep our critical healthcare system well-secured as it ensures the confidentiality, integrity, and availability of medical information. Moreover, it aligns with internationally recognized data protection standards and regulatory requirements, such as Health Insurance Portability and Accountability Act (HIPAA) and General Data Protection Regulation (GDPR), which mandate strong access controls for safeguarding personal health data.

4.2.3. Implementation of Multi-Factor Authentication

In the design of our healthcare website, we implemented Multi-Factor Authentication (MFA) system to provide a robust security framework for authenticating users. This implementation is achieved by enforcing the user to authenticate using two distinct authentication factors from two different categories:

- **First Factor (Knowledge-Based Factor)**
  Password which represents something that is already known by the user and is securely stored in the system database using cryptographic hashing algorithm.
- **Second Factor (Possession-Based Factor)**
  One-Time Passwords (OTP) which represents something that the user has and is dynamically generated and sent to the user via his/her registered email address during the login process.

A successful access to the system and a secure communication channel is not given to the user unless both authentication factors, password and OTP, are verified successfully. Without the verification of both security levels, the access of the user is denied and the session does not begin.

4.2.4. One-Time Password (OTP)

A One-Time Password (OTP) is a security mechanism that generates a unique, time-limited password for a single login or transaction. It introduces an extra layer of authentication based on possession category; something the user has.

In our design, the mechanism of OTP is implemented on a Node.js server using Express. The OTP is:

- **Randomly generated**
  - A 6-digit random number is generated ranging between 100000 and 999999.
- **Hashed for security**
  - For providing a higher security level, the generated OTPs are hashed instead of being stored in plaintext form. They are hashed using $bcrypt$ hashing algorithm.
  - $bcrypt$ is a password-hashing function designed to securely store passwords and mitigate attacks.
  - The $bcrypt.hash$ () takes the 6-digit OTP and a cost factor of 10, and returns a secure hashed version of the OTP.
  - $bcrypt$ algorithm incorporates a unique salt for each hashed value which makes it computationally expensive to reverse.
- **Sent to the user's registered email address via Nodemailer**
  - Once the OTP is generated and hashed, its original plaintext form is sent to the user's registered email address using Nodemailer library which is configured with Gmail SMTP transporter to deliver the email.
  - The email body contains the user's OTP which will be used for verification. This OTP is valid for $five\ minutes$. Once the validation duration ends, the OTP becomes expired and unusable.
- **Valid for a limited number of verification attempts**
  - The user submits the OTP to be verified where the server first checks whether a corresponding hashed OTP exists for the provided email address.
  - For a higher probability of mitigating attacks, each user has a maximum allowed number of three attempts to enter the correct OTP.
- **Verified securely**

  The $bcrypt.compare$ () function is used to securely compare the submitted OTP with the hashed stored one.

  If the OTP matches the stored hashed one and the comparison successes:
  - The OTP and attempts record are deleted from the memory and the OTP becomes unusable again.

- Access is granted for the user and a JWT token is given so that the session starts and the user is directed to the section in the system based on its role (doctor, nurse, or patient).

If the OTP does not match the stored hashed one and the comparison fails:

- The system tracks the number of attempts where the attempt counter is incremented until three attempts and then the OTP becomes invalid.
- On exceeding the maximum allowed attempts, the system bans the user for a duration of $two\ minutes$.

4.3. Role-Based Access Control (RBAC) [10]

Access control has many models and the model we used is role-based access control (RBAC) which is the approach of assigning users' authorization according to their position within an organization. It provides a simple, controllable method of managing access that is less error-prone than giving users separate permissions which allows scalability of the project.

Role management with RBAC involves analyzing user demands and assigning users to roles based on common responsibilities. Each user is then given one or more roles given one or more permissions. The user-role and role-permissions relationships make it simpler and much efficient to perform user assignments, but instead have privileges that conform to the permissions assigned to their roles since users no longer need to be managed separately.

Role assignment is essential for collection of permissions can be applied to users, which makes it easier to add, remove or modify the permissions than assigning permission to each user individually.

4.3.1. Benefits of RBAC

Role-based access control makes it easier to handle access management as long as role requirements are adhered strictly which helps to:

- Creating scalable, systematic, and repeatable assignment of permissions.
- Easier auditing of user privileges and correction of possible issues.
- Quicker addition and changing of roles as the RBAC are implemented to APIs.
- Reduction of potential errors when assigning user permissions.
- Effective compliance with regulatory requirements for confidentiality and privacy.

4.3.2. RBAC in our ZTA system

Users of the system are given roles such as "Doctor", "Nurse", or "Patient" which are used to define authority level as shown in *table 2*. When users are accessing the system, they are given an access token and this token contains their role as will be discussed later.

Throughout the session user accesses API endpoints as shown in *figure 9*, RBAC is done at the beginning of each API function whose role is checking whether the token given to this user authorizes him to access this API endpoint or not as in *figure 10* code snippet. This way the system performs RBAC and prevents any unauthorized access of the backend server endpoints. Moreover, even if the user is given the access token based on his role, the system keeps monitoring the session and the user's behavior and takes the proper decisions against the user in case of unusual behaviors according to the policy engine. Although the user is given some specified rights based on his role, he is prompted to re-verify his identity while trying to do some sensitive actions. For example, a doctor is asked to verify an OTP in case he wants to edit a patient's record.

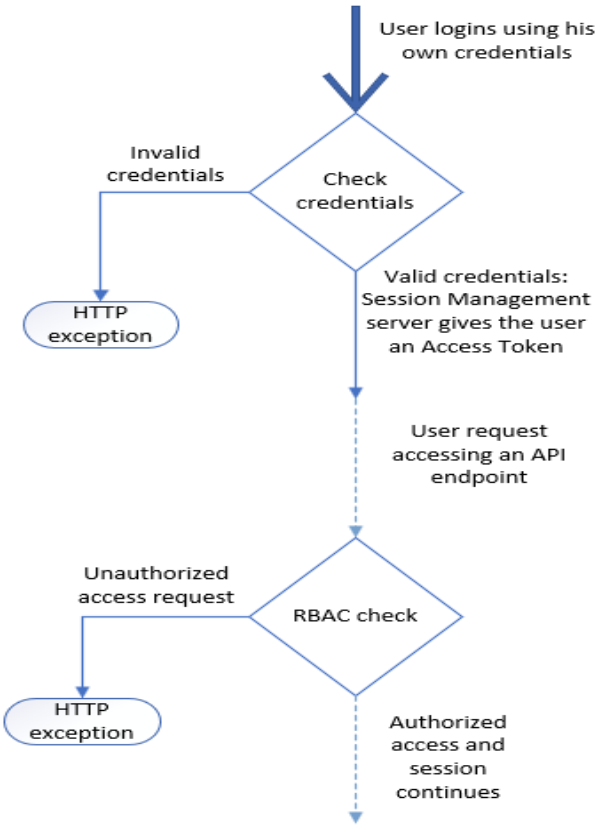

Figure 9 **Role-Based Access Control Flow in the System**

```python
@Patient_record_router.put("/clinical-services/{Patient_ID}", status_code=202)
async def buffer_clinical_update(
    Patient_ID: int,
    update_data: ClinicalServiceUpdate,
    current_user: dict = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    # Verify permissions
    if current_user["role"] != "Doctor":
        revoke_token(current_user["token"])
        raise HTTPException(status_code=403, detail="RBAC unauthorized")
```

Figure 10 **Code snippet of Edit Patient API Endpoint which should be accessed by doctors only**

Table 2 **Rights given to each user based on the user's role**

	Doctor	Nurse	Patient
Responsible Patients	✓	✓	✗
Edit clinical services	✓	✗	✗
Delete clinical services	✓	✗	✗
Add patient	✓	✗	✗
Patient Health overview	✓	✓	✓
Billing and Finance	✗	✗	✓
Patient clinical services	✓	✓	✓

The following figures; *figure 11*, *figure 12*, and *figure 13*, show the different rights and the resources each user; doctor, nurse, and patient respectively, can access based on his role applying RBAC.

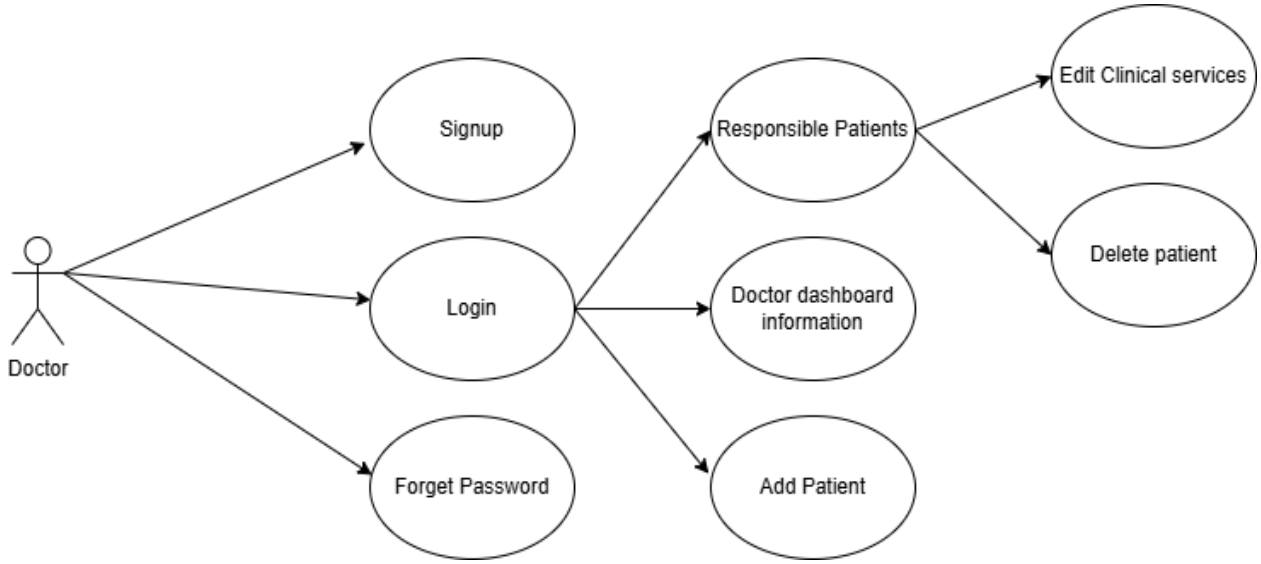

Figure 11 **Rights given to the doctor**

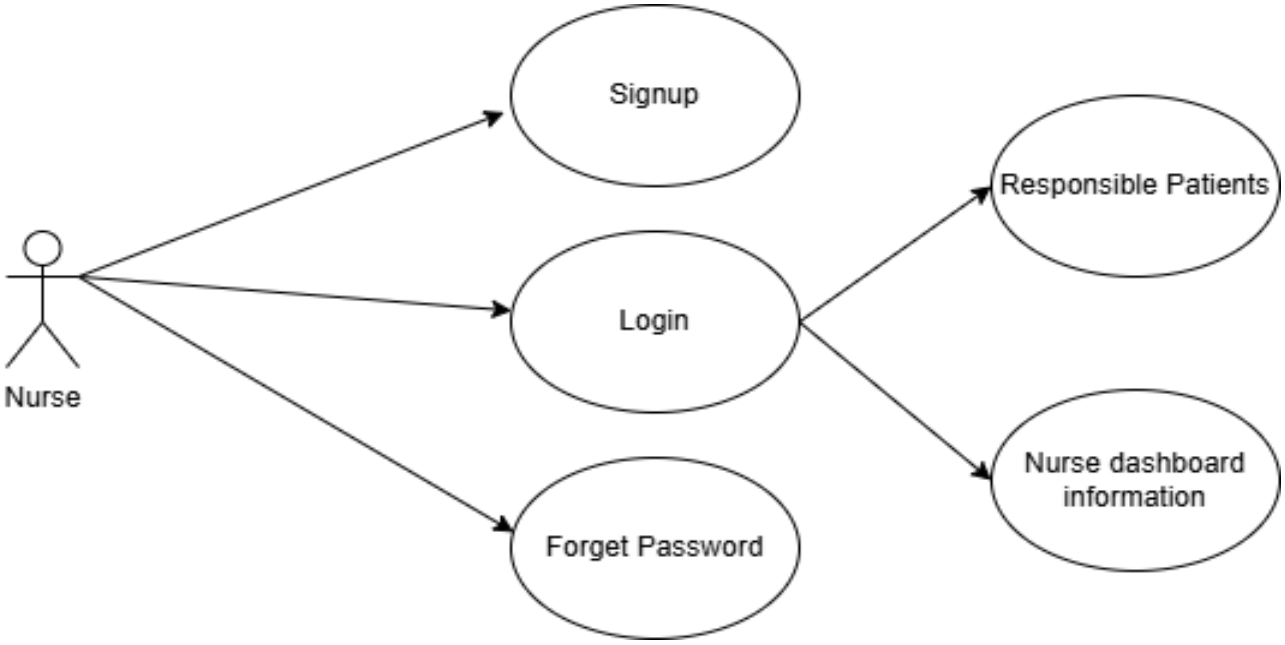

Figure 12 **Rights given to the nurse**

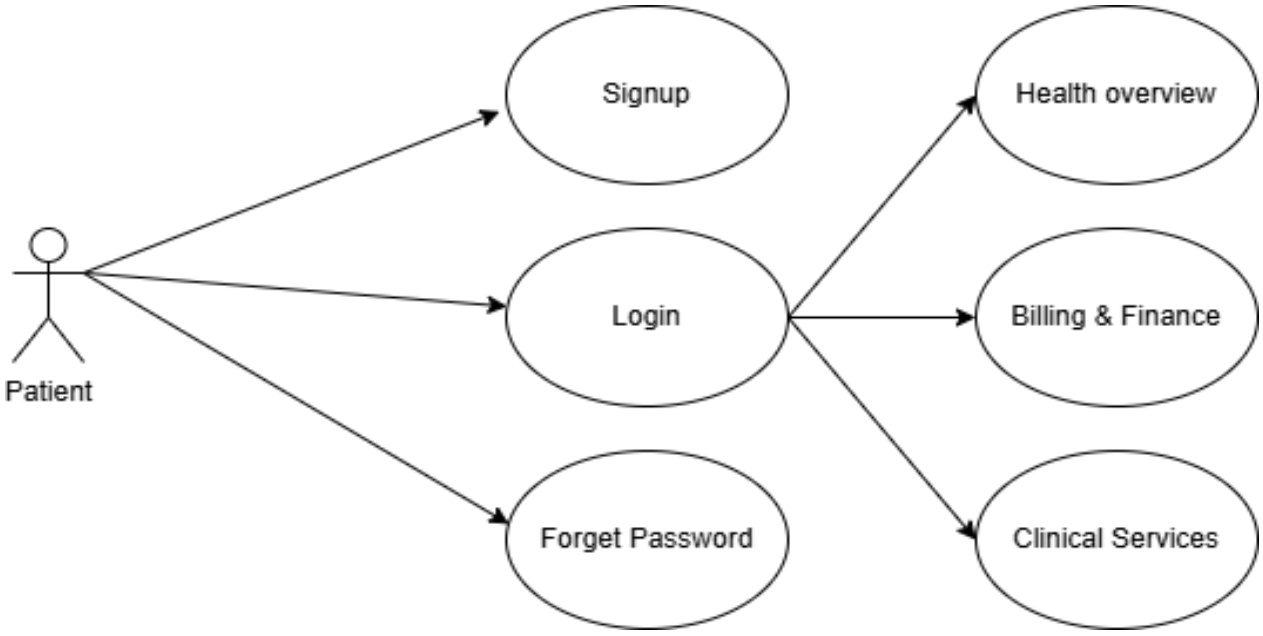

Figure 13 **Rights given to the patient**

4.4. JSON Web Token (JWT)

4.4.1. Introduction to JWT

- **What is a Token**

  Token-based authentication mainly depends on access tokens to allow users' access to an application API endpoint. Once the user passes the initial challenges necessary to be authenticated, he receives an access token for authorization throughout his session. Later, the user passes this access token as a credential once he requests access to a target API [11].

- **JSON Web Token (JWT)** [12]

  According to the JSON Web Token (JWT) Specification RFC 7519, JSON Web Tokens (JWT) are an open standard. Claims between two parties are safely represented by them. Although claims can be associated with any business process, they are usually utilized to represent an identity and its connections. For example, the JWT represents a user who is a member of an administrator role or group.

  A Message Authentication Code (MAC) is typically used to digitally sign the claims in a JWT, which are encoded as JSON objects. Authentication is the most common use case for a JWT. The JWT is included in every request after the user logs in, enabling the user to access services that are approved by that token.
  Because of its small size, a JWT can be sent quickly by a URL, a POST parameter, or an HTTP header. A JWT does not require the recipient to call a server to validate the token, and it contains all the necessary information about an entity to prevent repeated database queries, and that is what makes it much faster than using any method of authorization.

- **Why Tokens but not cookies** [13]

  Tokens, usually referring to JSON Web Tokens (JWTs), are signed credentials encoded into a long string of characters created by the server, whereas cookies are chunks of data created by the server and sent to the client for purposes of communication. Tokens are stateless, whereas cookies are stateful. So, Tokens being stateless makes it much faster to authorize without need to return to the database for every single refresh or API endpoint request (No server-side session storage is needed).

  The reason why we need either is because even if you authenticate with a single request, the server effectively "forgets" that authentication with subsequent queries because HTTP is stateless. As a result, you must provide the token or cookie whenever the server requests authentication. Until the cookie or token expires, the frontend keeps the token or cookie stored and uses it to send more requests to the server.

There are two main problems with cookies which made us prefer to use tokens over it:

- **Scaling issues**
  Because sessions are tied to a specific server, we may encounter problems when scaling our application into separate different servers. In a load-balanced application, the session data is lost if a logged-in user is redirected to a different server; to avoid this, sessions must be stored in a shared database or cache, which makes each interaction more complex and overloads the database.
- **Not suitable for API authentication**
  APIs provide one-time resources for authenticated end-users and do not need to keep track of user's sessions. Cookies do not work well in this situation because they track and validate active sessions, also this does not match our ZT principle. Tokens, meanwhile, provide authentication with a unique identifier on every request to the API endpoints.

4.4.2. Benefits of JWT

- **More compact**
  Compared to other types of tokens; when the JWT is encoded, it is much more compact and smaller in size.
- **More secure**
  JWT is securely signed by public/private key pair in the form of an X.509 certificate. It can also be symmetrically signed by a shared secret using the HMAC algorithm.
- **More common**
  Since JSON parsers translate directly to objects, they exist in the majority of programming languages. On the other hand, there is no inherent document-to-object mapping in XML. Because of this, working with JWT is simpler than legacy SAML representations.
- **Easy processing**
  JWT is easier to be processed on users' devices and mobiles specially as it is used at internet scale.

4.4.3. Use Cases of JWT [14]

- **Authentication**
  After logging in successfully of the user using his own credentials; ID token represented in JWT is returned from the authentication server back to the user.
- **Authorization**
  When the user successfully logs in, he may request to access any of routes, services, or resources as APIs. In our application, these API endpoints are secured and locked such that according to the user request to access this endpoint, the server is responsible for checking and verifying according to the token this user has so according to the server authorization policy it decides whether giving this user access or not.
- **Information Exchange**
  Because JWTs can be signed, you can be confident that the senders are who they claim to be, making them an excellent method of securely exchanging information between parties. A JWT structure also makes it possible to confirm that the content has not been tampered.

4.4.4. Why is JWT secure [14]

Because the JSON object is securely signed, the data it contains may be validated and trusted. $Auth0$-issued JWTs are JSON Web Signatures (JWS), which means they are signed instead of encrypted, even though JWTs can also be encrypted to provide confidentiality between parties. Therefore, we will concentrate on signed tokens since they can validate the accuracy of the claims they contain, whereas encrypted tokens hide those claims from other parties. In our system, we prefer partial encryption of the token, as will be discussed.

In general, JWTs can be signed using a secret (using the HMAC algorithm) or a public/private key pair using RSA or ECDSA. Signing a token with a public/private key combination also ensures that the only person who signed it is the one with the private key.

A JWT should be correctly authenticated using its signature before being utilized. Keeping in mind that a correctly validated token just indicates that no one else has changed the data it contains. This does not mean that others were not able to see the content, which is stored in plain text. This is why you should never put sensitive data inside a JWT. Other precautions are also taken to make sure that JWTs are not intercepted, such sending them only over HTTPS, using only secure and current libraries, and following to best practices.

4.4.5. Digital Footprint and its Implementation in the System [15] [16]

4.4.5.1. Digital Footprint

A digital footprint which may be called a "Digital Shadow" or an "Electronic Footprint" refers to the trail of data you leave when using the internet. It is a record of your online behavior and mobile device actions like websites you visit, emails you send, and information you submit. Internet users' digital footprint is created either actively or passively. A user does not leave every digital trace on purpose. Digital footprints play a crucial role in modern applications, offering benefits in security, analytics, and user experience. Sometimes a user leave information unknowingly and there are four types of digital footprint:

- **Active**
  When you purposely perform an action, place information online, or accept browser cookies and share information about yourself. Some examples: posting or participating on social networking sites, completing and submit online forums, and when user is logged into a website through a registered username or profile or password.
- **Passive**
  Data is collected without the user's direct knowledge. A lot of websites store statistical information about you behind the scenes. For example, some collect information about how many times users visit a page or a website, register your IP address and some device information.
- **Private**
  A private digital footprint consists of information that only some people can access, for example when a user belongs to a private chat group, there will be a record of text messages, but this user and the other members can access them.
- **Public**
  Your public digital footprint is all the information you have posted on public forums which anyone can find it.

Since Digital Footprint mainly depends on collection information about user there will be Ethical Considerations and challenges of Digital Footprint:

- **Balance between Surveillance and Privacy**
  Respecting individuals' privacy by collecting and using their digital data with explicit consent.
- **Purpose Limitations**
  Using the collected digital footprints solely for the predefined objectives of our security project.

- **Data Minimization**
  Only the necessary data for a specific purpose can be collected. This prevents over-collection of digital footprints and reduces privacy risks as an entity is responsible for securing the collected data and preventing it from data breaches.

4.4.5.2. Digital Footprint in the System

In this system implementation, we focus on three main elements and the three elements are used inside the access token and this collected information is considered passive information:

- **Operating System Detection:**
  It is collected by Extracting it from the User-Agent HTTP header to Identify the user's device OS (Windows, macOS, Android, iOS, etc.) as in *figure 14*, it is usually used for detection suspicious devices as unexpected OS in login attempts.
- **Browser Fingerprinting**
  This is parsed from the User-Agent string to captures browser type (Chrome, Firefox, Safari, Edge) and version as in *figure 15*. This can be used for detection suspicious devices as unexpected OS in login attempts, and detection bots or automated scripts (if browser behavior is abnormal) as well.

```
// Function to capture OS and Browser info accurately
function getOS() {
    let OSName = "Unknown OS";
    if (navigator.appVersion.indexOf("Win") != -1) OSName = "Windows";
    if (navigator.appVersion.indexOf("Mac") != -1) OSName = "MacOS";
    if (navigator.appVersion.indexOf("X11") != -1) OSName = "UNIX";
    if (navigator.appVersion.indexOf("Linux") != -1) OSName = "Linux";
    return OSName;
}
```

Figure 14 **Operating System detection in frontend**

```
function getBrowser() {
    let userAgent = navigator.userAgent;
    let browserName;

    if (userAgent.match(/edg/i)) {
        browserName = "Edge";
    } else if (userAgent.match(/opr\//i)) {
        browserName = "Opera";
    } else if (userAgent.match(/chrome|chromium|crios/i)) {
        browserName = "Chrome";
    } else if (userAgent.match(/firefox|fxios/i)) {
        browserName = "Firefox";
    } else if (userAgent.match(/safari/i) && !userAgent.match(/crios|chromium|edg|opr|fxios/i)) {
        browserName = "Safari";
    } else {
        browserName = "No browser detection";
    }

    return browserName;
}
```

Figure 15 **Browser Fingerprinting in frontend**

- **IP Geolocation Tracking**
  It is collected by using IP geolocation APIs or a self-hosted database as in *figure 16* to Estimates the user's location (country, city, ISP). May be used for flagging logins from unusual locations if location is outside the hospital another challenge will be taken to make sure the authority of user and limit the data that can be accessed.

```
// Get the user's location
const position = await getLocation().catch(async () => await getIPLocation());
if (!position || !position.coords) {
    throw new Error("Unable to retrieve location.");
}
const location = `${position.coords.latitude},${position.coords.longitude}`;
console.log("Retrieved location:", location); // Debugging log
```

Figure 16 **IP Geolocation Tracking in frontend**

4.4.6. JWT Structure

Considering the following JWT in *figure 17* returned to the user after authentication, token structure is obvious in *figure 18* after decoding the token.

```
Token:                                                                    doc.html:253
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkZW5pc2UudGVycnlAbWVkbWFpbC5jb20iLCJpZCI6Mjg3ODM4LCJyb2xlIjoiRG
9jdG9yIiwidHlwZSI6ImFjY2VzcyIsImV4cCI6MTc0Mzk0Nzc2MywidXNlcl9pbmZvIjoibG9jOiAzMC4wNjAyNCwzMC45NjExNDMsIG9zOiBXa
W5kb3dzLCBicm93c2VyOiBFZGdlIn0.NZ3OaR_id1H4LeZ5v9Rvrn0FRbWBS6fExzWDmQG4nws
```

Figure 17 **Token returned to user after authentication**

Figure 18 **Decoded Token**

Token structure consists of:

- **Jose Header**
  The parameters that describe the cryptographic techniques and parameters used are located in a JSON object. The hashing algorithm being utilized (such as HMAC SHA256 or RSA) as shown in *figure 19* and the type of JWT are a couple examples of the Header Parameters which comprise up the JOSE (JSON Object Signing and Encryption) Header, these parameters usually consist of a name/value pair [11].

Figure 19 **JWT Header**

- **JWS Payload**
  The payload contains statements about the entity which is typically, the user, and additional entity attributes, which are called claims. Here in our case, we included user information as in *figure 20*, which simply contributes the current user geolocation, operating system (OS) and browser. These parameters will be used later on to contribute the ZT principle. Also, it should be put into consideration that having such sensitive information explicitly means having user information susceptible to risk which is unacceptable, so we will handle this later on.
  The reason why the user information is stored in the token instead of storing in the database is the preservation of the statelessness principle of the token discussed before.

Figure 20 **JWT Payload**

- **JWS Signature**

  The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message was not changed along the way. To create the signature, the $Base64$-encoded header and payload are taken, along with a secret, and signed with the algorithm specified in the header. So, in our code in the session management server we have a "*get current user*" function which is responsible for authorization any API endpoint access request and according to the passed token it gets the user ID and the most important part is token signature verification represented in verifying that this token was originally made by our session management server and not modified or created by any other entity rather than system server responsible for handling tokens. Unsecure webserver accepts unsigned JWT as in *figure 21* which makes it possible for malicious user to send a modified or fake JWT and server accepts it [11].

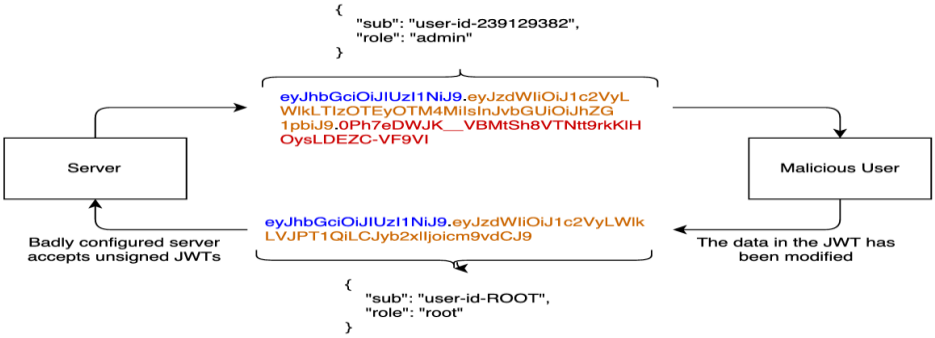

Figure 21 **Signature Role for JWT Integrity** [17]

Since the secret used in JWT signature is only stored in our server securely, the following mathematical operation in *figure 22* is done before using representing the signature can only be produced from our server and nobody else can produce the same signature without knowing the secret.

```
HMACSHA256(
      base64UrlEncode(header) + "." +
      base64UrlEncode(payload),
      secret)
```

Figure 22 **JWT Signature Operation**

HMAC-SHA256 consists of:

- **HMAC:** Hash-based Message Authentication Code.
- **SHA256:** Secure Hash Algorithm 256-bit.

HMAC serves both purposes of authentication and data integrity issued in RFC 2104. [17] So, it is mainly a non-reversible operation such that anyone having the token still cannot get the secret used to originally produce this signature. Common uses of HMAC can be: Session tokens, in transit message verification, and data integrity checks.

Figure 23 illustrates using HMAC for signing a token where message is the token payload and the shared secret is supposed to be secret shared only between servers concerned in session management operations.

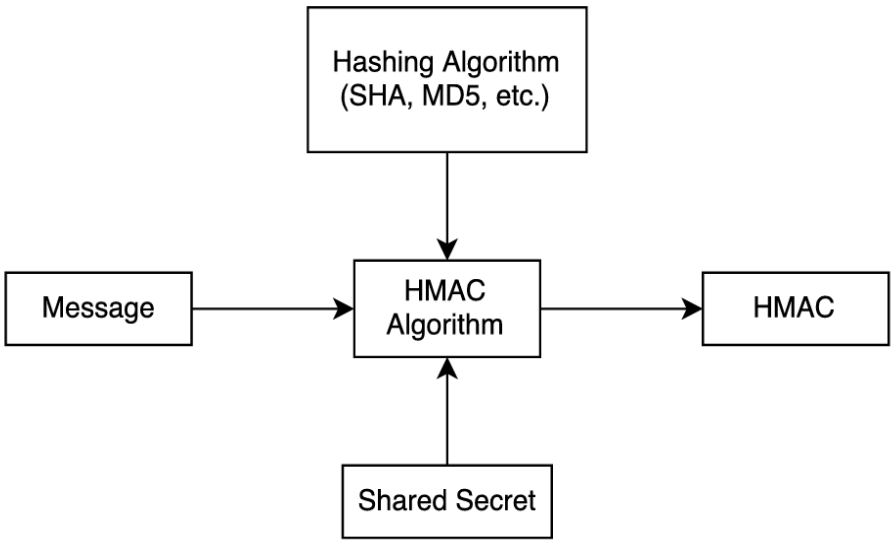

Figure 23 **HMAC for signing JWT** [17]

The following steps shows how HMAC-SHA256 works: [18]

- **Key Preparation**
  - If the key is longer than the block size of SHA-256 (64 bytes), it is first hashed using SHA-256.
  - If the key is shorter than the block size, it's padded with zeros to reach 64 bytes.
- **Inner Padding**
  - Create an inner pad by XOR the key with 0x36 repeated 64 times.
  - This gives $ipad = key \oplus 0x363636\ldots36$
- **Outer Padding**
  - Create an "outer pad" by XORing the key with 0x5C repeated 64 times.
  - This gives $opad = key \oplus 0x5C5C5C\ldots5C$
- **First Hashing (Inner)**
  - Concatenate the inner pad with the message: $ipad \,||\, message$.
  - Hash this combination with SHA-256: $inner\_hash = SHA-256(ipad \,||\, message)$.
- **Second Hashing (Outer)**
  - Concatenate the outer pad with the inner hash: $opad \,||\, inner\_hash$.
  - Hash this combination with SHA-256: $HMAC = SHA-256(opad \,||\, inner\_hash)$.

Mathematical Representation:

$$HMAC - SHA256(K, m) = SHA - 256((K \oplus opad) \,||\, SHA - 256((K \oplus ipad) \,||\, m))$$

Where:

- **K** is the secret key.
- **m** is the message.
- || denotes concatenation.
- ⊕ denotes XOR

Code snippet in *figure 24* of code section is responsible for getting user information of user accessing a web service API by passing JWT as authenticity method, so as obvious the first step done is checking integrity of this token by decoding it and verification of the signature. This way the server can make sure that this token was actually produced by an entity inside the system.

```python
async def get_current_user(token: Annotated[str, Depends(oauth2_bearer)]):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        if payload.get('type') != 'access':
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='Invalid token type')
        email: str = payload.get('sub')
        user_id: int = payload.get('id')
        role: str = payload.get('role')
        user_info: str = payload.get('user_info')  # Extract concatenated user info from the token payload
        if email is None or user_id is None or role is None or user_info is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='Could not validate user')
        return {'email': email, 'user_id': user_id, 'role': role, 'user_info': user_info}
    except JWTError as e:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail=f'Could not validate user: {str(e)}')
```

Figure 24 **Code snippet of server authenticating a user by checking user's access token**

4.4.7. Refresh Token

Access tokens may have a limited validity period for security reasons which is our case. Client apps can "refresh" the access token by using a refresh token after it expires. A refresh token is a credential token that enables a client program to obtain new access tokens without requiring the user to log in once more.

As long as the refresh token is still current and has not expired, the user application can obtain a new access token under certain validations according to the server configurations as clear in *figure 25*.

Consequently, a refresh token that has a very long lifespan could theoretically give infinite power to the token bearer to get a new access token to access protected resources anytime. An attacker or a genuine user could be the bearer of the refresh token, which requires validation done by server to produce a refresh token.

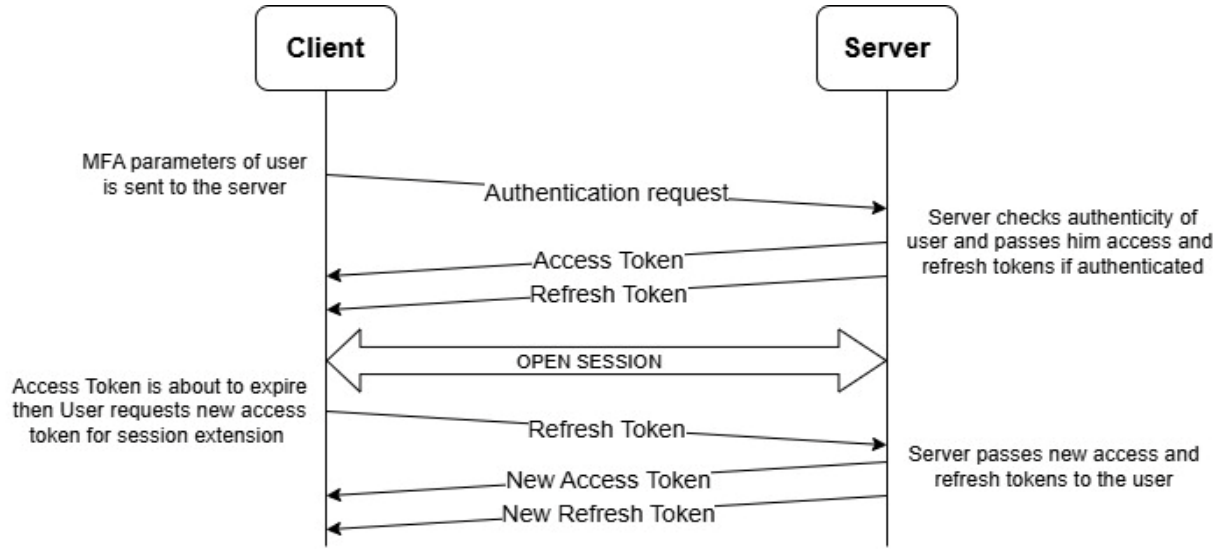

Figure 25 **Common Practice of JWT Sequence Diagram**

4.4.8. Zero Trust Contribution

What we will actually do is having the access token as relatively short-term as possible. Also, as mentioned before, it contains the user information, such as the user's geolocation, browser, and OS, at the time he logged in. Meanwhile, the refresh token will have a short expiry time but should be longer than that of the access token to prevent overloading the session management server. With every refresh token request the current user's geolocation, browser, and OS are taken with the request as well such that the server's responsibility is comparing the user information in the token needed to be refreshed with the current user info sent with the request and compare them if they are compatible the token is refreshed and a new token is returned to the user which will last for the refresh token expiry time set at the server, otherwise no refresh token is generated and this session is considered to be expired such that this user loses access as obvious in *figure 26*.

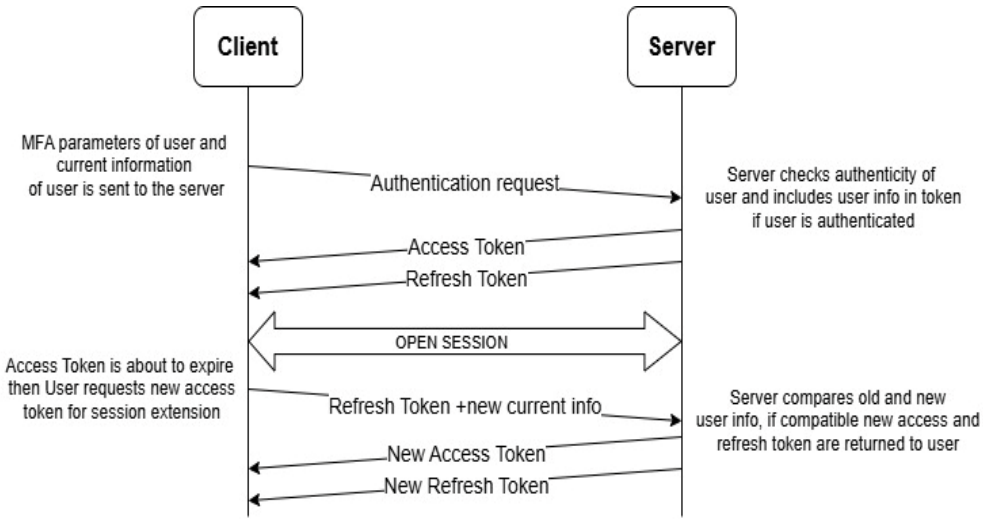

Figure 26 **System customized JWT Sequence Diagram**

The main challenge that should be considered is the need to compromise between having either long-life tokens and partially losing the accuracy of the user information and checking on it or short-life tokens which will overload the session management server and make it partially hard to manage all refresh requests made by all users but offers much more secure session and system practices.

The following *figure 27* illustrates how the server handles a refresh request:

Figure 27 **Server issuing Two Refresh Tokens during an Open Session**

4.4.9. Encryption Token's Sensitive Data in the Payload

Although we aim our token is stateless, we also need to continuously compare the user status accessing the token provided at login with his current status as a part of implementing zero trust principle. The only way to do this is encrypting the sensitive data stored in the token instead of the database to achieve statelessness.

The used encryption algorithm is AES-256(Advanced Encryption Standard with key size of 256) using cryptography library, which includes two core functions shown in *figure 28* and *figure 29*: "*encrypt_user_info*" and "*decrypt_user_info*", responsible for handling symmetric encryption / decryption of UTF-8 encoded strings and implementation of padding to ensure the plaintext aligns with AES 16-byte block size.

```python
# Encrypt a message using AES-256
def encrypt_user_info(key, plaintext):
    # ECB mode doesn't use an IV
    encryptor = Cipher(
        algorithms.AES(key),
        modes.ECB(),
        backend=default_backend()
    ).encryptor()

    # Pad the plaintext to be a multiple of 16 bytes (AES block size)
    pad_length = 16 - (len(plaintext) % 16)
    padded_plaintext = plaintext.encode('utf-8') + bytes([pad_length] * pad_length)

    # Encrypt the padded plaintext
    ciphertext = encryptor.update(padded_plaintext) + encryptor.finalize()
    return ciphertext
```

Figure 28 **Encryption function used for encrypting user information in Token Payload**

```python
# Decrypt a message using AES-256
def decrypt_user_info(key, ciphertext):
    decryptor = Cipher(
        algorithms.AES(key),
        modes.ECB(),
        backend=default_backend()
    ).decryptor()

    # Decrypt the ciphertext
    padded_plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    # Remove padding
    pad_length = padded_plaintext[-1]
    plaintext = padded_plaintext[:-pad_length]
    return plaintext.decode('utf-8')
```

Figure 29 **Decryption function used for decrypting user information in Token Payload**

The following shows a returned token to the user containing the user info encrypted as in *figure 30* and the token structure in *figure 31* illustrates the decoded token:

Token: doc.html:253

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkZW5pc2UudGVycnlAbWVkbWFpbC5jb20iLCJpZCI6Mjg3ODM4LCJyb2xlIjoiRG9jdG9yIiwidHlwZSI6ImFjY2VzcyIsImV4cCI6MTc0Mzk1MjIyMiwidXNlcl9pbmZvIjoiOEZSTDBXNmFmM3FQeXNWOW45OEFROUY3RUhOZkxWeE53VWN0b2hVZW5rUGQ5MDhtSG5FdGVpQnlNN0tLTm03YUxyWWNudHZyYTFFRUVSV0ZBTTZpNEFBaWZWZz0ifQ.EKLRKfJDX8SsDaDipRvoSTVyoEyhgjGoi5TSoZ_tyak

Figure 30 **Returned token after user information encryption**

Figure 31 **Decoded token after user information encryption**

The main difference here that the server decrypts the access token user information and compares it to the current user information sent with the refresh request such that it compares both entities and according to certain tolerance in the geolocation but no tolerance in browser or OS the server will decide either refreshing the token and sending the new token to user or session expiry. Server performance is shown in *figure 32*.

Figure 32 **Server-side performance after encryption of user information**

For scaling up this system and providing much more secure and less vulnerable to tampering user information data encryption, AES-GCM can be used for example which depends a uniquely used randomly generated initialization vector (IV) for every time encryption is done which makes it unique for every token produced by the system and the necessary modification to be done is including IV in the token payload to ensure the server is able to decrypt the user information. [17]

4.4.10. Token Revocation

Token revocation is primarily necessary when a session needs to be terminated due to the user's suspicious or malicious behavior during the session to ensure that this token is no longer valid, even though it has not yet expired.

A good practice for short-lived tokens is to store the revoked token in the Heap (RAM) to achieve fast response and statelessness and avoid including the database in this process, which is considered essential to follow the tokens approach. Periodically flushing the list containing the revoked tokens after their expiry time passes and becomes unusable.

4.4.11. Considerations about JWT

Generally, a denial-of-service (DoS) attack occurs when authorized users are unable to access information systems, devices, or other network resources due to the actions of a malicious cyber threat actor. [19]

In short-lived tokens, which is our case, a DoS attack may be possible due to the requirement for more frequent refreshes, and this can be a vulnerability targeting session management in the system unless handled cautiously. That is why the refreshing should be done at the right time, not too early to preserve the server capacity and not too late to avoid token expiry.

Consequently, an efficient refresh tokens rate limiter to prevent DoS attacks should be implemented, so the token should include the time it was issued such that the server compares the time a refresh token is requested and the time the refresh token is supposed to be requested based on the expiry time configured in server side. This process is shown clearly in *figure 33*.

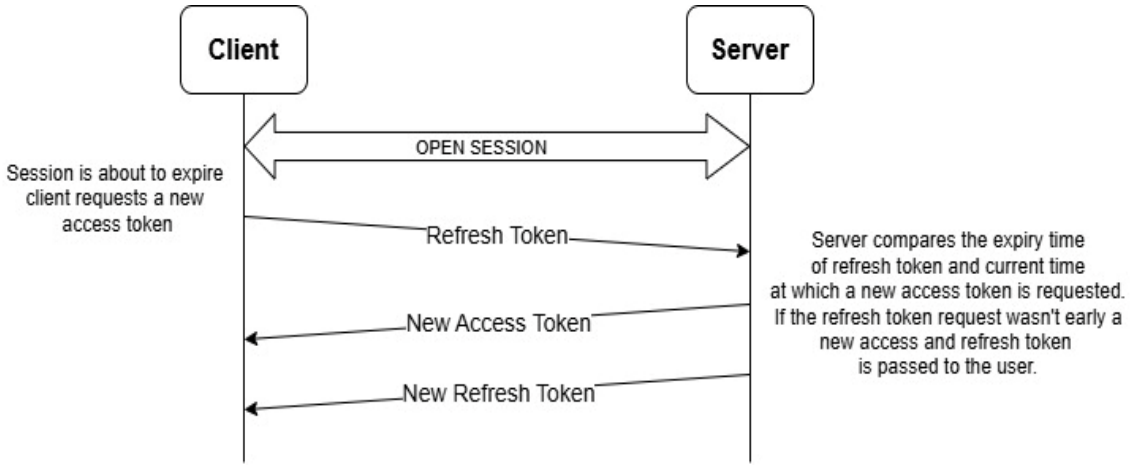

Figure 33 **System customized JWT Sequence Diagram with Refresh Rate Limiter**

The following *figure 34* illustrates attacker modified refresh token frequency in the frontend code which will cause server DoS due to not implementing the rate limiter at server side which caused refresh token overload.

```
const currentTime = Math.floor(Date.now() / 1000); // Current time in seconds
//const expiresIn = accessTokenExpires - currentTime; // Time remaining until expiration
const expiresIn = 0.25; // Time remaining until expiration
```

Figure 34 **Modified frontend code to overload server by Refresh Tokens**

As clear in *figure 35*, this is the code snippet of the necessary modification at server side to implement rate limiter on the issued refresh tokens which will offer efficient server capacity usage essential in short expiry time for tokens.

```
        # Time left before token expires
        current_time_utc = datetime.utcnow()
        current_time_egypt = current_time_utc + timedelta(hours=3)  # UTC+3 (Egypt Time)

        time_remaining =  -(current_time_egypt-datetime.fromtimestamp(payload.get('exp')) )
        # Check if more than (1/1.5 ≈ 60%) of the lifetime remains
        if (time_remaining > timedelta(minutes=REFRESH_TOKEN_EXPIRE_MINUTES / 1.5)) and (time_remaining >
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES / 1.5)):
            print("Refresh token overload")
            raise HTTPException(status_code=status.3HTTP_401_UNAUTHORIZED, detail='Refresh token
overload')
```

Figure 35 **Server Refresh Token Rate Limiter Code Snippet**

Figure 36 shows the result if server issues a refresh token request which has more than 60% (configurable according to server capabilities) of token lifetime remaining:

```
Decoded payload: {'sub': 'denise.terry@medmail.com', 'id': 287838, 'role': 'Doctor', 'type': 'refresh', 'exp': 1749190883, 'user_info': 'ry/CxLggvLKXkQ+yRmPGN3cFyPjP1dRRctoVl+HeD
RvqPgkuJ72mexXvTiNEU31I3gla7bEXEZB9MLNSGbksEg==''}
Refresh token overload
Unexpected error: 401: Refresh token overload
INFO:     127.0.0.1:54248 - "POST /auth/refresh HTTP/1.1" 400 Bad Request
```

Figure 36 **Server with Refresh Token Rate Limiter issuing an Early Refresh Request**

4.5. ELK SIEM Tool [20]

4.5.1. Why ELK Stack?

ELK is a Security Information and Event Management (SIEM) tool that consists of Elasticsearch, Kibana, and Logstash used for continuous monitoring of logs. It was selected as the monitoring and observability platform because it combines open-source flexibility with enterprise-grade features. All components of ELK are freely available under open-source licenses, eliminating per-GB or per-node licensing fees. This contrasts sharply with commercial SIEMs like Splunk or IBM QRadar whose charge is based on data volume or event rates. In practice, Splunk licenses can cost on the order of $1,800 per GB per year, and QRadar pricing is based on events-per-second (often tens of thousands of dollars annually). The ELK Stack avoids these costs by using commodity hardware and scaling horizontally.

Custom dashboards and observability are another strength of ELK. Kibana's visualization interface lets analysts build rich, interactive dashboards tailored to specific security metrics or compliance views. In practice this means that, for example, hospital administrators could create real-time dashboards showing authentication logs, application error rates, network intrusion alerts, or patient-data access patterns. Such customizability helps security teams quickly surface anomalies or policy violations relevant to a ZT context. Moreover, ELK includes built-in alerting: Kibana can raise real-time alerts on query conditions (e.g. an unusual login pattern) and integrate with common incident-response tools (PagerDuty, ServiceNow, email, Slack, etc.).

Community and industry support also favor ELK. A large open-source community continually contributes new plugins, dashboards, and integrations. This ecosystem means new log types and analytic features (even machine-learning anomaly detection) are rapidly incorporated.

4.5.2. ELK Stack Components

In our system as shown in *figure 37*:

- **Elasticsearch**
  It serves as the core of this architecture. It is a distributed search and analytics engine capable of indexing and querying large volumes of data in near real-time. Elasticsearch excels in handling both structured and unstructured data, and it offers powerful full-text search and aggregation features. These capabilities are crucial for detecting patterns and anomalies in user activity logs, making it a natural fit for security analytics applications.

Figure 37 **ELK Stack Main Components** [20]

- **Logstash**
  It plays the role of the data processing pipeline. It is responsible for receiving raw log data from various sources, in our case, from the FastAPI backend via HTTP POST requests.
  Once received, Logstash transforms and enriches this data into structured JSON documents. These are then forwarded to Elasticsearch for storage and indexing. Through this pipeline, logs generated by user activity (such as login attempts) are immediately captured, structured, and made searchable.

- **Kibana**
  It complements the stack by providing a user-friendly interface for data visualization and exploration. It connects directly to Elasticsearch and allows administrators to build dashboards, query logs, and monitor system health and security metrics in real-time. Kibana enables both developers and security teams to observe patterns in user behavior and to verify that the Policy Engine is functioning as expected.

Key Responsibilities:

- Generation & ingestion.
- Indexing & search.
- Real time enforcement.
- Visualization & reporting.

4.5.3. Component Integration in Our System

4.5.3.1. Log Generation and Ingestion

To demonstrate the interaction between these components, consider the detection and enforcement process for a common security rule: Three Failed Login Attempts as in *figure 40*. When a user repeatedly attempts to log in with incorrect credentials, the FastAPI backend generates a log entry. In *figure 38*, the entry includes information such as the user's email, ID, role and a status field indicating the nature of the request (e.g., "FailedLogin"). The log is sent as a POST request to the server running Logstash, using port 5044.

```
user_info1 = {
    "Email": form_data.username,
    "Role": user.Role,
    "User_id": user.User_ID,
    "Location": current_location,
    "OS": os,
    "Browser": browser,
    "status" : 'ThreeFailedLogins' }
requests.post(logstash_url, json=user_info1)
```

Figure 38 **Custom information sent with the log**

Logstash is configured to accept HTTP input on this port. Once received, the log is passed through the pipeline and forwarded to Elasticsearch. In the configuration file, Logstash defines the input plugin (HTTP in this case) and specifies Elasticsearch as the output as in *figure 39*. This ensures seamless integration and structured storage of logs.

```
input {
  # Input for FastAPI logs via HTTP POST
  http {
    host => "100.69.247.53"
    port => 5044
```

```
output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    user => "elastic"  # Replace with y
    index => "user-logins"
```

Figure 39 **Defining Logstash input and output**

4.5.3.2. Log Indexing and Search

Once the data is indexed in Elasticsearch, the policy engine begins its role. It issues queries directly to Elasticsearch using custom filters and time windows. For example, in *figure 40*, it can search for entries where the status field matches "FailedLogin" within the last three minutes. If such logs are found, the policy engine evaluates whether a threshold has been crossed and decides whether to trigger an enforcement action.

```
Policy_Engine.py > check_failed_logins
17    def check_failed_logins():
18        query = {
19            "size": 0,
20            "query": {
21                "bool": {
22                    "filter": [
23                        {"term": {"status.keyword": "FailedLogin"}},
24                        {
25                            "range": {
26                                "@timestamp": {
27                                    "gte": "now-3m/m",
28                                    "lte": "now"
```

Figure 40 **A Policy Engine Rule**

4.5.3.3. Real-Time Enforcement

When a violation is detected, the policy engine (PE) takes immediate action. For instance, if three failed login attempts are detected from the same user within a short time frame, the engine generates a POST request to the FastAPI backend, instructing it to ban the user for a predefined duration. In *figure 41*, the request includes information such as the email and the number of minutes the ban should remain in effect. This closed feedback loop ensures that policy enforcement occurs in real-time, which is a critical requirement of any ZT security model.

```
# Send alert to backend
backend_url = "http://100.89.45.27:8000/ban-user/"
alert_data = {
    "user_username": username,
    "ban_minutes": 15
}
try:
    response = requests.post(backend_url, json=alert_data)
```

Figure 41 **Action based on rule triggered**

This architecture allows the system to react proactively to threats, continuously verify user behavior, and maintain strong access controls. Moreover, with logs stored centrally and visualized through Kibana, system administrators gain full visibility into ongoing security operations. The combination of real-time detection, automated response, and intuitive monitoring makes this solution both effective and scalable for securing critical systems in environments like healthcare, where data protection is essential.

4.5.4. Diagram: System Architecture

The following figure, *figure 42*, shows the relation between the FastAPI Backend, ELK, and policy engine (PE).

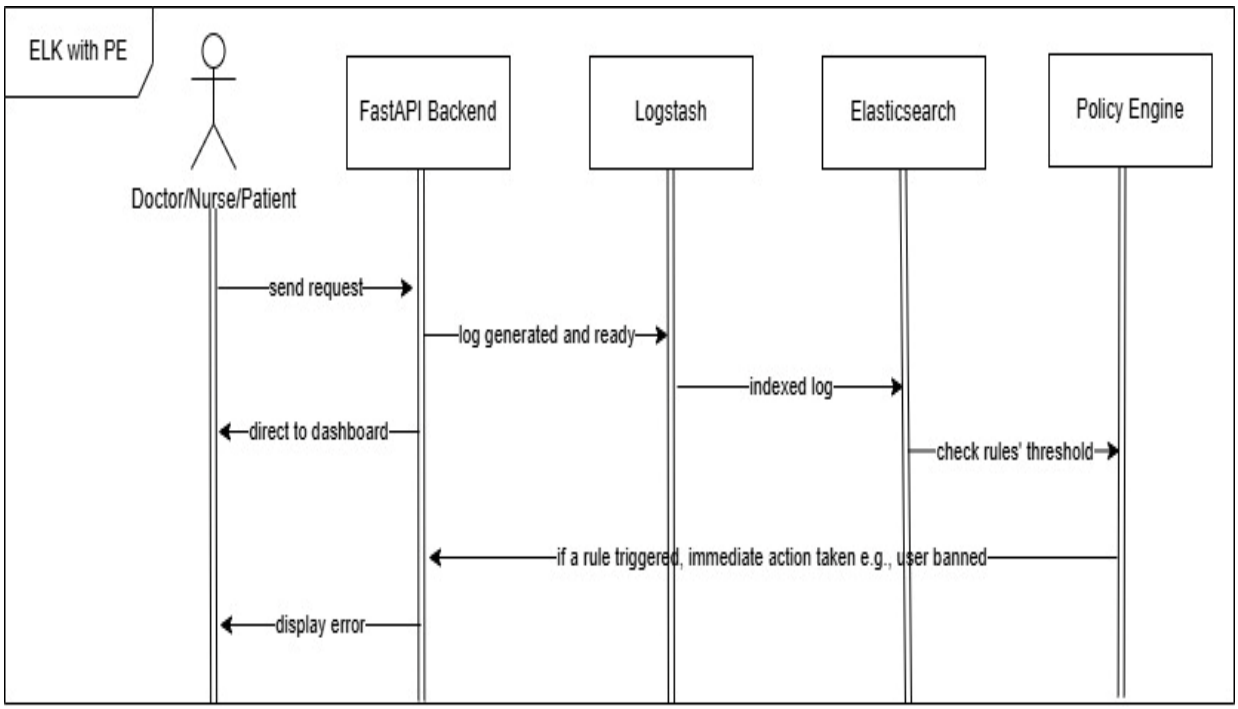

Figure 42 **ELK relation with Policy Engine (PE)**

4.6. Micro-segmentation and Tailscale

4.6.1. Micro-segmentation

Micro-segmentation is a security strategy that divides a network into smaller isolated segments to restrict unauthorized access , prevent lateral movement and denial-of-service (DoS) attacks .

In our project, Micro-segmentation plays a crucial role in enforcing zero-trust security principles and ensuring controlled communication between different components.

Micro-segmentation is applied to three servers, each represent a separate security segment as shown in *figure 43.*

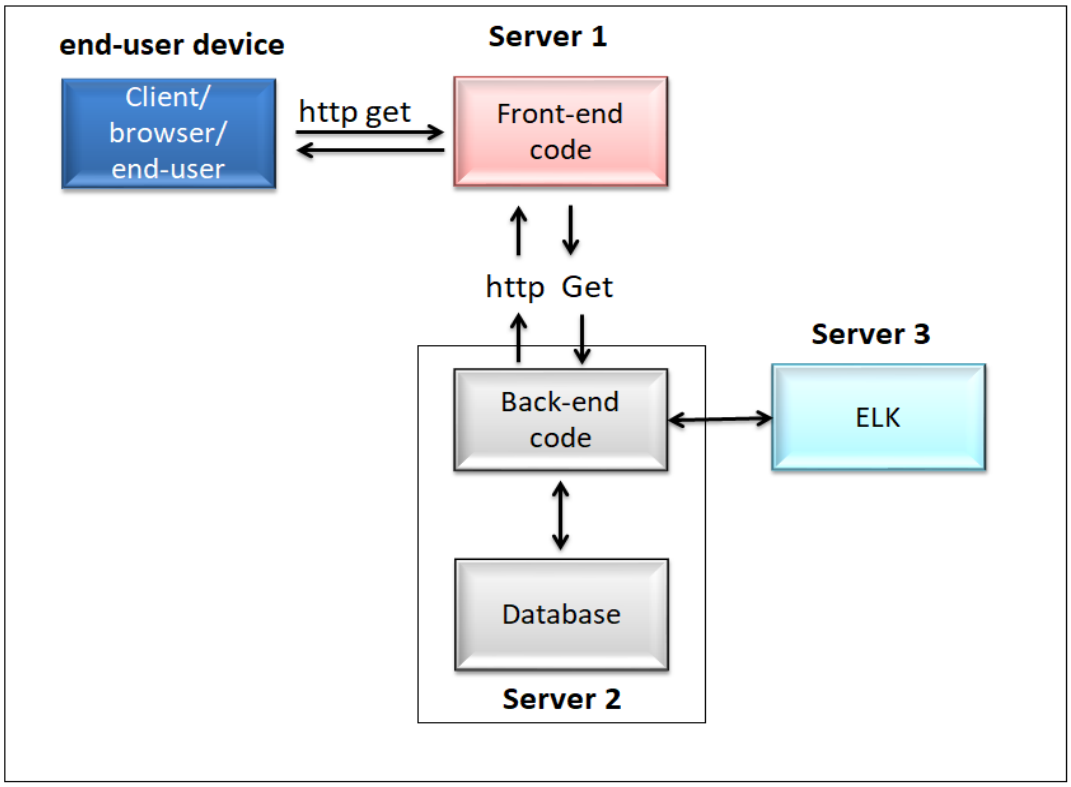

Figure 43 **Micro-segmentation Diagram**

Where:

- **Server 1:** contains the front-end code run on it.
- **Server 2:** contains the back-end code run on it and contain the database.
- **Server 3:** contains the policy engine and ELK server.

As we said our goal is to prevent Lateral Movement Attacks and Denial-of-Service (DoS) Attacks where they are :

- **Lateral Movement Attacks**
  The attackers gain access to one part of the system and attempt to move laterally to other components, compromising authentication, databases, or monitoring tools.
- **Denial-of-Service (DoS) Attacks**
  The attackers flood the system with excessive requests, disrupting authentication services and overloading network resources.

So, also the lateral movement allows attackers to spread deeper into a network, making DoS attacks more damaging.

The role of Micro-segmentation:

- Isolating critical services to ensure attackers cannot access authentication, database, or monitor components at the same time so when detecting an attack on some of these services or servers, it is immediately blocked before spreading and attacking the other devices or services.
- Limiting Lateral Movement so that attackers cannot jump between devices, reducing the impact of breaches.
- If a DoS attack happens on some servers, it is caught quickly without dropping the other servers.
- In case of a device that provides a service drops, it will be easily to fix it as other services are on other devices. So, it will be easier than fixing all services if they are on the same device.

4.6.2. Tailscale

Tailscale is a modern zero-configuration VPN and a peer-to-peer mesh VPN that simplifies secure networking by creating encrypted connections between servers using the WireGuard protocol. Unlike traditional VPNs that rely on a central gateway, Tailscale forms a tailnet, where servers communicate directly, reducing latency and improving security where [21]:

- **Tailnet**
  It is a private network created using Tailscale, allowing servers to securely communicate over the internet without exposing them to public threats.
- **Zero-Configuration Networking**
  This means no manual IP assignments or firewall rules needed.
- **End-to-End Encryption**
  It uses WireGuard for fast, secure connections.

- **Peer-to-Peer (P2P) Connections**
  This means that servers communicate directly when possible, reducing latency not rely on a central gateway.
- **WireGuard Protocol**
  It uses ChaCha20 which is a symmetric encryption performs 20 rounds of encryption, supports a 256-bit key (32 bytes), making it resistant to brute-force attacks.
- **DERP (Detour Encrypted Routing for Packets)**
  Tailscale fallback mechanism for encrypted packet relay when direct peer-to-peer (P2P) connections fail as Tailscale operates DERP servers worldwide to ensure connectivity. So, at first servers try UDP hole-punching for direct P2P communication and if direct connection fails, traffic is relayed through DERP.

4.6.2.1. Implementation

As we said in above when we apply the micro-segmentation we make:

- Server 1 will be the frontend and its name in the network is engy2002.
- Server 2 will be the backend and its name in the network is user.
- Server 3 will be the ELK with policy engine and its name in the network is mak.

We will define some Tailscale terminologies:

- **Shared out:** It means that the server can receive requests from the other servers it invites them in the Tailscale network.
- **Shared in:** It means that the server can send requests to the other servers which it was invited by them in the Tailscale network.

The steps of Tailscale Implementation:

- We need first to install Tailcale client on each server that needs to join the Tailscale network.
- After installing Tailscale client on each server, server 2 will invite server 1 and server 3 so it can receive only requests from both servers.
  On the machines section on the server 2, the Tailscale network will appear under the user server as "shared out".
- Then also server 1 and server 2 must invite server 3 so that server 3 can send requests to both servers (Two-way communication between both server 2 and server 1, server 2 and server 3).
  On the machines section on server 2, the Tailscale network will appear under the server 1 and server 3 as "shared in".

On server 2, the Tailscale client looks like this connecting to both server 1 and sever 3 taking the IP of 100.89.45.27 as shown in *figure 44*.

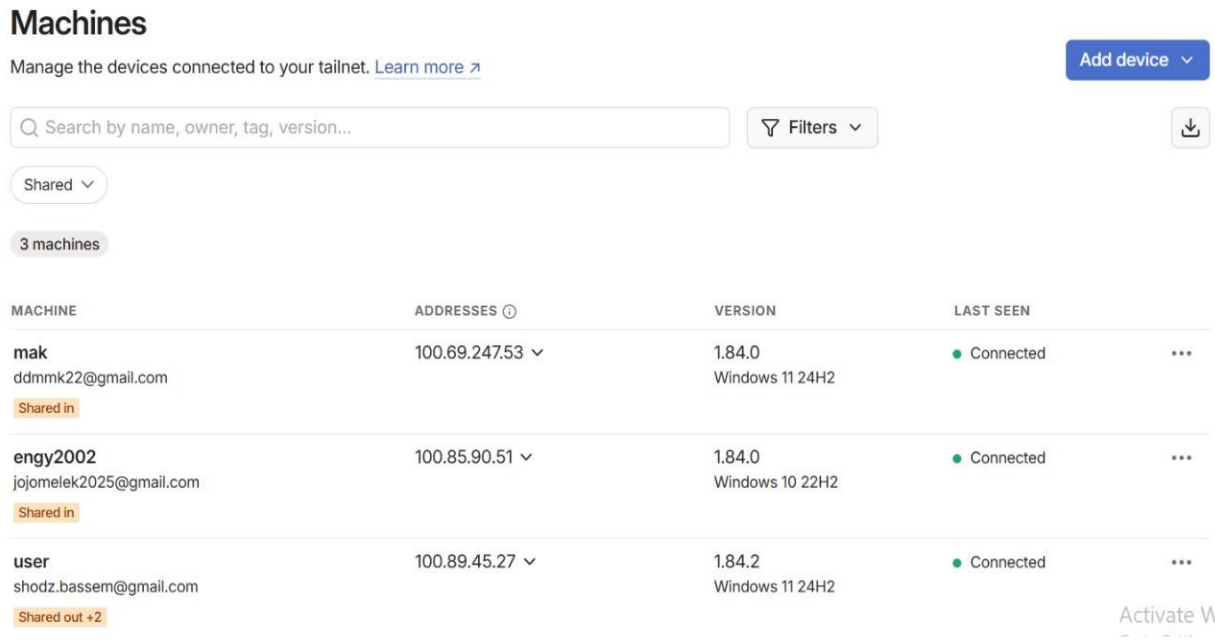

Figure 44 **Machine section on Server 2**

- Now we only want server 1 to receive requests only from server 2 not to connect to server 3 as we want to apply the micro-segmentation. Also, server 3 will invite only server 2 to receive requests only from it not from server 1.

On server 1, the Tailscale client looks like this connecting only to server 2 taking the IP of 100.85.0.51 as shown in *figure 45*.

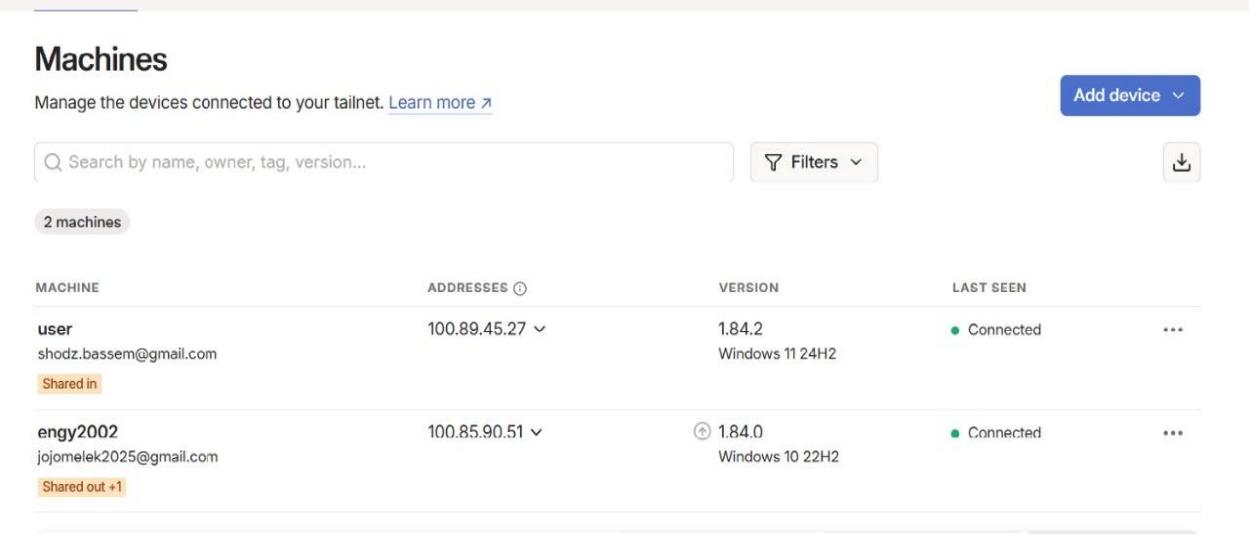

Figure 45 **Machine section on Server 1**

On server 3, the Tailscale client looks like this connecting only to server 2 taking the IP of 100.69.247.53 as shown in *figure 46*.

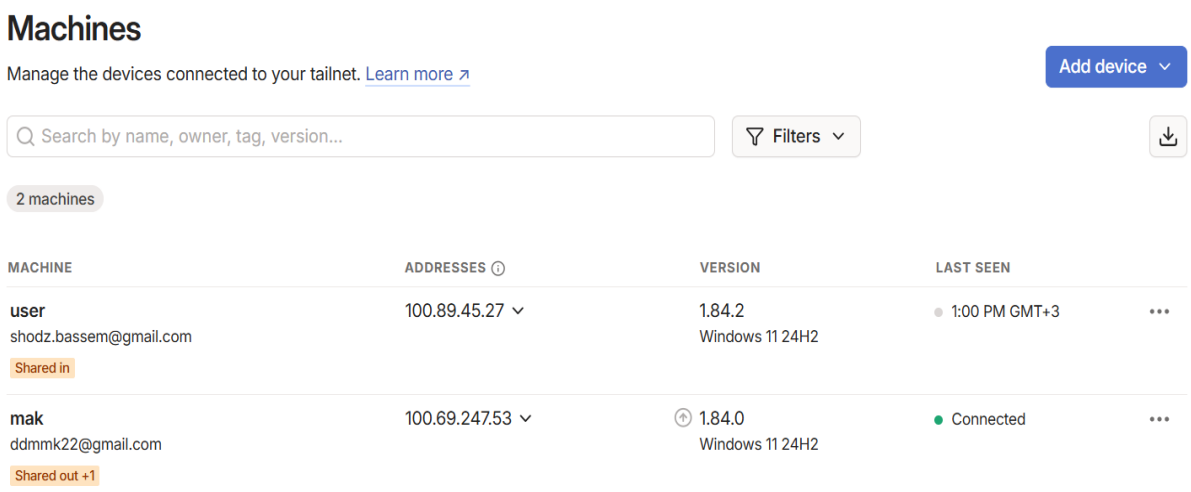

Figure 46 **Machine section on Server 3 and Policy Engine Server**

4.6.2.2. Tailscale Operational Scenario Diagram from Backend to ELK

Figure 47 shows the Tailscale operational scenario from server 2 to server 3.

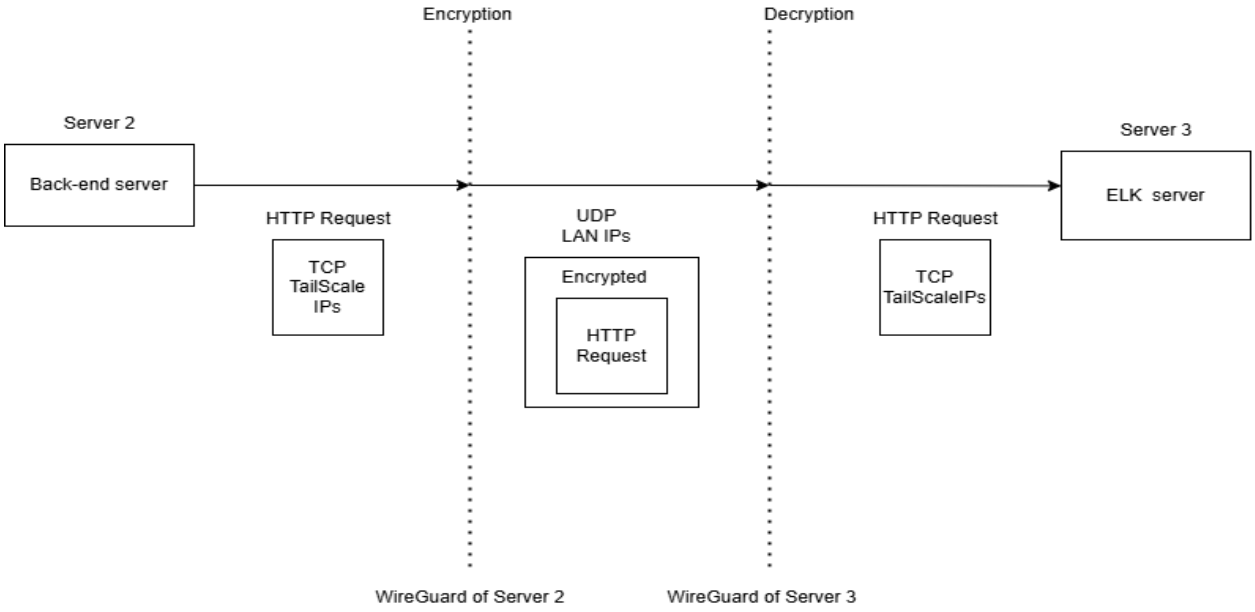

Figure 47 **Operational Scenario Diagram from Server 2 to Server 3 through Tailscale**

4.6.2.3. The Operational Scenario from Frontend to Backend

So now the three servers are joined together in the Tailscale network then when server 1 sends a request to server 2, the following sequence will happen:

- Server 2 will run on port 8000 and its Tailscale IP (100.89.45.27) as shown in *figure 48*.

```
PS D:\Graduation Project\pyth> uvicorn main:app --host 100.89.45.27 --reload
INFO:     Will watch for changes in these directories: ['D:\\Graduation Project\\pyth']
INFO:     Uvicorn running on http://100.89.45.27:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [1980] using StatReload
INFO:     Started server process [21260]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Figure 48 **Command to run Server 2**

Using this command to run server 2:

- To allow the servers only send requests on the Tailscale IP of server 2 to allow encryption of WireGuard using Tailscale and not allow requests on the LAN connection.
- Server 2 code should be running on its Tailscale IP to prevent any external connection form the client to access the server rather than being on the tailnet IP so the stream will be end-to-end encrypted.

- Server 2 code runs on 100.89.45.27 not 127.0.0.1 or 0.0.0.0 to enforce the connection through the Tailscale IP and any other try to access the server will be unreachable.

- The frontend server will run on port 5000 as shown in *figure 49*.

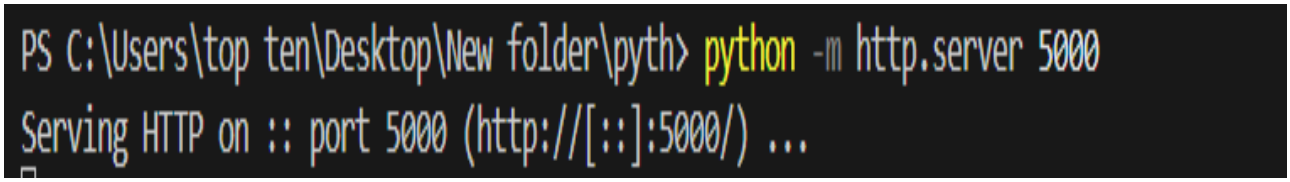

Figure 49 **Command to run Frontend Server (Server 1)**

- Server 1 will send request to server 2:
  - Server 1 initiates an HTTP request directed to server 2.
  - The request targets server 2 Tailscale virtual IP (100.89.45.27).
  - The Windows network stack prepares a TCP packet, ensuring it is destined for the correct virtual interface linked to Tailscale.

  IP details at this stage:
  - **Source IP:** 100.85.90.51 (Server 1 Tailscale IP).
  - **Destination IP:** 100.89.45.27 (Server 2 Tailscale IP).
- Tailscale client will intercept and encrypt the request:
  - The Tailscale windows service detects traffic meant for 100.85.90.51 using its virtual network adapter as it captures packets before they hit the physical network interface.
  - Instead of sending HTTP traffic across the network, it encapsulates the request in a WireGuard UDP packet for encryption.

  The UDP packet contains:
  - **Source IP:** Server 1 public/LAN IP (192.168.1.136).
  - **Destination IP:**

    Case (1): If a direct Peer-to-Peer Connection is available:
    - The destination IP in the encrypted UDP packet will be the real public or LAN IP of server 2.
    - This allows direct transmission between peers without needing relay servers.
    - **Destination IP:** Server 2 LAN IP (192.168.1.4).

    Case (2): If NAT/Firewall blocks direct connection, DERP relay is used:
    - The destination IP will be the public IP of the nearest Tailscale DERP server.
    - The DERP server relays the encrypted packet to server 2 Tailscale IP (100.89.45.27), ensuring connectivity.

- **Destination Port:** 41641 (WireGuard standard port in Tailscale).
- **Payload:** The encrypted HTTP request using ChaCha20 encryption.

Encryption processes at this stage where the contents of the HTTP request are fully encrypted using WireGuard pre-established keys, ensuring secure transmission, WireGuard transmits the encrypted packet depending on two cases discussed above.

If we captured the packet on LAN, it will appear that the protocol is UDP and the payload is encrypted as we will show clearly later, so this proves that the traffic is encrypted because we cannot see real headers or data.

- Server 2 receives and decrypts:
  - The Tailscale client on server 2 intercepts the incoming UDP packet.
  - Using WireGuard private keys, it decrypts the binary payload, recovering the original http request.
  - The decrypted request is then injected into server 2 network stack via the virtual adapter.
  - From Windows perspective, server 2 receives a normal HTTP request on its virtual NIC (Network Interface Card).

  Even though DERP relayed the packet, the decrypted request, the IP details after decryption and before sending to server 2 stack will be:
  - **Source IP:** 100.85.90.51 (Server 1 Tailscale IP).
  - **Destination IP:** 100.69.247.53 (Server 2 Tailscale IP).

- Server 2 processes the HTTP request:
  - Server 2 processes the HTTP request as usual, generating a response (200 OK).
  - The response is then prepared for transmission back to server 1.

- Response of server 2:
  - It will be a symmetric flow. Its Tailscale client encrypts the reply using WireGuard and sends it as UDP to server 1.
  - As shown the three servers we have: secure-patience (connected to MySQL database), OTP servers with 2 separate microservices and backend server.

4.6.2.4. Captured Packets using Wireshark from Frontend to Backend

We then used Wireshark and tried to capture packets flow in two cases to ensure data is encrypted while transmission and only servers in the Tailscale network can decrypt data.

- **Case (1): Capturing on Tailscale0 Interface (Decrypted)**
  If you run Wireshark on server 2 and capture on the Tailscale0 interface, as shown in *figure 50*:

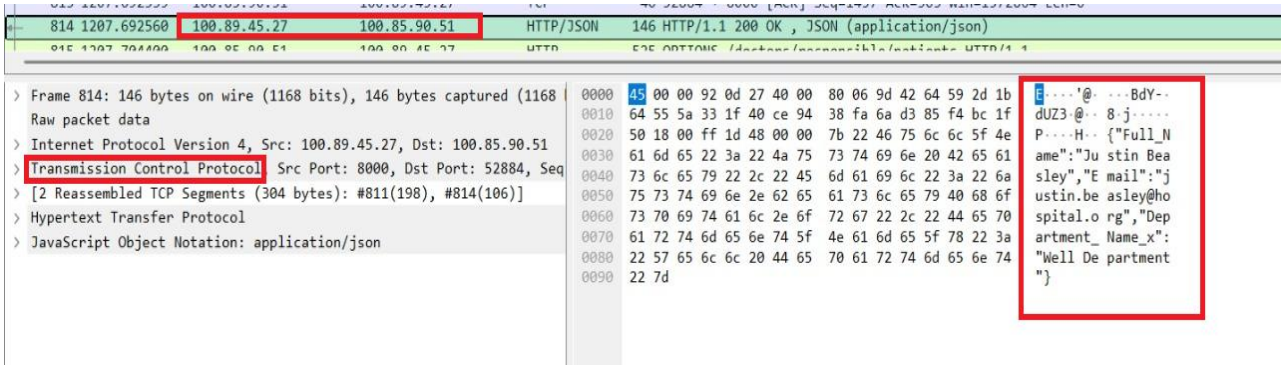

Figure 50 **Wireshark on Tailscale Network on Server 2**

The user info is cleared plaintext as seen not encrypted as the server 2 and server 1 are connected to each other inside the Tailscale network can decrypt data using WireGuard. The protocol is TCP and using the Tailscale IPs.

- **Case (2): Capturing on eth0 or wlan0 (Encrypted)**
  If you run Wireshark on the physical interface (e.g., eth0, wlan0), as shown in *figure 51*:

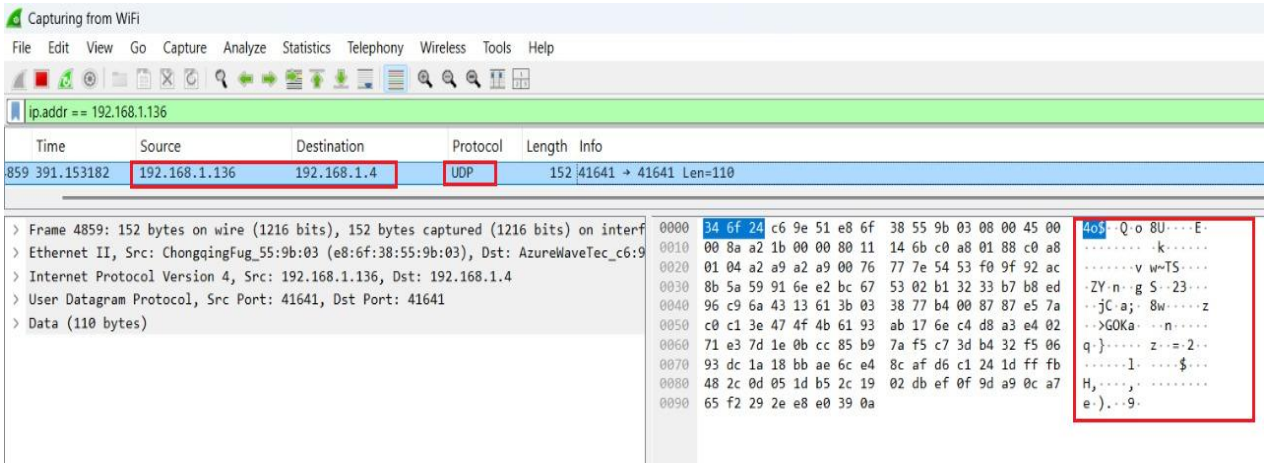

Figure 51 **Wireshark on LAN interface network of Server 2**

As we see here the data is encrypted over LAN or Wi-fi network. The protocol used is UDP due to WireGuard encryption and using the Public/LAN IPs.
The following *Table 3* represents a summary of Wireshark differences between the two cases above:

Table 3 Difference between Tailscale0 interface and eth0/wlan0 interface

Interface	Encryption	What You See	IP Addresses
Tailscale0	Decrypted	Full traffic (e.g. HTTP, SSH, etc.)	Tailscale IPs
eth0/wlan0	Encrypted	Encrypted UDP packets (WireGuard)	Public/Private IPs

4.6.2.5. The Operational Scenario from Backend to ELK

First, we need the Logstash on the ELK server to receive logs on the Tailscale IP of server 3, so the input in the Logstash configuration file will be like as shown in *figure 52*.

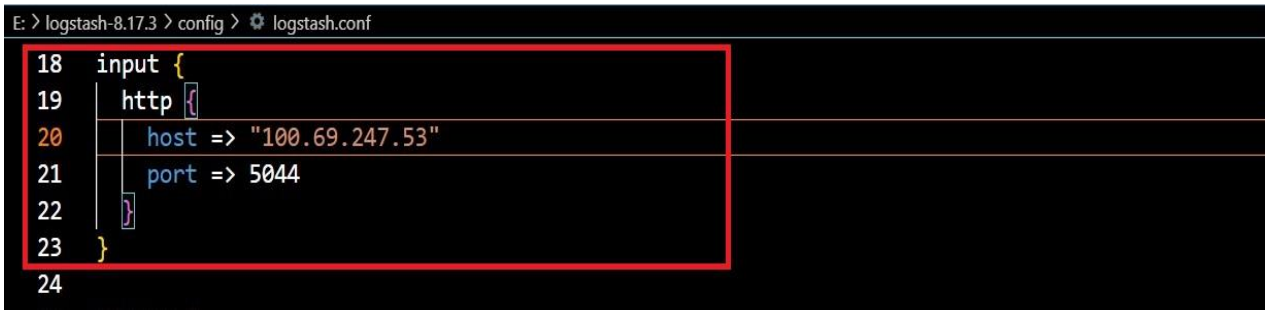

***Figure 52* Configuration file of Logstash**

In the Logstash input, it listens to the logs coming from server 2 on port 5044 and on the Tailscale IP (100.69.247.53) of server 3.

- Server 2 initiates log transmission:
  - Server 2 generates log entries and forwards them to server 3.

    Packet details at this stage:
  - **Source IP:** 100.89.45.27 (Server 2 Tailscale IP).
  - **Destination IP:** 100.69.247.53 (Server 3 Tailscale IP).
  - **Destination Port:** 5044 (Logstash listening port).

  This ensures requests go through Tailscale, avoiding exposure to the internet.

- Tailscale client intercepts and encrypts the request:
  - The Tailscale client running on server 2 detects outgoing traffic meant for (100.69.247.53).
  - The request is encrypted using WireGuard, encapsulating it within a UDP packet.

The UDP packet contains:

- **Source IP:** Server 2 public/LAN IP (192.168.1.4).
- **Destination IP:**

  Case (1): If a direct Peer-to-Peer connection is available:
  - The destination IP in the encrypted UDP packet will be the real public or LAN IP of server 3 (192.168.1.137).
  - This allows direct transmission between peers without needing relay servers.
  - **Destination IP:** Server 3 LAN IP (192.168.1.137).

  Case (2): If NAT/Firewall blocks direct connection, DERP relay is used:
  - The destination IP will be the public IP of the nearest Tailscale DERP server.
  - The DERP server relays the encrypted packet to server 3 Tailscale IP (100.69.247.53), ensuring connectivity.

Now if you capture the packet on LAN, it will appear that the protocol is UDP and the payload is encrypted as we will show clearly later.

- Server 3 receives and decrypts logs:
  - Server 3 Tailscale client intercepts the incoming encrypted UDP packet. Using pre-established WireGuard keys, Tailscale decrypts the payload.
  - The decrypted request is injected into server 3 network stack.

Even though DERP relayed the packet, the decrypted request, the IP details after decryption and before sending to server 2 will be:

- **Source IP:** 100.89.45.27 (Server 2 Tailscale IP).
- **Destination IP:** 100.69.247.53 (Server 3 Tailscale IP).

- Logstash processes and stores logs:
  - Now Logstash, running on 100.69.247.53: 5044, receives the logs as a standard HTTP/TCP request.
  - Logstash processes the logs so they are parsed, indexed, and forwarded to Elasticsearch for storage.
  - Server 3 dashboards (e.g., Kibana) visualize logs for monitoring.

As said in the ELK part, server 3 will run the policy engine file to apply the polices searching in the Elasticsearch. If trigger happens, it will send an alert to a function on server 2 and the same symmetric scenario will happen using Tailscale.

4.6.2.6. Captured Packets using Wireshark from Backend to ELK

We then used Wireshark and try to capture packets flow between server 3 and server 2 in two cases to ensure data is encrypted while transmission and only servers in the Tailscale network can decrypt data.

- **Case (1): Capturing on Tailscale0 Interface (Decrypted)**
  If you run Wireshark on server 3 and capture on the Tailscale0 interface, as shown in *figure 53*.

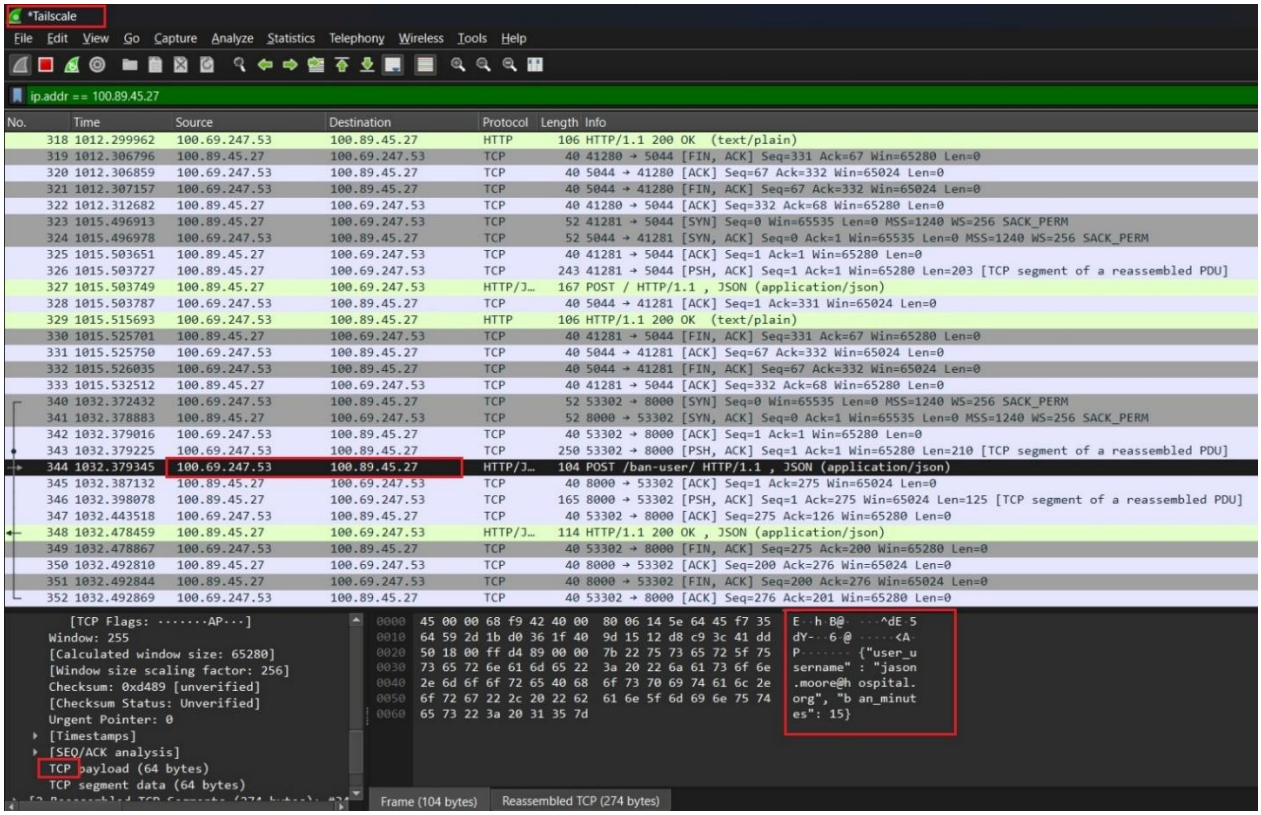

Figure 53 **Wireshark on Tailscale network of Server 3**

The data is a cleared plaintext as seen not encrypted as the server 2 and server 3 are connected to each other inside the Tailscale network can decrypt data using WireGuard. The protocol is TCP and using the Tailscale IPs.

- **Case (2): Capturing on eth0 or wlan0 (Encrypted)**
  If you run Wireshark on the physical interface (e.g., eth0, wlan0), as shown in *figure 54*.

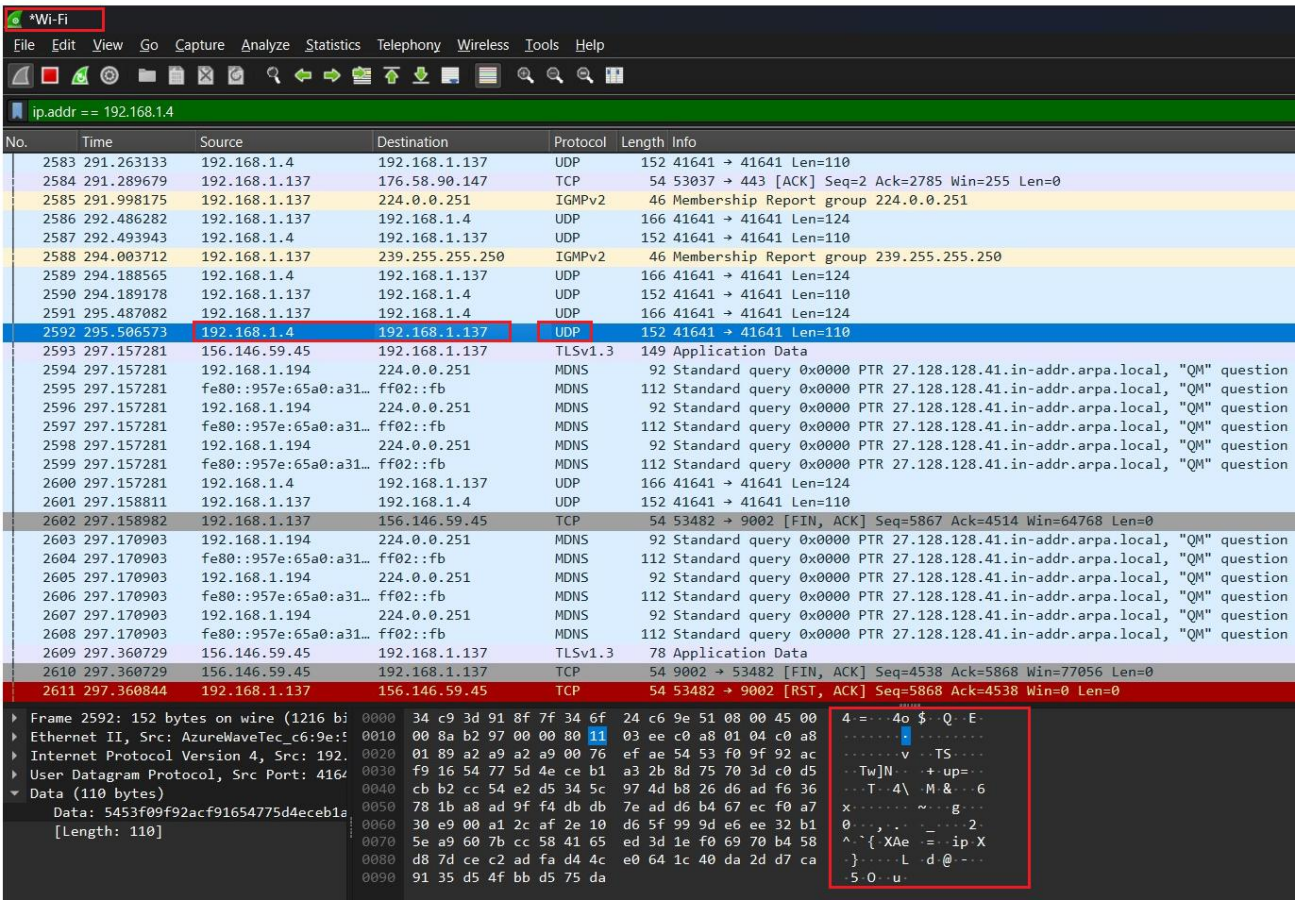

Figure 54 **Wireshark on LAN interface network of Server 3**

As we see here the data is encrypted over LAN or Wifi network. The protocol used is UDP due to WireGuard encryption and using the Public/LAN IPs.

CHAPTER FIVE: OPERATIONAL SCENARIOS

In this section, we will represent five different scenarios; four scenarios will be experienced by the user while interfacing on the website which are:

- Signup Scenario.
- Forget Password Scenario.
- Login Scenario.
- Add / Edit / Delete Operations Scenario.

and the fifth one is JWT scenario which is a backend scenario of how the session is controlled by the JWT as an access token.

5.1. Signup Process Scenario

The following diagram in *figure 55* shows the scenario of the process of creating a non-existing user's account.

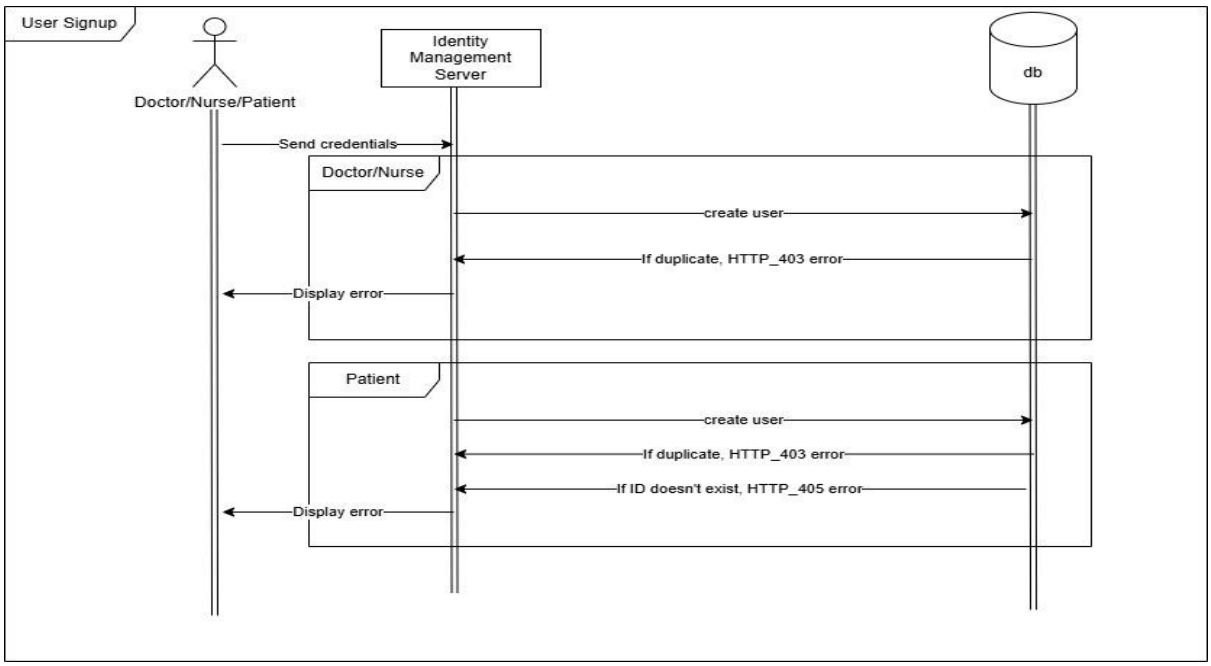

Figure 55 **Scenario of signup process**

Steps of Signup process:

- When the user clicks on the "Sign Up" button, he is prompted to enter his username, email, password, and choose role (Doctor – Nurse – Patient).
- In case the user is a patient, the user is prompted to enter his userID, too.
- Once the user submits the form, the system immediately checks if this user has already an account.

- For a higher level of security, in case the user is a patient, the identity management server also checks that a doctor has added this patient to his record before.
- After checking that no duplicates in the database exist, the server creates a new user in the database of the systems.
- By then, this user has the credentials to login to the website any time.

5.2. Forget Password Scenario

It is possible that a user can forget his password and still wants to login so he can change his password. The following diagram in *figure 56* shows this process.

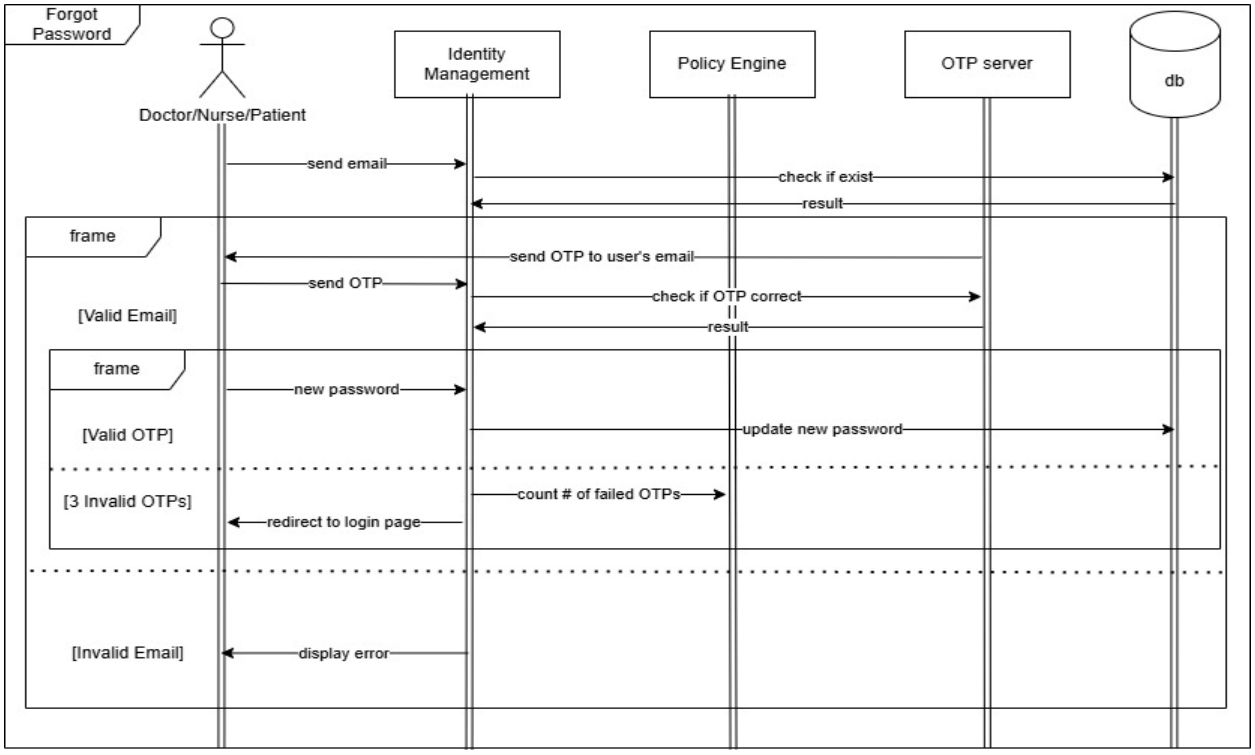

Figure 56 **Scenario of forgetting the password**

Steps of forgetting password:

- Once the user clicks on "forget password" button, a request is sent to the identity management (IM) server.
- The user is directed to a new page and asked to enter his registered email address.
- When the user clicks on "Send OTP" button:
  - The IM server checks that this email exists in the database.
  - In case the email is valid, the OTP server sends an OTP to the user.
  - In case the email is not valid, an HTTP error is triggered.
- The user enters the received OTP and the OTP server checks it.

Case (1):

- If the OTP is valid, the user is prompted to enter his new password twice for confirmation. The IM server updates the user password in the database and redirects the user to the login page where the user can login again with the new password.

Case (2):

- If the entered OTP is not correct, a counter of the failed attempts is initiated and sent to the policy engine. Every time the user enters an invalid OTP, the counter is incremented till it reaches three failed attempts.
- In case of three failed attempts of entering the correct OTP, the IM server redirects the user to the login page.

5.3. Login Process Scenario

The following diagram in *figure 57* shows the process of logging in with an existing user's account.

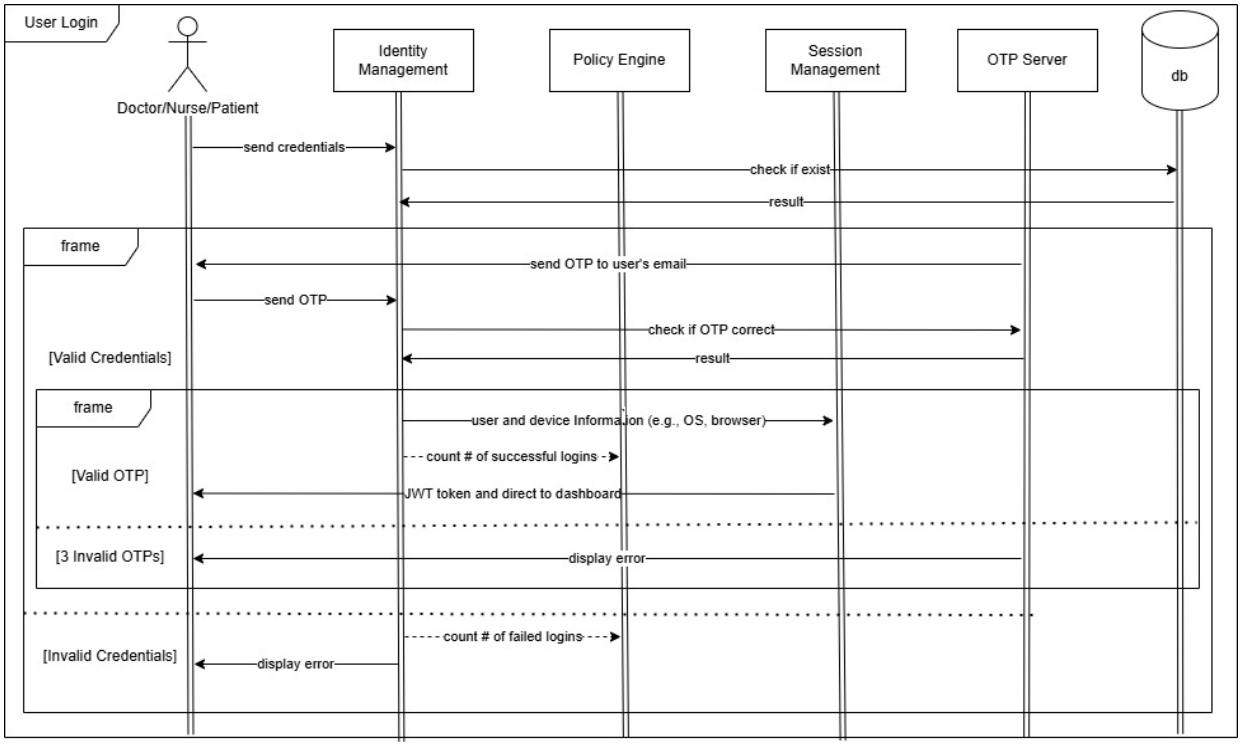

Figure 57 **Scenario of login process**

Steps of Login process:

- When the user clicks on the "Login" button, he is prompted to enter his email and password.
- Once the user clicks on the “Send OTP” button, the IM server user's credentials are checked against the database of the system.

- If the credentials are verified, the system retrieves the user's role (a doctor, a nurse, or a patient) and stores both the email address and the role in a local storage to support subsequent role-based access control and the OTP server sends an OTP to the user to be verified. By then, the first authentication factor is applied successfully.
- In case the password is entered wrong for three times, the policy engine bans the users.
- For verified credentials, the user enters the received OTP and the OTP server checks it.

Case (1):

- If the OTP is valid, the access is granted to the user and a JWT access token is given to the user where he is directed to his dashboard based on his role and the session begins.

Case (2):

- If the entered OTP is not correct, a counter of the failed attempts is initiated. Every time the user enters an invalid OTP, the counter is incremented till it reaches three failed attempts.
- In case of three failed attempts of entering the correct OTP, the user is banned.

5.4. Add / Edit Operations Scenario

For applying role-based access control model (RBAC), each user has a specific set of rights based on his role. So, these operations can only be done by a doctor while the other roles (nurse - patient) have no rights to add, edit, or delete a patient record. The following diagram in *figure 58* shows these operations.

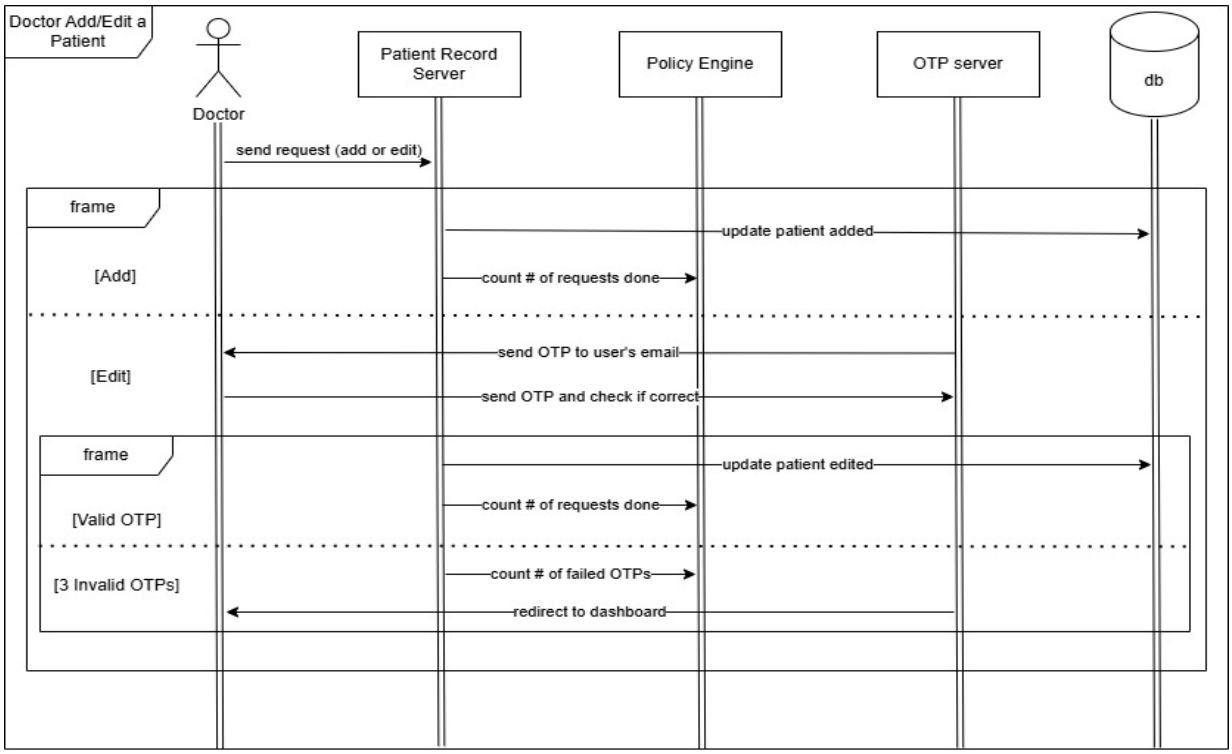

Figure 58 **Scenario of adding / editing a patient record**

Steps of adding / editing a patient record:

- The doctor sends an add / edit / delete request to the patient record server.
- In case the request is add a patient, the doctor fills the form of a new patient and submits it, so the patient record server updates the database with the new patient.
- In case the request is edit an existing patient record, the user is prompted to verify a received OTP.
- The doctor enters the received OTP and the OTP server checks it.

Case (1):

- If the OTP is valid, the doctor is directed to update the patient record. The patient record server updates the database with the updated data of the patient.

Case (2):

- If the entered OTP is not correct, a counter of the failed attempts is initiated. Every time the user enters an invalid OTP, the counter is incremented till it reaches three failed attempts.
- In case of three failed attempts of entering the correct OTP, the doctor is redirected to his dashboard without editing any data.

5.5. JWT Scenario

The following diagram in *figure 59* shows all operational scenarios of a session controlled by JWT as an access token that identifies the user and validates him for the server.

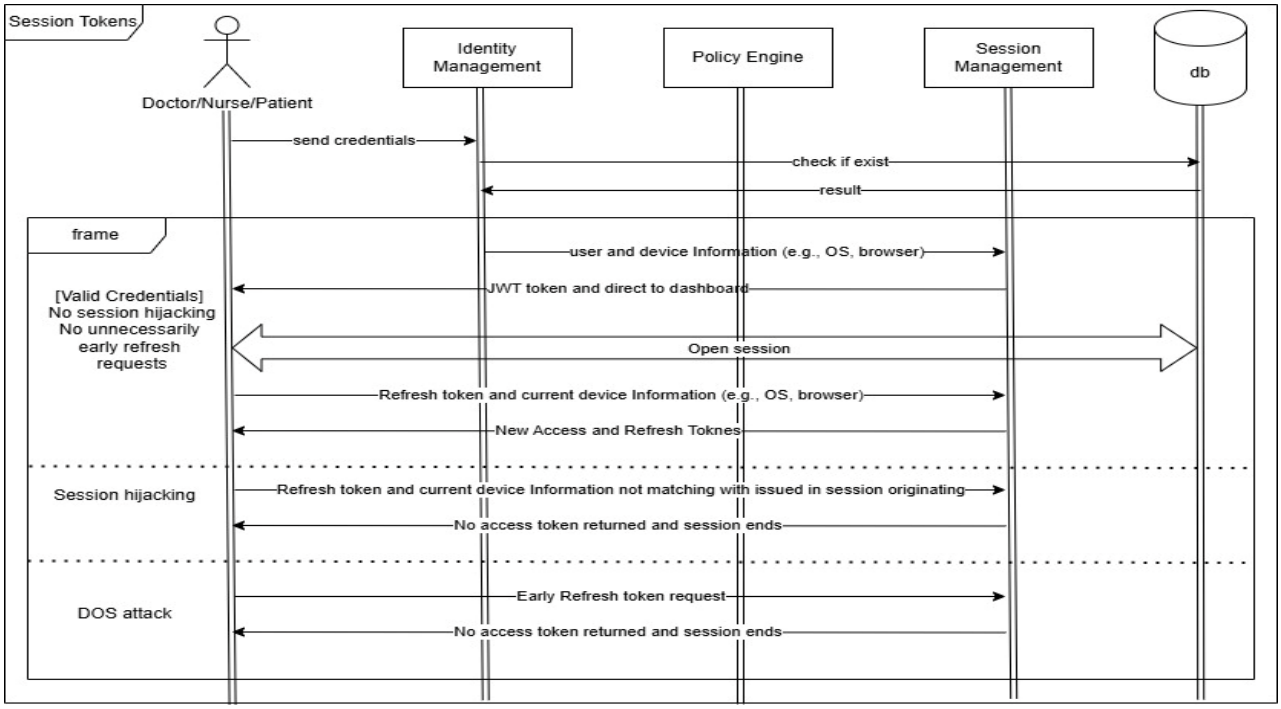

Figure 59 **Scenario of JWT**

Steps of session controlled by JWT :

After the OTP verification, the IM server sends the user and device information to the session management server (e.g., OS, Browser) where the session management server gives the user the JWT token and directs him to his dashboard based on his role. Let us discuss the different operational scenarios of a session controlled by JWT.

- **Normal Session Operation:**
  For an open session, if the expiry of token is soon, the user side will request refreshing his access token to extend the session. In order to continue the session, the user sends to the server his new current information and the refresh token. If the old and the new user information are compatible the refreshment is done successfully.

- **Session Hijacking:**
  For a changed and illogical new user information sent along with refresh request, the server refuses to refresh this user's access token and thus access token expires and session ends.

- **DoS attack on session management server:**
  For unnecessarily early refresh request for the access token, the server considers it a DoS attack and denies this refreshment request due to refresh rate limiter.

CHAPTER SIX: CONCLUSION AND FUTURE WORK

6.1. Conclusion

In the process of developing this graduation project, we aimed to develop a secure, scalable, and practical system using the modern security principle Zero Trust Architecture (ZTA) model. The main goal was to protect sensitive patient data while ensuring simplicity of access for authorized users such as doctors, nurses, and patients. With the rise of cyberattacks, especially in the healthcare sector, it was clear that the traditional perimeter-based security frameworks no longer functioned. This prompted us to adopt a newer, identity-based model like ZTA with the "Never Trust, Always Verify" principle. This project wasn't just a technical project as taught us how to think like cybersecurity professionals. We had to consider real-world threats, user actions, ethical considerations, and compliance with regulations such as GDPR and HIPAA. The developed system is a real-world example of the implementation of Zero Trust Architecture in healthcare. It shows how security can be integrated into every layer of a web platform, from authentication of the user through encryption and monitoring of data, without a loss of function or user experience. While there is always more to do, especially as threats evolve, we believe this project lays a good foundation for building more secure, intelligent, and user-centric applications in the years ahead.

6.2. Future Work

6.2.1. Backend Full Micro-segmentation

Each module should be fully isolated in terms of codebase and then these modules are expected to be running on different VMs or cloud-based containers. These modules will communicate through Tailscale network to ensure encryption through data transition between these servers. This setup promotes ZTA and improves fault and achieves both scalability and micro-segmentation.

6.2.2. Hosting Integration along with ELK

Clouding solution as: AWS, Azure, or GCP, can be used to host the backend server. The previously set up locally Elastic Stack (ELK – Elasticsearch, Logstash, and Kibana), will be reconfigured in the cloud environment. Thus, hosting the website will include the ELK and policy enforcement ensuring real-time monitoring and policy enforcement.

6.2.3. AI-trained Policy Engine

An artificial intelligence should analyze historical access and activity logs to detect anomalies, flag suspicious behaviors, and dynamically adapt access policies to train the engine on the usual behavioral patterns of users. This will enhance the security and provide a more intelligent automated policy engine.

6.2.4. AI Data Entry Validation

The data modified or entered by the user should be reviewed to check the information inconsistency or this user is an attacker which tries to mitigate system data. Also, AI can help check the doctor entries for example to automatically correct and validate his syntax or semantic errors.

6.2.5. Mobile Application Development

A mobile application will be developed to enhance accessibility and usability of the system. Mobile application can send notifications to patients to remind of upcoming appointments, doctors to remind of surgical operations and security experts as well to aware him of essential or important logs.

Developing Mobile application can provide Biometric MFA such as Facial or voice recognition and fingerprints which introduces an extra layer of MFA to emphasize authenticity.

6.2.6. ZTA in 5G and Beyond [22]

ZTA using in 5G enhances the ability to dynamically control access behaviors of users' equipment (UEs), which helps prevent the spread of computer viruses, mitigates distributed denial of service (DDoS) attacks, and further security features improvements. There may be several approaches for implementing ZTA in 5G networks as it is extensively researched .

De-centralized Identity Management

This design aspect can be considered near to the JWT approach used for continuous monitoring done by the backend centralized session management server.
Since 6G is concerned in massive connectivity the centralized concept was hard to be implemented and may be neither feasible nor scalable.

Decentralized digital certificate-based authentication and identity management, shown in *figure 60*, is done through these following steps:

- **Certificate Generation**
  This step is initiated by the UE as it generates its certificate as customized by the home community it belongs to. This certificate should contain the UE type, OS, other UE related proof of identity information. After that the certificate will be encrypted and submitted to the controller to be approved.
- **Certificate Registration**
  Local controller of the community decrypts the received certificate and if approved a unique ID is given to this certificate. After signed by the private key of the controller, the certificate is returned to UE to be verified and authorized.
- **Certificate Updating**
  UE may require updates for the digital certificate after updating content of the certificate, which requires certificate regeneration and performing the registration again.
- **Certificate Revocation**
  It is done when the UE changes its home community.

Keeping in mind that each UE and Controller has its own public and private keys, also the encryption algorithms that can be used is asymmetric encryption.

Certificate ID is sufficient for verifying the user, reviewing certificate, resigning certificate, and returning the certificate to its holder if updating was required.

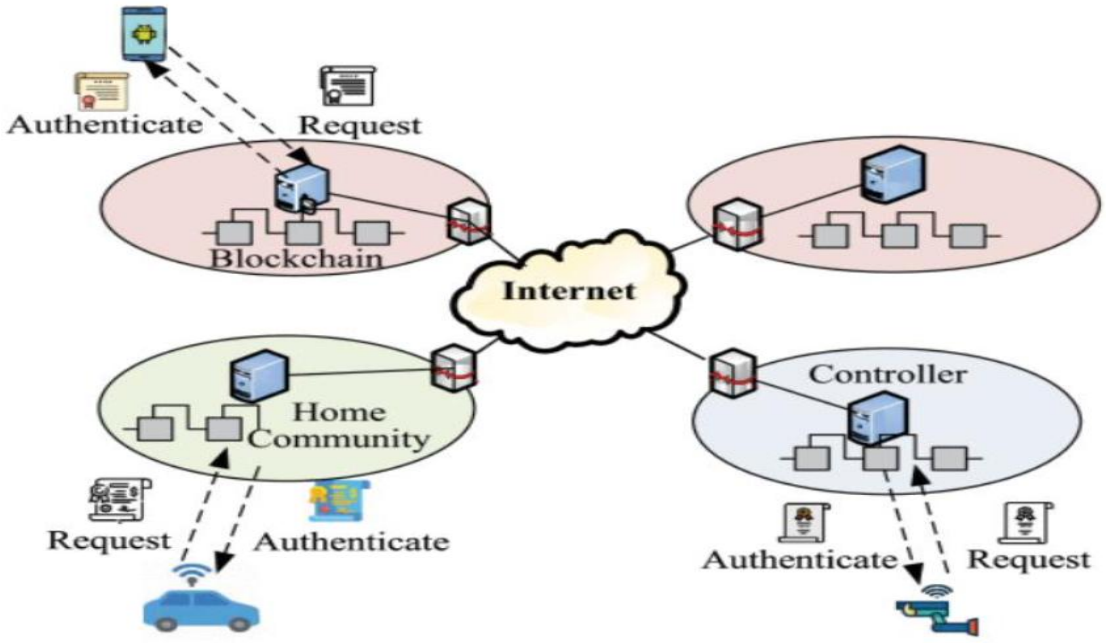

Figure 60 **Community-based de-centralized Identity Management** [22]

CHALLENGES

- Learning and understanding the frontend stack: HTML, CSS and JavaScript, how to make a specific email to act as an OTP server to send, receive and verify the OTP, FastAPI backend framework and how to handle multiple users, distinguish, verify and authenticate them using JWTs, how to connect backend and frontend codes also how to handle data returned from each of them, and how to use SIEM tool for continuous monitoring and understand the ZTA requirements of real-time monitoring and according to it taking the necessary action to enforce policies.
- Learning how to use Tailscale to:
  - Connect servers to its network.
  - The meaning of shared-in and shared-out between servers to meet our micro-segmentation goal.
  - Pings requests work when server 1 send to server 2 not vice-versa also happen; when we understood how the invitation works in Tailscale we could solve the problem as it was tricky to understand. For example, when server 1 invites server 2, it allows server 2 to send requests to it but server1 cannot send to server 2 until server 2 invites him also then it transforms into two-way communication.
  - Understanding how to use Wireshark for both Tailscale and LAN interfaces to verify and show what should appear on each interface when using Tailscale network as Wireshark traffic on the Tailscale connection was decrypted but same traffic from the LAN connection perspective was encrypted.
- Solving database Foreign-key Error:
  - It was a MySQL error as we couldn’t add or update a row as "foreign key constraint failed".
  - The reason was that we tried inserting a row into a table with a reference to a non-existent foreign key.
  - The solution was ensuring the correct order of inserting tables as the parent table should be inserted before the child table and also consider the order of deleting tables.
- Solving CORS (Cross Origin Resource Sharing) Error:
  - There was an error in connecting frontend with backend (FastAPI) as the frontend URL has been blocked by CORS policy.
  - The solution was installing fastapi.middleware.cors and configured it to allow the frontend origin.

- Solving $Bcrypt$ hash comparison failed during login:
  - "Invalid salt" always appeared even though the credentials of the user are correct.
  - The problem was trying to compare the password to the hashed one incorrectly as the password was rehashed again before comparing even though it was already hashed once.
  - The solution was to use $bcrypt.checkpw()$ to prevent double hashing in any step during login check.
- Solving role-based Access control misbehavior:
  - We were facing issues accessing the dashboards of the "Patient" role while "Doctor" and "Nurse" roles worked fine. The backend was filtering data based on roles.
  - We traced the backend endpoint logic. The route was trying to fetch services based on user ID but the relations between tables in database was not correct as there was a relation missing so the function could not fetch the patient details based on user ID.
  - By fixing the relations, after authenticating the user and getting his user ID from authentication table, the record of the patient could be fetched correctly from the patient's table.
- Solving JWTs implementation errors:
  - Session expiry without issuing a refresh request from the user the reason for this was our frontend code having problem with sending token refreshment request according to expiry time of the token so the solution was defining the refresh time at frontend code as it was given as a parameter in the token payload such that the refresh request is sent to backend server on time without passing expiry time.
  - Understand how tokens and refresh tokens are implemented in FastAPI was complicated.
  - When trying to implement the rate limiter time difference between the time a refresh request made and time access token will expire was 3 hours, we then realized that this was due to the server calculating the current time in UTC, while Egypt is UTC+3. To fix this, we needed to configure the server to account for Egypt's time zone (UTC+3).
- Solving problems to set up Kibana, it opened at the first time of setup only and crashes when we try to open it again => we solve it by changing the IP in the configuration file of Kibana to the local host (127.0.0.1) as the IP given by the DHCP server to the server of ELK changes periodically every time the server restarts and according to network conditions which was the cause of crashing.

- Handle sending an alert from Elasticsearch to our backend server we needed a license around $10K\ EGP$. Also sending an alert was complicated so we searched for other methods to be able to fetch the logs indexed in the Elasticsearch in real-time to take instant actions; we used a feedback loop as we said in the ELK part so we wrote our system polices in a python file. It fetches the logs and applies some polices and rules on it instantly and during the session for efficient real-time policy enforcement.
- Solving problems that we faced when applying the Micro-segmentation:
  - Changing and setting the IPs and Port numbers in requests and triggers in frontend, backend and Policy Engine codes to be able to be connected.
  - Change the Logstash input IP to be the Backend server's IP.

REFERENCES

[1] J. Kindervag, S. Rose, O. Borchert, S. Mitchell, and S. Connelly, *Zero Trust Architecture*, NIST Special Publication 800-207, National Institute of Standards and Technology, Gaithersburg, MD, USA, Aug. 2020. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-207

[2] O. C. Edo, D. Ang, P. Billakota, and J. C. Ho, "A zero trust architecture for health information systems," *Health Technol.*, vol. 14, pp. 189–199, Dec. 2023.

[3] M. L. Gambo and A. Almulhem, "Zero Trust Architecture: A Systematic Literature Review," arXiv, Tech. Rep. arXiv:2503.11659v2 [cs.CR], Mar. 21, 2025.

[4] Cybersecurity and Infrastructure Security Agency (CISA), "Zero Trust Maturity Model," U.S. Dept. of Homeland Security, [Online]. Available: https://www.cisa.gov/zero-trust-maturity-model

[5] R. Terry, "Zero Trust Security Explained: Principles of the Zero Trust Model," *Cybersecurity 101*, CrowdStrike, Mar. 13, 2025. [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/zero-trust-security/

[6] "Fernet (symmetric encryption)," *Cryptography 46.0.0.dev1 documentation*, The Cryptography Developers, May 22, 2025. [Online]. Available: https://cryptography.io/en/latest/fernet/

[7] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," *RFC 4648*, Internet Engineering Task Force (IETF), Oct. 2006. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4648#section-4

[8] "Modular Crypt Format," *Passlib v1.7.4 Documentation*, The Passlib Developers. [Online]. Available: https://passlib.readthedocs.io/en/stable/modular_crypt_format.html

[9] M. McCormack, "HIPAA Multi Factor Authentication Requirements," *Compliancy Group*, Oct. 26, 2023. [Online]. Available: https://compliancy-group.com/hipaa-multi-factor-authentication-requirements/

[10] Okta, Inc., "Role-Based Access Control," *Auth0 Documentation*, 2025. [Online]. Available: https://auth0.com/docs/manage-users/access-control/rbac

[11] Okta, Inc., "JSON Web Tokens," *Okta Website*. [Online]. Available: https://auth0.com/docs/secure/tokens/json-web-tokens#security-of-jwts

[12] IBM, "JSON Web Token (JWT)," *IBM Documentation*. [Online]. Available: https://www.ibm.com/docs/en/cics-ts/6.x?topic=cics-json-web-token-jwt

[13] T. Fatunmbi, "A Comparison of Cookies and Tokens for Secure Authentication," *Okta Developer Blog*, Feb. 8, 2022. [Online]. Available: https://developer.okta.com/blog/2022/02/08/cookies-vs-tokens

[14] D. Arias and S. Bellen, “What Are Refresh Tokens and How to Use Them Securely,” Auth0 Blog, Oct. 7, 2021. [Online]. Available: https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/

[15] Kaspersky, "What is a digital footprint? And how to protect it from hackers," Kaspersky. Available: https://www.kaspersky.com/resource-center/definitions/what-is-a-digital-footprint

[16] GeeksforGeeks, "What is a Digital Footprint?," GeeksforGeeks, 2024.Available: https://www.geeksforgeeks.org/what-is-a-digital-footprint/

[17] Auth0, *The JWT Handbook*. Auth0, 2021. [Online]. Available: https://auth0.com/resources/ebooks/jwt-handbook

[18] Q. Dang, “Recommendation for Applications Using Approved Hash Algorithms,” *NIST Special Publication 800-107 Revision 1*, National Institute of Standards and Technology, Gaithersburg, MD, USA, Aug. 2012. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf

[19] Cybersecurity and Infrastructure Security Agency (CISA), “Understanding Denial-of-Service Attacks,” *CISA Website*, Feb. 1, 2021. [Online]. Available: https://www.cisa.gov/news-events/news/understanding-denial-service-attacks

[20] GeeksforGeeks, “What is Elastic Stack and Elasticsearch?,” *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/what-is-elastic-stack-and-elasticsearch/

[21] Tailscale Inc., “What is Tailscale?,” *Tailscale Documentation*, May 19, 2025. [Online]. Available: https://tailscale.com/kb/1151/what-is-tailscale

[22] X. Chen, W. Feng, N. Ge, and Y. Zhang, “Zero Trust Architecture for 6G Security,” *IEEE Network*, vol. 38, no. 4, pp. 224–232, Jul. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10288499