# **REPORT CYBERSECURITY**

L’obbiettivo del nostro progetto è quello di creare un server rest API, con dati prodotti casualmente, da affiancare ad un servizio reale e critico che nel nostro caso riguarda il sistema informatico di un ospedale.Al giorno d’oggi molte applicazioni sono considerabili come un front-end per una serie di chiamate API. Le API sono necessarie per il corretto funzionamento di un’applicazione e se non fossero protette adeguatamente attaccanti malevoli potrebbero esfiltrare dati, effettuare attacchi DDoS ai server o abusarne.  
Il sistema che abbiamo realizzato è composto da un authentication server (che si occupa di gestire l’autenticazione Oauth), il server API e un esempio di applicazione client che vuole utilizzare le API disponibili.

**Authorization Server**  
Per proteggere le nostre API abbiamo utilizzato il protocollo **Oauth** che permette l’accesso alle risorse solamente ad utenti autenticati. In particolare, abbiamo utilizzato l’authorization code flow che viene rappresentato nella seguente immagine.

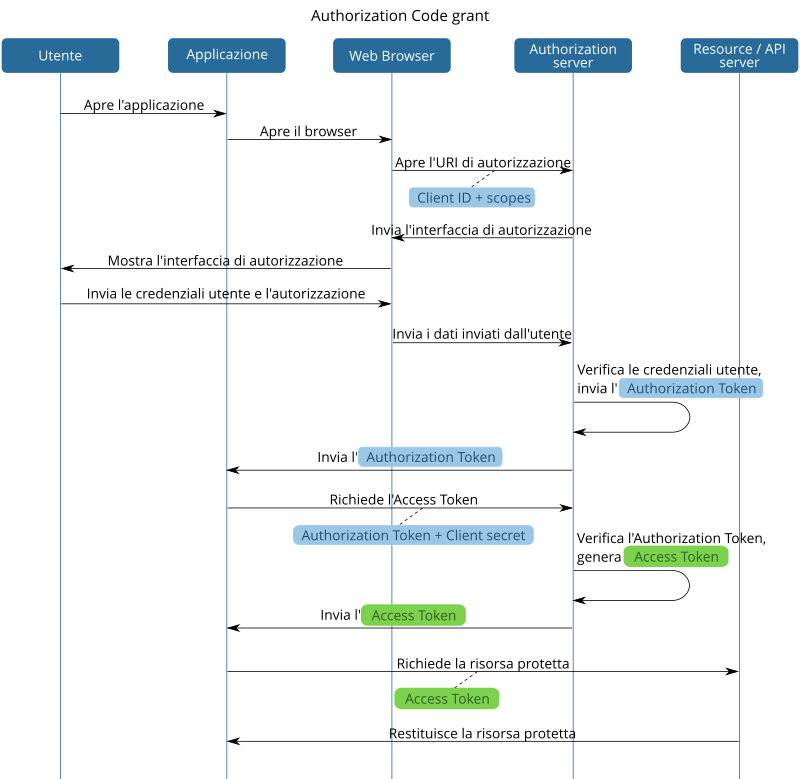


Figura 1: flusso di esecuzione del protocollo OAuth

Per rendere ulteriormente più sicuro il protocollo abbiamo utilizzato l’estensione PKCE, il quale consiste nella generazione, da parte dell’applicazione client, di un code verifier e di un code challenge che permettono di evitare un’eventuale intercettazione da parte di un attaccante dell’authorization code.

Per gestire il processo di autenticazione l’authorization server dispone di tre endpoint: /auth, /signin e /token:

**/auth**: il client invia il proprio client\_id, redirect\_url e code\_challenge per avviare il processo di creazione di un nuovo authorization code e conseguente access token. In particolare, l’authorization server andrà a verificare se il client\_id e redirect\_url corrispondono ai dati salvati nella precedente fase di registrazione, se corretti si viene reindirizzati a /signin.  
Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 2: gestione /auth

**/signin**:viene presentata all’utente una pagina di login per la verifica delle sue credenziali personali. Se la verifica ha successo viene creato un **authorization** **code** in cui viene memorizzato client\_id, redirect\_url, code\_challenge, username e expiration\_date. Al termine viene eseguito un redirect a redirect\_url inviando l’authorization code.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 3: gestione /signin

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 4: creazione authorization code

**/token**: il client invia il proprio client\_id, client\_secret, redirect\_url, code\_verifier e authorization code all’authorization server, il quale verifica la validità dell’authorization code e sfrutta il code verifier per controllare se il code\_challenge inviato precedentemente è corretto. In caso di successo viene restituito un access token, con il quale l’applicazione per conto dell’utente può accedere a determinate risorse del server API.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 5: gestione /token

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 6: creazione access\_token

L’authorization sever offre anche una pagina dedicata alla registrazione (**/register**) di nuove applicazioni che vogliono utilizzare le API. Effettuata la registrazione dell’applicazione, l’authentication server restituisce un client\_id e un client\_secret che verranno utilizzati successivamente per la generazione di nuovi access token per l’accesso all’API.

## **Immagine che contiene testo, schermata, Carattere Descrizione generata automaticamente**

Figura 7: gestione /register

## **Server API**

Per la realizzazione del server API abbiamo utilizzato fastAPI, un framework moderno e ad alte prestazioni per la creazione di API.  
Abbiamo definito diverse routes raggiungibili solamente se l’utente autenticato possiede i diritti per accedervi.  
Per questo motivo abbiamo inserito diversi ruoli che permettono di simulare un sistema reale, in cui a tipi di utenti differenti corrispondono privilegi differenti. Nel nostro caso specifico abbiamo definito tre ruoli: dottore, infermiere, paziente (e ciascuno di essi appartiene ad uno specifico reparto).  
In questo modo ogni volta che il client invia una richiesta al server API si verifica se l’access token è ancora valido e in caso affermativo viene fatta una richiesta all’authorization server per ricavare lo username dell’utente a cui è associato l’access token.  
L’username viene, quindi, utilizzato per ricavare il ruolo e il reparto di appartenenza dell’utente e per stabilire se esso ha il permesso o meno per accedere alla risorsa richiesta. Nelle seguenti figure viene mostrato il messaggio di errore che viene inviato nel caso in cui l’utente non ha i permessi sufficienti per accedere ad una risorsa, o nel caso in cui richiede risorse che appartengono ad un reparto diverso dal suo.



Figura 8: esempio di richiesta ad una risorsa senza avere i permessi corretti



Figura 9: esempio di richiesta per una risorsa che appartiene ad un reparto non di nostra competenza

Invece, in tutti gli altri casi il server risponderà con i dati richiesti (come si può vedere nella seguente figura).

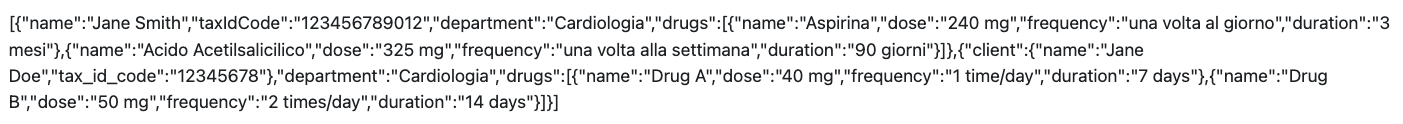


Figura 10: esempio di dati prodotti correttamente dal server API

In figura viene, quindi, mostrato il flusso di esecuzione in caso di richiesta da un utente che dispone dei permessi corretti:

Immagine che contiene testo, diagramma, schermata, cerchio

Descrizione generata automaticamente

Figura 11: flusso di esecuzione corretto

Per la generazione di dati casuali abbiamo utilizzato la libreria Llama.cpp che supporta l'inferenza per molti modelli LLMs (in particolare noi abbiamo utilizzato il modello LLaMa 2 7B) e abbiamo seguito i passi descritti nella guida <https://python.langchain.com/docs/integrations/llms/llamacpp>.

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 12: metodo per la generazione di dati casuali

Per vedere l’API specification basta cliccare nel seguente link (da mettere il link github).  
(In realtà penso che sia meglio metterlo direttamente qui, ma non so come farlo)

## **Client**

Per una completa simulazione di un caso reale abbiamo realizzato anche una semplice applicazione client con la quale è possibile testare il corretto funzionamento dell’authentication e dell’API server.  
(non ho idea di cosa poter scrivere)

## **Esecuzione del sistema**

Per l’esecuzione del sistema abbiamo utilizzato Docker. (Credo che sia cosi) In particolare, viene avviato un container in cui sono in esecuzione sia l’authentication che l’API server, mentre in un secondo container può essere avviata l’applicazione client per la simulazione del sistema. Nelle seguenti figure vengono mostrati i dockerfile per l’esecuzione del sistema.

[IMMAGINI DOCKER FILE]

-aggiungere immagine iniziale con architettura sistema (client, authServer, APIServer)  
-aggiungere dettagli sul flow (PKCE, redirect\_url, session)  
-aggiungere parti di codice  
-aggiungere descrizione API  
-spiegare funzionamento client registration.