**Cybersecurity M report**Deception component generator

Rest API server

A cura di:

Luca Dominici  
Enrico Zacchiroli  
Lorenzo Ziosi

# Introduzione

L'obiettivo di questo progetto è creare un generatore di risorse false e casuali per un specifico tipo di risorsa da proteggere, nel nostro caso un sistema informatico di un ospedale.

Ad oggi il carico computazionale è scaricato su server decentralizzati lasciando quindi ai client il solo compito di renderizzare il front-end. Questi server quindi diventano bersagli di attacchi malevoli che, nel caso in cui non fossero protetti adeguatamente, potrebbero comportare l’esfiltrazione di dati o la non disponibilità dei servizi erogati.

Il sistema che abbiamo realizzato utilizza il protocollo **OAuth** per l’autorizzazione sicura seguento il cosiddetto “Authorization code flow” [Figura 1]. Abbiamo quindi implementato un server API (resource server) che eroga i servizi utilizzati da un ulteriore applicativo chiamato “client”.

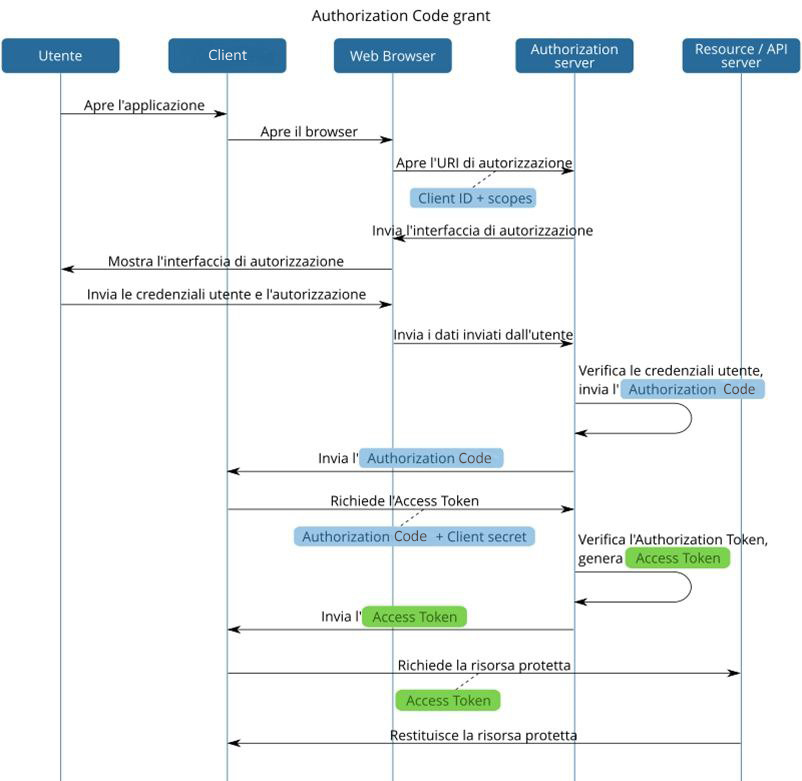


Figura 1: Flusso di esecuzione OAuth in versione “Authorization code flow”

Per rendere ulteriormente più sicuro il protocollo abbiamo utilizzato l’estensione PKCE. Essa consiste nella generazione, da parte dell’applicazione client, di un code verifier e di un code challenge che rendono il sistema robusto ad eventuali intercettazioni dell’authorization code.

# Authorization Server

Come da specifiche, 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.



Figura 2: endpoint /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.



Figura 3: endpoint /signin



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 sua validità 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.



Figura 5: endpoint /token  
  
  


Figura 6: creazione access\_token

L’authorization sever offre anche una pagina dedicata alla registrazione (**/client-signup**) di nuove applicazioni che vogliono utilizzare le API. Effettuata la registrazione dell’applicazione, verranno restituiti un *client\_id* e un *client\_secret* generati casualmente che verranno utilizzati successivamente per l’autenticazione del client presso l’authorization server.



Figura 7: endpoint /client-signup

# Server API

Per la realizzazione del server API abbiamo utilizzato FastAPI, un recente framework 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 e paziente ognuno dei quali appartiene ad uno specifico reparto, con conseguenti limitazioni nell’accesso alle risorse appartenti ad altri reparti.

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 effettuata 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 [Figura 8], o nel caso in cui richiede risorse che appartengono ad un reparto diverso dal suo [Figura 9].



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 [Figura 10].

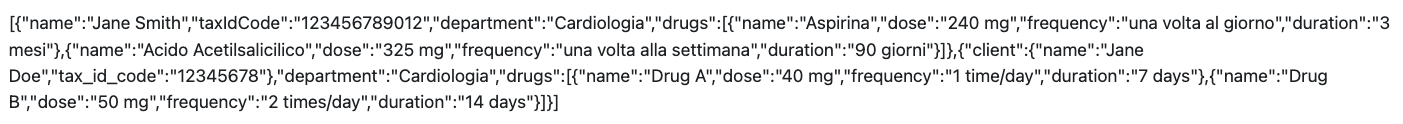


Figura 10: esempio di dati prodotti correttamente dal server API

Di seguito viene mostrato il flusso di esecuzione in caso di richiesta da un utente che dispone dei permessi corretti [Figura 11]:

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. Nel nostro caso è stato utilizzato il modello LLaMa 2 7B. Per l’installzione sono stati seguiti i passi descritti nella guida disponibile al seguente link: [LangChain Llama.cpp](https://python.langchain.com/docs/integrations/llms/llamacpp) (<https://python.langchain.com/docs/integrations/llms/llamacpp>).



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’authorization server e del server API.

L’utente appena collegato sarà invitato ad effettuare l’accesso all’authorization server per ottenere l’access token. Se questo è presente verranno invece mostrate le API disponibili fornite dal resource server.

# Esecuzione del sistema

Per l’esecuzione del sistema abbiamo utilizzato Docker. (Credo che sia cosi)

In particolare, dopo l’esecuzione del comando *docker-compose*, saranno avviati tre container che eseguiranno rispettivamente authorization server, API server e client.

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.

Oauth: implementazione protocollo, pkce

Auth server + api server

* + accept configurations for oauth authentication (at generation time ) to secure the api endpoint and provide an out of the box functionality

Client + docker + demo

1.

Obiettivo, (intro) risorse API vulnerabili e da proteggere, cos’è OAuth

2.

ACF, PKCE, auth server

3.

OpenAPI (server API), docker, demo