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 seguendo 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".

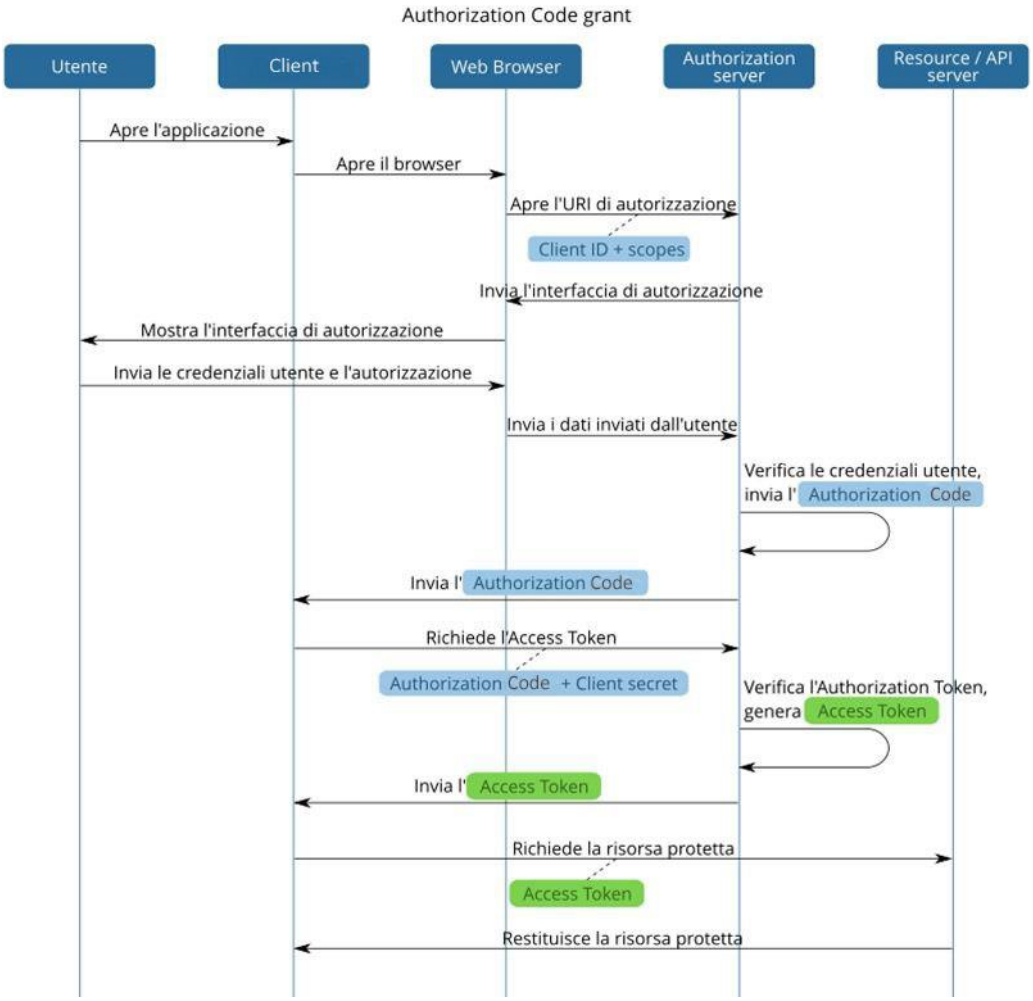

Figura 1: Flusso di esecuzione OAuth in versione "Authorization code flow"

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 autorizzazione 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.

```
@app.route('/auth')
def auth():
  # Parametri della richiesta di accesso del client
  client_id = request.args.get('client_id')
  redirect_url = request.args.get('redirect_url')
  code_challenge = request.args.get('code_challenge')

  #Controllo che siano stati inviati tutti i dati dal client
  if None in [ client_id, redirect_url, code_challenge ]:
    return json.dumps({
      "error": "invalid_request"
    }), 400

  #Controllo i dati del client siano corretti
  if not verify_client_info(client_id, redirect_url):
    return json.dumps({
      "error": "invalid_client"
    })

  #Tutto ok -> mostro la pagina di login
  return render_template('AC_PKCE_grant_access.html',
                         client_id = client_id,
                         redirect_url = redirect_url,
                         code_challenge = code_challenge)
```

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* ed *expiration_date*. Al termine viene eseguito un redirect a *redirect_url* inviando l'authorization code.

```python
@app.route('/signin', methods = ['POST'])
def signin():
  # Parametri richiesta
  username = request.form.get('username')
  password = request.form.get('password')
  client_id = request.form.get('client_id')
  redirect_url = request.form.get('redirect_url')
  code_challenge = request.form.get('code_challenge')

  # Controllo che siano stati inviati tutti i dati
  if None in [username, password, client_id, redirect_url, code_challenge]:
    return json.dumps({
      "error": "invalid_request"
    }), 400

  # Controllo che il redirect url dato sia valido
  if not verify_client_info(client_id, redirect_url):
    return json.dumps({
      "error": "invalid_client"
    })

  # Verifico se username e password dell'utente che ha fatto il login siano giusti
  if not authenticate_user_credentials(username, password):
    return json.dumps({
      'error': 'access_denied'
    }), 401

  # Creazione authorization code
  authorization_code = generate_authorization_code(client_id, redirect_url,
                                                   code_challenge,username)

  url = process_redirect_url(redirect_url, authorization_code)

  return redirect(url, code = 303)
```

Figura 3: endpoint /signin

```python
def generate_authorization_code(client_id, redirect_url, code_challenge, username):
  # Costruzione e cifratura dell'authorization code
  authorization_code = f.encrypt(json.dumps({
      "client_id": client_id,
      "redirect_url": redirect_url,
  }).encode())

  # Encoding dell'authorization code in base 64 e rimozione padding
  authorization_code = base64.b64encode(
      authorization_code, b'-_').decode().replace('=', '')

  expiration_date = time.time() + CODE_LIFE_SPAN

  # Memoizzazione locale dell'authorization code
  authorization_codes[authorization_code] = {
      "client_id": client_id,
      "redirect_url": redirect_url,
      "exp": expiration_date,
      "code_challenge": code_challenge,
      "username": username
  }

  return authorization_code
```

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. Il client recupera il *code_verifier* utilizzato nello step iniziale utilizzando una variabile alfanumerica identificativa chiamata *state* inserita direttamente nel *redirect_url*.

La struttura di un redirect URL è la seguente:

```
{{dest}}?response_type=code&client_id={{client_id}}&code_challenge={{code_challenge
}}&redirect_url={{redirect_url}}?state={{state}}
```

```python
@app.route('/token', methods = ['POST'])
def exchange_for_token():
  # Parametri richiesta
  authorization_code = request.form.get('authorization_code')
  client_id = request.form.get('client_id')
  client_secret = request.form.get('client_secret')
  code_verifier = request.form.get('code_verifier')
  redirect_url = request.form.get('redirect_url')

  # Verifica che tutti i parametri siano settati
  if None in [ authorization_code, client_id, code_verifier, redirect_url ]:
    return json.dumps({
      "error": "invalid_request"
    }), 400

  # Autentica il client con client_id e client_secret
  if not authenticate_client(client_id, client_secret):
    return json.dumps({
      'error': 'access_denied'
    }), 401

  # Verifica authorization code
  if not verify_authorization_code(authorization_code, client_id, redirect_url,
                                   code_verifier):
    return json.dumps({
      "error": "access_denied"
    }), 400

  # Genera access token
  access_token = generate_access_token(authorization_code)

  return json.dumps({
    "access_token": access_token,
    "token_type": "JWT",
    "expires_in": JWT_LIFE_SPAN
  })
```

Figura 5: endpoint /token

```python
def generate_access_token(authorization_code):

  record = authorization_codes.get(authorization_code)
  username = record['username']

  payload = {
      "iss": ISSUER,
      "exp": time.time() + JWT_LIFE_SPAN,
  }

  access_token = JWT().encode(payload, private_key, alg='RS256')

  users_db[username]['access_token'] = access_token

  del authorization_codes[authorization_code]

  return access_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.

```
@app.route('/client-signup', methods = ['POST'])
def client_signup():
  urls = request.form.getlist('redirect_url[]')

  for url in urls:
    if not is_valid_url(url):
      return render_template('AC_PKCE_register_client.html', error = 'Invalid URLs')

  (client_id, client_secret) = register_client(urls)

  return render_template('AC_PKCE_register_client.html',
                         client_id = client_id,
                         client_secret = client_secret)
```

Figura 7: endpoint /client-signup

Server API

Per la realizzazione del server API abbiamo utilizzato FastAPI, un recente framework 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].

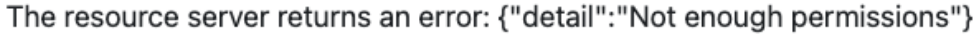

Figura 8: esempio di richiesta ad una risorsa senza avere i permessi corretti

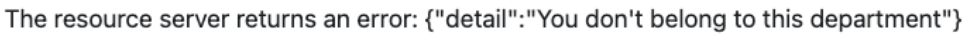

Figura 9: esempio di richiesta per una risorsa che appartiene ad un reparto non di nostra competenza

Invece, in tutti gli altri casi il server risponderà con i dati richiesti [Figura 10].

[{"name":"Jane Smith","taxIdCode":"123456789012","department":"Cardiologia","drugs":[{"name":"Aspirina","dose":"240 mg","frequency":"una volta al giorno","duration":"3 mesi"},{"name":"Acido Acetilsalicilico","dose":"325 mg","frequency":"una volta alla settimana","duration":"90 giorni"}]},{"client":{"name":"Jane Doe","tax_id_code":"12345678"},"department":"Cardiologia","drugs":[{"name":"Drug A","dose":"40 mg","frequency":"1 time/day","duration":"7 days"},{"name":"Drug B","dose":"50 mg","frequency":"2 times/day","duration":"14 days"}]}]

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]:

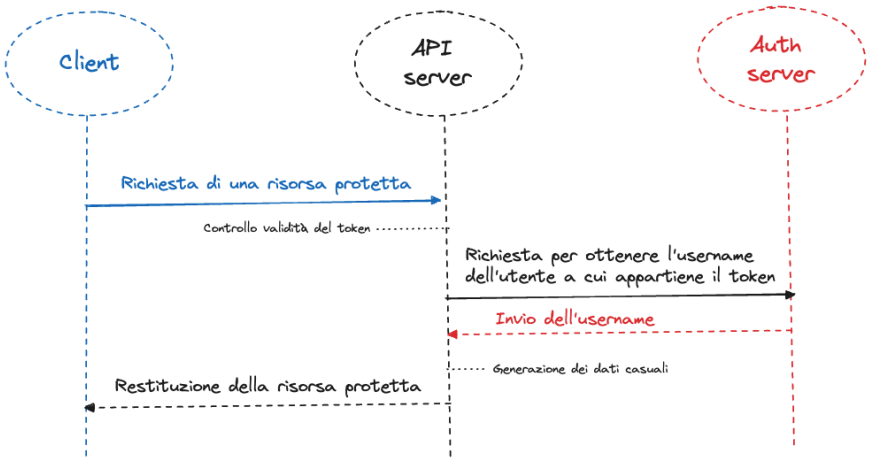

Figura 11: flusso di esecuzione corretto

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).

```
def create_data(prompt:str, num:int):
    res_list = []
    for _ in range(num):
        result = llm(prompt)
        json_object = json.loads(result)
        res_list.append(json_object)
    return res_list
```

Figura 12: metodo per la generazione di dati casuali

Per vedere l'OpenAPI specification basta cliccare nel seguente link: [openapi.json](https://github.com/zack-99/restAPIServer/blob/main/openapi.json) (https://github.com/zack-99/restAPIServer/blob/main/openapi.json).

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. Nel caso quest'ultimo sia già presente verranno invece mostrate le API disponibili fornite dal resource server.

Esecuzione del sistema

Per l'esecuzione del sistema abbiamo utilizzato Docker. In particolare, dopo l'esecuzione del comando *docker-compose* (Figura 13)Figura 13: docker-compose, saranno avviati tre container che eseguiranno rispettivamente authorization server, API server e client.

```yaml
version: '3'
services:
  client:
    image: zacken1999/apiclientimage:1.0
    container_name: apiclient
    ports:
      - 5000:5000
    environment:
      - TOKEN_PATH=http://authserver:5001/token
      - RES_PATH=http://apiserver:8000/
      - VERIFY_TOKEN_PATH=http://authserver:5001/user
  authserver:
    image: zacken1999/authserverimage:1.0
    container_name: authserver
    ports:
      - 5001:5001
  apiserver:
    image: zacken1999/apiserverimage:1.0
    container_name: apiserver
    ports:
      - 8000:8000
    environment:
      - VERIFY_TOKEN_PATH=http://authserver:5001/user
```

Figura 13: docker-compose

Il comando da eseguire per avviare il sistema è il seguente:

```
docker-compose -f docker-compose.yaml up
```