Sicurezza nelle API RESTful

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *Questo documento si basa sulle slide del corso di Sviluppo Web Avanzato, riorganizzate per migliorare l'esperienza di lettura. Non è un libro di testo completo o un manuale tecnico, e deve essere utilizzato insieme a tutti gli altri materiali didattici del corso. Si prega di segnalare eventuali errori o omissioni all'autore.*

La sicurezza nelle API (RESTful)

La sicurezza *è parte integrante del design* di una API:

- Le API sono spesso esposte su Internet
- Trasportano dati sensibili
- Sono un bersaglio tipico degli attacchi automatizzati come le Botnet.

Molte vulnerabilità derivano da errori di configurazione o di design, *non da bug*.

Tecniche per proteggere le API RESTful

E' possibile applicare alle API tecniche di protezione ben note già in ambiente web e tecniche sviluppate *ad-hoc*.

- API Keys
- Basic Authentication
- Token-based Authentication
- Mutual TLS (mTLS)
- OAuth *(standard di fatto per API pubbliche e microservizi)*

API Keys

> Le API Keys sono semplici *shared secrets*. Vengono generate sul server (solitamente tramite una interfaccia web) e conservate sul client.

Pro

- Semplici da generare
- Facili da usare lato client
- Buone per servizi server-to-server semplici

Contro

- Non identificano un utente
- Difficili da revocare *granularmente*
- Non supportano *scope*
- Non sicure se usate lato client (mobile, browser)

Basic Authentication

> Trasmette le credenziali dell'utente con ogni richiesta tramite l'header *Authorization*. Accettabile solo in contesti interni e protetti.

```
Authorization: Basic base64(username:password)
```

Pro

- Semplice da implementare
- Supportata nativamente da HTTP

Contro

- Password in ogni richiesta
- Richiede TLS obbligatorio

Token-based Authentication

> Il server rilascia (ad esempio tramite una procedura di login) un token firmato (JWT) che il client invia in ogni richiesta tramite l'header *Authorization*.

```
Esempio header HTTP
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

Pro

- Scalabile: il server non deve mantenere stato di sessione.
- Self-contained: il token contiene claim utili (sub, scope, ruoli, ecc.).

Contro

- Revoca non banale: finché non scade, il token resta valido (a meno di *blacklist*).

Mutual TLS (mTLS)

> Autenticazione *reciproca* client - server tramite certificati X.509.

Pro

- Autenticazione forte del client (basata su certificati).
- Molto robusto contro il furto di credenziali.
- Ideale per comunicazioni interne tra microservizi.

Contro

- Gestione complessa dei certificati (emissione, rinnovo, revoca).
- Non adatto a client di tipo utente (browser,...).

OAuth 2.0: lo standard moderno

- Elimina i problemi delle API Keys e della Basic Authentication
- Supporta autorizzazione delegata (*"Accedi con Google"*, *"Concedi accesso a GitHub"*,...)
- Supporta *scope* granulari
- Funziona con *JWT*
- Ideale per API pubbliche e microservizi

OAuth 2.0 (per API RESTful)

Cos'è OAuth 2.0

- Framework di **autorizzazione delegata**
- Evita la condivisione di password
- Basato su **token** (a vita breve)
- Autorizzazioni **granulari** (scope)
- **Revoca** semplice

Ruoli in OAuth 2.0

I processi descritti in OAuth 2.0 prevedono la partecipazione di diversi attori, ognuno con un proprio ruolo. I ruoli principali sono i seguenti:

- Resource owner *U*
  ad esempio l'utente
- Client *C*
  l'applicazione usata dall'utente per accedere alla risorsa
- Authorization server *S*
  gestisce l'autorizzazione vera e propria
- Resource server *R*
  nel nostro caso la API

Token e Codici

I processi descritti in OAuth 2.0 prevedono lo scambio di diversi codici e token:

- Authorization code
  permette di generare l'access token dopo la verifica delle credenziali
- Access Token
  usato per accedere effettivamente alla risorsa
- Refresh Token
  usato per rigenerare l'access token a intervalli regolari

Authorization Code Flow

A) Registrazione del client

> Il client *C* deve essere registrato una tantum presso l'Authorization Server *S*.

1. Lo sviluppatore registra *C* su *S*.
2. *S* restituisce un **client_id** e, se previsto, un **client_secret**.
   Questi identificativi sono le "credenziali" di *C*, usate per dimostrare la sua identità verso *S*. Non vanno mai esposti pubblicamente (es. in codice lato browser).

B) Autorizzazione dell'utente

> *U* chiede tramite *C* di accedere a *R* con la propria identità e con specifici permessi (*scope*).

1. *U* interagisce con *C* per accedere a *R*.

2. *C* redirige *U* verso la pagina di login di *S*.
3. *U* si autentica con *S* e approva gli *scope* richiesti.
4. *S* restituisce a *C* un **authorization code**.
    - È un codice breve, monouso, con validità molto limitata (tipicamente < 1 minuto).
    - Serve solo per ottenere l'*access token*, non per accedere direttamente a *R*.

L'**authorization code** rappresenta la prova che l'utente ha concesso l'autorizzazione al client per accedere a certe risorse con certi permessi. È molto più sicuro delle credenziali dell'utente, che restano confinate all'Authorization Server (il client non le vede).

Nota: l'**Implicit Grant** rilasciava direttamente l'access token a questo punto, ma è oggi **deprecato** e non va usato.

C) Scambio del codice con un access token

> *C* scambia l'authorization code con un access token presso *S*.

1. *C* invia a *S* l'authorization code, il **client_id** e, se previsto, il **client_secret**.
2. *S* restituisce a *C*:
    - un **access token** (valido per accedere a *R*)
    - opzionalmente un **refresh token** (per ottenere nuovi access token senza coinvolgere di nuovo *U*).

In pratica, il client sta dimostrando due cose contemporaneamente:

- che l'utente ha autorizzato (tramite l'**authorization code**),
- che il client stesso è legittimo (tramite le proprie credenziali). Il risultato è l'**access token**, che permette di accedere alle risorse protette.

D) Accesso alla API

> *C* accede a *R* usando l'access token.

- L'access token viene passato nell'header HTTP:

```
Authorization: Bearer <access_token>
```

- *R* verifica la validità del token.
  Se il token è un JWT (*Json Web Token*), *R* può leggernne direttamente i *claim* (*issuer*, *scadenza*, *scope*, *userId*, ecc.) e verificarne quindi la validità. Se il token non è un JWT (token opaco), *R* deve chiamare l'Authorization Server *S* per verificare il token e ottenere i dati associati.

E) Refresh del token di accesso

> *C* rigenera l'access token quando scade, usando il refresh token.

1. *C* invia a *S* il **refresh token** insieme al **client_id** (e, se previsto, il **client_secret**).
2. *S* risponde con un nuovo **access token** e, spesso, un nuovo **refresh token**.

- Il refresh token ha durata più lunga dell'access token, ma non illimitata: dipende dalla policy di *S*.
- Viene memorizzato da *C* per evitare di chiedere ogni volta all'utente di autenticarsi. È però una credenziale molto sensibile e va protetta come una password.

Questo passo è simile al precedente scambio (C), ma invece dell'authorization code si usa il refresh token.

Documentazione con OpenAPI 2.0

Definire la sicurezza OAuth2

```yaml
securityDefinitions:
  oauth2:
    type: oauth2
    flow: accessCode
    authorizationUrl: https://auth.example.com/oauth/authorize
    tokenUrl: https://auth.example.com/oauth/token
    scopes:
      read: Lettura risorse
      write: Scrittura risorse
```

Applicare la sicurezza agli endpoint

```yaml
paths:
  /orders:
    get:
      security:
        - oauth2:
            - read
```