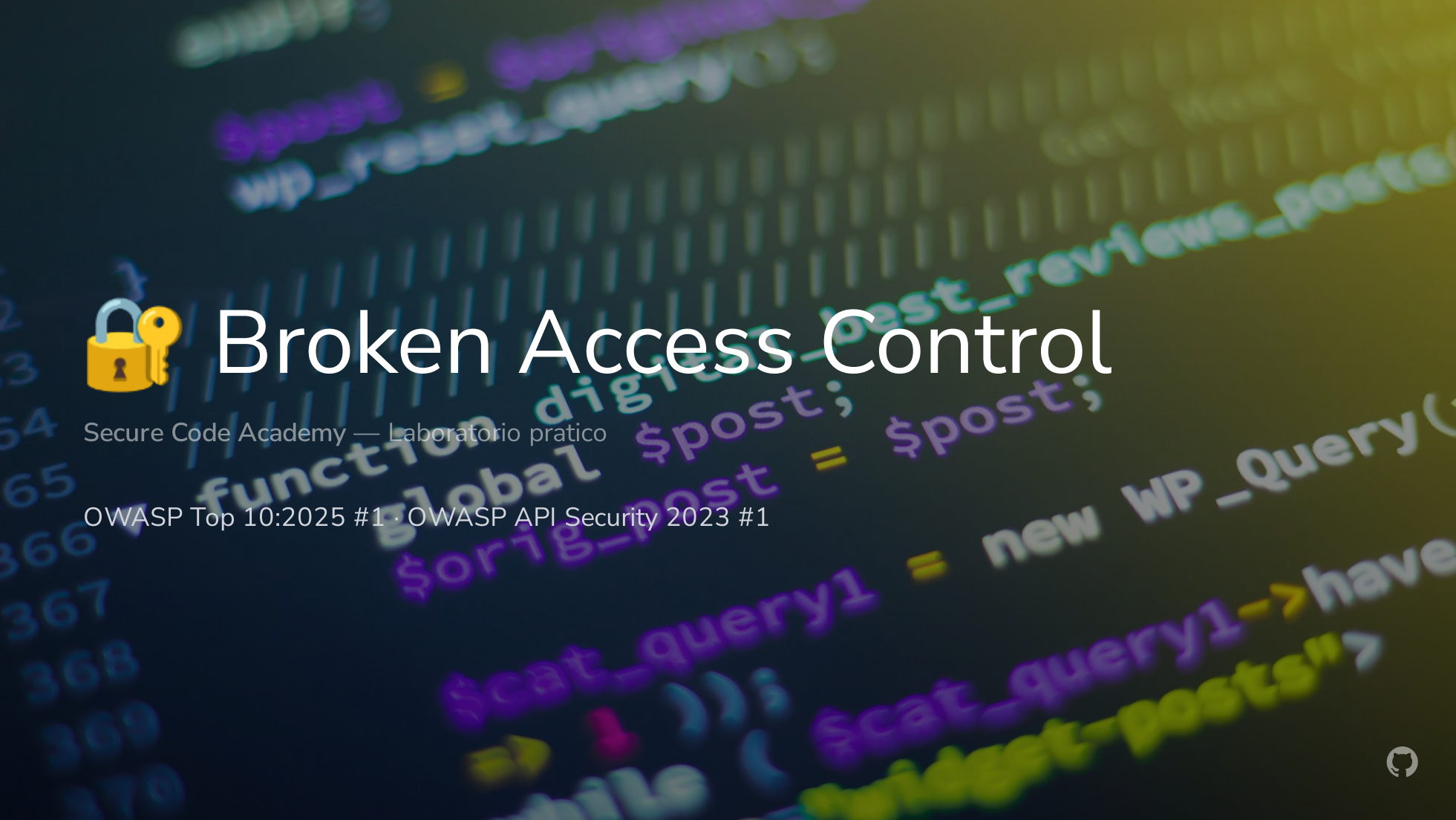


Broken Access Control

Secure Code Academy — Laboratorio pratico

OWASP Top 10:2025 #1 · OWASP API Security 2023 #1

Contenuto

Teoria

1. 🎯 Broken Access Control: Cosa sono?
2. 📊 Dati e impatto (OWASP 2025)
3. 🔐 Authentication & Authorization
4. 🔎 Tipologie di vulnerabilità
5. 🧪 OWASP Testing Guide
6. 🌐 OWASP API Security: BOLA
7. 🛡️ Remediation & Best Practice

Pratica

8. 🧪 Struttura del laboratorio
9. 💻 Le 6 vulnerabilità del lab
10. 🔴/✅ Vuln (1) — ID Enumeration
11. 🔴/✅ Vuln (2) — Privilege Escalation (Data)
12. 🔴/✅ Vuln (3) — Privilege Escalation (Action)
13. 🔴/✅ Vuln (4) — Broken Object Authorization
14. 🔴/✅ Vuln (5) — Missing Authentication
15. 🎁/✅ Vuln (X) — Hidden Vulnerability
16. ✅ Approccio TDD & Verifica

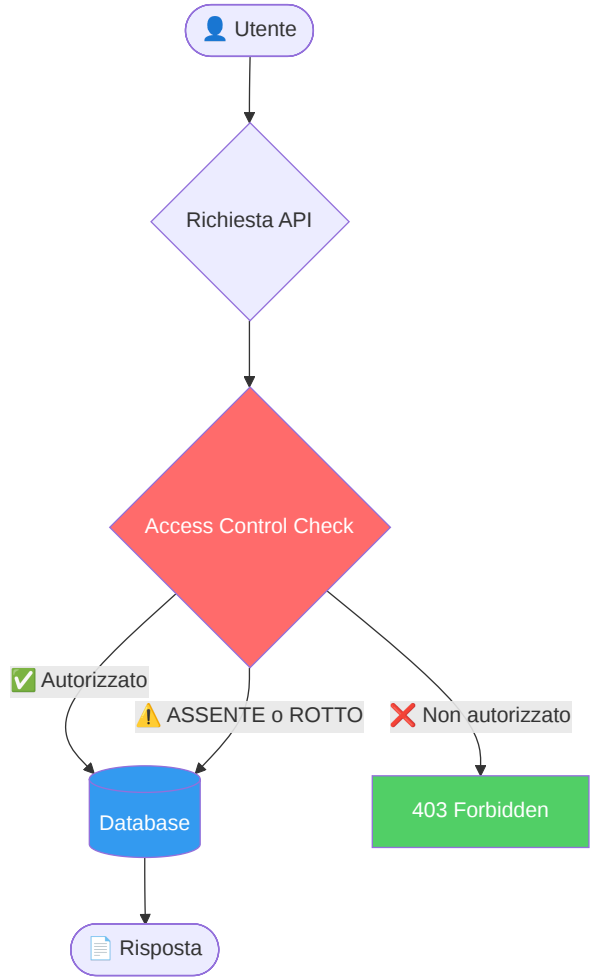
Broken Access Control: Cosa Sono?

Broken Access Control

Il controllo degli accessi garantisce che gli utenti **non possano agire al di fuori dei loro permessi previsti.**

Un fallimento di questo meccanismo porta tipicamente a:

- **Divulgazione non autorizzata** di informazioni
- **Modifica o distruzione** di dati altrui
- **Esecuzione di funzioni privilegiate** senza averne il diritto



OWASP Top 10:2025 — #1

100%
delle applicazioni testate presenta qualche forma di BAC

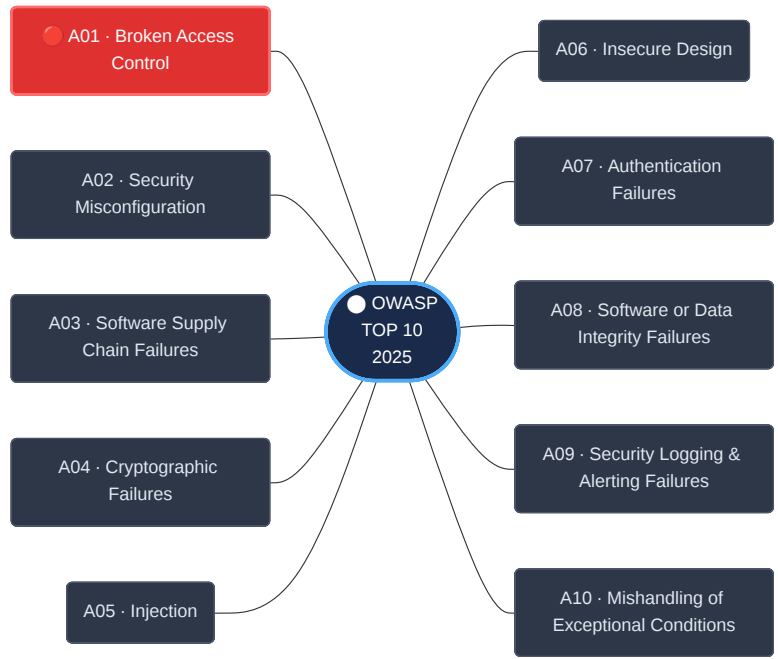
1.8M+
occorrenze rilevate nei dati raccolti

32.654
CVE correlati — il secondo numero più alto in assoluto

Metrica	Valore
CWE mappate	40
Max Incidence Rate	20,15%
Avg Weighted Exploit	7,04 / 10
Avg Weighted Impact	3,84 / 10

Fonte: OWASP Top 10:2025 — A01

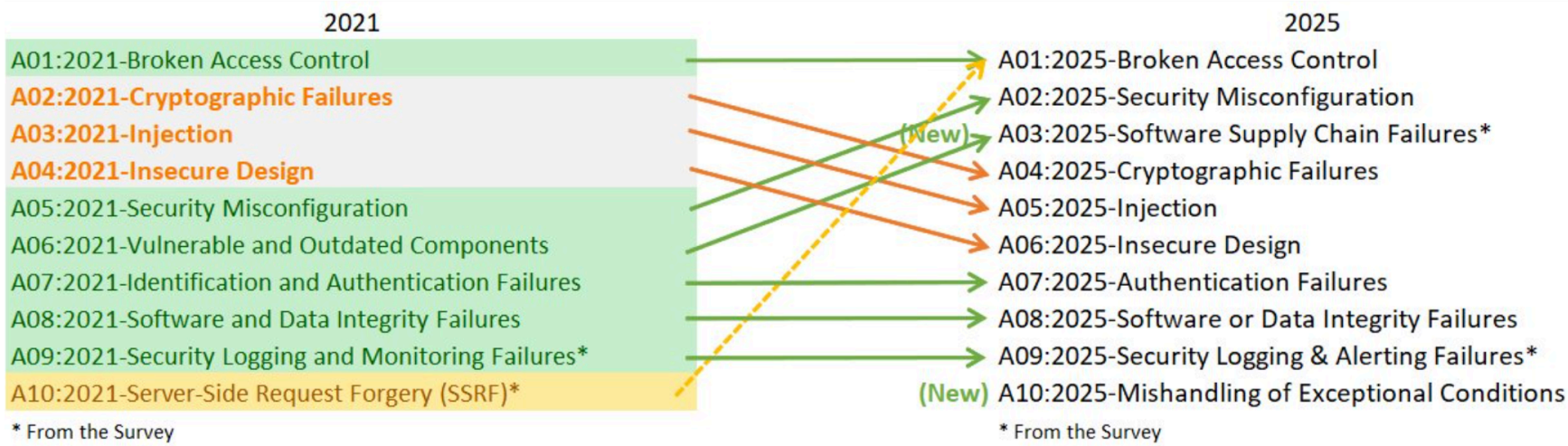
OWASP Top 10 : 2025



https://owasp.org/Top10/2025/

OWASP Top 10 2021 vs 2025

A01 Broken Access Control



Authentication & Authorization

Authentication

≠

Authorization

Authentication & Authorization

🔍 Processo mediante il quale un'applicazione verifica l'identità di un utente. In altre parole, risponde alla domanda: "Chi sei?"

🔑 Gli utenti devono dimostrare la loro identità, di solito attraverso l'inserimento di credenziali, come username e password.

🛠️ Esistono diversi metodi di autenticazione: Password, MFA, SSO, Biometrica, Certificato, etc.

Authentication

Authentication & Authorization

Authorization

🔍 Processo mediante il quale un'applicazione verifica cosa può fare l'utente. In altre parole, risponde alla domanda: "Cosa sei autorizzato a fare?"

🛡️ L'autorizzazione definisce quali risorse e funzionalità un utente può accedere o modificare.

🛠️ Esistono diverse politiche autorizzative: RBAC, ABAC, MAC, DAC, etc.

Broken Access Control — Tipologie

Vertical Access Control

User → Admin (privilegi superiori)

Horizontal Access Control

User A → User B (stesso livello)

Broken Access Control — Verticale

Vertical Access Control

🎯 Il controllo accessi di tipo «verticale» è un meccanismo che regola l'accesso a funzionalità dell'applicazione in base al profilo dell'utente autenticato.

👥 Utenti con profili differenti hanno accesso a funzionalità differenti. Per esempio un utente può avere il permesso di modificare o cancellare alcuni documenti mentre un altro utente può solamente visualizzare i documenti presenti a sistema.

⚙️ Ogni applicazione può avere regole estremamente personalizzate e dettagliate ed implementare principi come *separation of duties* o *least privilege*.

Broken Access Control — Verticale

Vertical
Access Control

Utente autenticato — ruolo: **user**

`GET /api/users/read_all_news`

✅ Risposta: `200 OK` — accesso alle notizie consentito

Broken Access Control — Verticale

Vertical
Access Control

Utente — ruolo: **user**

`GET /api/users/read_all_news`

✅

Attaccante — ruolo: **user** che tenta accesso da **redazione**

`POST /api/redazione/write_news`

🔓

⚠️ Se il controllo verticale è assente o rotto, l'utente user riesce a pubblicare notizie — funzionalità riservata alla redazione.

Broken Access Control — Orizzontale

Horizontal Access Control

🎯 Il controllo accessi di tipo «orizzontale» è un meccanismo che regola l'accesso a risorse dell'applicazione in base all'identità dell'utente autenticato.

👥 Utenti differenti con lo stesso profilo hanno accesso allo stesso insieme di funzionalità ma possono visualizzare dati differenti.

📁 Per esempio un utente può avere il permesso di caricare e modificare i documenti personali o dell'ufficio di appartenenza, ma non di eseguire le stesse operazioni su documenti di altri utenti o altri uffici.

Broken Access Control — Orizzontale

Horizontal Access Control

Utente autenticata — accede al **proprio** profilo

`GET /api/account/4df66f4/profile`

✅ Risposta: `200 OK` — profilo personale restituito correttamente

Broken Access Control — Orizzontale

Horizontal Access Control

Utente — accede al **proprio** profilo

`GET /api/account/4df66f4/profile`

✅

Attaccante — stesso ruolo, modifica l'ID per accedere al profilo **altrui**

`GET /api/account/8rt77p6/profile`

🔓

⚠️ Tecnica IDOR (Insecure Direct Object Reference): sostituendo l'ID nell'URL, l'attaccante accede ai dati di un altro utente con lo stesso livello di privilegi.

Testing — OWASP Testing Guide

4.5.1 Testing Directory Traversal File Include

4.5.2 Testing for Bypassing Authorization Schema

4.5.3 Testing for Privilege Escalation

4.5.4 Testing for Insecure Direct Object References

4.5.5 Testing for OAuth Weaknesses

- ↳ 4.5.5.1 Testing for OAuth Authorization Server Weaknesses
- ↳ 4.5.5.2 Testing for OAuth Client Weaknesses

Testing — OWASP Testing Guide

4.5.1 Testing Directory Traversal File Include

🔍 Verificare se l'applicazione permette di accedere a file o cartelle non autorizzate usando sequenze come `../` o `..\` o varianti con URL encoding.

🥷 L'attaccante tenta di "uscire" dalla directory prevista e accedere a file di sistema, configurazioni, o dati di altri utenti. Testare anche varianti come `..%2F` o sequenze multiple.

✅ Verificare che ci sia adeguata **sanitizzazione e validazione dei path**. Usare **whitelist** di file permessi piuttosto che blacklist di caratteri vietati.

Testing — OWASP Testing Guide

4.5.2 Testing for Bypassing Authorization Schema

🔓 Accesso non autenticato (forced browsing): è possibile accedere a pagine o funzioni dell'applicazione senza autenticazione conoscendo semplicemente la URL?

↔️ Accesso orizzontale: con un utente normale è possibile vedere o modificare dati di un altro utente dello stesso ruolo?

↕️ Accesso verticale: un utente normale può accedere a funzioni amministrative? Per testare efficacemente, creare utenti con ruoli diversi e provare sistematicamente ad accedere a risorse di altri ruoli.

Testing — OWASP Testing Guide

4.5.3 Testing for Privilege Escalation

Cercare di elevare i privilegi da utente normale ad amministratore. Tecniche comuni:

🎭 Manipolazione di parametri di ruolo negli header o nel body (es. `role=user` → `role=admin`)

🔑 Manipolazione di cookie, JWT token o campi nascosti contenenti informazioni di ruolo

🧪 Test con utenti di ruoli diversi tentando di accedere a funzioni riservate come `/admin/users`, `/admin/deleteUser`, `/admin/config`

❓ L'applicazione restituisce 403 Forbidden o esegue l'operazione? Verificare anche cambio ruolo, promozione utenti, gestione permessi.

Testing — OWASP Testing Guide

4.5.4 Testing for Insecure Direct Object References

🔍 Identificare le chiamate HTTP contenenti parametri che referenziano oggetti: ID nelle URL, nei form, nelle API. Tipicamente sono numeri sequenziali o UUID.

🥷 Creare risorse con un utente, leggere gli ID, poi con un altro utente tentare di accedere a quegli ID. L'applicazione lascia l'utente B vedere, modificare o cancellare le risorse dell'utente A?

👥 Il test per essere efficace deve essere effettuato con almeno due utenti.

Testing — OWASP Testing Guide

4.5.5 Testing for OAuth Weaknesses

1. Vulnerabilità dell'Authorization Server

Verificare la correttezza della redirect URI, la protezione contro CSRF sul flusso OAuth e la corretta validazione degli scope.

2. Vulnerabilità del Client

Il token viene salvato in modo sicuro? Viene trasmesso solo su HTTPS? Timeout e revoca sono gestiti correttamente? Gli scope richiesti rispettano il least privilege?

⚠️ OAuth è complesso e facile da sbagliare. Configurazioni errate possono permettere authorization code interception, token leakage, scope escalation. Se si usano provider OAuth di terze parti, non presumere che siano sicuri: testare comunque l'integrazione.

Tipologie di Vulnerabilità

🔓 Violazione del Least Privilege

Risorse accessibili a chiunque invece che solo agli utenti autorizzati

🔗 IDOR — Insecure Direct Object Reference

Accedere all'account altrui modificando un ID nella richiesta

🚀 Privilege Escalation

Agire come utente non autenticato, o ottenere privilegi admin senza averne diritto

🌐 CORS Misconfiguration

Configurazione errata che permette accesso API da origini non autorizzate

🔑 JWT / Metadata Manipulation

Replay o tampering di token JWT per elevare i propri privilegi

🗺️ Force Browsing

Accedere direttamente a URL privilegiati senza autenticazione

BOLA — Perché le API sono a rischio

Le API sono particolarmente vulnerabili perché:

- Il server **non traccia lo stato** del client
- Le decisioni di accesso si basano su **parametri inviati dal client** (object ID, VIN, documentId...)
- La risposta HTTP è spesso **sufficiente** per capire se l'attacco ha avuto successo

> BOLA ≠ BFLA
> In BOLA l'endpoint è accessibile, il problema è a livello di **oggetto**.
> In BFLA (API5) l'utente non dovrebbe accedere all'endpoint stesso.

Scenario reale

```
# Utente autenticato accede al proprio profilo
GET /api/v1/users/1337/profile
Authorization: Bearer eyJ...

# Attaccante prova ad accedere ad altri utenti
GET /api/v1/users/1338/profile  ← ID modificato
GET /api/v1/users/1/profile     ← Prova admin!
Authorization: Bearer eyJ...    ← Stesso token!
```

Rischi concreti:

- Data breach
- Manipolazione dati altrui
- Account takeover completo

Come Prevenire il BAC

✅ Lato Server

- **Deny by default**: nega tutto ciò che non è esplicitamente permesso
- Implementa i controlli di accesso **una sola volta** e riusali
- Valida che l'utente sia il proprietario della risorsa (**record ownership**)
- Centralizza la logica di autorizzazione (no duplicazioni!)

🔑 Gestione Token e Sessioni

- Invalida i session token **lato server** al logout
- Usa JWT **short-lived** (breve durata)
- Per JWT long-lived: usa **refresh token** con revoca OAuth2
- Non basarti mai su claim del token senza validarli

🌐 API & CORS

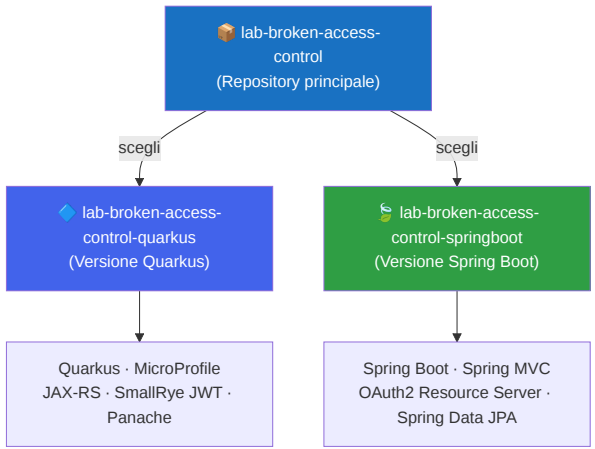
- Disabilita il **directory listing** del server
- Minimizza l'uso di CORS, configura allowed origins in modo restrittivo
- Applica **rate limiting** su endpoint API e controller
- Rimuovi backup e metadata (.git) dalla web root

🧪 Test & Monitoring

- Scrivi **unit e integration test** per il controllo accessi
- **Logga** ogni fallimento di accesso e avvisa gli admin
- Usa **GUID casuali** come ID degli oggetti (non ID sequenziali!)
- Includi test di autorizzazione nelle pipeline CI/CD

Il Laboratorio Pratico

Struttura del Laboratorio



Repository principale: github.com/lab-sca/lab-broken-access-control

Le 6 Vulnerabilità del Laboratorio

#	Tipo	Classificazione	Endpoint
(1)	ID Enumeration	IDOR	`GET /person/find/{id}`
(2)	Privilege Escalation — Dati	BOLA	`GET /doc/example.md` · `/doc/person/list`
(3)	Privilege Escalation — Azione	BOLA	`DELETE /doc/person/delete/{id}`
(4)	Broken Object Authorization	BOLA	`GET /doc/person/find/{id}`
(5)	Missing Authentication	Access Control	`GET /doc/example.md`
(X)	🎁 Hidden Vulnerability (BONUS)	Access Control	`PUT /person/add`

📄 File da modificare:
`DocResource.java` · `PersonResource.java` · `PersonRepository.java`

🧪 Test di riferimento:
`DocResourceSicurezzaTest.java`

💡 Ogni vulnerabilità fa fallire almeno un test. La **(2)** ne fa fallire 2. La **(X)** non è coperta dai test — trovala tu!

Vuln (1) — ID Enumeration

IDOR · `GET /person/find/{id}`

Vuln (1) — ID Enumeration ❌ Codice Vulnerabile

Il problema: restituendo `404` per ID inesistenti e `403` per ID esistenti ma non autorizzati, un attaccante può enumerare gli ID validi nel database.

```
GET /person/999   → 404 Not Found   ← questo ID non esiste
GET /person/10002 → 403 Forbidden   ← questo ID esiste! 😱
```

File: `PersonResource.java`

```java
@GET
@Path("/person/find/{id}")
@RolesAllowed({ "admin", "user" })
@Transactional
public Response findPerson(@PathParam("id") Long id) {
    Person person = this.personRepository.findById(id);
    if (person == null) {
        return Response.status(Response.Status.NOT_FOUND).build(); // ← rivela l'esistenza dell'oggetto!
    } else {
        return Response.status(Response.Status.OK).entity(person.toDTO()).build();
    }
}
```

🎯 **Tecnica di attacco:** iterare sugli ID e distinguere `404` da `403` per costruire una mappa completa degli oggetti esistenti nel database.

Vuln (1) — ID Enumeration ✅ Soluzione

La fix: restituire sempre `403 FORBIDDEN` indipendentemente dal fatto che l'oggetto esista o meno, rendendo impossibile l'enumerazione.

File: `PersonResource.java`

```java
@GET
@Path("/person/find/{id}")
@RolesAllowed({ "admin", "user" })
@Transactional
public Response findPerson(@PathParam("id") Long id) {
    Person person = this.personRepository.findById(id);
    if (person == null) {
        // SOLUTION (1): restituiamo FORBIDDEN invece di NOT_FOUND
        // per non rendere gli oggetti enumerabili
        return Response.status(Response.Status.FORBIDDEN).build();
    } else {
        return Response.status(Response.Status.OK).entity(person.toDTO()).build();
    }
}
```

✅ **Principio applicato:** un utente non autorizzato non deve sapere se una risorsa esiste o meno — la risposta deve essere identica in entrambi i casi.

Vuln (2) — Privilege Escalation (Data)

BOLA · `GET /doc/person/list`

Vuln (2) — Privilege Escalation (Data) ❌ Codice Vulnerabile

Il problema: `findByRolesOrderedByName()` riceve i ruoli dell'utente come parametro ma **li ignora completamente nella query**, restituendo tutti i record senza filtro.

File: `PersonRepository.java`

```
/**
 * Restituisce l'elenco delle persone ordinate per cognome e nome,
 * filtrate per MIN_ROLE (NULL oppure presente nella collection di ruoli fornita)
 */
public List<Person> findByRolesOrderedByName(Collection<String> roles) {
    // Il parametro 'roles' è ricevuto ma completamente ignorato!
    return find("order by lastName, firstName").list();
}
```

🎯 **Effetto:** un utente con ruolo `user` vede anche le persone con `minRole=admin` (es. Richard Feynman).
⚠️ Questa vulnerabilità fa fallire **2 casi di test**: uno per il formato MD e uno per HTML.

Vuln (2) — Privilege Escalation (Data) ✅ Soluzione

La fix: usare il parametro `roles` come filtro nella query Panache, includendo solo le persone con `minRole` nullo (visibili a tutti) o corrispondente ai ruoli dell'utente.

File: `PersonRepository.java`

```java
/**
 * Restituisce l'elenco delle persone ordinate per cognome e nome,
 * filtrate per MIN_ROLE (NULL oppure presente nella collection di ruoli fornita)
 */
public List<Person> findByRolesOrderedByName(Collection<String> roles) {
    // SOLUTION (2): usiamo 'roles' come filtro nella query.
    // minRole null = visibile a tutti i ruoli autenticati
    return find("minRole is null or minRole in ?1 order by lastName, firstName", roles).list();
}
```

✅ **Principio applicato:** il filtraggio dei dati in base ai ruoli deve avvenire **lato server nella query**, non affidarsi al client o a logica applicativa successiva.

Vuln (3) — Privilege Escalation (Action)

BOLA · `DELETE /doc/person/delete/{id}`

Vuln (3) — Privilege Escalation (Action) ❌ Codice Vulnerabile

Il problema: `@RolesAllowed` include erroneamente il ruolo `user`, permettendo a qualsiasi utente autenticato di **cancellare** persone — operazione che dovrebbe essere riservata solo agli `admin`.

File: `DocResource.java`

```
1   @DELETE
2   @Path("/person/delete/{id}")
3   @RolesAllowed({ "admin", "user" }) // ← 'user' non dovrebbe poter cancellare!
4   @Transactional
5   public Response deletePerson(@PathParam("id") Long id) {
6       Person person = this.personRepository.findById(id);
7       if (person == null) {
8           return Response.status(Response.Status.FORBIDDEN).build();
9       }
10      person.delete();
11      return Response.status(Response.Status.OK).build();
12  }
```

🎯 **Principio violato:** Least Privilege — un utente ha ottenuto più permessi di quelli necessari al suo ruolo, probabilmente per un errore di copia-incolla dell'annotazione.

Vuln (3) — Privilege Escalation (Action) ✅ Soluzione

La fix: rimuovere `"user"` da `@RolesAllowed`, lasciando solo `"admin"` come da specifiche.

File: `DocResource.java`

```java
@DELETE
@Path("/person/delete/{id}")
// SOLUTION (3): rimuoviamo il ruolo 'user'.
// Secondo le specifiche, la cancellazione è consentita solo ad 'admin'
@RolesAllowed({ "admin" })
@Transactional
public Response deletePerson(@PathParam("id") Long id) {
    Person person = this.personRepository.findById(id);
    if (person == null) {
        return Response.status(Response.Status.FORBIDDEN).build();
    }
    person.delete();
    return Response.status(Response.Status.OK).build();
}
```

✅ **Buona pratica:** ogni operazione distruttiva (DELETE, modifica dati critici) dovrebbe richiedere una revisione esplicita dei ruoli autorizzati, separata dalla logica di lettura.

Vuln (4) — Broken Object Authorization

BOLA · `GET /doc/person/find/{id}`

Vuln (4) — Broken Object Authorization ❌ Codice Vulnerabile

Il problema: anche se l'utente è autenticato e autorizzato all'endpoint, non viene verificato se possiede il ruolo minimo richiesto **dalla singola risorsa** (`person.getMinRole()`). Contiene anche la vuln (1).

File: `DocResource.java`

```java
@GET
@Path("/person/find/{id}")
@RolesAllowed({ "admin", "user" })
@Transactional
public Response findPerson(@PathParam("id") Long id) {
    Person person = this.personRepository.findById(id);
    if (person == null) {
        return Response.status(Response.Status.NOT_FOUND).build(); // ← vuln (1): enumerable!
    } else {
        // Nessun controllo su person.getMinRole() vs ruoli dell'utente!
        return Response.status(Response.Status.OK).entity(person.toDTO()).build();
    }
}
```

🎯 **Effetto:** un utente `user` può leggere dati di persone con `minRole=admin` conoscendone l'ID diretto, anche se la lista è filtrata correttamente (vuln 2).

Vuln (4) — Broken Object Authorization ✅ Soluzione

La fix: verificare che i ruoli dell'utente autenticato contengano il `minRole` richiesto dalla risorsa. Questa soluzione ingloba anche la fix della vuln (1).

File: `DocResource.java`

```java
@GET
@Path("/person/find/{id}")
@RolesAllowed({ "admin", "user" })
@Transactional
public Response findPerson(@PathParam("id") Long id) {
    Person person = this.personRepository.findById(id);
    if (person == null) {
        // SOLUTION (1): FORBIDDEN invece di NOT_FOUND — oggetti non enumerabili
        return Response.status(Response.Status.FORBIDDEN).build();
    } else {
        // SOLUTION (4): verifica se l'utente ha il ruolo minimo richiesto dalla risorsa
        if (person.getMinRole() == null || this.securityIdentity.getRoles().contains(person.getMinRole())) {
            return Response.status(Response.Status.OK).entity(person.toDTO()).build();
        } else {
            return Response.status(Response.Status.FORBIDDEN).build();
        }
    }
}
```

Vuln (5) — Missing Authentication

Access Control · `GET /doc/example.md`

Vuln (5) — Missing Authentication ❌ Codice Vulnerabile

Il problema: il metodo ha `@SecurityRequirement` (visibilità Swagger) ma **manca completamente di** `@RolesAllowed`, rendendolo accessibile senza alcuna autenticazione.

File: `DocResource.java`

```
1   @GET
2   @Path("/example.md")
3   @SecurityRequirement(name = "SecurityScheme")
4   // @RolesAllowed mancante!
5   public Response markdownExample() throws IOException {
6       return Response.status(Response.Status.OK)
7                      .entity(processDocument(DocConfig.TYPE_MD))
8                      .build();
9   }
```

⚠️ **Errore comune:** confondere `@SecurityRequirement` (documentazione OpenAPI) con `@RolesAllowed` (controllo di sicurezza reale). Il primo non protegge nulla.

Vuln (5) — Missing Authentication ✅ Soluzione

La fix: aggiungere `@RolesAllowed` con i ruoli previsti dalle specifiche. Il ruolo `guest` è il minimo richiesto per accedere al documento.

File: `DocResource.java`

```
1   @GET
2   @Path("/example.md")
3   @SecurityRequirement(name = "SecurityScheme")
4   // SOLUTION (5): aggiungiamo i ruoli autorizzati previsti dalle specifiche.
5   // 'guest' è il ruolo minimo — il path non è ad accesso pubblico
6   @RolesAllowed({ "admin", "user", "guest" })
7   public Response markdownExample() throws IOException {
8       return Response.status(Response.Status.OK)
9                      .entity(processDocument(DocConfig.TYPE_MD))
10                     .build();
11  }
```

✅ **Regola generale:** ogni endpoint deve avere esplicitamente `@RolesAllowed`. Se un endpoint deve essere pubblico, va dichiarato esplicitamente con `@PermitAll`, non semplicemente omettendo l'annotazione.

Vuln (5) — Missing Authentication 💡 Spring Boot

La soluzione varia a seconda del framework. In Spring Boot abbiamo usato UN `SecurityFilterChain` per gestire i path permessi a tutti.

Versione Spring Boot — `SecurityConfig.java` ❌ Vulnerabile

```
1   .authorizeHttpRequests(auth → auth
2       .requestMatchers("/demo/**").permitAll()
3       .requestMatchers("/h2-console/**").permitAll()
4       .requestMatchers("/swagger-ui/**", "/swagger-ui.html", "/v3/api-docs/**").permitAll()
5       .requestMatchers("/actuator/health").permitAll()
6       .requestMatchers("/doc/example.md").permitAll()  // ← chiunque può accedere!
7       .anyRequest().authenticated()
8   )
```

🎯 **Effetto:** l'endpoint `/doc/example.md` è dichiarato esplicitamente come pubblico nel `SecurityFilterChain` — qualsiasi richiesta, anche non autenticata, viene accettata.

Vuln (5) — Missing Authentication 💡 Spring Boot ✅

La fix: rimuovere la riga `permitAll()` per `/doc/example.md` — l'endpoint ricadrà automaticamente nella regola `anyRequest().authenticated()`.

Versione Spring Boot — `SecurityConfig.java` ✅ Soluzione

```
  .authorizeHttpRequests(auth → auth
      .requestMatchers("/demo/**").permitAll()
      .requestMatchers("/h2-console/**").permitAll()
      .requestMatchers("/swagger-ui/**", "/swagger-ui.html", "/v3/api-docs/**").permitAll()
      .requestMatchers("/actuator/health").permitAll()
      // SOLUTION (5): rimosso /doc/example.md — ora ricade in anyRequest().authenticated()
      .anyRequest().authenticated()
  )
```

✅ **Principio generale:** indipendentemente dal framework, la regola è sempre **deny by default**. Ogni endpoint pubblico dovrebbe essere una eccezione esplicita e consapevole, non il comportamento di default.

Vuln (X) — Hidden Vulnerability

BONUS · `PUT /person/add`

Vuln (X) — Hidden Vulnerability 🎁 ❌ Codice Vulnerabile

Il problema: un metodo `PUT` alternativo per aggiungere persone è rimasto attivo **senza alcun controllo di autenticazione o autorizzazione**. Non è censito nei test — va trovato analizzando il codice.

File: `PersonResource.java`

```
1   @APIResponse(responseCode = "201", description = "La persona è stata creata")
2   @APIResponse(responseCode = "401", description = "Se l'autenticazione non è presente")
3   @APIResponse(responseCode = "403", description = "Se l'utente non è autorizzato per la risorsa")
4   @Tag(name = "person")
5   @Operation(operationId = "addPersonPut", summary = "Aggiunge una persona al database (ruoli: admin)")
6   @PUT
7   @Path("/person/add")
8   @Transactional
9   // Nessun @RolesAllowed — nessun @SecurityRequirement
10  // Chiunque, anche non autenticato, può aggiungere persone!
11  public Response addPersonPut(AddPersonRequestDTO request) {
12      return this.addPerson(request);
13  }
```

🎁 **Nota:** persino la documentazione OpenAPI dichiara `401` e `403` come risposte possibili… ma senza `@RolesAllowed` non verranno mai restituite!

Vuln (X) — Hidden Vulnerability 🎁 ✅ Soluzione

La fix: rimuovere completamente il metodo. Esiste già `addPersonPost()` con i controlli di sicurezza appropriati — `addPersonPut()` è un duplicato dimenticato.

File: `PersonResource.java`

```
1   // SOLUTION (X): una PUT senza controllo di autorizzazione
2   // è rimasta abilitata per errore.
3   //
4   // La soluzione corretta è rimuovere totalmente il metodo addPersonPut().
5   // Esiste già addPersonPost() con @RolesAllowed({ "admin" }) che svolge
6   // la stessa funzione in modo sicuro.
7   //
8   // ← metodo rimosso
```

🎁 **Lezione:** le API dimenticate o duplicate sono tra le vulnerabilità più insidiose — difficili da trovare senza una revisione sistematica del codice o un inventario degli endpoint. Strumenti come Swagger UI o test di superficie API aiutano a scoprirle.

Approccio TDD: prima i test, poi il codice

Il laboratorio segue il **Test-Driven Development**: i test di sicurezza sono scritti prima e **falliscono** finché le vulnerabilità non vengono corrette.

❌ Prima della fix

```
mvn test
```

```
Tests run: 11, Failures: 6

FAIL testFindPersonKoNotFound
FAIL testOkMarkDownConVerificaContenutoUser
FAIL testListPersonsResultKo
FAIL testDeletePersonUserKo
FAIL testFindPersonKoForbidden
FAIL testMarkdown401NoAuthorizationBearer
```

✅ Dopo la fix

```
mvn test
```

```
Tests run: 11, Failures: 0

PASS testFindPersonKoNotFound
PASS testOkMarkDownConVerificaContenutoUser
PASS testListPersonsResultKo
PASS testDeletePersonUserKo
PASS testFindPersonKoForbidden
PASS testMarkdown401NoAuthorizationBearer
```

Come Affrontare il Laboratorio

1️⃣

Setup

Clona il repo e avvia con `mvn quarkus:dev`. Esegui i test: vedrai 6 fallimenti.

2️⃣

Analizza & Attacca

Cerca i commenti `// VULNERABILITY: (n)`. Prova a sfruttare ogni falla con curl o Swagger UI.

3️⃣

Correggi & Verifica

Applica le fix una alla volta. Riesegui i test finché tutti e 11 passano. Poi cerca la (X)!

File da modificare

`PersonResource.java` — vuln 1, X
`PersonRepository.java` — vuln 2
`DocResource.java` — vuln 3, 4, 5

Come verificare le soluzioni

Cerca: `// VULNERABILITY: (n)` nel codice
Confronta con il branch `solution`
oppure cerca: `// SOLUTION: (n)`

Riferimenti e Risorse

OWASP

- OWASP Top 10:2025 — A01 Broken Access Control
- OWASP API Security 2023 — API1 BOLA
- OWASP Authorization Cheat Sheet
- OWASP Proactive Controls: C1 Access Control
- OWASP ASVS V8 Authorization

Laboratorio

- 📦 Repository principale
- 🔷 Lab Quarkus
- 🍃 Lab Spring Boot

Strumenti Utili

- Fugerit Venus Doc
- PortSwigger Web Academy — Access Control
- OAuth 2.0 — Revoking Access

Buon Laboratorio! 🚀

Domande? Apri una issue su **GitHub**

🔗 github.com/lab-sca/lab-broken-access-control

Basato su OWASP Top 10:2025 · OWASP API Security Top 10:2023 · Licenza MIT

lab broken accesso control

github.com/lab-sca/lab-broken-access-control

Questions & Answers

"I'm not a great programmer; I'm just a good programmer with great habits." — Kent Beck

❓ Come possiamo assicurarci che l'applicazione sia stata testata correttamente?

Una buona suite di test non si misura solo dal fatto che i test passino, ma da **quanto codice viene effettivamente esercitato** e da **quanto i test rispecchiano scenari reali** di utilizzo — inclusi quelli legati alla sicurezza. Le slide seguenti approfondiscono due pratiche fondamentali per rispondere a questa domanda.

💡 A tool di analisi statica della sicurezza come **Checkmarx** è possibile affiancare strumenti come **SonarQube** che monitorano la qualità e la coverage del codice — due prospettive complementari per una strategia di analisi completa.

Unit Test & Code Coverage

*"Testing shows the presence, not the absence, of bugs." — **Edsger W. Dijkstra***

🧪 Code Coverage: misura la percentuale di codice sorgente eseguita durante i test. Gli unit test verificano che il codice funzioni correttamente e possono testare anche aspetti come la sicurezza — i test di questo laboratorio ne sono un esempio.

⚠️ Attenzione: una coverage bassa aumenta la probabilità di bug e regressioni. Codice mai eseguito durante i test è codice di cui non conosciamo il comportamento — un endpoint dimenticato o un controllo mancante potrebbero nascondere vulnerabilità come la Vuln (X) del laboratorio.

✅ Buona pratica: integrare la misurazione della coverage nella pipeline CI/CD e definire una soglia minima al di sotto della quale la build fallisce.

JaCoCo

Strumento di coverage per Java, si integra con Maven e Gradle. Genera report HTML e XML.

SonarQube

Piattaforma di analisi statica del codice. Aggrega coverage, code smells, vulnerabilità e security hotspot in un'unica dashboard.

Integration Test

*"Il tutto è maggiore della somma delle sue parti." — **Aristotele***

🔗 **Integration Test**: verificano che i diversi componenti del sistema funzionino correttamente **insieme**. A differenza degli unit test, testano il comportamento dell'applicazione end-to-end — inclusi database, API e controlli di sicurezza.

⚠️ **Attenzione:** un componente può funzionare correttamente in isolamento ma fallire quando integrato con gli altri. I controlli di sicurezza come autenticazione e autorizzazione emergono spesso solo a livello di integrazione — i test di questo laboratorio ne sono un esempio concreto.

✅ **Buona pratica:** affiancare gli unit test con integration test che verifichino i flussi critici, in particolare quelli legati a **autenticazione, autorizzazione e gestione dei dati sensibili**.

🔷 Quarkus Test

`@QuarkusTest` + JUnit 5 + RestAssured — avvia l'intera applicazione in modalità test, inclusa la sicurezza.

🍃 Spring Boot Test

`@SpringBootTest` + JUnit 5 + MockMvc — carica il contesto completo, incluso il `SecurityFilterChain`.