



University of Calabria
A.A. 2025/2026

CYBER OFFENSE AND DEFENSE

FINAL PROJECT

NoSQL injection

Presented by

Frandina Ilaria 264232 - Siciliano Samuele 263633 - Villella Luigi 269404

Date

December 2025



Overview sui database NoSQL

> I DATABASE NOSQL

I database NoSQL (MongoDB, Redis, CouchDB, Cassandra) rappresentano un approccio diverso alla gestione dei dati, nato per rispondere alle esigenze del web moderno e delle applicazioni distribuite.

> SQL VS NOSQL

A differenza dei database relazionali, i NoSQL utilizzano **schemi flessibili** e memorizzano i dati in formati come **JSON**. Le query non sono più stringhe SQL ma oggetti strutturati che manipolano documenti, chiavi-valori o grafi.

> UTILIZZO E DIFFUSIONE

La loro forza sta nella **scalabilità orizzontale** e nelle **performance elevate**, che li rendono ideali per big data, applicazioni real-time e architetture cloud-native. Oggi li troviamo ovunque: e-commerce, social network, IoT, sistemi di analytics.



NoSQL injection

➤ COME FUNZIONA NOSQL INJECTION

Invece di manipolare stringhe SQL, l'attaccante sfrutta gli **operatori speciali del linguaggio di query**. In MongoDB, alcune delle varianti più comuni per effettuare delle injection, sono le seguenti:

\$ne (Not Equal) - Bypass Autenticazione

Cerca utente admin con password diversa da null, se admin esiste restituirà sempre un risultato.

```
{"username": "admin", "password": {"$ne": null}}
```

\$where - Esecuzione codice JavaScript

Esegue JavaScript arbitrario all'interno del database. Con `|| 1==1` la where perde di valore.

```
{"$where": "this.password == 'x' || '1'=='1'"}
```

\$regex - Esecuzione di Regex

Cerca le password che iniziano per "a". È utile per ricostruire la password, carattere per carattere.

```
{"password": {"$regex": "^a"}}
```

Quando l'input utente **non viene validato correttamente**, un attaccante può: **autenticarsi senza credenziali valide, estrarre dati sensibili dal database**, o nei casi peggiori **eseguire codice arbitrario** sul server. Tuttavia l'input validation non è un problema di SQL (o NoSQL), è un problema di sicurezza generale. È necessario **non fidarsi mai dell'input utente**, qualunque sia il database utilizzato.



1° - Exploiting NoSQL operator injection to bypass authentication

Soluzione via script

➤ **TEST INIZIALI** Abbiamo testato se l'applicazione fosse vulnerabile a NoSQL injection provando a bypassare l'autenticazione attraverso questi due step:

01

Tentativo di entrare senza conoscere lo username usando l'operatore \$gt
`{"username": {"$gt": ""}, "password": "peter"}`

02

Tentativo di entrare senza conoscere la password usando l'operatore \$gt
`{"username": "wiener", "password": {"$gt": ""}}`

➤ **SOLUZIONE** Dopo aver verificato la vulnerabilità dell'applicazione, abbiamo costruito il payload finale per concludere l'exploit. Utilizzando l'operatore \$gt (greater than) nel campo password per bypassare il controllo di autenticazione (poiché qualsiasi password nel database sarà maggiore di una stringa vuota, rendendo la condizione sempre vera) e l'operatore \$regex per cercare un account il cui nome inizia con "admin"

03

Tentativo di entrare cercando qualsiasi username che inizia per "admin" utilizzando l'operatore \$regex e bypassando il controllo della password usando l'operatore \$gt
`{"username": { "$regex": "admin.*"}, "password": {"$gt": ""}}`



2° - Exploiting NoSQL injection to extract data

Soluzione via script

- > **TEST INIZIALI** Abbiamo testato se l'applicazione fosse vulnerabile verificando se eseguisse codice JavaScript iniettato attraverso **\$where** aggiungendo una condizione OR sempre vera alla GET di lookup:
administrator' || '1'=='1

- > **SOLUZIONE** Dopo aver verificato la vulnerabilità dell'API di lookup, abbiamo effettuato i seguenti step per ricostruire la password dell'account administrator.
 - 01** Ricerca del range di lunghezza testando a blocchi di 10 (1-10, 10-20 etc.) fino a trovare il range corretto
administrator' & this.password.length < 10 || 'a'=='b

 - 02** Determinazione della lunghezza esatta testando ogni valore nel range trovato
administrator' & this.password.length == 8 || 'a'=='b

 - 03** Estrazione carattere per carattere della password usando l'operatore di accesso agli array e iterando su tutto l'alfabeto per ogni posizione
administrator' & this.password[0] == 'a' || 'a'=='b

- > La condizione finale **|| 'a'=='b** aggiunta ad ogni richiesta, serve per bilanciare gli apici della query originale.
ORIGINALE: db.users.find({ \$where: "this.username == 'USER_INPUT'" })
CON INJECTION: db.users.find({ \$where: "this.username == 'administrator' & CONDIZIONE || 'a'=='b'" })



3° - Exploiting NoSQL operator injection to extract unknown fields

Soluzione via script

➤ **TEST INIZIALI** Abbiamo verificato la vulnerabilità iniettando JavaScript (**\$where**) nel login. Analizzando il traffico con ZAP, abbiamo notato comportamenti distinti:

1. Su **utenti attivi** (es. wiener), il bypass della password garantisce l'accesso immediato.
2. Sull'utente vittima Carlos, essendo il suo **account disabilitato**, il bypass restituisce l'errore "Account locked". Questo messaggio diventa il nostro Oracolo Booleano.

➤ **SOLUZIONE** Lo script automatizza l'attacco su Carlos. Per forzare l'esecuzione della clausola **\$where** nel database, bypassiamo prima il controllo della password usando l'operatore **\$ne** (not equal):
{ "username": "carlos", "password": { "\$ne": "invalid"}, "\$where": "..." }

01

Lo script itera su Object.keys(this). Per ogni indice, ricostruisce il nome del campo carattere per carattere, verificando ogni tentativo tramite regex contro l'oracolo.

Object.keys(this)[i].match("^parziale.*")

02

Individuato il campo target, lo script ne estrae il valore (il token). Ogni carattere viene indovinato testando se la regex **^tokenParziale.*** genera l'errore "Account locked".

this.nomeCampoTrovato.match("^tokenParziale.*")



Prevention

> LINEE GUIDA GENERALI

Le contromisure variano in base alla specifica tecnologia NoSQL utilizzata, ma è possibile seguire alcune linee guida generali per mitigare il rischio di injection.

01 VALIDAZIONE E SANITIZZAZIONE DELL'INPUT

Implementare una whitelist di caratteri consentiti e rifiutare qualsiasi input che contenga caratteri o pattern non previsti, validando sempre formato e tipo di dato.

02 QUERY PARAMETRIZZATE

Evitare la concatenazione diretta dell'input utente nelle query, utilizzando invece le funzionalità di binding dei parametri offerte dal database per separare la logica della query dai dati forniti dall'utente.

03 CONTROLLO DEGLI OPERATORI

Applicare una whitelist di operatori consentiti e bloccare operatori pericolosi come **\$where** o **\$regex** quando non strettamente necessari, validando la struttura degli oggetti JSON prima di processarli.



University of Calabria
A.A. 2025/2026

GRAZIE PER L'ATTENZIONE

Presented by

Frandina Ilaria 264232 - Siciliano Samuele 263633 - Villella Luigi 269404

Date

December 2025