NoSQL INJECTION

1. CORE CONCEPTS

NoSQL Injection is the manipulation of queries in non-relational databases (like MongoDB, CouchDB) to bypass auth, steal data, or execute code.
The Shift from SQL:

- **SQL:** Queries are Strings. Vulnerability comes from unescaped characters (e.g., `' OR 1=1`).
- **NoSQL:** Queries are often **JSON Objects**. Vulnerability comes from injecting malicious **Operators** (Keys) or executing **JavaScript** logic.

---

2. ATTACK VECTORS

There are two primary distinct vectors in NoSQLi.

A. Syntax Injection (The "Classic" Approach)

Similar to SQLi. You break the query structure because the application constructs queries by concatenating strings.

- **Goal:** Break out of the intended query syntax and inject your own.
- **Detection:** Fuzz inputs with characters that have special meaning in the target language.
- **MongoDB Fuzz List:** `' "` `{ } ; $ \0`

B. Operator Injection (The "New" Approach)

Unique to NoSQL. You inject query *operators* (JSON keys) to alter the condition logic.

- **Goal:** Change a simple value check (`username="admin"`) into a logic check (`username={"$ne": "invalid"}`).
- **Mechanism:** If the app accepts JSON or URL parameters blindly, it treats your injected object as part of the query logic.

---

3. SYNTAX INJECTION (Methodology)

Phase 1: Detection

Submit fuzz strings to trigger syntax errors or behavioral changes.

- **Payload:** `'`
  - *Result:* Server returns 500 Error (Syntax broken).
- **Payload:** `\'` (Escaped)
  - *Result:* Server returns 200 OK (Syntax repaired).
  - *Conclusion:* Vulnerable.

Phase 2: Boolean Inference

Test if you can control the logic (True vs. False).

- **False Test:** `fizzy' && 0 && 'x` -> No results.
- **True Test:** `fizzy' && 1 && 'x` -> Returns results.
- **Bypass:** `fizzy' || '1'=='1` -> Returns **ALL** results (Standard "OR 1=1" bypass).

Phase 3: The Null Byte

MongoDB often stops parsing after a Null Byte.

- **Scenario:** Query checks `category == 'fizzy' && released == 1`.
- **Attack:** Input `fizzy'\0`.
- **Result:** `category == 'fizzy'\0' && ...` (The "released" check is ignored).

---

4. OPERATOR INJECTION (Methodology)

This is the primary way to bypass authentication in NoSQL.

Common Operators

- `$ne`: Not Equal
- `$in`: Matches any value in an array
- `$regex`: Matches a regular expression
- `$where`: Executes arbitrary JavaScript

Authentication Bypass

- **Target:** A login form expecting `{"user": "u", "pass": "p"}`.
- **Attack:** Inject `$ne` (Not Equal).
- **Payload:**

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

- **Logic:** "Find user 'admin' where password is NOT empty." -> **True**. Access Granted.
- **Blind Bypass:**

```
{"username": {"$ne": "invalid"}, "password": {"$ne": "invalid"}}
```

- *Result:* Logs you in as the *first* user in the database (often Admin).

Injection via URL

If the app reads URL parameters into the query object:

- `user=admin&password[$ne]=invalid`

---

5. DATA EXFILTRATION (JavaScript & Regex)

If the database uses `$where` or `mapReduce`, it evaluates JavaScript. This allows for complex extraction.

A. JavaScript Injection ($where)

- **Concept:** Similar to Blind SQLi. Ask the DB questions via JS code.
- **Payload (Boolean Check):**

```
admin' && this.password[0] == 'a' || 'a'=='b
```

- **Field Enumeration:**
  - Use `Object.keys(this)` to find hidden fields.
  - Payload: `"$where": "Object.keys(this)[0].match('^token')"`

B. Regex Extraction (Non-JS)

If JS is disabled, use `$regex` to blindly guess data character by character.

- **Payload:**

```
{"username": "admin", "password": {"$regex": "^a.*"}}
```

- *If True:* Password starts with 'a'.

- *If False:* Password does not start with 'a'. Try 'b'.

C. Timing Attacks

If errors are suppressed, use time delays to confirm logic.

- **Payload:**

```
admin' + function(x){ if(x.password[0]==='a'){ sleep(5000) }; }(this) + '
```

---

6. MITIGATION

1. **Sanitization:** Validate input against an allowlist (e.g., Alphanumeric only).
2. **No String Concatenation:** Never build NoSQL queries by concatenating strings.
3. **Type Checking:** Ensure input intended to be a String is actually a String, not an Object/JSON.
   - *Defense:* If `req.body.password` is an object, reject the request.
4. **Disable Scripting:** Disable JavaScript execution (e.g., `security.javascriptEnabled = false` in MongoDB) if not strictly necessary.