

# Java Spring / Spring Boot Security

Spring Boot Application:

AppSecSpringSecurityPart2

# Agenda

- [Overview](#)
- [SQL Injection Vulnerability & Prevention – Named Parameters](#)
- [SQL Injection – Spring Data JPA Methods](#)

# Overview

# Overview

- The purpose of this document is to give a “walkthrough” of the security concepts configured in the AppSecSpringSecurityPart2 Spring Boot Project.
- The main security related concepts implemented in this project are:
  - SQL Injection
- Each Class/method in the Spring Boot project contains comments for each functionality.
- This document will have screenshots of the main ideas for this project, so it is not required to run it locally to see how it works

# How to download GitHub repo and import project into Eclipse (Optional)

- **Download GitHub repository:**
- Go to the repository -> Click on the “Code” drop down button (next to the “About” section) -> Choose “Download ZIP”
- **Import the code into Eclipse:**
- Open Eclipse IDE
- Ensure Spring Tool Suite is installed. Click on – Help -> Eclipse Market Place -> Search for “Spring Tools Suite” -> Select & Install “Spring Tools 4 (aka Spring Tool Suite 4)”
- Click on - File -> Import -> Maven -> Existing Maven Projects
- In the “Root Directory” section, browse to the folder that holds the code for the Spring Boot project
- Click on Finish
- Now the project should be available on the Eclipse IDE UI, and we can run it as a “Spring Boot App”.

# SQL Injection – Named Parameters

# SQL Injection

- In Java, when using prepared statements to prevent SQL injection, we need to use the correct place holders ( ? ) for each parameter in the query.
- Example:
  - `String sqlString = "select * from db_user where username = ? and password = ?";`
- Both the username and password parameter's arguments contain the ? character, which will prevent SQL injection vulnerability if a malicious value is injected into the query.
- If these placeholders are not used correctly then the query is vulnerable to SQL injection.
- Reference: <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection>

# SQL Injection

- In Spring Boot, it is common to use the NamedParameterJdbcTemplate Class to execute SQL queries.
- For prepared statements here we need to use different type of placeholder (:paramName).
- Example:
  - String sql = "select \* from school where name = **:name**"
- Named parameters help to provide more clarity by replacing ? characters with actual names for each argument.
- If these placeholders are not used correctly then the query is vulnerable to SQL injection.
- Examples of the secure/insecure use of these named parameters will be on next couple slides.



## Secure Code

- The “findAllByName1” method is using the correct implementation of the Named Parameters class (line->40) by using the correct place holders -> :paramName
- This will prevent an SQL injection on the query
- The following endpoint will be calling to this secure method:
- /getAllSchoolsByNameDAO1

```
37     @Override
38     public List<School> findAllByName1(String name) {
39
40         String sql = "select * from school where name = :name";
41
42         Map<String, String> parameters = new HashMap<String, String>();
43         parameters.put("name", name);
44
45         return namedParameterJdbcTemplate.query(sql, parameters, new UserMapper() );
46
47     } // end findAllByName1
48
```

```
57     */
58     @GetMapping("/getAllSchoolsByNameDAO1/{SchoolName}")
59     public List<School> getOneSchoolByNameDAO1(@PathVariable("SchoolName") String name) {
60
61         return dao.findAllByName1(name);
62
63     } // end getOneSchoolByNameDAO1
64
```

# Secure

## Postman Example:

- After creating 2 School Objects and saving it in the database using the POST request, use the endpoint that is using the secure named parameter implementation to retrieve the record for:
  - AUTHOR
- Now inject the following value:
  - Test' or 1=1--
- When injecting a malicious SQL payload to view all the records of the current table, we get an empty list since the value of "Test' or 1=1--" does not exist in the table.
- SQL injection is prevented.

GET http://localhost:9091/schoolApi/getAllSchoolsByNameDAO1/AUTHOR

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 2,
4     "name": "AUTHOR",
5     "rooms": 20,
6     "students": 100
7   }
8 ]
```

GET http://localhost:9091/schoolApi/getAllSchoolsByNameDAO1/Test' or 1=1--

Params Authorization Headers (6) Body Pre-request Script Tests Settings

☒ none ☐ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [ ]
```

This request failed

## Insecure Code

- The “findAllByName2” method is not using the correct implementation of the Named Parameters class (line->64).
- Instead, here the String format method is being used, which is not secure and vulnerable to SQL injection.
- The following endpoint will be calling to this insecure method:
- /getAllSchoolsByNameDAO2

```
61  @Override
62  public List<School> findAllByName2(String name) {
63
64      String sql = String.format("select * from school where name = '%s'", name);
65
66      Map<String, String> parameters = new HashMap<String, String>();
67      parameters.put("name", name);
68
69      return namedParameterJdbcTemplate.query(sql, parameters, new UserMapper() );
70
71  } // end findAllByName2
```

```
69  /
70  @GetMapping("/getAllSchoolsByNameDAO2/{SchoolName}")
71  public List<School> getOneSchoolByNameDAO2(@PathVariable("SchoolName") String name) {
72
73      return dao.findAllByName2(name);
74
75  } // end getOneSchoolByNameDAO2
76
```

## Insecure

### Postman Example:

- After creating 2 School Objects and saving it in the database using the POST request, use the endpoint that is using the insecure named parameter implementation to retrieve the record for:
  - AUTHOR
- Only 1 record for that name is returned in the response.

GET http://localhost:9091/schoolApi/getAllSchoolsByNameDAO2/AUTHOR

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 2,
4     "name": "AUTHOR",
5     "rooms": 20,
6     "students": 100
7   }
8 ]
```

- Now inject the following value:
  - Test' or 1=1--
- When injecting a malicious SQL payload to view all the records of the current table, this time we do get all the information stored in the current table.
- SQL injection is executed.
- Final query:
- select \* from school where name = 'Test' or 1=1--'

GET http://localhost:9091/schoolApi/getAllSchoolsByNameDAO2/Test' or 1=1--

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 1,
4     "name": "ADMIN",
5     "rooms": 20,
6     "students": 100
7   },
8   {
9     "id": 2,
10    "name": "AUTHOR",
11    "rooms": 20,
12    "students": 100
13  }
14 ]
```

### More tests

- When injecting a single quote to the vulnerable endpoint/field, a 500 Internal Server Error is returned.

GET

▼

http://localhost:9091/schoolApi/getAllSchoolsByNameDAO2/Test"

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settin

Query Params

	KEY	VALUE
--	-----	-------

Body

Cookies

Headers (5)

Test Results

Pretty

Raw

Preview

Visualize

JSON ▼

↻

1

[ ]

GET

▼

http://localhost:9091/schoolApi/getAllSchoolsByNameDAO2/Test'

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Set

Query Params

	KEY	VALUE
--	-----	-------

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

Visualize

JSON ▼

↻

1

{

2

"timestamp": "2022-11-22T22:59:36.726+00:00",

3

"status": 500,

4

"error": "Internal Server Error",

5

"path": "/schoolApi/getAllSchoolsByNameDAO2/Test'"

6

}

- When injecting 2 single quotes to the vulnerable endpoint/field, an empty list is returned, and error message is gone. Behavior is highly likely for SQL injection vulnerability.

# SQL Injection – Spring Data JPA Methods

# SchoolRepo.java Class

- The SchoolRepo.java Class contains 3 Spring Data JPA methods. The Spring Data JPA module will define SQL queries automatically from these methods.
- `List<School> findByName(String name);`
- `List<School> findByNameOrderByRoomsAsc(String name);`
- `List<School> findByStudentsLessThan(int num);`



- **Are these Spring Data JPA methods safe from SQL injection?**
- It was hard to find references/documentation specifically that mention they are safe from SQL injection.
- However, from the few references mentioned in the SchoolRepo.java Class, if these queries are not created dynamically and the parameters are not being concatenated dynamically to the queries, these type of method queries are safe from SQL injection.
- After performing a few SQLi tests, the methods were not vulnerable to SQLi.
- **However, still recommended to test any of these methods out if planning to use them in a real application.**

# Testing for SQL injection

Normal request/response

GET

http://localhost:9091/schoolApi/jpa/CHS

ParamsAuthorizationHeaders (6)BodyPre-request ScriptTests

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualize

JSON

1

[

2

{

3

"id": 1,

4

"name": "CHS",

5

"rooms": 1,

6

"students": 50

7

}

8

]

GET

http://localhost:9091/schoolApi/jpa/CHS'

ParamsAuthorizationHeaders (6)BodyPre-request ScriptTestsS

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualize

JSON

1


[ ]

Injecting a single quote returns a 200 OK with an empty list

GET ▼ http://localhost:9091/schoolApi/jpa/CHS"

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼ 

1 `[]`


Injecting 2 single quotes returns a 200 OK with an empty list

Injecting the ' or 1=1-- payload returns a 200 OK with an empty list

GET ▼ http://localhost:9091/schoolApi/jpa/CHS' or 1=1--

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼ 

1 `[]`

# Resources

- <https://www.devglan.com/spring-jdbc/working-with-springboot-namedparameter-jdbctemplate>
- <https://www.stackhawk.com/blog/sql-injection-prevention-spring/>
- [https://docs.guardrails.io/docs/vulnerabilities/java/insecure use of sql queries](https://docs.guardrails.io/docs/vulnerabilities/java/insecure_use_of_sql_queries)
- <https://dzone.com/articles/how-to-specify-named-parameters-using-the-namedpar>
- <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection>