| SQL Injection | | | | Severity: High |
|---|---|---|---|---|
| **CVSS v3.1** | 8.1 | **Vector** | AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N | |
| **Privileges** | Read only user | | | |
| **Version** | Lucht Probst Associates GmbH 1.0 | | | |

SQL injection attacks consist in inserting SQL queries through the usage of an input data field from the client application. This kind of attack allows to obtain access to the database which could then lead to the exfiltration, modify or deletion of existing data.

SQL Injection vulnerability was discovered in the application's search functionality, allowing an attacker to manipulate database queries. The vulnerability enables both **Boolean-Based SQL Injection** and **Time-Based Blind SQL Injection**, leading to potential data exfiltration and database enumeration.

The vulnerable resource:

- https://<application-baseurl>:443/DPP/api/customer/search?q=

**Follow the steps below to replicate the results**

The search functionality is vulnerable to SQL injection, as demonstrated by the following behaviors:

When injecting a single quote ('), the application returns a 500 Internal Server Error.

The error message reveals the column *legalEntityId*.

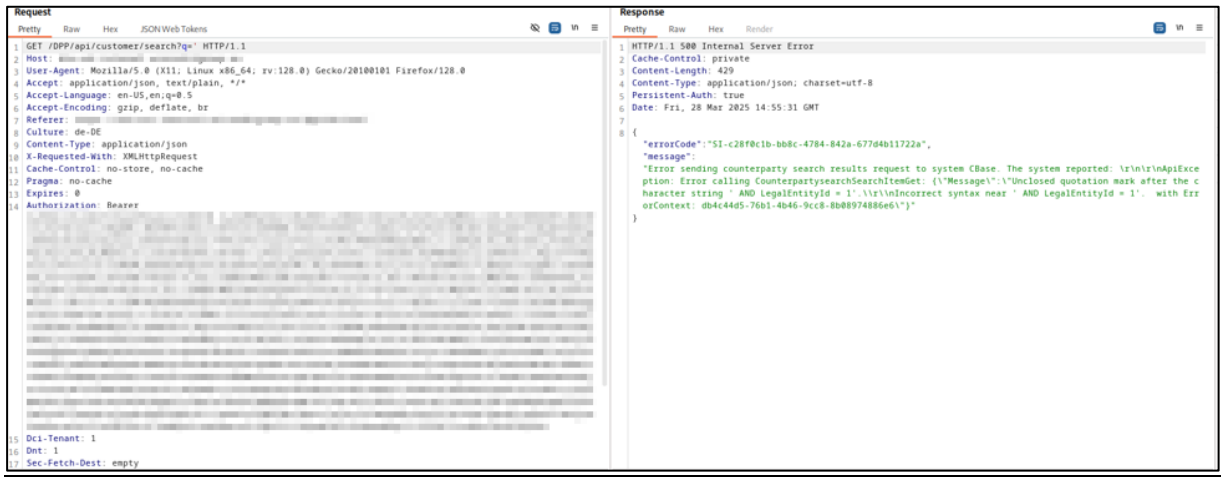


**Figure 1:** Single Quote (') Injection.

When injecting an escaped quote (''), the query execution is correctly terminated, and the application returns 200 OK.

This indicates that user input is directly concatenated into the SQL query without proper sanitization.

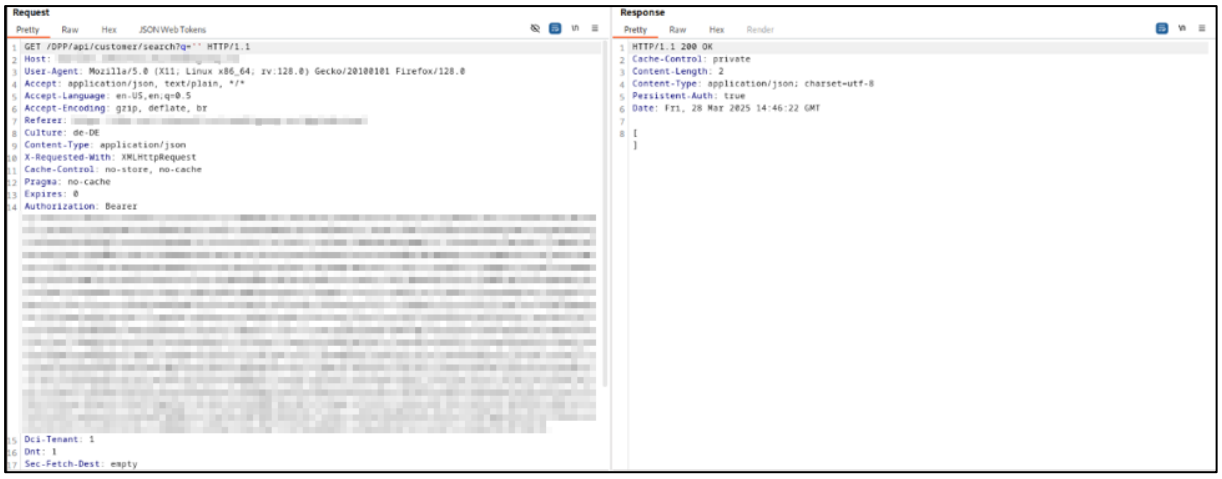


**Figure 2:** Closing the quote ('')

When testing for column presence using invalid column names, the application returns specific error messages: The response includes an error stating that the column password does not exist:

***Invalid column name 'password'***

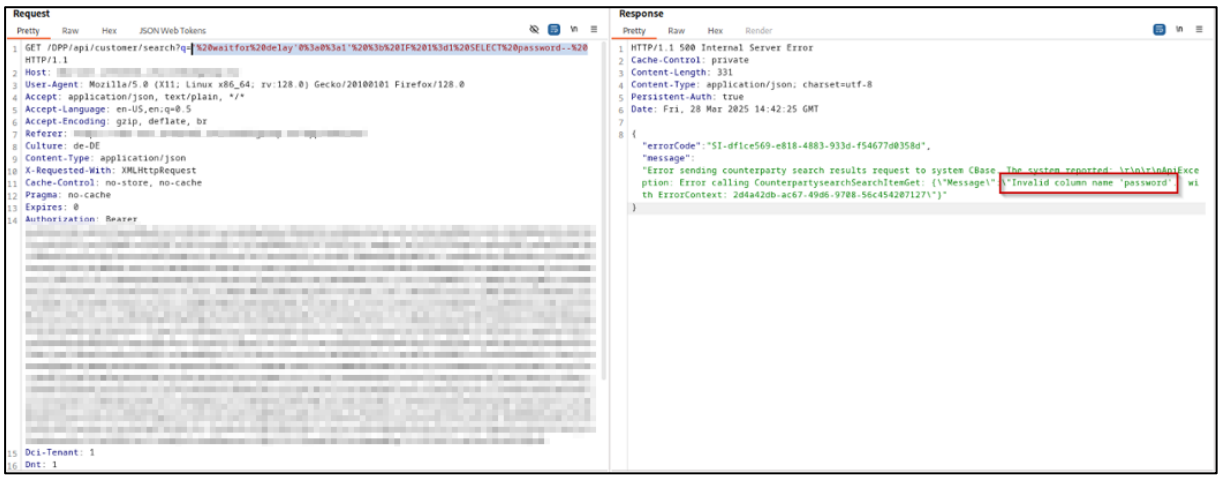


**Figure 3:** Non-existent column (password)

The application behaves differently when the column exists, allowing further interaction.

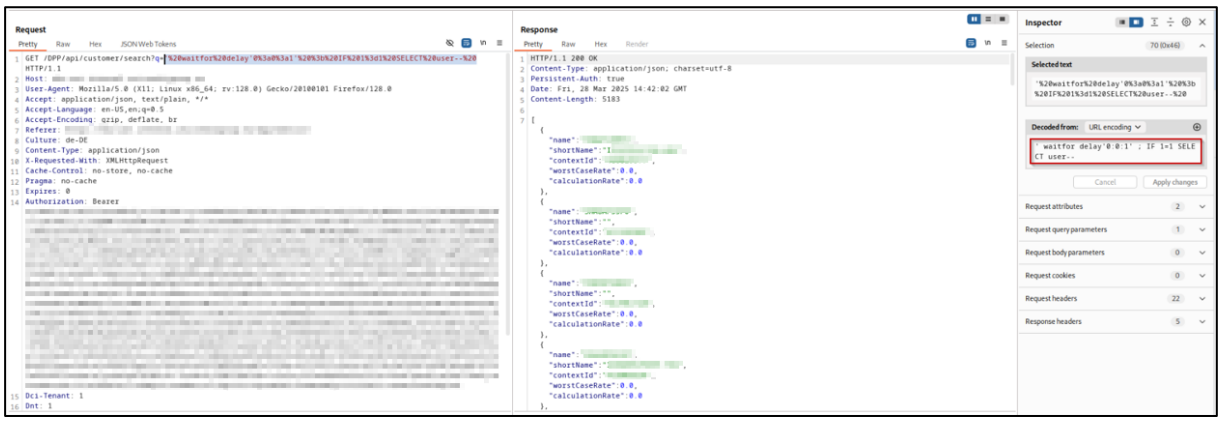


**Figure 4:** Existing column (user).

By injecting SQL queries with **time delays**, it is possible to confirm database interaction and infer query execution:

**Delay Mechanism:**

Using time-based SQL injection techniques (e.g., waitfor delay‘0:0:n’), the database execution time **doubles** the expected delay.
Example behavior:

- **waitfor delay‘0:0:10’** → Response time ~ **20 seconds**
- **waitfor delay‘0:0:15’** → Response time ~ **30 seconds**

This confirms that the injected SQL query is being executed by the database.

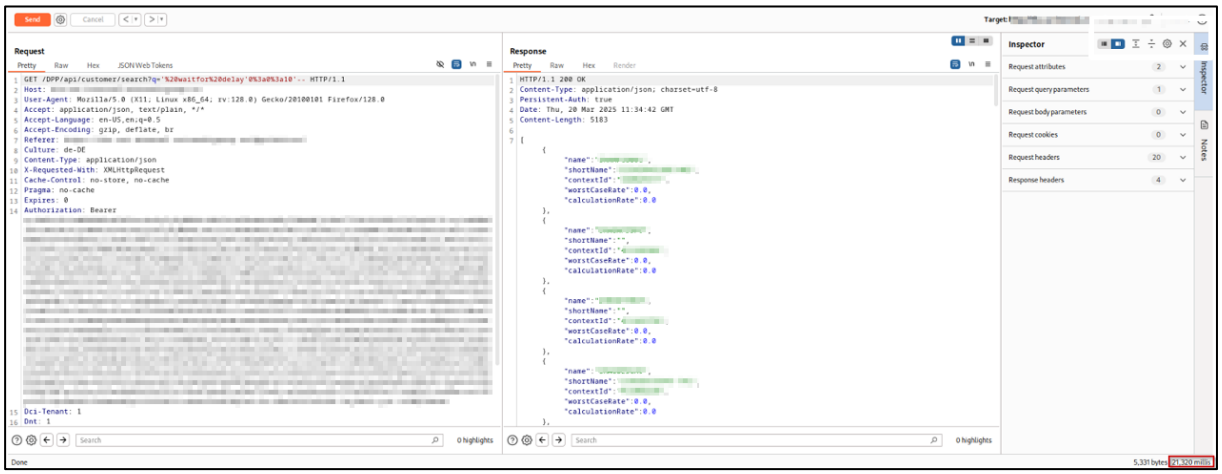

**Figure 5:** Payload 10 seconds delay.

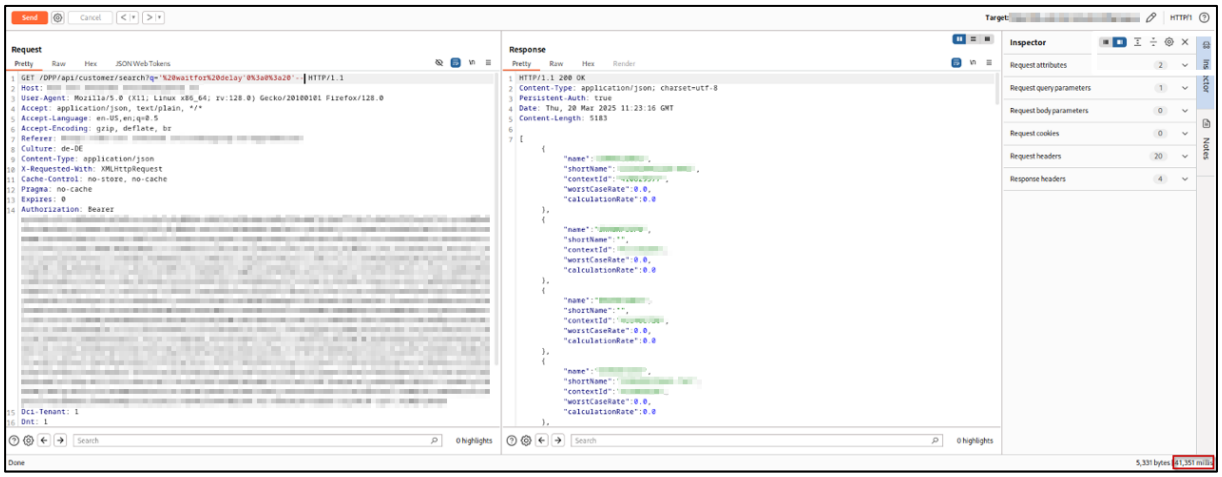

**Figure 6:** Payload 20 seconds delay

When searching using a non-existent legalEntityId (e.g., 255), the application returns an empty result set.

**Figure 7:** Empty Response with Non-Existent legalEntityId.

However, when injecting the following payload:

```
OR 1=1--+-
```

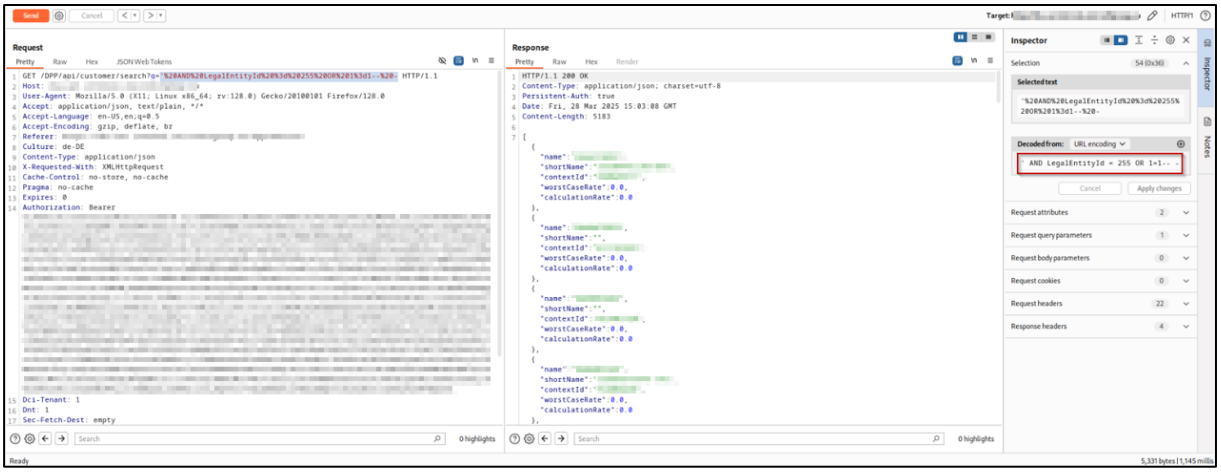

**Figure 8:** Data Extraction Using OR-Based SQL Injection (OR 1=1--+-)

The query condition always evaluates to TRUE, causing the application to return all available records.

This confirms that the application is vulnerable to Boolean-based SQL Injection and allows unauthorized access to the dataset.

**REMEDIATION:**

SQL injection occurs when user-controlled input is included into queries without prior sanitization. In order to prevent this, it is suggested to:

- Prefer the usage of prepared statements instead of concatenating user input into existing queries.
- Use properly constructed stored procedure.
- Sanitize and escape any user provided input.

For further information, please refer to:

- https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html