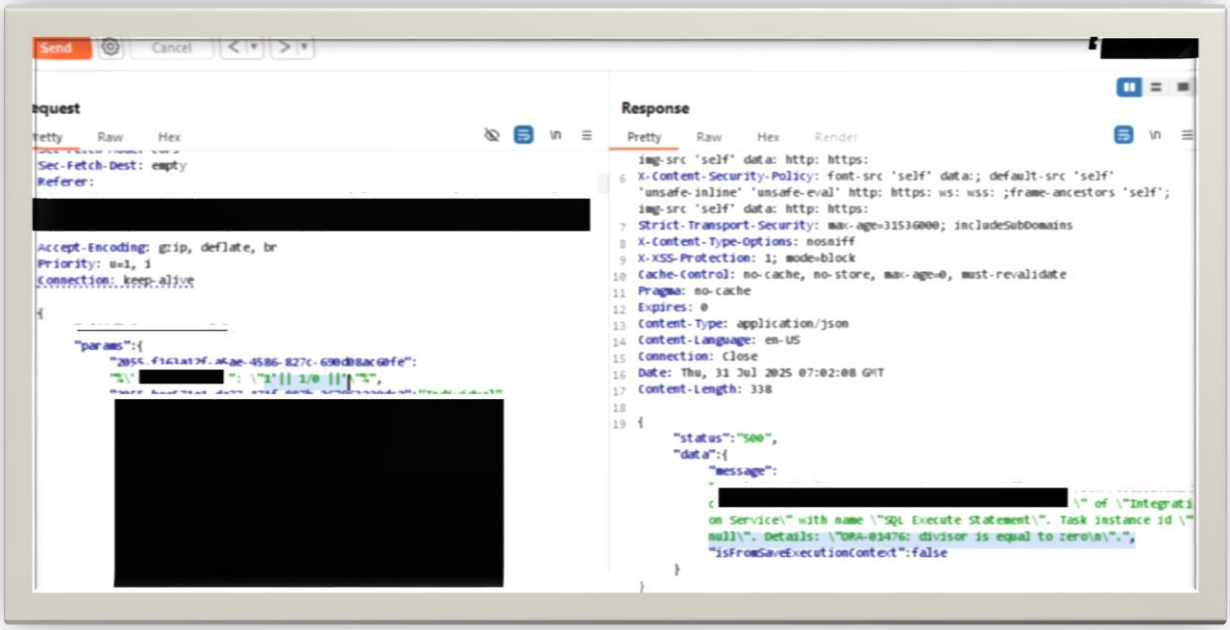
ORACLE DIVISION BY ZERO- ERROR BASED SQLI.

This document describes a repeatable, authorized test procedure used to confirm error-based SQL injection against an Oracle backend by using a division-by-zero payload. The technique uses Oracle's ORA-01476 error (divisor is equal to zero) as an oracle to confirm that attacker-controlled input is evaluated by the database and to extract information (via boolean checks).

Simple division-by-zero probe:

Inject a direct division-by-zero expression to see if the DB error surfaces. Oracle divisor by zero confirms we're directly injecting into SQL execute



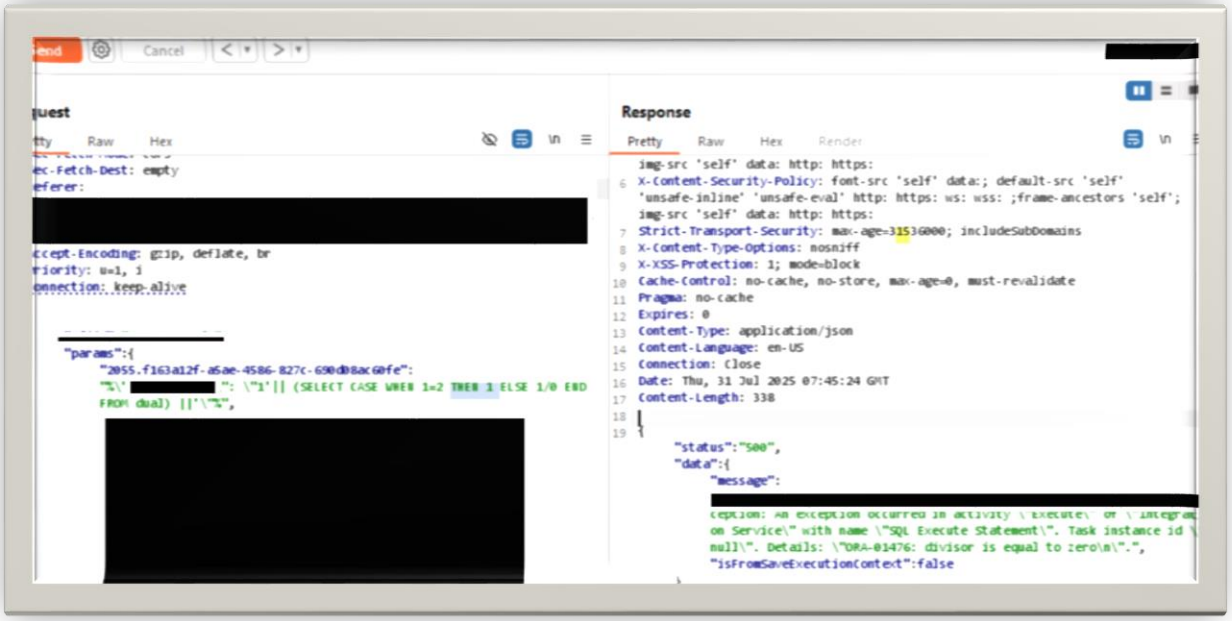
Use CASE to convert boolean checks to error/no-error (error-oracle)

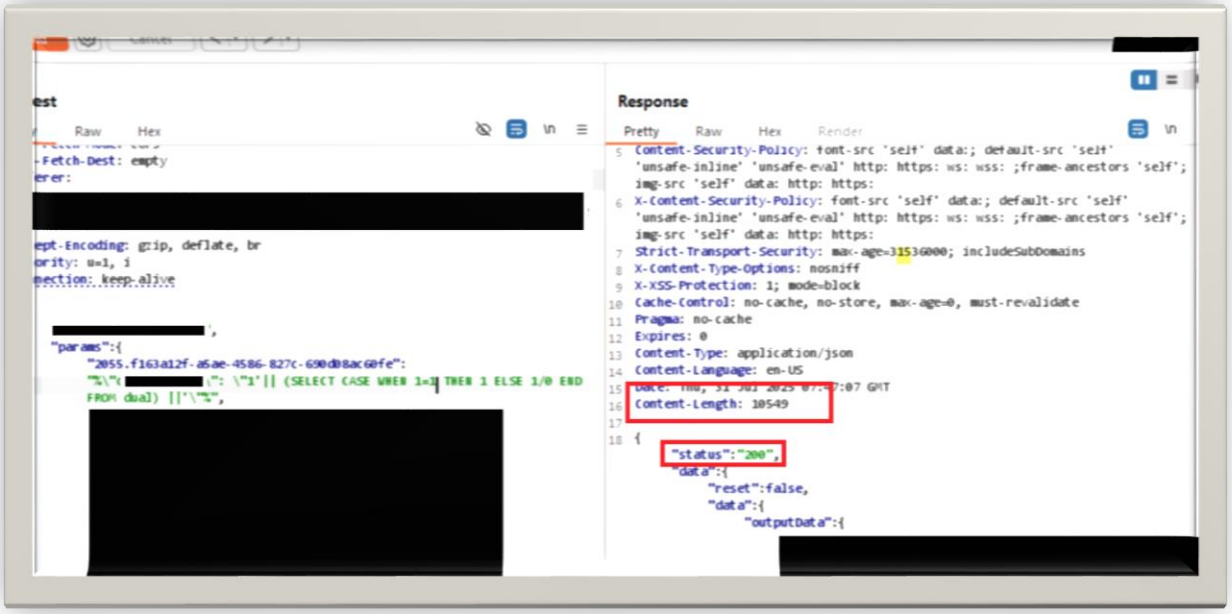
Provide a payload like:

' || (SELECT CASE WHEN 1=1 THEN 1 ELSE 1/0 END FROM DUAL) ||'.

Interpretation behind this is: With 1=1 the CASE returns 1 (no error). With 1=2 the CASE evaluates 1/0 and triggers ORA-01476.

Send the request and observe results. If the DB error appears (HTTP 500 and an ORA-01476 reference), you have direct evidence that injected content is being evaluated by Oracle.





Finding the length of current database:

Form a conditional payload that only triggers 1/0 when the boolean test is false.

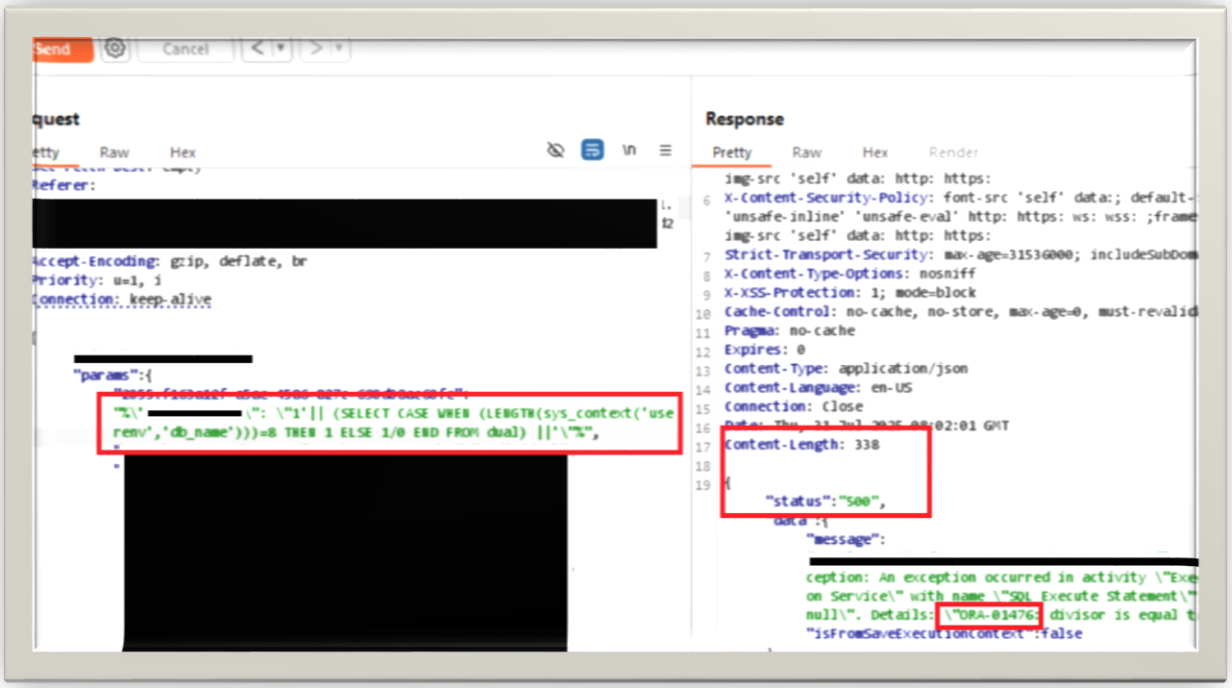
'|| (SELECT CASE WHEN (LENGTH(sys.context('userenv','db_name')))=<n> THEN 1 ELSE 1/0 END FROM DUAL) ||'.

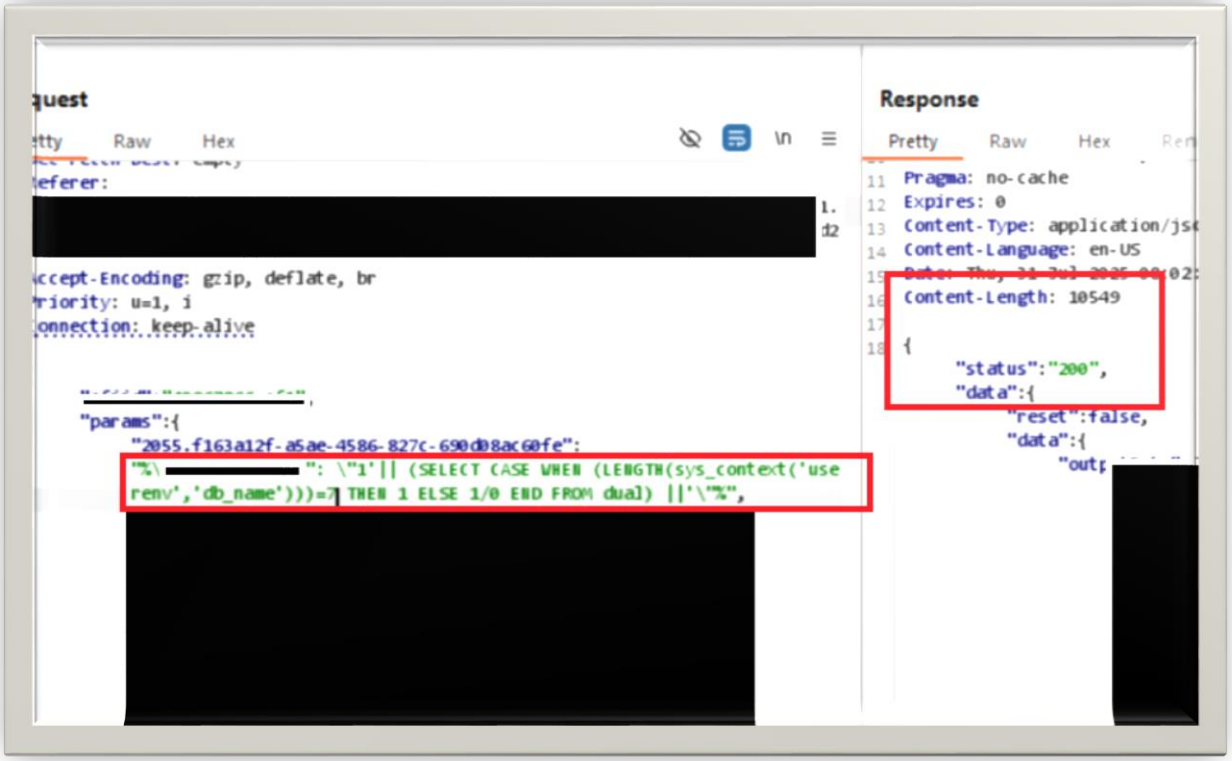
Replace <n> with the guessed length value.

For each guessed value of <n>:

- Send the crafted request.
- If response is 200 (and large Content-Length) → the CASE returned 1 and the tested condition is **true**.
- If response is 500 and/or response contains ORA-01476 (and Content-Length is significantly smaller) → the CASE evaluated 1/0 and the tested condition is **false** (you triggered the division-by-zero).

Using this technique, iterate values (e.g., n=6,7,8...) until you identify the true value. The screenshot show that guessing 7 returned 200 while 8 produced a 500 and included ORA-01476, confirming the condition.





Impact Rationale:

- The application is vulnerable to **error-based SQL injection**, which allows an attacker to confirm injection and extract information (schema names, DB properties, data) via iterative checks.
- Exposed DB errors amplify the risk by leaking internal DB details and enabling faster exploitation.

Remediation & recommendations

1. **Use parameterized queries / prepared statements** — avoid concatenating user input into SQL strings.
2. **Do not leak DB errors to the client.** Log full error details server-side; return safe, generic error messages to users.
3. **Apply least privilege** to the database account used by the application — restrict what queries and metadata can be accessed.
4. **Input validation**— validate and sanitize JSON input server-side; treat inputs as data, not code.
5. **Add runtime protections** — WAF rules detecting SQLi patterns and monitoring for repeated probes or DB error spikes.