|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Web Application with SQL Injection | | |
| **Findings Categorization** | | | | |
| **Business Impact** | | High | **CVSS v4.0 Score** | 9.1 (High) |
| **CVSS Vector** | | CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:L/SI:L/SA:L | | |

Technical Description

A SQL Injection vulnerability was discovered in the application’s input handling mechanisms. During testing, it was observed that user-supplied input was not being properly sanitized before being used in SQL queries. This flaw allows an attacker to manipulate the SQL query structure, potentially leading to unauthorized access to data stored within the backend database.

The affected parameter, <PARAMETER\_NAME>, was identified on the <ENDPOINT> endpoint. By injecting malicious SQL payloads, engineers were able to enumerate database tables and extract sensitive data such as user credentials and internal application records.

Business Impact Description

SQL Injection vulnerabilities pose a critical risk to business operations as they can lead to full compromise of the backend database, exposure of sensitive business or customer information, and even remote code execution in certain cases.

In this instance, engineers were able to extract <DATA\_TYPE>, including but not limited to usernames, email addresses, and password hashes. Such an exploit could allow attackers to escalate privileges within the application or pivot to other internal systems.

Affected Systems

* hostname:PORT/PROTOCOL (include affected endpoint or API where possible)
* Endpoint: https://example.com/<ENDPOINT>

Steps to Reproduce

Engineers navigated to the endpoint <ENDPOINT> and intercepted the request. Then, modified the input parameter <PARAMETER\_NAME> with the following payload: <INSERT PAYLOAD>

/Screenshot 1

Caption 1: SQL payload used in intercepted request

The server responded with a full list of user records, indicating that the input was directly injected into the SQL query.

/Screenshot 2

Caption 2: Response showing database contents as a result of SQLi

Further payloads like <PAYLOADS> confirmed the vulnerability and demonstrated time-based and error-based exploitation methods.

Mitigations

* Implement parameterized queries or prepared statements across all database interactions.
* Input validation and output encoding should be enforced at all user-controlled entry points.
* Consider deploying a Web Application Firewall (WAF) to detect and block injection attempts.
* Perform regular security reviews and code audits to identify improper query handling.

References

* **OWASP – SQL Injection**
  + [**https://owasp.org/www-community/attacks/SQL\_Injection**](https://owasp.org/www-community/attacks/SQL_Injection)
* **MITRE ATT&CK – T1505.001: SQL Injection**
  + [**https://attack.mitre.org/techniques/T1505/001/**](https://attack.mitre.org/techniques/T1505/001/)
* **NIST SP 800-64 Rev.2 – Security Considerations in the System Development Life Cycle**
  + [**https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-64r2.pdf**](https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-64r2.pdf)