

**Finding Name:** SQL Injestion

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Team** | **Role** | **Project** | **Quality Assurance** | **Is this a re-tested Finding?** |
| Manasa Vallaboju | SCR | Team member | Ontrack |  | No |
|  |  |  |  |  |  |

|  |
| --- |
| **Was this Finding Successful?** |
| Yes |

**Finding Description**

The given code sample includes a SQL query that was created with user input but was not properly sanitized, leaving it open to SQL injection attacks. An application's complete breach, data leaking, manipulation, or unauthorized database access are all possible outcomes of SQL injection, a security flaw that lets hackers alter SQL queries through user input.

An SQL injection attack allows a user to access a database directly by submitting a SQL query and obtaining access without the necessary credentials. Subsequently, the attacker has the ability to alter passwords and other authentication details, read, export, change, and remove sensitive data, and potentially even access other systems on the network. Although this is one of the most often exploited vulnerability categories, it is mostly preventable with proper coding techniques.

**Risk Rating**  
Impact: Significant  
Likelihood: High

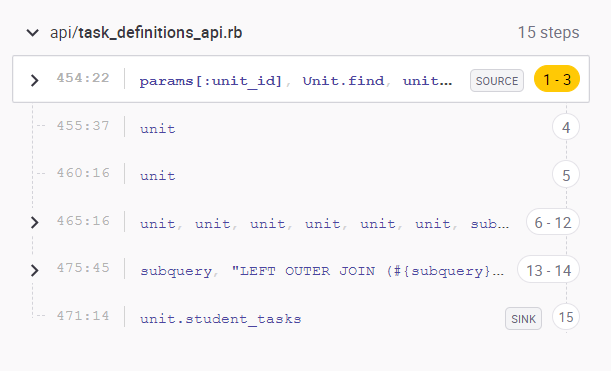
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Impact values** | | | | |
| **Very Minor** | **Minor** | **Significant** | **Major** | **Severe** |
| Risk that holds little to no impact. Will not cause damage and regular activity can continue. | Risk that holds minor form of impact, but not significant enough to be of threat. Can cause some damage but not enough to impede regular activity. | Risk that holds enough impact to be somewhat of a threat. Will cause damage that can impede regular activity but will be able to run normally. | Risk that holds major impact to be of threat. Will cause damage that will impede regular activity and will not be able to run normally. | Risk that holds severe impact and is a threat. Will cause critical damage that can cease activity to be run. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Likelihood** | | | | |
| **Rare** | **Unlikely** | **Moderate** | **High** | **Certain** |
| Event may occur and/or if it did, it happens in specific circumstances. | Event could occur occasionally and/or could happen (at some point) | Event may occur and/or happens. | Event occurs at times and/or probably happens a lot. | Event is occurring now and/or happens frequently. |

**Business Impact**

Due to this vulnerability, the application is susceptible to SQL injection attacks, which could give attackers the power to run any SQL query on the database. This might lead to data tampering, illegal access to private information, or even the whole compromise of the program and its related data.

**Affected Assets**

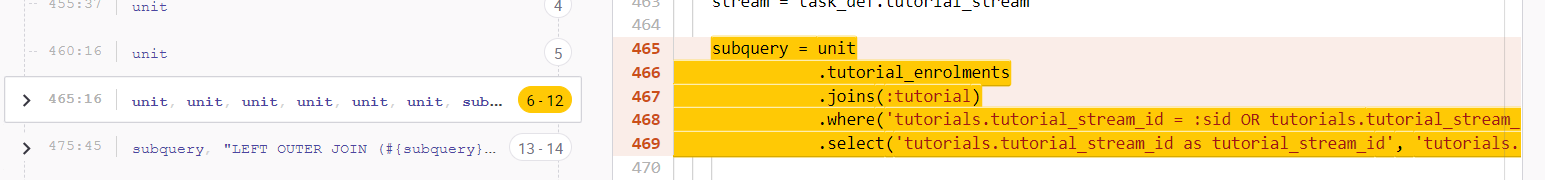
**\**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated** **A screenshot of a computer

Description automatically generated**

**Evidence**

Analyzing the code with the help of the tool name Synk, Which identifies a risks/ vulnerabilities which is in the code which leads to a data leak or unauthorized access.

Step 1: Without first undergoing sanitization, suspicious input is interpolated straight into a SQL query.   
Step 2: To insert malicious SQL code, an attacker modifies the input.   
Step 3: The database executes the malicious SQL code, which may cause data leaks or unauthorized access.   
Step 4: The attacker enters the program and takes illegal actions or obtains access to sensitive data.

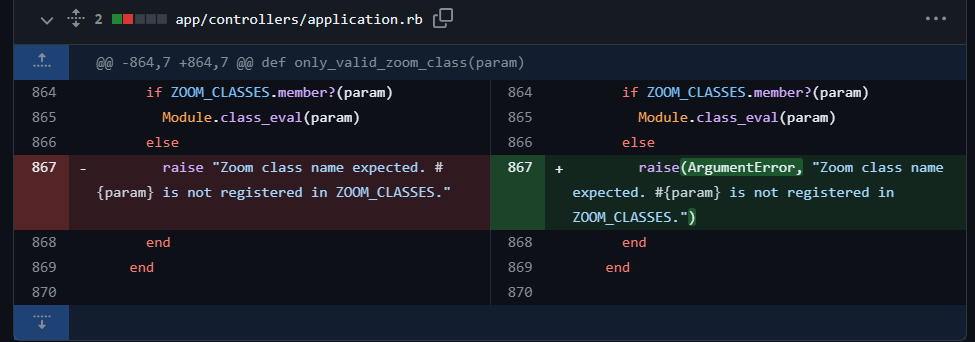
****

**Remediation Advice**

* Refrain from providing the SQL server with user-entered parameters directly.
* Don't construct SQL queries from user-entered parameters by string concatenation.
* When writing code, pass in parameters after defining the SQL code. When using parameterized queries, use prepared statements. SqlCommand() in.NET and bindParam() in PHP are two examples.
* To prevent unexpected user data from being rejected, utilize strong typing for all parameters.
* When it is not possible to prevent direct user input due to performance concerns, validate input using an extremely stringent allowlist of characters that are allowed, avoiding special characters like?, &, /, >, ; -, "," and spaces. If you can, use an escape routine that comes with the vendor.   
  Create your application in a setting that prevents SQL injection, or use libraries that do so.

Fix examples:

@user = user.find\_by(params[:user][:email] , params[:user][:password]); @user = User.find\_by(email: params[:user][:email], password: params[:user][:password]);



A screenshot of a computer program

Description automatically generated

**References**

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') <https://cwe.mitre.org/data/definitions/89.html>

Fix Example references: <https://github.com/kete/kete/commit/51e3dc3024996907696ff47b3abd562cb467ca6d?diff=split#diff-51e5beb8a535f5ab4ef70738b7aa8b45d52fd1fe713cb4e8932ee6d65cf6c84aL-1>

[**https://github.com/textlab/glossa/commit/5e373b6795167a8a751a830619d2058a8badc136?diff=split#diff-013605412b0f98983e90a7dac04810be988ca6cb243acecb7ef7c13a95b60513L-1**](https://github.com/textlab/glossa/commit/5e373b6795167a8a751a830619d2058a8badc136?diff=split#diff-013605412b0f98983e90a7dac04810be988ca6cb243acecb7ef7c13a95b60513L-1)

**Contact Details**

Name: Manasa Vallaboju

Email: s223001068@deakin.edu.au

**Pentest Leader Feedback.**

The lead will provide feedback to enact on.