UNIVERSITY OF INSUBRIA

Department of Theoretical and Applied Sciences – DiSTA
Master of Science in Computer Science

DATA SECURITY AND PRIVACY

Project Work: Oracle Database for a Student Capture The Flag (CTF) Training Platform

Author:
Roberto Vicario

Student ID:
744072

Advisor:
Barbara Carminati

Academic Year 2024-2025

Contents

1 Task 1: Database Design

1.1 Overview: Capture The Flag (CTF) Training Platform

The developed system is a training platform modeled after *Capture The Flag (CTF)* competitions. In this environment, users engage with a variety of security-focused challenges to accumulate points and improve their ranking.

The platform is designed around distinct user roles each with specific permissions and workflows. The database schema and platform logic are structured to support these roles and their interactions, ensuring secure authentication, challenge management, and real-time tracking of user progress across three primary usage scenarios.

1.2 Student Scenario

Here is a breakdown of the operations that a *Student* can perform on the platform:

1. Sign in into the platform.
2. View *Visible* challenges.
3. Submit challenge solution using flag format `flag{...}`:
    - If the solution is correct:
        1. The challenge will be marked as solved only for the student who submitted the solution;
        2. The user score will be increased with points from the solved challenge.
    - Otherwise:
        1. The platform returns an error message indicating the solution is incorrect.

IMPLEMENTATION

Designing this scenario requires the following considerations:

1. **User Authentication:** To access the platform, users must log in using their credentials. This needs a `Users` table.
2. **Challenge Visibility:** To see only visible challenges, the platform should filter challenges based on their visibility setting. This requires a `Challenges` table with a `is_visible` field.
3. **Flag Submission:** The backend validates the submitted flag against the `flag` field into `Challenges` table:
    - If the solution is correct:

    1. Will be created a new row in the `Solved_Challenges` table, linking the student using their `id` with the challenge `code`.
    2. The user `score` will be increased with `points` from the solved challenge.
- Otherwise:
    1. The platform response is completely a frontend/backend job.

To get started, we need to create the following database schema:

```sql
-- Users
CREATE TABLE Users (
    id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    username VARCHAR2(50) UNIQUE NOT NULL,
    password VARCHAR2(255) NOT NULL,
    role VARCHAR2(25) NOT NULL,
    score NUMBER DEFAULT 0 CHECK (score >= 0) NOT NULL,
    -- -------------------------
    CHECK (role IN ('Student', 'Tutor', 'Admin'))
);

-- Challenges
CREATE TABLE Challenges (
    code VARCHAR2(25) PRIMARY KEY,
    title VARCHAR2(100) NOT NULL UNIQUE,
    body CLOB NOT NULL,
    category VARCHAR2(50) NOT NULL,
    flag VARCHAR2(255) NOT NULL UNIQUE,
    is_visible NUMBER(1) DEFAULT 1 CHECK (is_visible IN (0, 1)) NOT
       NULL,
    points NUMBER DEFAULT 0 CHECK (points >= 0) NOT NULL
);

-- Solved Challenges
CREATE TABLE Solved_Challenges (
    id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    solved_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    user_id NUMBER NOT NULL,
    challenge_code VARCHAR2(25) NOT NULL,
    -- -------------------------
    CONSTRAINT fk_user_id FOREIGN KEY (user_id) REFERENCES Users(id) ON
        DELETE CASCADE,
    CONSTRAINT fk_challenge_code FOREIGN KEY (challenge_code)
       REFERENCES Challenges(code) ON DELETE CASCADE,
    CONSTRAINT uq_user_challenge UNIQUE (user_id, challenge_code)
);
```

1.2.1 Leaderboard

To be completed, the platform should have a leaderboard that displays the top students based on their scores and the number of challenges solved. It is not necessery to create a new table for this, as it can be achieved using a view `Leaderboard` that aggregates data from the `Users` and `Solved_Challenges` tables:

```sql
-- Leaderboard
CREATE VIEW Leaderboard AS
SELECT u.id, u.username, u.score, COUNT(sc.challenge_code) AS
    Solved_Challenges
FROM Users u
LEFT JOIN Solved_Challenges sc ON u.id = sc.user_id
WHERE u.role = 'Student'
GROUP BY u.id, u.username, u.score
ORDER BY u.score DESC, COUNT(sc.challenge_code) DESC;
```

1.3 Tutor Scenario

Here is a breakdown of the operations that a *Tutor* can perform on the platform:

1. Sign in into the platform.
2. View *Available* challenges.
3. Update challenges *Visibility* setting.

IMPLEMENTATION

Designing this scenario requires the following considerations:

1. **User Authentication**: Similar to the previous scenario, users must log in to access the platform.
2. **Challenge Availability:** To see all available challenges, the platform should filter challenges based on their availability setting. This requires a `is_available` field in the `Challenges` table.
3. **Challenge Management:** This role can update the `is_visible` field in the `Challenges` table, allowing tutors to control which challenges are visible to students.

To continue, it is necessary to update the `Challenges` table schema to include the `is_available` field:

```sql
-- Challenges
```

```
CREATE TABLE Challenges (
    code VARCHAR2(25) PRIMARY KEY,
    title VARCHAR2(100) NOT NULL UNIQUE,
    body CLOB NOT NULL,
    category VARCHAR2(50) NOT NULL,
    flag VARCHAR2(255) NOT NULL UNIQUE,
    is_visible NUMBER(1) DEFAULT 1 CHECK (is_visible IN (0, 1)) NOT
       NULL,
    is_available NUMBER(1) DEFAULT 1 CHECK (is_available IN (0, 1)) NOT
        NULL,
    points NUMBER DEFAULT 0 CHECK (points >= 0) NOT NULL
);
```

1.4 Admin Scenario

Here is a breakdown of the operations that an *Admin* can perform on the platform:

1. Sign in into the platform.
2. Update challenges *Visibility* setting.
3. Create/update/remove challenges.
4. If user activity is malicious, the admin can ban the user.

IMPLEMENTATION

Designing this scenario requires the following considerations:

1. **User Authentication**: Similar to the previous scenario, users must log in to access the platform.
2. **Availability Management:** The ADMIN role should have the permission to update the `is_visible` field in the `Challenges` table, allowing them to control which challenges are visible to students.
3. **Challenge Management:** This role has the permission to add a new row in the `Challenges` table, update existing challenges, or delete them. *ADMIN* can update the `is_available` field in the `Challenges` table, allowing tutors to control which challenges are available to students.
4. **User Status:** If user activity is malicious, the admin can ban the user by updating their `status` field in the `Users` table. This feature allows more general admin to manage user accounts effectively, such as suspending or reactivating accounts.

To implement this scenario, we need to ensure that the database schema supports this management:

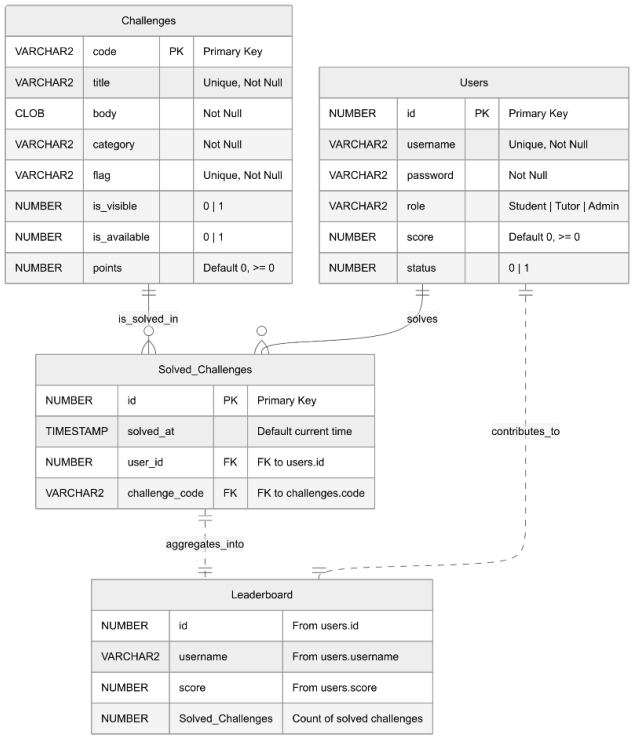
```
-- Users
CREATE TABLE Users (
```

```
    id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    username VARCHAR2(50) UNIQUE NOT NULL,
    password VARCHAR2(255) NOT NULL,
    role VARCHAR2(25) NOT NULL,
    score NUMBER DEFAULT 0 CHECK (score >= 0) NOT NULL,
    status NUMBER(1) DEFAULT 1 CHECK (status IN (0, 1)) NOT NULL,
    -- ------------------------
    CHECK (role IN ('Student', 'Tutor', 'Admin'))
);

-- Leaderboard
CREATE VIEW Leaderboard AS
SELECT u.id, u.username, u.score, COUNT(sc.challenge_code) AS
   Solved_Challenges
FROM Users u
LEFT JOIN Solved_Challenges sc ON u.id = sc.user_id
WHERE u.status = 1 AND u.role = 'Student'
GROUP BY u.id, u.username, u.score
ORDER BY u.score DESC, COUNT(sc.challenge_code) DESC;
```

1.5 Final Schema

The following *Figure 1* illustrates the relationships between the main entities in the training platform:



> **Figure 1:** An ER Diagram of the training platform designed to illustrate the relationships between main entities.

To summarize, the complete SQL schema based on the assumptions we made for the platform is as follows:

```sql
-- Task 1: Database Design

-- DDL

-- Users
CREATE TABLE Users (
    id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
```

```
    username VARCHAR2(50) UNIQUE NOT NULL,
    password VARCHAR2(255) NOT NULL,
    role VARCHAR2(25) NOT NULL,
    score NUMBER DEFAULT 0 CHECK (score >= 0) NOT NULL,
    status NUMBER(1) DEFAULT 1 CHECK (status IN (0, 1)) NOT NULL,
    -- -------------------------
    CHECK (role IN ('Student', 'Tutor', 'Admin'))
);

-- Challenges
CREATE TABLE Challenges (
    code VARCHAR2(25) PRIMARY KEY,
    title VARCHAR2(100) NOT NULL UNIQUE,
    body CLOB NOT NULL,
    category VARCHAR2(50) NOT NULL,
    flag VARCHAR2(255) NOT NULL UNIQUE,
    is_visible NUMBER(1) DEFAULT 1 CHECK (is_visible IN (0, 1)) NOT
       NULL,
    is_available NUMBER(1) DEFAULT 1 CHECK (is_available IN (0, 1)) NOT
        NULL,
    points NUMBER DEFAULT 0 CHECK (points >= 0) NOT NULL
);

-- Solved Challenges
CREATE TABLE Solved_Challenges (
    id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    solved_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    user_id NUMBER NOT NULL,
    challenge_code VARCHAR2(25) NOT NULL,
    -- -------------------------
    CONSTRAINT fk_user_id FOREIGN KEY (user_id) REFERENCES Users(id) ON
        DELETE CASCADE,
    CONSTRAINT fk_challenge_code FOREIGN KEY (challenge_code)
       REFERENCES Challenges(code) ON DELETE CASCADE,
    CONSTRAINT uq_user_challenge UNIQUE (user_id, challenge_code)
);

-- Leaderboard
CREATE VIEW Leaderboard AS
SELECT u.id, u.username, u.score, COUNT(sc.challenge_code) AS
   Solved_Challenges
FROM Users u
LEFT JOIN Solved_Challenges sc ON u.id = sc.user_id
WHERE u.status = 1 AND u.role = 'Student'
GROUP BY u.id, u.username, u.score
ORDER BY u.score DESC, COUNT(sc.challenge_code) DESC;
```

1.5.1 Sample Data

To test the platform, we can insert some sample data into the tables. The following SQL statements provide an example of how to populate the `Users`, `Challenges`, and `Solved_Challenges` tables with initial data:

```sql
-- Task 1: Database Design

-- DML

-- Users
INSERT INTO Users (username, password, role)
VALUES ('student1', 'student1', 'Student');
INSERT INTO Users (username, password, role)
VALUES ('student2', 'student2', 'Student');
INSERT INTO Users (username, password, role)
VALUES ('student3', 'student3', 'Student');
INSERT INTO Users (username, password, role)
VALUES ('tutor1', 'tutor1', 'Tutor');
INSERT INTO Users (username, password, role)
VALUES ('admin1', 'admin1', 'Admin');

-- Challenges
INSERT INTO Challenges (code, title, body, category, flag, is_visible, is_available, points)
VALUES ('CH_1', 'Challenge 1', '...', 'Cryptography', 'flag{1}', 1, 1, 500);
INSERT INTO Challenges (code, title, body, category, flag, is_visible, is_available, points)
VALUES ('CH_2', 'Challenge 2', '...', 'Web', 'flag{2}', 1, 1, 500);
INSERT INTO Challenges (code, title, body, category, flag, is_visible, is_available, points)
VALUES ('CH_3', 'Challenge 3', '...', 'Binary', 'flag{3}', 1, 1, 500);

-- Student 1 solved Challenge 1
INSERT INTO Solved_Challenges (user_id, challenge_code)
VALUES ((SELECT id FROM Users WHERE username = 'student1'), 'CH_1');
UPDATE Users
SET score = score + (SELECT points FROM Challenges WHERE code = 'CH_1')
WHERE id = (SELECT id FROM Users WHERE username = 'student1');

-- Student 1 solved Challenge 2
INSERT INTO Solved_Challenges (user_id, challenge_code)
VALUES ((SELECT id FROM Users WHERE username = 'student1'), 'CH_2');
UPDATE Users
SET score = score + (SELECT points FROM Challenges WHERE code = 'CH_2')
WHERE id = (SELECT id FROM Users WHERE username = 'student1');

-- Student 2 solved Challenge 1
```

```sql
INSERT INTO Solved_Challenges (user_id, challenge_code)
VALUES ((SELECT id FROM Users WHERE username = 'student2'), 'CH_1');
UPDATE Users
SET score = score + (SELECT points FROM Challenges WHERE code = 'CH_1')
WHERE id = (SELECT id FROM Users WHERE username = 'student2');
```