

Comprehensive Security & RBAC Audit Report

This document summarizes the complete security and architectural overhaul performed on the centraljuanv2 backend to enforce strict Role-Based Access Control (RBAC), patch critical vulnerabilities, and guarantee database execution integrity.

1. Backend RBAC Enforcement

The Problem: Previously, the system relied entirely on the frontend to hide CRUD (Create, Read, Update, Delete) buttons based on a user's permissions. The backend API did not verify if the user actually had the right to perform the action.

The Solution: We deployed a global automated patch across **133+ backend endpoints**, injecting a strict `require_permission()` check before any data modification could occur.

Code Change Example (e.g., `delete_department.php`):

```
<?php
include("../server/cors.php");
include("../server/connection.php");
require_admin_state();

+require_permission($conn, 'can_delete');
+
$data = get_sanitized_input();
```

2. Preventing IDOR (Insecure Direct Object Reference)

The Problem: Even if a user had the `can_delete` permission, they could theoretically delete entities (like Employees) that belonged to brands or branches they had no access to, simply by guessing the `employee_id` in a direct POST request.

The Solution: We modified critical deletion endpoints to actively verify that the target entity belongs to a brand that the authenticated user is authorized to manage.

Code Change Example (delete_employee.php):

```
// User brands check
$user_brands = $_SESSION['user_brands']; // e.g., ['BRAND_A',
      'BRAND_B']

-$sql = "DELETE FROM employees WHERE employee_id = '$id'";
-$conn->query($sql);
+ // 1. Verify the employee belongs to an allowed brand
+ $check_sql = "SELECT brand_id FROM employees WHERE employee_id
      = ?";
+ $check_stmt = $conn->prepare($check_sql);
+ $check_stmt->bind_param("s", $id);
+ $check_stmt->execute();
+ $emp = $check_stmt->get_result()->fetch_assoc();
+
+ if (!in_array($emp['brand_id'], $user_brands)) {
+     die(json_encode(['success' => false, 'message' =>
      'Unauthorized brand access.']));
+ }
+
+ // 2. Proceed with deletion
+ $del_stmt = $conn->prepare("DELETE FROM employees WHERE
      employee_id = ?");
+ $del_stmt->bind_param("s", $id);
+ $del_stmt->execute();
```

3. SQL Injection Remediation

The Problem: Over 16+ core backend files were vulnerable to SQL Injection because user-supplied input (like IDs or names) was concatenated directly into SQL query strings rather than being safely parameterized.

The Solution: We developed a Python AST-parsing script to systematically locate raw `$conn->query()` executions and refactor them into highly secure Prepared Statements (`$conn->prepare()`).

Code Change Example (delete_department.php):

```
- $id = $_POST['id'];
- $sql = "DELETE FROM departments WHERE department_id = '$id'";
- if ($conn->query($sql) === TRUE) {
-     echo json_encode(["success" => true]);
- }
+ $id = $_POST['id'];
+ $sql = "DELETE FROM departments WHERE department_id = ?";
```

```
+ $stmt = $conn->prepare($sql);
+ $stmt->bind_param("s", $id);
+ if ($stmt->execute()) {
+     echo json_encode(["success" => true]);
+ }
```

4. Execution Logic & Data Integrity

The Problem: Complex endpoints that perform multiple database modifications sequentially (like Overtime and Attendance updates) were prone to partial-writes. If the server crashed or a subsequent query failed mid-execution, the database would be left with permanently corrupted or orphaned data.

The Solution: We manually wrapped the most vulnerable operational endpoints (update_attendance.php and add_overtime_request.php) in strict MySQL Transactions.

Code Change Example (add_overtime_request.php):

```
- $updateStmt->execute();
- overtime_refresh_request_snapshot($conn, (int)$request_id,
    false);
- $emailSent = sendOvertimeEmail(...);
- echo json_encode(["success" => true]);
+ $conn->begin_transaction();
+ try {
+     if ($updateStmt->execute()) {
+         overtime_refresh_request_snapshot($conn, (int)
+             $request_id, false);
+         $conn->commit();
+
+         // Non-critical external services (emails) are kept
+         OUTSIDE the transaction
+         $emailSent = sendOvertimeEmail(...);
+         echo json_encode(["success" => true]);
+     } else {
+         $conn->rollback();
+     }
+ } catch (\Throwable $e) {
+     $conn->rollback();
+     echo json_encode(["success" => false, "message" =>
+         "Transaction failed."]);
+ }
```

Conclusion

The CRUD infrastructure of the application is now deeply fortified. RBAC policies are strictly enforced at the database gateway, data isolation boundaries are respected, and logical operations run atomically.