Assignment Guidance and Front Sheet

This front sheet for assignments is designed to contain the brief, the submission instructions, and the actual student submission for any WMG assignment. As a result the sheet is completed by several people over time, and is therefore split up into sections explaining who completes what information and when. Yellow highlighted text indicates examples or further explanation of what is requested, and the highlight and instructions should be removed as you populate 'your' section.

This sheet is only to be used for components of assessment worth more than 3 CATS (e.g. for a 15 credit module, weighted more than 20%; or for a 10 credit module, weighted more than 30%).

To be completed by the student(s) prior to final submission:

Your actual submission should be written at the end of this cover sheet file, or attached with the cover sheet at the front if drafted in a separate file, program or application.

Student ID or IDs for group work	**U2136249**

To be completed (highlighted parts only) by the programme administration after approval and prior to issuing of the assessment; to be consulted by the student(s) so that you know how and when to submit:

Date set	20/02/2023
Submission date (excluding extensions)	15/05/23
Submission guidance	Submission details are provided in the assignment sheet (A single ZIP file to be submitted to Tabula)
Marks return date (excluding extensions)	12/06/23
Late submission policy	If work is submitted late, penalties will be applied at the rate of **5 marks per University working day** after the due date, up to a **maximum of 10 working days** late. After this period the mark for the work will be reduced to 0 (which is the maximum penalty). "Late" means **after the submission deadline time as well as the date** – work submitted after the given time even on the same day is counted as 1 day late. For **Postgraduate** students only, who started their **current course before 1 August 2019**, the daily penalty is **3 marks** rather than 5.
Resubmission policy	If you fail this assignment or module, please be aware that the University allows students to remedy such failure (within certain limits). Decisions to authorise such resubmissions are made by Exam Boards. Normally these will be issued at specific times of the year, depending on your programme of study. More information can be found from your programme office if you are concerned.

To be completed by the module owner/tutor prior to approval and issuing of the assessment; to be consulted by the student(s) so that you understand the assignment brief, its context within the module, and any specific criteria and advice from the tutor:

Module title & code	WM243 Information Management
Module owner	Anita Khadka
Module tutor	Same as above
Assessment type	Report / database/test scripts
Weighting of mark	50%

Assessment brief
Details of the assessment brief are provided in each case study and a separate note section.

Word count	The word count for the PDF report must not exceed 2000 words (Excluding cover sheet, tables, diagrams and references).
Module learning outcomes (numbered)	• Critically identify entities, attributes and their relationships • Interact with database(s) of information through suitable programming queries • Critically evaluate the cyber consequences that flow from management of information in a given scenario. • Critically identify and evaluate database security and take useful measures
Learning outcomes assessed in this assessment (numbered)	As above
Marking guidelines	Marking guidelines are provided in the Assessment Brief.
Academic guidance resources	Academic guidance will be provided throughout the module.

Table of Contents

I. Case 1

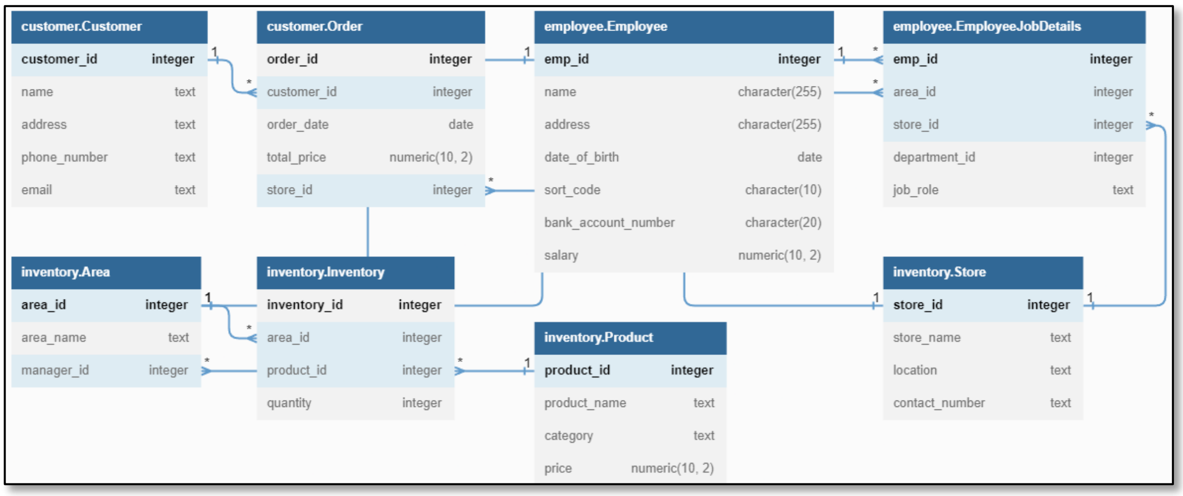
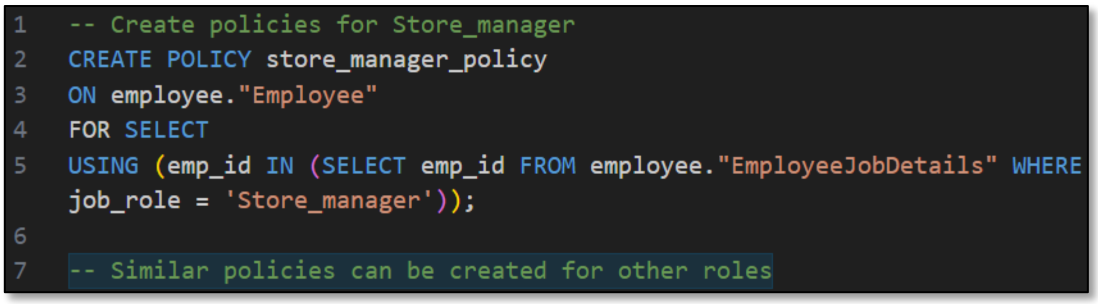


Figure 1 – EDR for a Supermarket Database

When implementing the database for the supermarket database, I focused on ensuring that the data is only accessible to those who require it. My goal was to implement a database that aligns with the principle of least privilege – granting only the necessary access to employees based on their job roles (CSRC, 2017).

I started by separated the database into different schemas to store information about, its customers, employees, inventory and products. I, then created its corresponding tables and relations between them as shown in Figure 1. The critical table was the "Employee" table, which stored sensitive PII of employees such as names, addresses, date of births, and salaries. To ensure the integrity and confidentiality the information.

The primary challenge was to ensure that each role could access only the specific data relevant to its responsibilities. For instance, a "Store_Manager" should only be able to access employee names and addresses, while an "HR_manager" should have access to the date of birth and salary information. By granting specific "SELECT" permissions on individual columns for each role using CLS – restrict user to view only a particular column by blocking access to all other columns in a table (PostgreSQL, 2023), I was able to control the information each role could view from the "Employee" table.

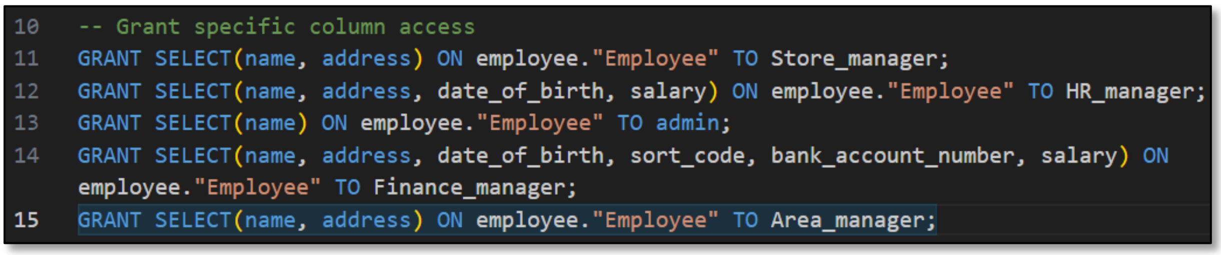


```sql
-- Create policies for Store_manager
CREATE POLICY store_manager_policy
ON employee."Employee"
FOR SELECT
USING (emp_id IN (SELECT emp_id FROM employee."EmployeeJobDetails" WHERE
job_role = 'Store_manager'));

-- Similar policies can be created for other roles
```

Figure 2 - SQL Policy for "Store_manager" Role

Some roles were required to have access only to the information of employees within their own store or area. To enforce this restriction, I implemented RLS by defining specific policies for each role, as shown in Figure 2. RLS is a security feature that facilitates policies definitions that restrict, on a per-user basis, whether a row of a table can be returned by normal queries or modified by data modification commands (PostgreSQL, 2022). I used RLS to control access to rows in the "Employee" table based on the employee's job role and assigned store. This was accomplished by creating policies that used the "USING" clause to filter the rows visible to a role, based on the "job_role" attribute in the "EmployeeJobDetails" table. The policy for the "Store_manager" role restricts access to rows where the “emp_id” in the “Employee” table matches those in the “EmployeeJobDetails” with a “job_role” of “Store_manager”. This way, a store manager can only view data of employees from their own store.

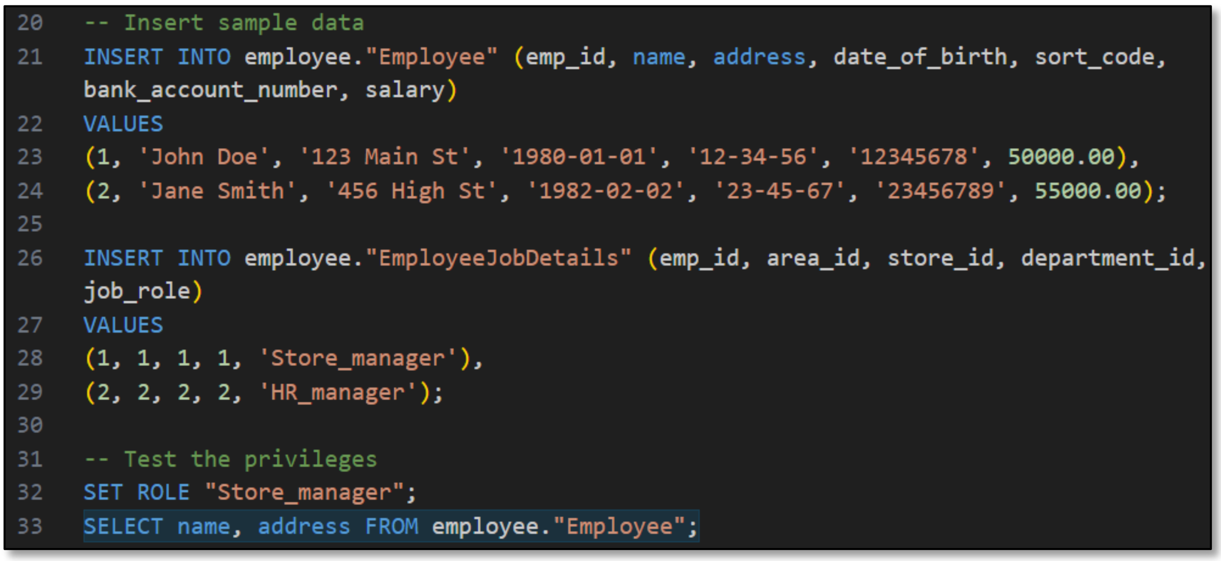
I followed by using PostgreSQL’s “GRANT” command to assign the column-level permissions to each role according to the requirements. For isntance, the “Store_manger” and “Area_manager” roles were granted access to the “name” and “address” columns of the “Employee” table shown in Figure 3.



```sql
-- Grant specific column access
GRANT SELECT(name, address) ON employee."Employee" TO Store_manager;
GRANT SELECT(name, address, date_of_birth, salary) ON employee."Employee" TO HR_manager;
GRANT SELECT(name) ON employee."Employee" TO admin;
GRANT SELECT(name, address, date_of_birth, sort_code, bank_account_number, salary) ON
employee."Employee" TO Finance_manager;
GRANT SELECT(name, address) ON employee."Employee" TO Area_manager;
```

Figure 3 - CLS for Each Role

To validate the effectiveness of the implemented security measures, I created test data for the “Employee” and “EmployeeJobDetails” tables. I inserted records into these tables with hypothetical employee details and their corresponding job roles. Figure 4 proves the accuracy of the roles and the privileges assigned. This script should only display the employee’s name and address for the rows where “job_role” is “Store_manager”. If the user tries to access other columns the system will output an error message.

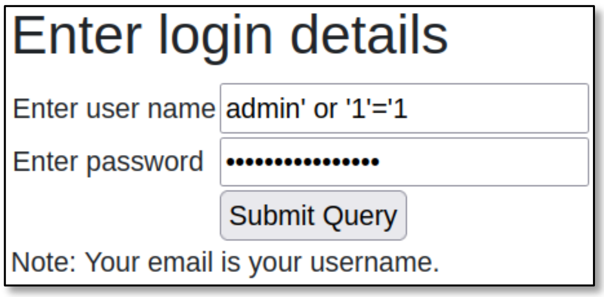


```sql
-- Insert sample data
INSERT INTO employee."Employee" (emp_id, name, address, date_of_birth, sort_code,
bank_account_number, salary)
VALUES
(1, 'John Doe', '123 Main St', '1980-01-01', '12-34-56', '12345678', 50000.00),
(2, 'Jane Smith', '456 High St', '1982-02-02', '23-45-67', '23456789', 55000.00);

INSERT INTO employee."EmployeeJobDetails" (emp_id, area_id, store_id, department_id,
job_role)
VALUES
(1, 1, 1, 1, 'Store_manager'),
(2, 2, 2, 2, 'HR_manager');

-- Test the privileges
SET ROLE "Store_manager";
SELECT name, address FROM employee."Employee";
```

Figure 4 - Test Data to Verify Security Measures

II. Case 2

1) An SQL injection attack involves the malicious insertion of SQL code into user-provided data, often executed through web-based inputs fields such as a login screen (OWASP, 2013).



Enter login details

Enter user name	admin' or '1'='1
Enter password	••••••••••••••••
	Submit Query

Note: Your email is your username.

Figure 5 – SQL Authentication Bypass Injection Example

For instance, in the web-application shown in Figure 5, I was able to login without correct credentials by employing "true" SQL input statements to successfully bypass authentication security, such as; "' OR '1'='1" into the username field of the login screen of the application. This was successful because the back-end database is vulnerable to SQL injection, meaning that it would interpret the SQL statements as true and proceed to log in the user, effectively bypassing the authentication process (PortSwigger, 2019).

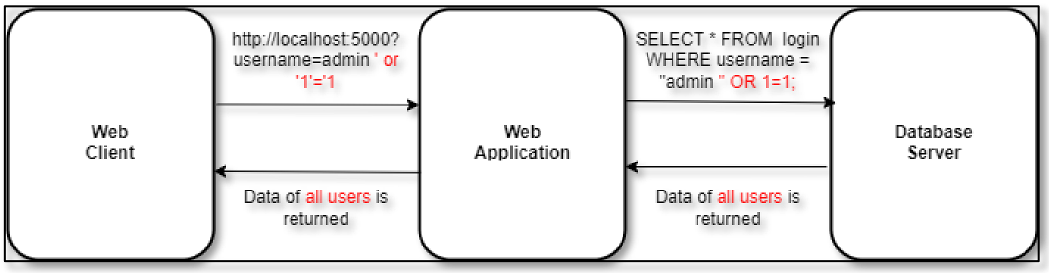
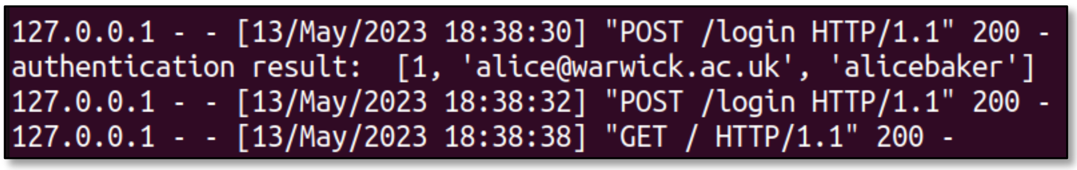

Web Client
http://localhost:5000?username=admin ' or '1'='1
Data of all users is returned
Web Application
SELECT * FROM login WHERE username = "admin " OR 1=1;
Data of all users is returned
Database Server

Figure 6 - SQL Injection Attack Example

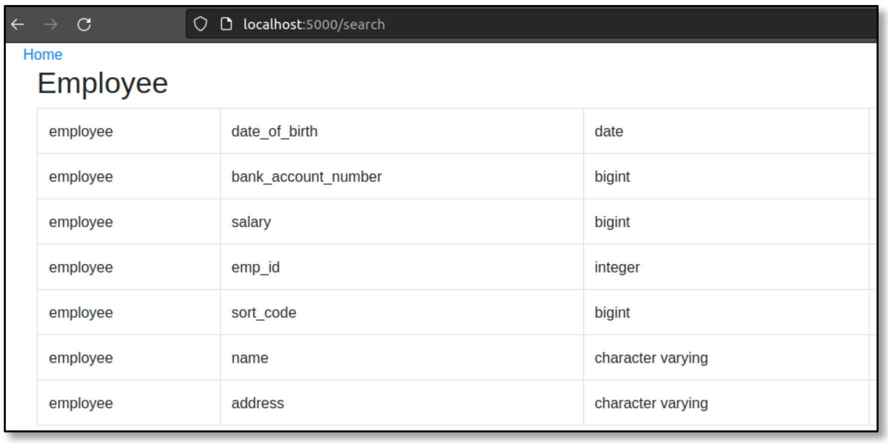
Figure 6, illustrates how the SQL injection attack was successful. In this case, the input query "' ; OR '1'='1" becomes "SELECT * FROM login WHERE username="admin" OR '1'='1';". The "'1'='1'" condition is always true, so the query will always return at least one row, thus bypassing the authentication check. This is proven by Figure 7, which shows a successful login when this input query is entered.



```
127.0.0.1 - - [13/May/2023 18:38:30] "POST /login HTTP/1.1" 200 -
authentication result:  [1, 'alice@warwick.ac.uk', 'alicebaker']
127.0.0.1 - - [13/May/2023 18:38:32] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [13/May/2023 18:38:38] "GET / HTTP/1.1" 200 -
```

Figure 7 - Successful Login Attempt

2) To insert a new employee into the employee table, I started by enumerating the database using SQL queries such as: "… ; SELECT table_name, column_name, data_type FROM information_schema.columns WHERE table_name ='employee';--" to figure out what does the employee table contain. After executing the command successfully into the "/search" page, I was able to obtain all the columns names along with its corresponding data types belonging to the employee table, shown in Figure 8.


localhost:5000/search

Home

Employee

employee	date_of_birth	date
employee	bank_account_number	bigint
employee	salary	bigint
employee	emp_id	integer
employee	sort_code	bigint
employee	name	character varying
employee	address	character varying

Figure 8 - Columns Names and Data Types in Employee Table

I then followed by crafting a malicious SQL query string, that would request the database to insert a new employee. First, I used a simple query " '; SELECT * FROM employee; --'" to obtain all the current employees inside the employee table. I did this to figure out the last "emp_id" to insert the new employee into the table appropriately. With this attribute value, I then crafted the SQL injection string:

" '; INSERT INTO employee VALUES('5', 'Attacker 1', '10 Down Street', '1999-11-01', '112233', '123456789', '999000'); --'"

In this query, the initial statement is ended prematurely by the semicolon (";"), then an entirely new "INSERT" statement is introduced to add a new row to the employee table. The double hyphen ("--") at the end is a comment marker in SQL, which causes the database to ignore anything that follows it, effectively nullifying the rest of the original query. Because of this, I was able to successfully input a new employee record into the employee table as proven in Figure 9.

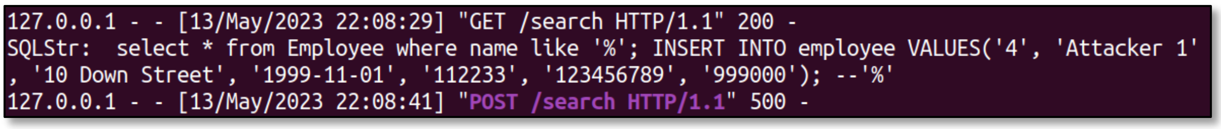
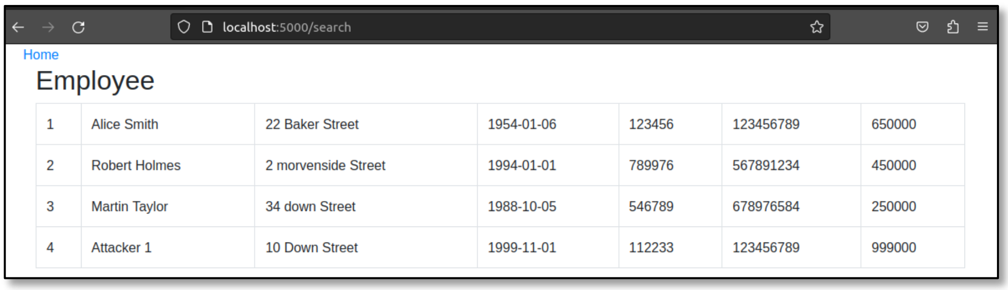

```
127.0.0.1 - - [13/May/2023 22:08:29] "GET /search HTTP/1.1" 200 -
SQLStr:  select * from Employee where name like '%'; INSERT INTO employee VALUES('4', 'Attacker 1'
, '10 Down Street', '1999-11-01', '112233', '123456789', '999000'); --'%'
127.0.0.1 - - [13/May/2023 22:08:41] "POST /search HTTP/1.1" 500 -
```

Figure 9 - Malicious SQL query string injection

To validate this, I used again a simple query " ' ; SELECT * FROM employee; --' " to obtain all the current employees inside the employee table, Figure 10, shows the output of this query.


localhost:5000/search

Home

Employee

1	Alice Smith	22 Baker Street	1954-01-06	123456	123456789	650000
2	Robert Holmes	2 morvenside Street	1994-01-01	789976	567891234	450000
3	Martin Taylor	34 down Street	1988-10-05	546789	678976584	250000
4	Attacker 1	10 Down Street	1999-11-01	112233	123456789	999000

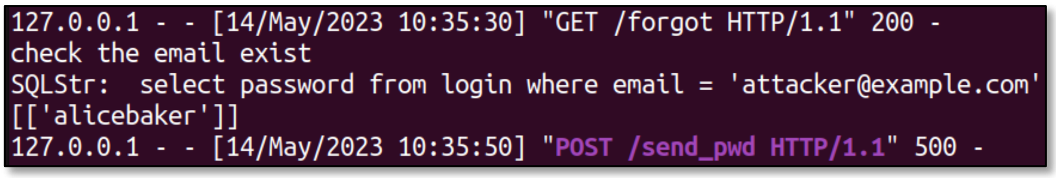
Figure 10 - Successful Employee Record Insertion

3) In the previous case, the SQL injection attack could fail for serval reasons. If the system uses SQL error message handling, the system would not real any sensitive information in error messages, which can prevent an attack from getting information about current employee information such as "email" addresses or attributes values like "emp_id". The system could also be integrating SQL parameterisation, so any SQL injection would be treated as a string rather than as an intended command. Other defences include the use of a web application firewalls (WAF), which can detect and block SQL injection attempts, limiting database permissions so that even successful SQL injection cannot make any changes to the database.

4) To accomplish this, I performed initial reconnaissance to obtain a current employee email address information, to then alter that email to an email I am in control of to finally request a new password. First, I successfully attempted to login into the website using a previously crafted SQL query " ' ; '1'='1;--". Upon the successful login, I was able to obtain a list of current employees along with their corresponding emails, I used this information to craft another SQL query:

"alice@warwick.ac.uk'; UPDATE login SET email='attacker@example.com' WHERE email='alice@warwick.ac.uk'; --'"

5) The above SQL string will update the system's records and alter an email address of a current employee to an email address that a threat actor has control over, in this case, "attacker@example.com". Following, a successful email change, the attacker then could employ a "forgot password" feature, to retrieve the corresponding password to that email or receive a password reset link. Figure 11 illustrates that after entering the updated email address the system returns the corresponding password that was linked to the previous email address in this case "alicebaker". With this information, a threat actor could access the system using valid credentials without raising any suspicious activity.



```
127.0.0.1 - - [14/May/2023 10:35:30] "GET /forgot HTTP/1.1" 200 -
check the email exist
SQLStr:  select password from login where email = 'attacker@example.com'
[['alicebaker']]
127.0.0.1 - - [14/May/2023 10:35:50] "POST /send_pwd HTTP/1.1" 500 -
```

Figure 11 - Access to Employee's Password

6) The risks of SQL injection are significant, as illustrated above if a threat actor is able to successfully execute a SQL injection attack, they could; modify or delete data, gain unauthorised access to data or even take control of the database server. To prevent this type of attacks, it is crucial to implement the following mitigating measures:

1. **Use of Parameterised Queries**: this method is the most effective way to prevent SQL injection, it enables a web application to distinguish between code and data, no matter what user input is supplied. It works by defining all the SQL code first and then passing each parameter to the query later, this ensures that a potential attacker cannot change the intent of a query, even if they intentionally insert malicious SQL queries strings. (Fadlallah, 2022).
2. **Input Validation and Sensitisation**: input validation is a method that allows a system to check user input, so it conforms to the required format, using a whitelist approach – define what is allowed, by default. This method reduces the chances of malicious inputs passing though that could lead to a SQL injection attack (OWASP, 2021).
3. **Least Privilege**: this principle enforces that each account in the system must have the minimum levels of access necessary to perform its tasks, to limit the potential damage in the event of SQL vulnerability. For instance, if an application only needs to extract data, the database account it uses should not have insert or delete rights (UC Berkeley, 2023).

4. **Regular Updates and Patching:** it is essential to ensure the web application and its back-end database is updated and patched to the latest version to ensure that known bugs and vulnerabilities are fixes, reducing the risk of SQL injection attacks, which can be done through database management systems (DBMS).

III. Case 3

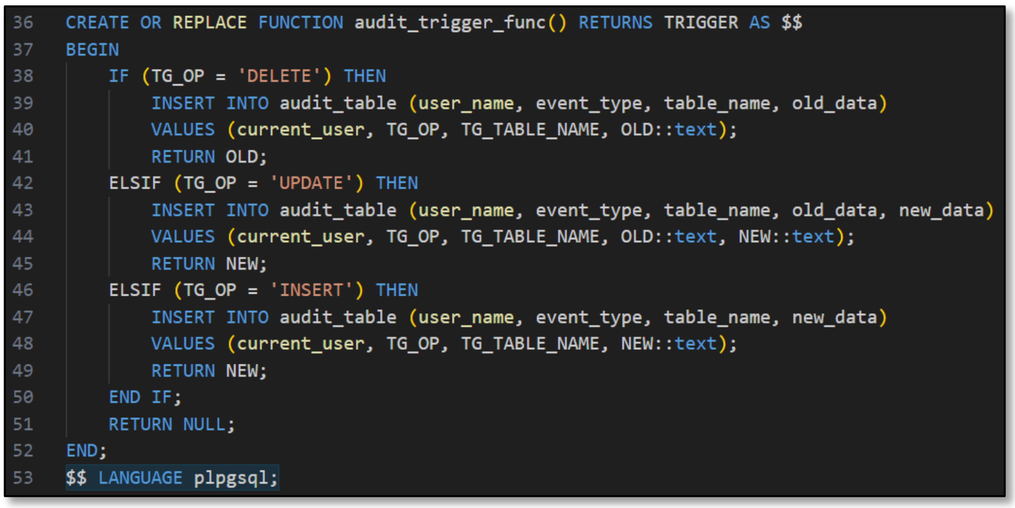
The failed attempts to access the database objects indicates that someone is trying to gain unauthorised access, to sensitive database objects using the credentials of a staff member. This is a significant threat as it could allow a threat actor – individual(s) posing a threat (CSRC, 2018), to access, alter, or delete data within the database. This situation can lead to data breaches or misuse of sensitive employee data, threatening the confidentiality, integrity, and availability of the data stored in the system.

The fact that the manager was in the office late at night raises suspicion. The unauthorised access attempts occurred while the manager was present is concerning, it could mean that the manager was attempting to access the database, which would represent a misuse of authority or a potential insider threat. If the manager's account is compromised, it could be used to perform malicious activities such as data manipulation or deletion, posing a threat to the integrity of the database.

To understand the nature of these threats, additional information is needed. It would be helpful to know the:

- Exact time the unauthorised access attempts were made and compare that with the time the manager was in the office.
- Specific database objects the manager tried to access.
- Manager's role and privileges in the database, and whether there are any existing security measures in place to prevent such incidents.
- Specific database objects that were targeted could help identify the potential motives behind the access attempts.

To mitigate these threats, it is essential to implement an auditing procedure. Auditing can track all changes made to specific tables, including insertions, updates, deletions, and the users who performed these actions (ECA, 2013). Figure 12, creates an audit table that logs all changes in the database. It defines a function that logs changes depending on the operation performed; if data is inserted, updated, or deleted. This function is then used in a trigger event that is attached to the tables that need to be monitored.



```
CREATE OR REPLACE FUNCTION audit_trigger_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO audit_table (user_name, event_type, table_name, old_data)
        VALUES (current_user, TG_OP, TG_TABLE_NAME, OLD::text);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO audit_table (user_name, event_type, table_name, old_data, new_data)
        VALUES (current_user, TG_OP, TG_TABLE_NAME, OLD::text, NEW::text);
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO audit_table (user_name, event_type, table_name, new_data)
        VALUES (current_user, TG_OP, TG_TABLE_NAME, NEW::text);
        RETURN NEW;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Figure 12 - Auditing Implementation

IV. References

Cloudflare (2018). What is a WAF? | Web Application Firewall explained | Cloudflare UK. *Cloudflare*. [online] Available at: https://www.cloudflare.com/en-gb/learning/ddos/glossary/web-application-firewall-waf/.

CSRC (2017). *least privilege - Glossary | CSRC*. [online] csrc.nist.gov. Available at: https://csrc.nist.gov/glossary/term/least_privilege#:~:text=Definition(s)%3A.

CSRC (2018). *threat actor - Glossary | CSRC*. [online] csrc.nist.gov. Available at: https://csrc.nist.gov/glossary/term/threat_actor.

CSRC (2019). *unauthorized access - Glossary | CSRC*. [online] csrc.nist.gov. Available at: https://csrc.nist.gov/glossary/term/unauthorized_access#:~:text=Definition(s)%3A.

ECA (2013). *Audit procedures*. [online] methodology.eca.europa.eu. Available at: https://methodology.eca.europa.eu/aware/PA/Pages/Planning/Audit-procedures.aspx#:~:text=An%20audit%20procedure%20is%20a [Accessed 12 May 2023].

Ellingwood, J. (2013). *How To Use Roles and Manage Grant Permissions in PostgreSQL on a VPS | DigitalOcean*. [online] www.digitalocean.com. Available at: https://www.digitalocean.com/community/tutorials/how-to-use-roles-and-manage-grant-permissions-in-postgresql-on-a-vps-2 [Accessed 8 May 2023].

Fadlallah, H. (2022). *Using parameterized queries to avoid SQL injection*. [online] SQL Shack - articles about database auditing, server performance, data recovery, and more. Available at: https://www.sqlshack.com/using-parameterized-queries-to-avoid-sql-injection/.

Nettles, R.A. (2019). *Data Manipulation: Attacks and Mitigation*. [online] Cyber Security & Information Systems Information Analysis Center [CSIAC]. Available at: https://csiac.org/articles/data-manipulation-attacks-and-mitigation/#:~:text=Data%20manipulation%20attacks%20occur%20when [Accessed 12 May 2023].

OWASP (2013). *SQL Injection*. [online] OWASP. Available at: https://owasp.org/www-community/attacks/SQL_Injection.

OWASP (2021). *SQL Injection Prevention · OWASP Cheat Sheet Series*. [online] Owasp.org. Available at: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

PortSwigger (2019). *What is SQL Injection? Tutorial & Examples*. [online] Portswigger.net. Available at: https://portswigger.net/web-security/sql-injection.

PostgreSQL (2012a). *Creating or Modifying a Table — pgAdmin 4 7.1 documentation*. [online] www.pgadmin.org. Available at: https://www.pgadmin.org/docs/pgadmin4/development/modifying_tables.html [Accessed 8 May 2023].

PostgreSQL (2012b). *Database Roles and Privileges*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/8.1/user-manag.html [Accessed 8 May 2023].

PostgreSQL (2022). *5.8. Row Security Policies*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/ddl-rowsecurity.html.

PostgreSQL (2023). *37.15. column_privileges*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/infoschema-column-privileges.html [Accessed 12 May 2023].

Statista (2022). *Frequency of password resets worldwide 2022*. [online] Statista. Available at: https://www.statista.com/statistics/1303484/frequency-of-password-resets-worldwide/.

UC Berkeley (2023). *How to Protect Against SQL Injection Attacks | Information Security Office*. [online] security.berkeley.edu. Available at: https://security.berkeley.edu/education-awareness/how-protect-against-sql-injection-attacks.