Critical Threats in Modern Systems
Step 4 - Second Implementation of the Database Application System (PostgreSQL)

Jair Ramirez

CS 4332 Introduction to Database Systems

Texas State University

April 20th, 2025

Abstract

In this step, we port our fully normalized schema to PostgreSQL to demonstrate cross-DBMS compatibility and robustness. After creating tables with all keys and constraints, we load sample data and execute the suite of ten core queries—capturing psql outputs (including the PostgreSQL version banner) as screenshots. This phase validates our design in a second environment and highlights any SQL dialect considerations for future application development.

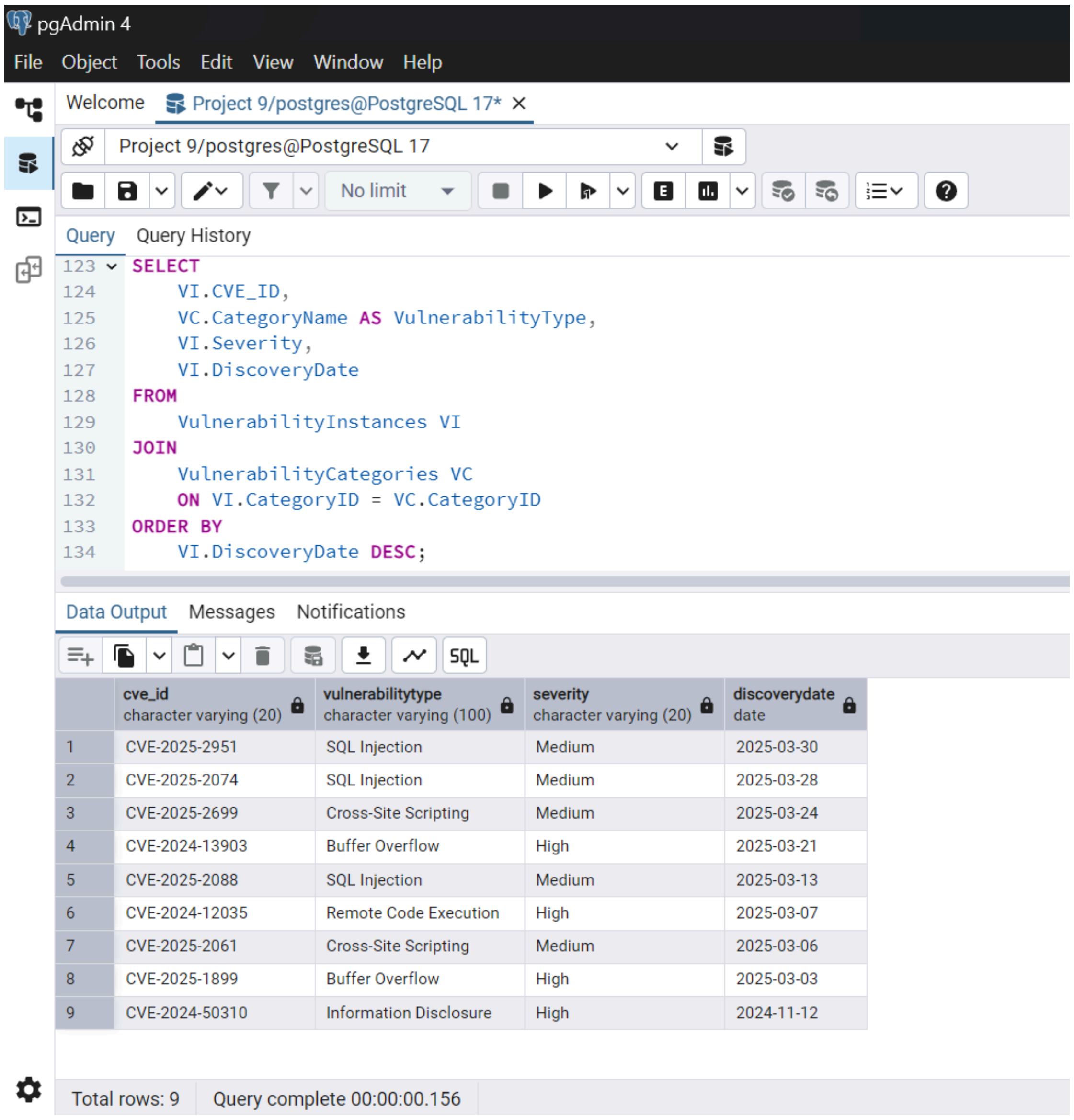
Section A - Second Implementation (PostgreSQL)

Here we implement the database in PostgreSQL using psql. We recreate our tables, enforce all primary/foreign keys and constraints, then run each of the ten core queries from Step 3. Screenshots include the PostgreSQL version banner to prove platform specificity.

1. ***Query 1:*** Retrieve a list of all vulnerability instances, showing each CVE ID, the corresponding vulnerability type, severity, and discovery date, sorted by discovery date in descending order (most recent first).

```sql
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Severity,
    VI.DiscoveryDate
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC
    ON VI.CategoryID = VC.CategoryID
ORDER BY
    VI.DiscoveryDate DESC;
```

This query provides a comprehensive overview of all vulnerabilities recorded in the system. By retrieving the CVE IDs, vulnerability types, severity levels, and discovery dates, it allows security analysts and decision makers to quickly identify the most recent vulnerabilities. This information is crucial for prioritizing remediation efforts and ensuring that the latest threats are addressed promptly.


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query
Query History
Data Output
Messages
Notifications
SQL

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Severity,
    VI.DiscoveryDate
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC
    ON VI.CategoryID = VC.CategoryID
ORDER BY
    VI.DiscoveryDate DESC;
```

	cve_id character varying (20)	vulnerabilitytype character varying (100)	severity character varying (20)	discoverydate date
1	CVE-2025-2951	SQL Injection	Medium	2025-03-30
2	CVE-2025-2074	SQL Injection	Medium	2025-03-28
3	CVE-2025-2699	Cross-Site Scripting	Medium	2025-03-24
4	CVE-2024-13903	Buffer Overflow	High	2025-03-21
5	CVE-2025-2088	SQL Injection	Medium	2025-03-13
6	CVE-2024-12035	Remote Code Execution	High	2025-03-07
7	CVE-2025-2061	Cross-Site Scripting	Medium	2025-03-06
8	CVE-2025-1899	Buffer Overflow	High	2025-03-03
9	CVE-2024-50310	Information Disclosure	High	2024-11-12

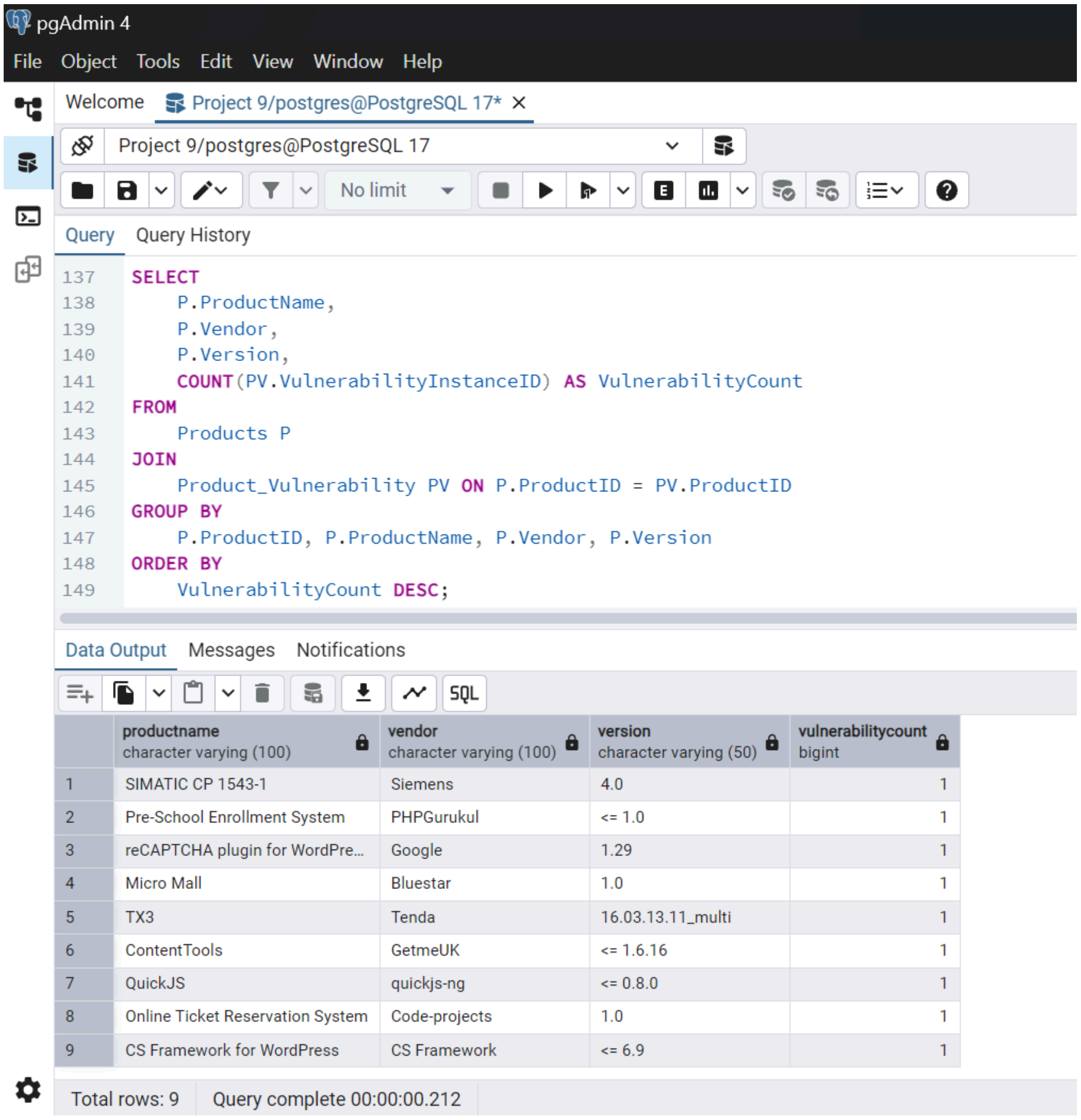
```
Total rows: 9    Query complete 00:00:00.156
```

Figure 1: Execution Output for Query 1 - Vulnerability Overview Dashboard

2. ***Query 2:*** Display a list of products including the vendor and version, along with the number of vulnerabilities affecting each product, sorted by the vulnerability count in descending order.

```
SELECT
    P.ProductName,
    P.Vendor,
    P.Version,
    COUNT(PV.VulnerabilityInstanceID) AS VulnerabilityCount
FROM
    Products P
JOIN
    Product_Vulnerability PV ON P.ProductID = PV.ProductID
GROUP BY
    P.ProductID, P.ProductName, P.Vendor, P.Version
ORDER BY
    VulnerabilityCount DESC;
```

This query aggregates data from the Products and Product_Vulnerability tables. By joining these tables, the query counts the number of vulnerabilities associated with each product. It then displays the product's name, vendor, and version along with this count, sorted from highest to lowest vulnerability count. This is useful for identifying which products are at greatest risk, allowing security teams to prioritize remediation efforts


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query
Query History
Data Output
Messages
Notifications

```
SELECT
    P.ProductName,
    P.Vendor,
    P.Version,
    COUNT(PV.VulnerabilityInstanceID) AS VulnerabilityCount
FROM
    Products P
JOIN
    Product_Vulnerability PV ON P.ProductID = PV.ProductID
GROUP BY
    P.ProductID, P.ProductName, P.Vendor, P.Version
ORDER BY
    VulnerabilityCount DESC;
```

	productname character varying (100)	vendor character varying (100)	version character varying (50)	vulnerabilitycount bigint
1	SIMATIC CP 1543-1	Siemens	4.0	1
2	Pre-School Enrollment System	PHPGurukul	<= 1.0	1
3	reCAPTCHA plugin for WordPre...	Google	1.29	1
4	Micro Mall	Bluestar	1.0	1
5	TX3	Tenda	16.03.13.11_multi	1
6	ContentTools	GetmeUK	<= 1.6.16	1
7	QuickJS	quickjs-ng	<= 0.8.0	1
8	Online Ticket Reservation System	Code-projects	1.0	1
9	CS Framework for WordPress	CS Framework	<= 6.9	1

Total rows: 9   Query complete 00:00:00.212

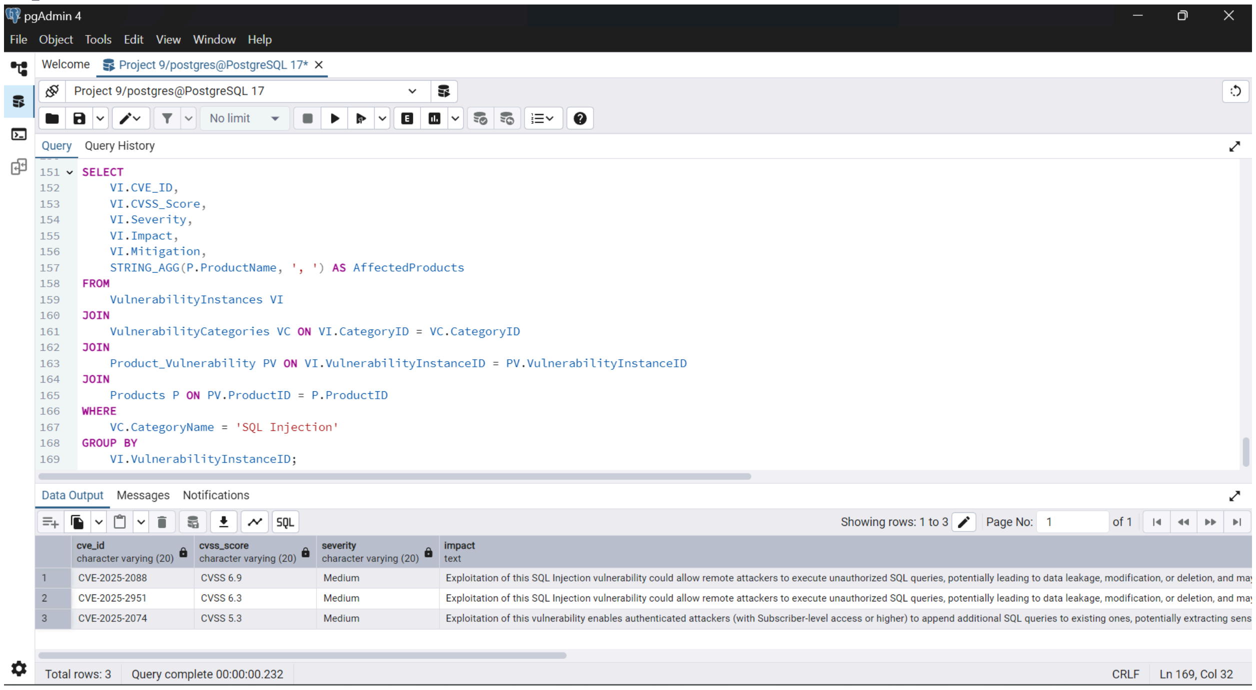
Figure 2: Execution Output for Query 2 – Vulnerability Count by Product

3. ***Query 3:*** Retrieve details for all vulnerabilities of type 'SQL Injection'—including the CVE ID, CVSS score, severity, impact, and mitigation details—and list the names of all products affected by each of these vulnerabilities.

```
SELECT
    VI.CVE_ID,
    VI.CVSS_Score,
    VI.Severity,
    VI.Impact,
    VI.Mitigation,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VC.CategoryName = 'SQL Injection'
```

```
GROUP BY
    VI.VulnerabilityInstanceID;
```

This query narrows down the results to only include vulnerabilities classified as 'SQL Injection.' For each case, it pulls essential details like the CVE ID, CVSS score, severity level, potential impact, and recommended mitigation steps. It also compiles the names of all affected products using the STRING_AGG function, creating a unified view. This output gives security teams a clearer understanding of the risks tied to SQL Injection flaws and highlights exactly which products need focused remediation.

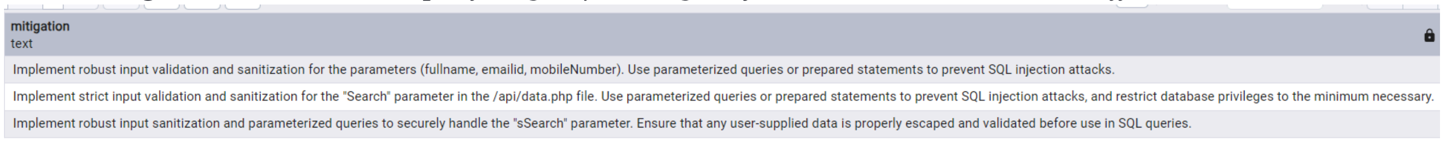


```
SELECT
    VI.CVE_ID,
    VI.CVSS_Score,
    VI.Severity,
    VI.Impact,
    VI.Mitigation,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VC.CategoryName = 'SQL Injection'
GROUP BY
    VI.VulnerabilityInstanceID;
```

	cve_id character varying (20)	cvss_score character varying (20)	severity character varying (20)	impact text
1	CVE-2025-2088	CVSS 6.9	Medium	Exploitation of this SQL Injection vulnerability could allow remote attackers to execute unauthorized SQL queries, potentially leading to data leakage, modification, or deletion, and ma
2	CVE-2025-2951	CVSS 6.3	Medium	Exploitation of this SQL Injection vulnerability could allow remote attackers to execute unauthorized SQL queries, potentially leading to data leakage, modification, or deletion, and ma
3	CVE-2025-2074	CVSS 5.3	Medium	Exploitation of this vulnerability enables authenticated attackers (with Subscriber-level access or higher) to append additional SQL queries to existing ones, potentially extracting sens

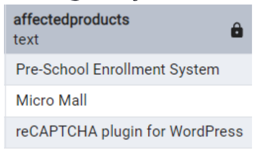
Total rows: 3   Query complete 00:00:00.232   CRLF   Ln 169, Col 32

Figure 3: Execution Output for Query 3 – SQL Injection Vulnerabilities and Affected Products



mitigation text
Implement robust input validation and sanitization for the parameters (fullname, emailid, mobileNumber). Use parameterized queries or prepared statements to prevent SQL injection attacks.
Implement strict input validation and sanitization for the "Search" parameter in the /api/data.php file. Use parameterized queries or prepared statements to prevent SQL injection attacks, and restrict database privileges to the minimum necessary.
Implement robust input sanitization and parameterized queries to securely handle the "sSearch" parameter. Ensure that any user-supplied data is properly escaped and validated before use in SQL queries.

Figure 4: Execution Output for Query 3 – SQL Injection Vulnerabilities and Affected Products (2)



affectedproducts text
Pre-School Enrollment System
Micro Mall
reCAPTCHA plugin for WordPress

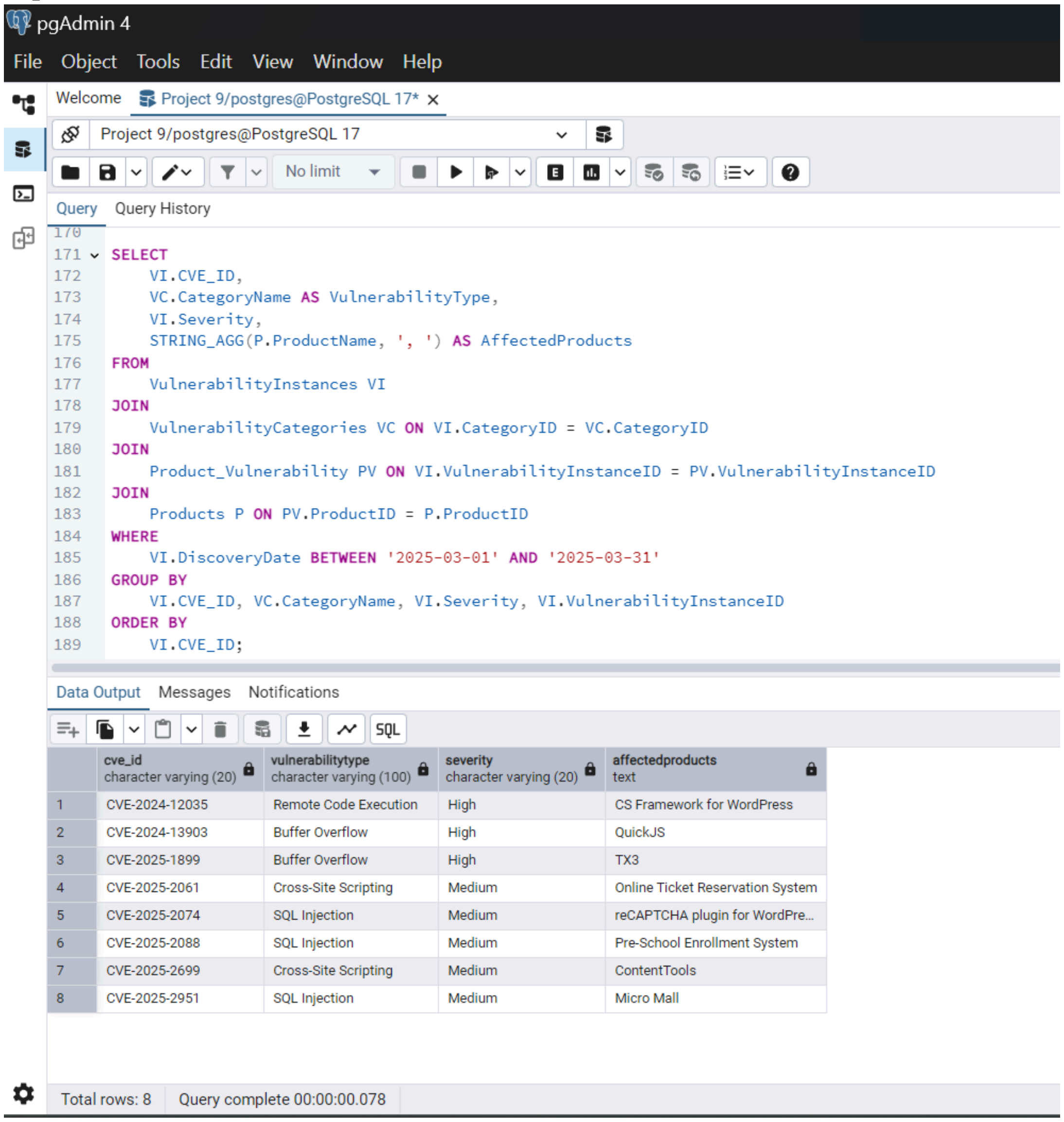
Figure 5: Execution Output for Query 3 – SQL Injection Vulnerabilities and Affected Products (3)

4. ***Query 4:*** Retrieve all vulnerabilities that were discovered in March 2025, displaying their CVE IDs, vulnerability types, severity levels, and the names of the products affected.

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Severity,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
```

```
WHERE
    VI.DiscoveryDate BETWEEN '2025-03-01' AND '2025-03-31'
GROUP BY
    VI.CVE_ID, VC.CategoryName, VI.Severity, VI.VulnerabilityInstanceID
ORDER BY
    VI.CVE_ID;
```

This query is useful for focusing on a specific time period, allowing security teams to analyze vulnerabilities discovered during that month and quickly identify the risk associated with the affected products.



```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Severity,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VI.DiscoveryDate BETWEEN '2025-03-01' AND '2025-03-31'
GROUP BY
    VI.CVE_ID, VC.CategoryName, VI.Severity, VI.VulnerabilityInstanceID
ORDER BY
    VI.CVE_ID;
```

	cve_id character varying (20)	vulnerabilitytype character varying (100)	severity character varying (20)	affectedproducts text
1	CVE-2024-12035	Remote Code Execution	High	CS Framework for WordPress
2	CVE-2024-13903	Buffer Overflow	High	QuickJS
3	CVE-2025-1899	Buffer Overflow	High	TX3
4	CVE-2025-2061	Cross-Site Scripting	Medium	Online Ticket Reservation System
5	CVE-2025-2074	SQL Injection	Medium	reCAPTCHA plugin for WordPre...
6	CVE-2025-2088	SQL Injection	Medium	Pre-School Enrollment System
7	CVE-2025-2699	Cross-Site Scripting	Medium	ContentTools
8	CVE-2025-2951	SQL Injection	Medium	Micro Mall

Total rows: 8   Query complete 00:00:00.078

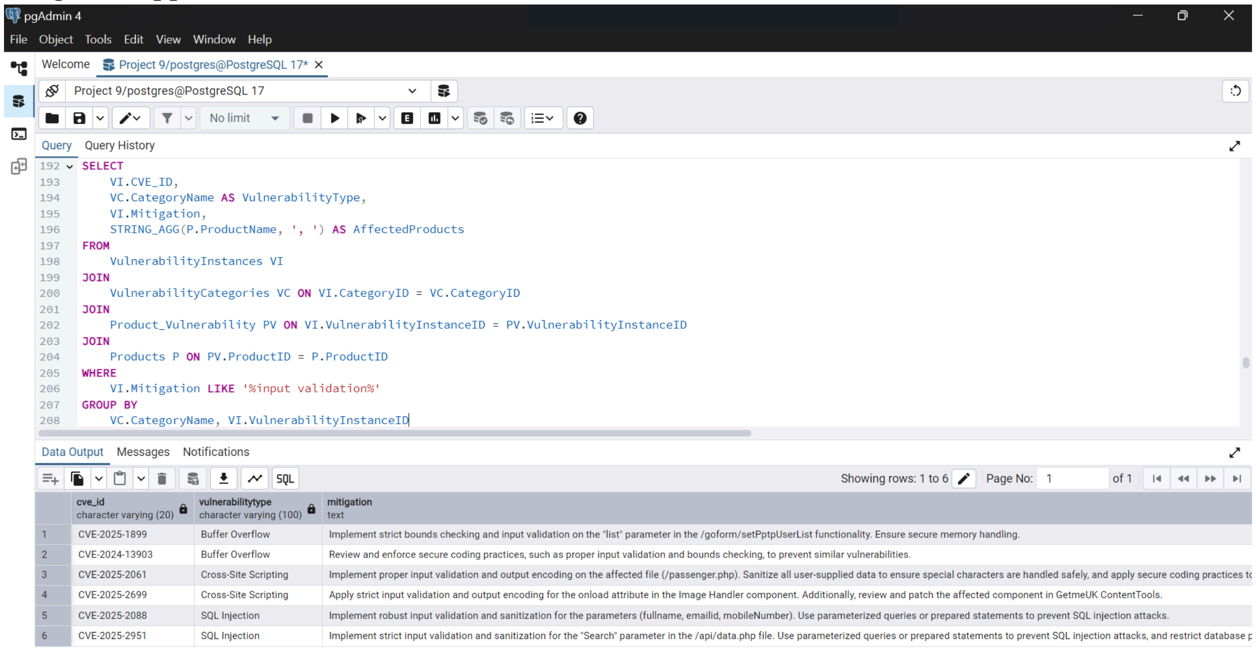
Figure 6: Execution Output for Query 4 – Vulnerabilities Discovered in March 2025

5. ***Query 5:*** Retrieve all vulnerabilities where the mitigation strategy includes the phrase 'input validation'. For each matching record, display the CVE ID, vulnerability type, detailed mitigation steps, and the names of all products affected by these vulnerabilities.

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Mitigation,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
```

```sql
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VI.Mitigation LIKE '%input validation%'
GROUP BY
    VI.CVE_ID, VC.CategoryName, VI.Severity, VI.VulnerabilityInstanceID
```

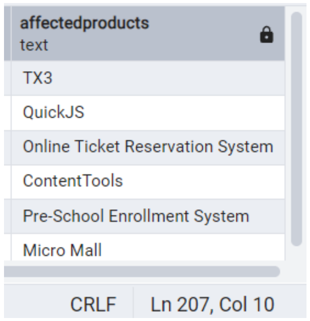
This query filters the vulnerability records to include only those where the mitigation strategy mentions 'input validation'. This is useful for quickly identifying vulnerabilities that rely on input validation as part of their remediation strategy, helping security teams focus on a common mitigation approach.


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query Query History
Data Output Messages Notifications
Showing rows: 1 to 6
Page No: 1 of 1

```sql
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Mitigation,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VI.Mitigation LIKE '%input validation%'
GROUP BY
    VC.CategoryName, VI.VulnerabilityInstanceID
```

	cve_id character varying (20)	vulnerabilitytype character varying (100)	mitigation text
1	CVE-2025-1899	Buffer Overflow	Implement strict bounds checking and input validation on the "list" parameter in the /goform/setPptpUserList functionality. Ensure secure memory handling.
2	CVE-2024-13903	Buffer Overflow	Review and enforce secure coding practices, such as proper input validation and bounds checking, to prevent similar vulnerabilities.
3	CVE-2025-2061	Cross-Site Scripting	Implement proper input validation and output encoding on the affected file (/passenger.php). Sanitize all user-supplied data to ensure special characters are handled safely, and apply secure coding practices t
4	CVE-2025-2699	Cross-Site Scripting	Apply strict input validation and output encoding for the onload attribute in the Image Handler component. Additionally, review and patch the affected component in GetmeUK ContentTools.
5	CVE-2025-2088	SQL Injection	Implement robust input validation and sanitization for the parameters (fullname, emailid, mobileNumber). Use parameterized queries or prepared statements to prevent SQL injection attacks.
6	CVE-2025-2951	SQL Injection	Implement strict input validation and sanitization for the "Search" parameter in the /api/data.php file. Use parameterized queries or prepared statements to prevent SQL injection attacks, and restrict database p

Figure 7: Execution Output for Query 5 – Vulnerabilities with 'Input Validation' in Mitigation



affectedproducts text
TX3
QuickJS
Online Ticket Reservation System
ContentTools
Pre-School Enrollment System
Micro Mall

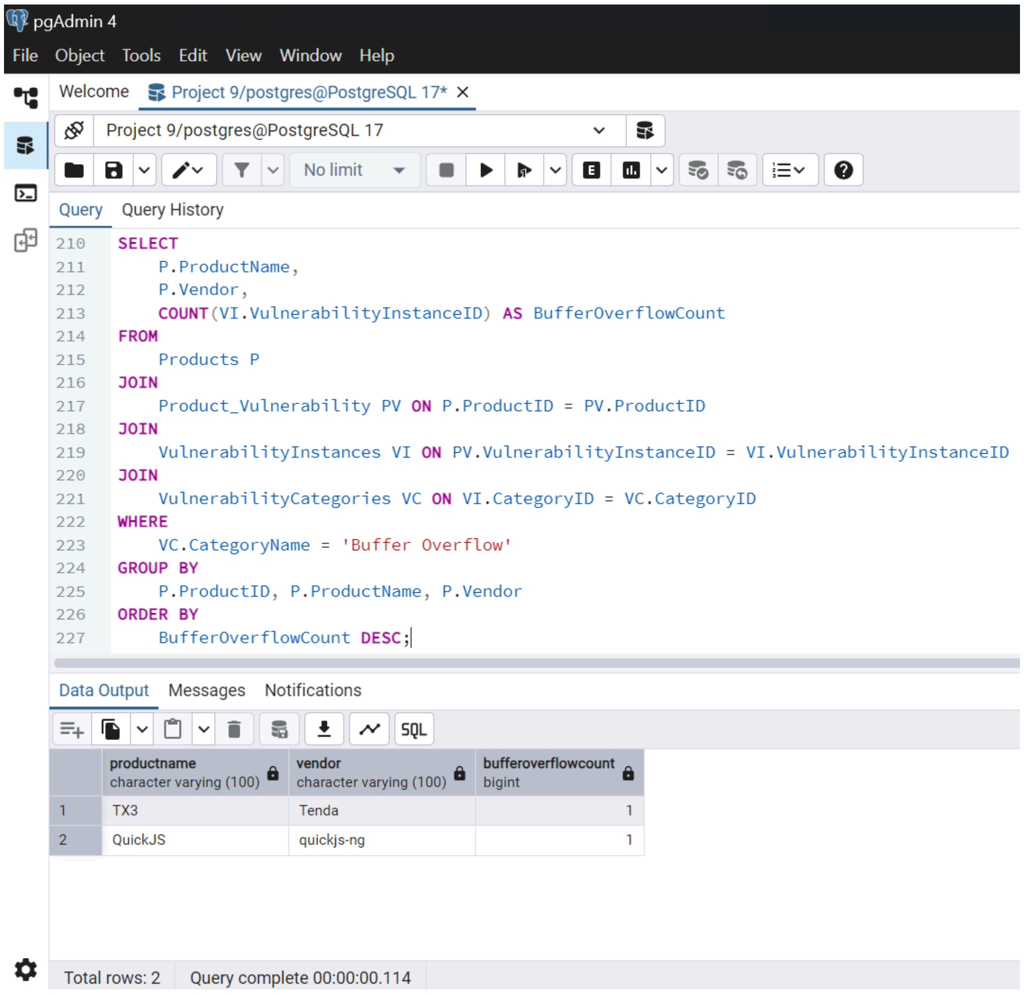
Figure 8: Execution Output for Query 5 – Vulnerabilities with 'Input Validation' in Mitigation (2)

6. ***Query 6:*** Retrieve all products affected by 'Buffer Overflow' vulnerabilities. For each product, display the product name, vendor, and the count of 'Buffer Overflow' vulnerabilities associated with it, ordered by the vulnerability count in descending order.

```sql
SELECT
    P.ProductName,
    P.Vendor,
    COUNT(VI.VulnerabilityInstanceID) AS BufferOverflowCount
FROM
    Products P
JOIN
```

```
    Product_Vulnerability PV ON P.ProductID = PV.ProductID
JOIN
    VulnerabilityInstances VI ON PV.VulnerabilityInstanceID = VI.VulnerabilityInstanceID
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
WHERE
    VC.CategoryName = 'Buffer Overflow'
GROUP BY
    P.ProductID, P.ProductName, P.Vendor
ORDER BY
    BufferOverflowCount DESC;
```

This query focuses on identifying products that are affected by 'Buffer Overflow' vulnerabilities. It joins the Products, Product_Vulnerability, and VulnerabilityInstances tables along with the VulnerabilityCategories table to filter for vulnerabilities that are classified as 'Buffer Overflow'.



```
SELECT
    P.ProductName,
    P.Vendor,
    COUNT(VI.VulnerabilityInstanceID) AS BufferOverflowCount
FROM
    Products P
JOIN
    Product_Vulnerability PV ON P.ProductID = PV.ProductID
JOIN
    VulnerabilityInstances VI ON PV.VulnerabilityInstanceID = VI.VulnerabilityInstanceID
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
WHERE
    VC.CategoryName = 'Buffer Overflow'
GROUP BY
    P.ProductID, P.ProductName, P.Vendor
ORDER BY
    BufferOverflowCount DESC;
```

	productname character varying (100)	vendor character varying (100)	bufferoverflowcount bigint
1	TX3	Tenda	1
2	QuickJS	quickjs-ng	1

Total rows: 2   Query complete 00:00:00.114

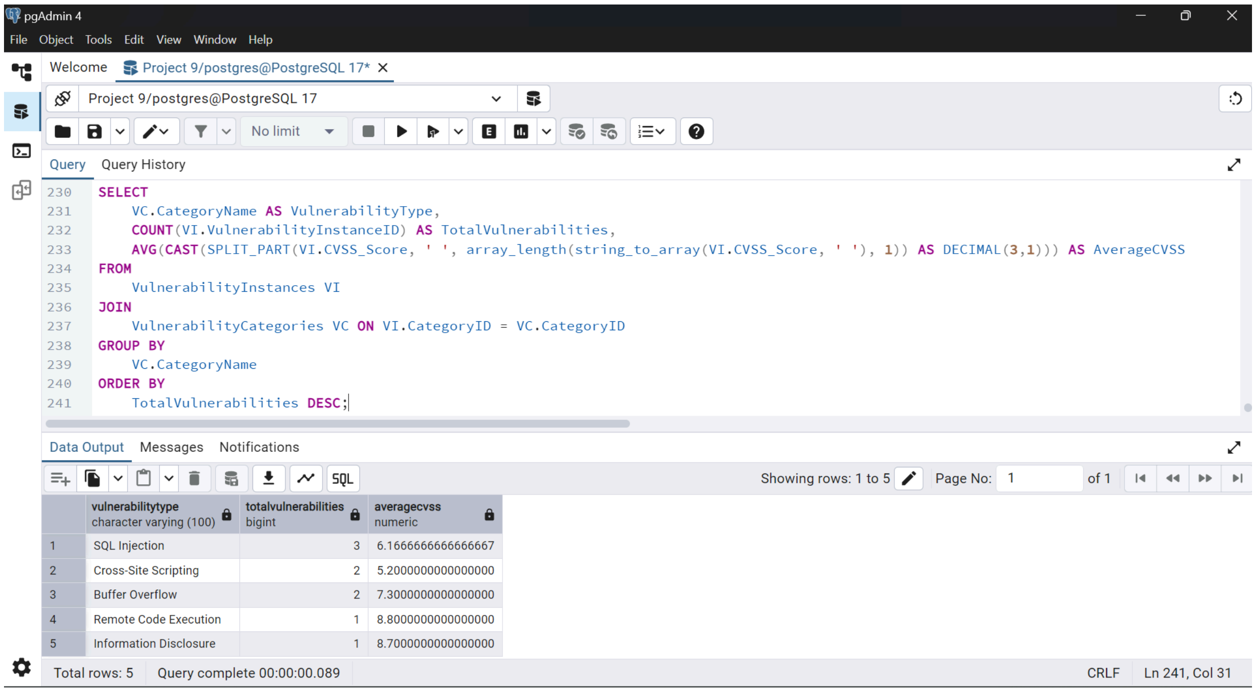
Figure 9: Execution Output for Query 6 – Products Affected by 'Buffer Overflow' Vulnerabilities

7. ***Query 7:*** Retrieve aggregated vulnerability data: for each vulnerability type, display the total number of vulnerabilities and the average CVSS score, sorted by the total count in descending order.

```
SELECT
    VC.CategoryName AS VulnerabilityType,
    COUNT(VI.VulnerabilityInstanceID) AS TotalVulnerabilities,
    AVG(CAST(SPLIT_PART(VI.CVSS_Score, ' ', array_length(string_to_array(VI.CVSS_Score, ' '), 1)) AS
DECIMAL(3,1))) AS AverageCVSS
FROM
```

```
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
GROUP BY
    VC.CategoryName
ORDER BY
    TotalVulnerabilities DESC;
```

This query aggregates vulnerability data by vulnerability type. For each type (e.g., 'SQL Injection', 'Buffer Overflow', etc.), it counts the total number of vulnerability instances. Calculates the average CVSS score. Sorting by the total count in descending order helps identify the vulnerability types with the highest number of occurrences, which can be useful for prioritization and risk assessment.


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query
Query History
Data Output
Messages
Notifications
SQL

```
SELECT
    VC.CategoryName AS VulnerabilityType,
    COUNT(VI.VulnerabilityInstanceID) AS TotalVulnerabilities,
    AVG(CAST(SPLIT_PART(VI.CVSS_Score, ' ', array_length(string_to_array(VI.CVSS_Score, ' '), 1)) AS DECIMAL(3,1))) AS AverageCVSS
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
GROUP BY
    VC.CategoryName
ORDER BY
    TotalVulnerabilities DESC;
```

Showing rows: 1 to 5 Page No: 1 of 1

	vulnerabilitytype character varying (100)	totalvulnerabilities bigint	averagecvss numeric
1	SQL Injection	3	6.1666666666666667
2	Cross-Site Scripting	2	5.2000000000000000
3	Buffer Overflow	2	7.3000000000000000
4	Remote Code Execution	1	8.8000000000000000
5	Information Disclosure	1	8.7000000000000000

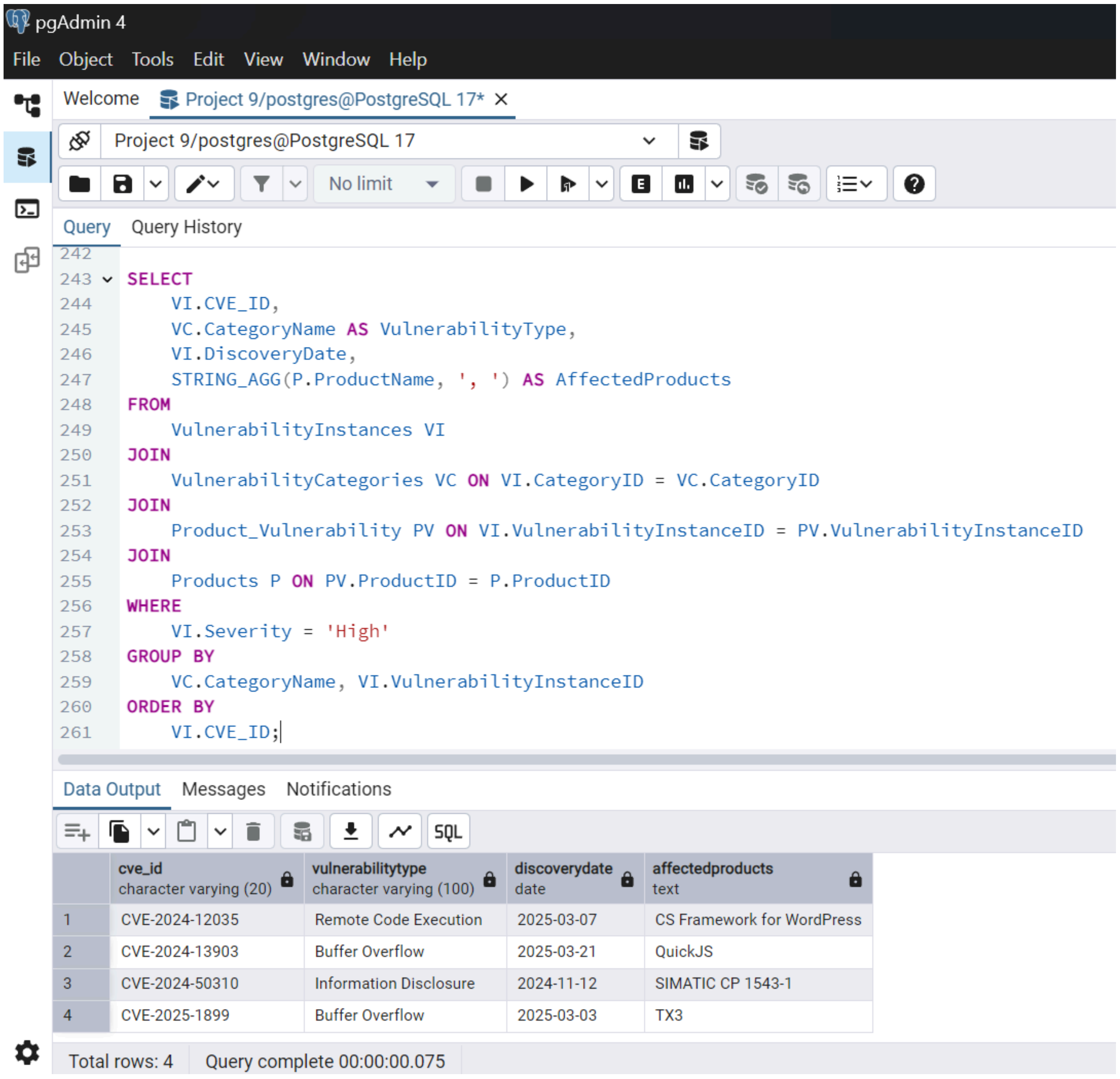
Total rows: 5 Query complete 00:00:00.089 CRLF Ln 241, Col 31

Figure 10: Execution Output for Query 7 – Aggregated Vulnerability Data

8. ***Query 8:*** Retrieve all vulnerabilities with a severity level of 'High' and display their CVE IDs, vulnerability types, and discovery dates, along with the names of the products affected by them. Sort the results by CVE ID.

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.DiscoveryDate,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VI.Severity = 'High'
GROUP BY
    VC.CategoryName, VI.VulnerabilityInstanceID
ORDER BY
    VI.CVE_ID;
```

This query focuses on vulnerabilities with a high severity level. For each vulnerability we collect CVE_ID, VulnerabilityType, DiscoveryDate, AffectedProducts Sorting by CVE_ID provides an organized, easily navigable list of high-severity vulnerabilities, enabling security teams to quickly identify and address the most critical issues.


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query Query History
Data Output Messages Notifications
SQL

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.DiscoveryDate,
    STRING_AGG(P.ProductName, ', ') AS AffectedProducts
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
JOIN
    Product_Vulnerability PV ON VI.VulnerabilityInstanceID = PV.VulnerabilityInstanceID
JOIN
    Products P ON PV.ProductID = P.ProductID
WHERE
    VI.Severity = 'High'
GROUP BY
    VC.CategoryName, VI.VulnerabilityInstanceID
ORDER BY
    VI.CVE_ID;
```

	cve_id character varying (20)	vulnerabilitytype character varying (100)	discoverydate date	affectedproducts text
1	CVE-2024-12035	Remote Code Execution	2025-03-07	CS Framework for WordPress
2	CVE-2024-13903	Buffer Overflow	2025-03-21	QuickJS
3	CVE-2024-50310	Information Disclosure	2024-11-12	SIMATIC CP 1543-1
4	CVE-2025-1899	Buffer Overflow	2025-03-03	TX3

Total rows: 4   Query complete 00:00:00.075

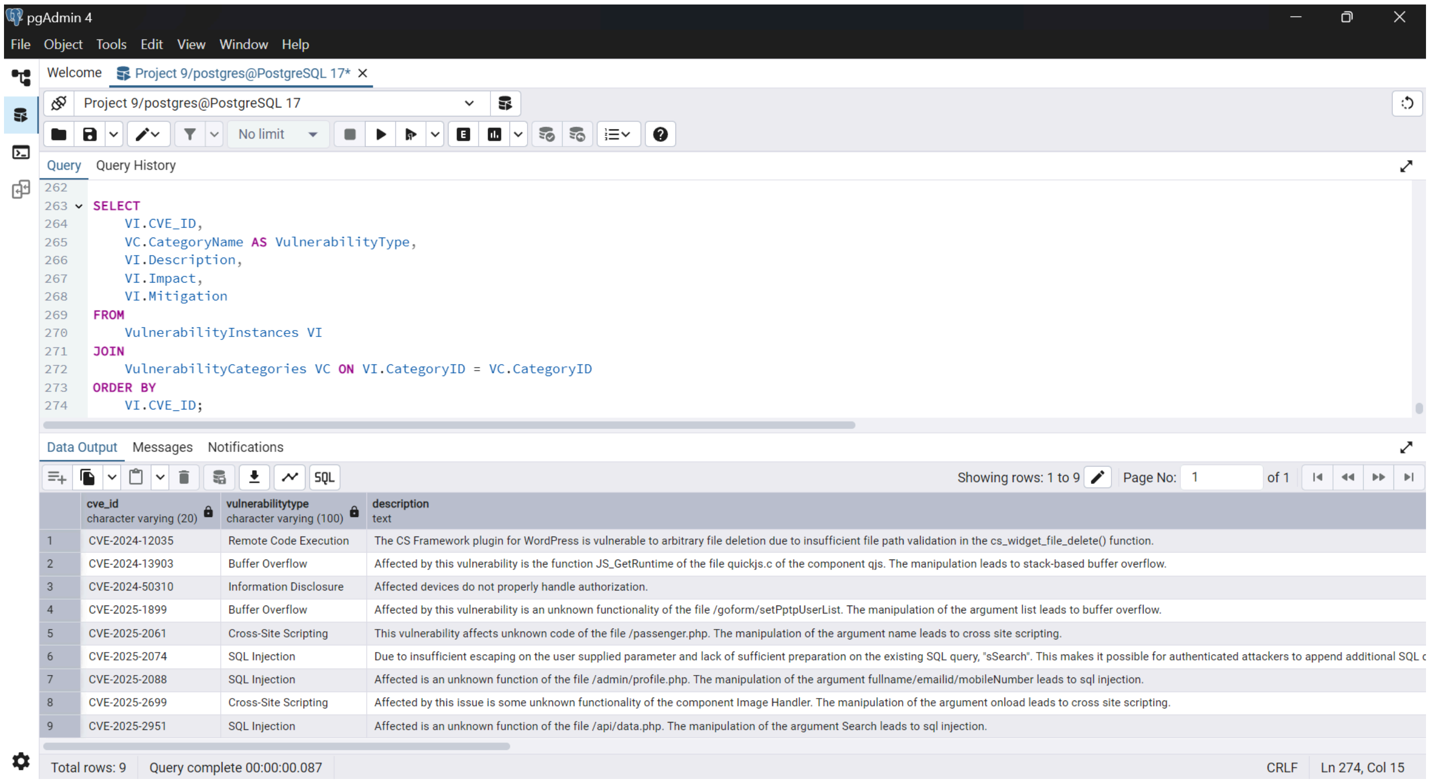
Figure 11: Execution Output for Query 8 – High Severity Vulnerabilities

9. ***Query 9:*** Retrieve a list of all vulnerabilities along with their CVE IDs, vulnerability types, detailed descriptions, impact, and corresponding mitigation strategies, sorted by the CVE ID.

```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Description,
    VI.Impact,
    VI.Mitigation
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
```

```
ORDER BY
    VI.CVE_ID;
```

This query is particularly useful for providing a detailed overview of all vulnerabilities, which helps security analysts, auditors, and management in understanding the nature of each threat and planning appropriate mitigation strategies.



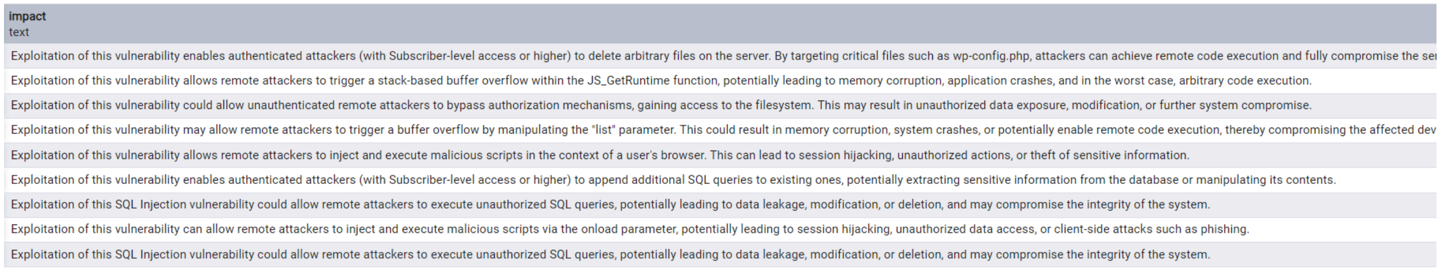
```
SELECT
    VI.CVE_ID,
    VC.CategoryName AS VulnerabilityType,
    VI.Description,
    VI.Impact,
    VI.Mitigation
FROM
    VulnerabilityInstances VI
JOIN
    VulnerabilityCategories VC ON VI.CategoryID = VC.CategoryID
ORDER BY
    VI.CVE_ID;
```

Showing rows: 1 to 9 Page No: 1 of 1

	cve_id character varying (20)	vulnerabilitytype character varying (100)	description text
1	CVE-2024-12035	Remote Code Execution	The CS Framework plugin for WordPress is vulnerable to arbitrary file deletion due to insufficient file path validation in the cs_widget_file_delete() function.
2	CVE-2024-13903	Buffer Overflow	Affected by this vulnerability is the function JS_GetRuntime of the file quickjs.c of the component qjs. The manipulation leads to stack-based buffer overflow.
3	CVE-2024-50310	Information Disclosure	Affected devices do not properly handle authorization.
4	CVE-2025-1899	Buffer Overflow	Affected by this vulnerability is an unknown functionality of the file /goform/setPptpUserList. The manipulation of the argument list leads to buffer overflow.
5	CVE-2025-2061	Cross-Site Scripting	This vulnerability affects unknown code of the file /passenger.php. The manipulation of the argument name leads to cross site scripting.
6	CVE-2025-2074	SQL Injection	Due to insufficient escaping on the user supplied parameter and lack of sufficient preparation on the existing SQL query, "sSearch". This makes it possible for authenticated attackers to append additional SQL c
7	CVE-2025-2088	SQL Injection	Affected is an unknown function of the file /admin/profile.php. The manipulation of the argument fullname/emailid/mobileNumber leads to sql injection.
8	CVE-2025-2699	Cross-Site Scripting	Affected by this issue is some unknown functionality of the component Image Handler. The manipulation of the argument onload leads to cross site scripting.
9	CVE-2025-2951	SQL Injection	Affected is an unknown function of the file /api/data.php. The manipulation of the argument Search leads to sql injection.

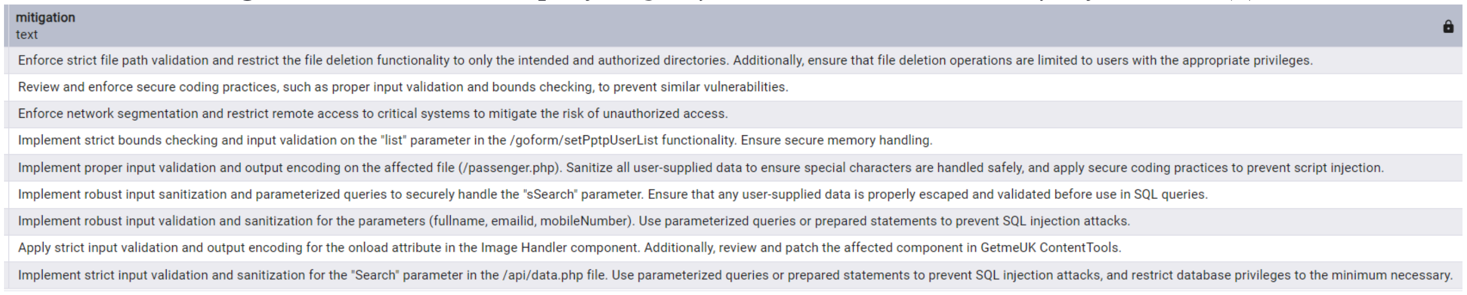
Total rows: 9 Query complete 00:00:00.087 CRLF Ln 274, Col 15

Figure 12: Execution Output for Query 9 – Detailed Vulnerability Information



impact text
Exploitation of this vulnerability enables authenticated attackers (with Subscriber-level access or higher) to delete arbitrary files on the server. By targeting critical files such as wp-config.php, attackers can achieve remote code execution and fully compromise the se
Exploitation of this vulnerability allows remote attackers to trigger a stack-based buffer overflow within the JS_GetRuntime function, potentially leading to memory corruption, application crashes, and in the worst case, arbitrary code execution.
Exploitation of this vulnerability could allow unauthenticated remote attackers to bypass authorization mechanisms, gaining access to the filesystem. This may result in unauthorized data exposure, modification, or further system compromise.
Exploitation of this vulnerability may allow remote attackers to trigger a buffer overflow by manipulating the "list" parameter. This could result in memory corruption, system crashes, or potentially enable remote code execution, thereby compromising the affected dev
Exploitation of this vulnerability allows remote attackers to inject and execute malicious scripts in the context of a user's browser. This can lead to session hijacking, unauthorized actions, or theft of sensitive information.
Exploitation of this vulnerability enables authenticated attackers (with Subscriber-level access or higher) to append additional SQL queries to existing ones, potentially extracting sensitive information from the database or manipulating its contents.
Exploitation of this SQL Injection vulnerability could allow remote attackers to execute unauthorized SQL queries, potentially leading to data leakage, modification, or deletion, and may compromise the integrity of the system.
Exploitation of this vulnerability can allow remote attackers to inject and execute malicious scripts via the onload parameter, potentially leading to session hijacking, unauthorized data access, or client-side attacks such as phishing.
Exploitation of this SQL Injection vulnerability could allow remote attackers to execute unauthorized SQL queries, potentially leading to data leakage, modification, or deletion, and may compromise the integrity of the system.

Figure 13: Execution Output for Query 9 – Detailed Vulnerability Information (2)



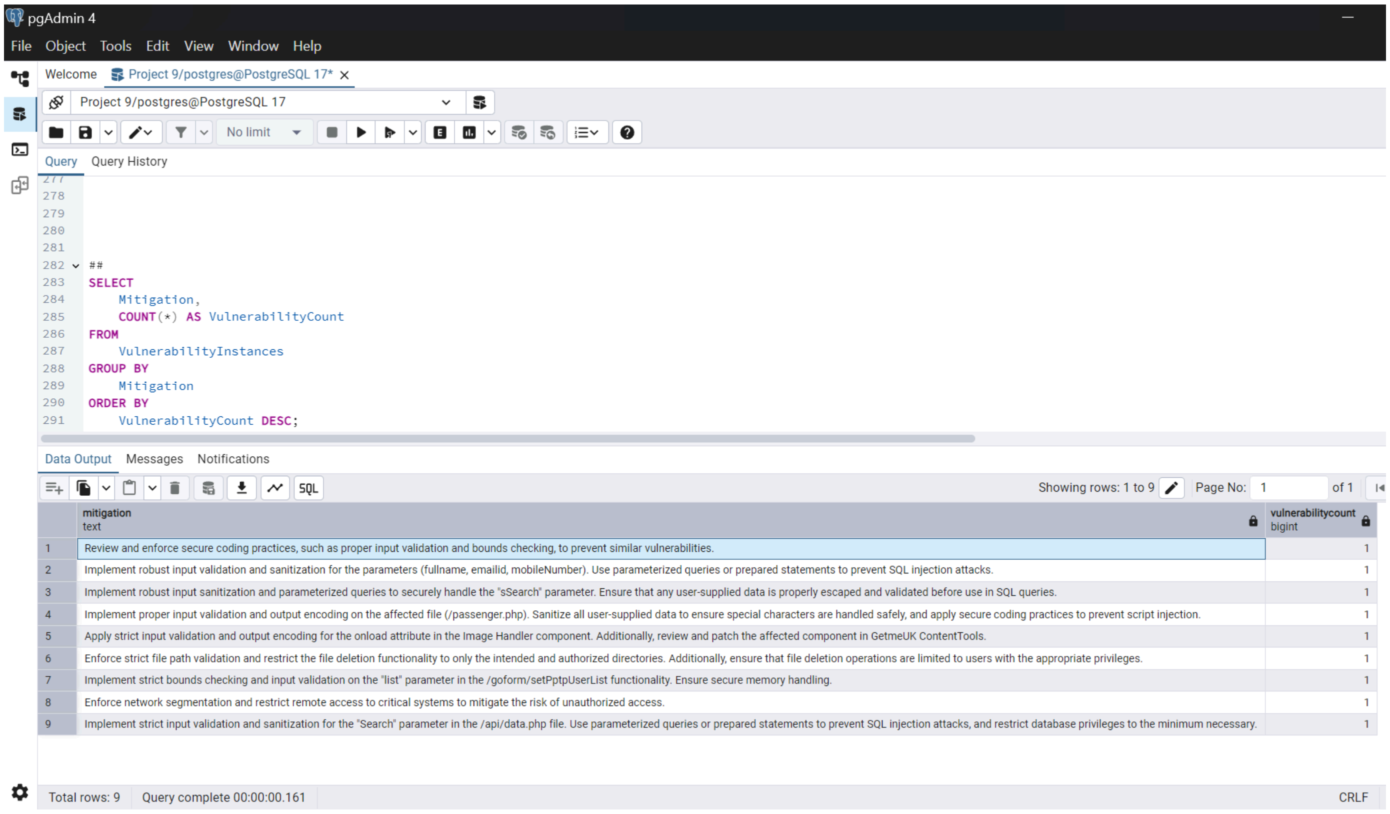
mitigation text
Enforce strict file path validation and restrict the file deletion functionality to only the intended and authorized directories. Additionally, ensure that file deletion operations are limited to users with the appropriate privileges.
Review and enforce secure coding practices, such as proper input validation and bounds checking, to prevent similar vulnerabilities.
Enforce network segmentation and restrict remote access to critical systems to mitigate the risk of unauthorized access.
Implement strict bounds checking and input validation on the "list" parameter in the /goform/setPptpUserList functionality. Ensure secure memory handling.
Implement proper input validation and output encoding on the affected file (/passenger.php). Sanitize all user-supplied data to ensure special characters are handled safely, and apply secure coding practices to prevent script injection.
Implement robust input sanitization and parameterized queries to securely handle the "sSearch" parameter. Ensure that any user-supplied data is properly escaped and validated before use in SQL queries.
Implement robust input validation and sanitization for the parameters (fullname, emailid, mobileNumber). Use parameterized queries or prepared statements to prevent SQL injection attacks.
Apply strict input validation and output encoding for the onload attribute in the Image Handler component. Additionally, review and patch the affected component in GetmeUK ContentTools.
Implement strict input validation and sanitization for the "Search" parameter in the /api/data.php file. Use parameterized queries or prepared statements to prevent SQL injection attacks, and restrict database privileges to the minimum necessary.

Figure 14: Execution Output for Query 9 – Detailed Vulnerability Information (3)

10. ***Query 10:*** Retrieve a distinct list of mitigation strategies used for vulnerabilities, along with the count of vulnerabilities that use each mitigation strategy, sorted by the count in descending order.

```sql
SELECT
    Mitigation,
    COUNT(*) AS VulnerabilityCount
FROM
    VulnerabilityInstances
GROUP BY
    Mitigation
ORDER BY
    VulnerabilityCount DESC;
```

This query retrieves a distinct list of mitigation strategies that are employed to address vulnerabilities across the system and counts the number of occurrences for each strategy. By grouping the vulnerabilities by their mitigation field, the query shows how frequently each mitigation approach is used. This insight can help in understanding prevalent remedial practices and may assist in further optimizing security measures by highlighting the most relied-upon strategies.


pgAdmin 4
File Object Tools Edit View Window Help
Welcome
Project 9/postgres@PostgreSQL 17*
Project 9/postgres@PostgreSQL 17
No limit
Query
Query History
Data Output
Messages
Notifications
SQL

```sql
##
SELECT
    Mitigation,
    COUNT(*) AS VulnerabilityCount
FROM
    VulnerabilityInstances
GROUP BY
    Mitigation
ORDER BY
    VulnerabilityCount DESC;
```

Showing rows: 1 to 9 Page No: 1 of 1

	mitigation text	vulnerabilitycount bigint
1	Review and enforce secure coding practices, such as proper input validation and bounds checking, to prevent similar vulnerabilities.	1
2	Implement robust input validation and sanitization for the parameters (fullname, emailid, mobileNumber). Use parameterized queries or prepared statements to prevent SQL injection attacks.	1
3	Implement robust input sanitization and parameterized queries to securely handle the "sSearch" parameter. Ensure that any user-supplied data is properly escaped and validated before use in SQL queries.	1
4	Implement proper input validation and output encoding on the affected file (/passenger.php). Sanitize all user-supplied data to ensure special characters are handled safely, and apply secure coding practices to prevent script injection.	1
5	Apply strict input validation and output encoding for the onload attribute in the Image Handler component. Additionally, review and patch the affected component in GetmeUK ContentTools.	1
6	Enforce strict file path validation and restrict the file deletion functionality to only the intended and authorized directories. Additionally, ensure that file deletion operations are limited to users with the appropriate privileges.	1
7	Implement strict bounds checking and input validation on the "list" parameter in the /goform/setPptpUserList functionality. Ensure secure memory handling.	1
8	Enforce network segmentation and restrict remote access to critical systems to mitigate the risk of unauthorized access.	1
9	Implement strict input validation and sanitization for the "Search" parameter in the /api/data.php file. Use parameterized queries or prepared statements to prevent SQL injection attacks, and restrict database privileges to the minimum necessary.	1

Total rows: 9 Query complete 00:00:00.161 CRLF

Figure 15: Execution Output for Query 10 – Mitigation Strategies and Vulnerability Counts