Part 1. Software Security

1. Context

Scottishglen is a small energy company with a 6-person development team managing the technical infrastructure. This infrastructure has been highly customized by the team to fit the company's unique business requirements. Recently, the company began receiving messages from a hacktivist group that threatened to target the company based on posts made by the CEO with no clear vector for this attack being clear. With this in mind, the CEO has suggested that security needs to be prioritized in order to potentially secure the business against this oncoming threat. As the IT manager, the researcher has been asked to identify one of the key systems for improvement and to detail how that would be conducted.

The software system chosen by the IT manager for security improvement was the PostgreSQL system. PostgreSQL is an open-source database management system employing structured query language (SQL). This is typically used for a variety of purposes such as account details, product history or any other form of mass information storage – with it being used for payroll and HR systems at Scottishglen .

This report intends to explain core concepts integral to understanding potential vulnerabilities, identify relevant and similar vulnerabilities that are present in systems making use of PostgreSQL and recommend a software design practice to prevent these vulnerabilities in future. It will then be demonstrated practically how static analysis tools can be used to prevent and mitigate these vulnerabilities prior to exploitation.

CVE (Common Vulnerabilities and Exposures) Database

The CVE (Common Vulnerabilities and Exposures) database maintained by MITRE represents an attempt to identify and categorize known software vulnerabilities in a standardized format in order to make sharing data easier across tools and organizations. It stores a repository of reported vulnerabilities with a large amount of relevant information such as the type (Injection, Overflow, XSS), severity, and typically comes accompanied by a proof of concept. The CVE database makes use of a scoring system to identify the severity, called a CVSS (Common Vulnerability Scoring System) wherein 10 represents is the highest and 0 is the lowest.

A variety of types of CVE have been identified in systems making use of PostgreSQL (MITRE, 2025). These include:

- CVE-2024-27299 – 8.8 CVSS – Injection - phpMyFAQ SQL Injection at "Save News"
- CVE-2021-44427 – 9.8 CVSS – Injection - An unauthenticated SQL Injection vulnerability in Rosario Student Information System allows remote attackers to execute PostgreSQL statements
- CVE-2016-0773 – 7.5 CVSS – Overflow & Denial of service – Specific PostgreSQL versions allow remote attackers to cause a denial of service (infinite loop or buffer overflow and crash) via a large Unicode character range in a regular expression
- CVE-2018-1058 – 8.8 CVSS – Input validation & Code execution - Postgresql allowed a user to modify the behaviour of a query for other users. An attacker with a user account could use this flaw to execute code with the permissions of superuser in the database.

SQL Injection

Based on this, it is clear one of the most well-known and common forms of vulnerability impacting a system employing SQL relates to injection, the third most common form of vulnerability overall as per the OWASP top 10 (OWASP, 2024) which details the most critical security risks to web applications. Injection refers to the introduction of user-provided data which lacks appropriate input validation or filtering and allows for unintended behaviour - often to execute commands. Specifically relevant in this case was SQL injection, which ranks third on the CWE most dangerous software weaknesses for 2024 (CWE-89 - Improper Neutralization of Special Elements used in an SQL Command) (CWE, 2025). This can allow for an attacker to query data from a database beyond the intended scope, potentially leaking passwords or other highly sensitive data and therefore understandably represents a significant risk (see Figure 1).

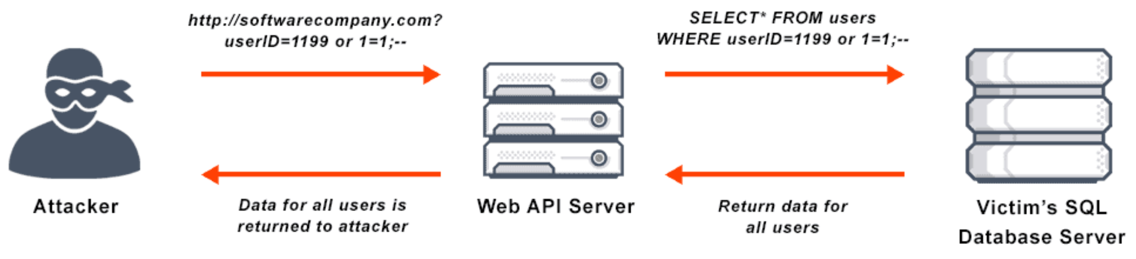


Figure 1 - SQL Injection diagram (VMware, n.d.)

This is especially true in the case of Scottishglen given it is used for the HR and payroll systems, which contain both economically and personally significant data.

2. Recommendation

The development practice deemed best suited to detect this form of security risk was deemed to be static analysis. Static analysis is the process by which source code or a compiled program is analyzed without actually running the program. This is an inexpensive and relatively simple method of identifying potential security faults in an automated fashion, with a number of open source programs available to perform this, ideal for a small 6-person team such as the one at Scottishglen. Given the unrestricted access to source code the development team has, static analysis stands a good chance at identifying potential issues with the server if they exist. The relative automation of this process is of particular note here given the comparatively small team size where dedicating one team member to this long-term may not be an effective use of resources.

Given this, when compared to other methods of assessment, static analysis represents the best combination of success potential and time investment for a small team to catch injection vulnerabilities. For example, a structured code walkthrough or peer review may catch this, though this would likely be significantly more time-consuming depending on the scope of the system and due to the intricacy of SQL queries wherein elements such as a stray colon or other outwardly insignificant mistakes can cause an issue; programmatic tools are ideal.

Dynamic analysis would be likely to find this form of vulnerability but may require greater time investment due to running dynamic analysis tools repeatedly with specific test cases, and may produce a greater number of false positives. Dynamic analysis would also require a fully functional database to provide relevant data, whereas static analysis can find potential security flaws during development as it analyzes source code rather than active environments. Tools such as SQLMap while designed for penetration testing are effective at finding exploits such as this in active SQL systems, though, so this would be a theoretical option should static analysis be ruled out or prove too difficult.

The main difficulty that arises for static analysis of SQL injection relates to the fact that tools that perform this for SQL are generally hard to find, especially given most examples of SQL injection are contained within .php files or other interfaces, as would be the likely case for the HR system as it was unlikely to be directly interfacing with the database and are likely to use a frontend. Most SQL static analysis tools work by analyzing raw .sql files, which are typically not vulnerable to injection exploits as they do not contain dynamic input handling and are instead static scripts which contain other vulnerabilities – meaning those tools tend to highlight structure. There are however a few examples of tools that can perform this role.

3. Implementation

Tool selection

One tool identified for this was Semgrep. Semgrep (Semantic grep) is a free open source Python tool for static analysis with support for over 30 programming languages. Semgrep is notable for being free when compared to other static code analysis tools that would likely also work such as SonarQube but require upfront payment and use a software as a service model. While semgrep simply requires an account for free access to all of their identification rulesets. It has both GUI and CLI variations available, with the CLI version being chosen in this case as it was deemed more accessible for a developer and reduces bloat.

A second static analysis tool in the form of Snyk was employed. Snyk is a software-as-a-service platform for code analysis that has a CLI interface and advertises itself on its IDE integration capabilities. It can be used for free for non business users Snyk has both the advantages and disadvantages that come with being always online – allowing for things such as continuous scanning and an assurance of constantly updated rules, but also requires an internet connection meaning it has less usability overall. This was used to ensure that concurrent results could be obtained and so that the relative usefulness of the tools could be compared. Practically both of these tools are free for an individual user or small teams, but both Snyk and Semgrep have paid options – with Snyk specifically advertising paid subscriptions recommended for development teams so has the option for expansion in future should it be desired.

The RIPS PHP security scanner was attempted to be used to further prove the effectiveness of static analysis but failed to function with contemporary installs of PHP on both Windows and Linux owing to its deprecated state, with it being absorbed into SonarQube, making this unfit for purpose.

CVE

One example of a CVE that could impact a PostgreSQL server relates to the PhpMyFAQ webapp which supports PostgreSQL - namely CVE-2006-6912 – a 7.5 CVSS scoring vulnerability which allows attackers to execute arbitrary SQL commands through the attachment.php file. This exploit relates to the fact that the “uin” variable when submitted on the admin/attachment.php file is improperly sanitized which allows for subsequent code injection. This is considered a simple example of injection as a checklist in a structured code review would likely catch these issues, as the SQL fails to make use of well-known precepts such as prepared statements. However, this is a part of a system that contains over 300 files – if it is considered that there is a similar (or more likely, larger) system in the case of Scottishglen, manual assessment would represent a significant time commitment.

Semgrep

Semgrep was installed through Python with the command:

```
pip install semgrep
```

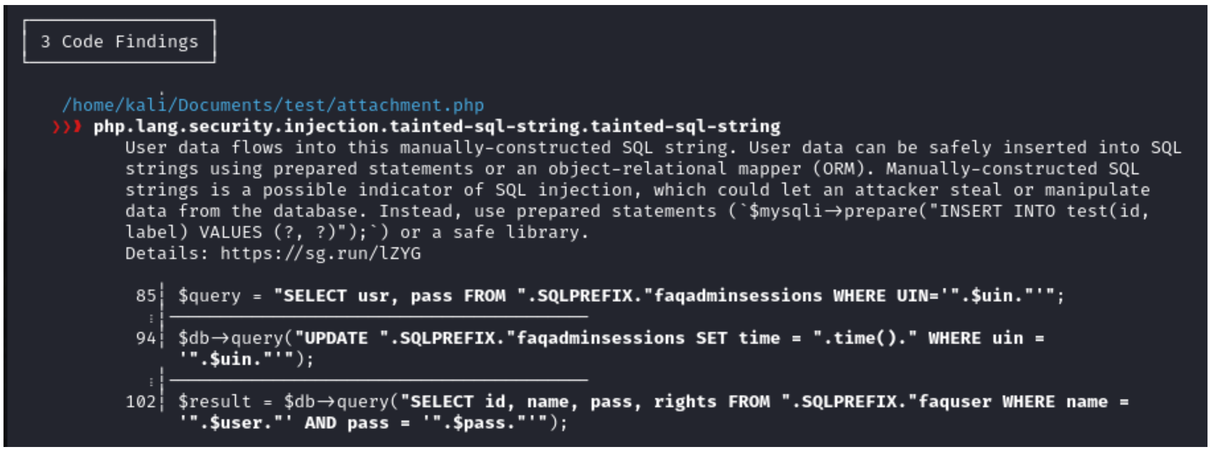
An account was then made on the semgrep website (https://semgrep.dev/) which allowed access to the full rulesets. This provided a token to register the local installation, which was used in the command format:

```
SEMGREP_APP_TOKEN=[TOKEN] semgrep login
```

Semgrep was then ran with the SQL injection identification ruleset using the format:

```
semgrep --config=p/sql-injection [FILE OR FOLDER TO BE ANALYZED]
```

When ran against the source code of the PHP system the following output was visible (see Figure 2)



```
 3 Code Findings

  /home/kali/Documents/test/attachment.php
 ❯❯❱ php.lang.security.injection.tainted-sql-string.tainted-sql-string
       User data flows into this manually-constructed SQL string. User data can be safely inserted into SQL
       strings using prepared statements or an object-relational mapper (ORM). Manually-constructed SQL
       strings is a possible indicator of SQL injection, which could let an attacker steal or manipulate
       data from the database. Instead, use prepared statements (`$mysqli→prepare("INSERT INTO test(id,
       label) VALUES (?, ?)");`) or a safe library.
       Details: https://sg.run/lZYG

        85┆ $query = "SELECT usr, pass FROM ".SQLPREFIX."faqadminsessions WHERE UIN='".$uin."'";
         ⋮┆----------------------------------------
        94┆ $db→query("UPDATE ".SQLPREFIX."faqadminsessions SET time = ".time()." WHERE uin =
          ┆ '".$uin."'");
         ⋮┆----------------------------------------
       102┆ $result = $db→query("SELECT id, name, pass, rights FROM ".SQLPREFIX."faquser WHERE name =
          ┆ '".$user."' AND pass = '".$pass."'");
```

Figure 2 - Semgrep output when ran against the vulnerable page

This clearly provided a verbose and clear identification of the issue related to the "uin" variable as well as suggestions for remediation, doing so almost instantly. Based on this had the secure software practice of static analysis been used then the SQL injection vulnerability would have been prevented through the introduction of prepared statements as per the semgrep recommendations, in turn demonstrating how static analysis can be used to prevent SQL injection in PostgresSQL database systems.

Snyk

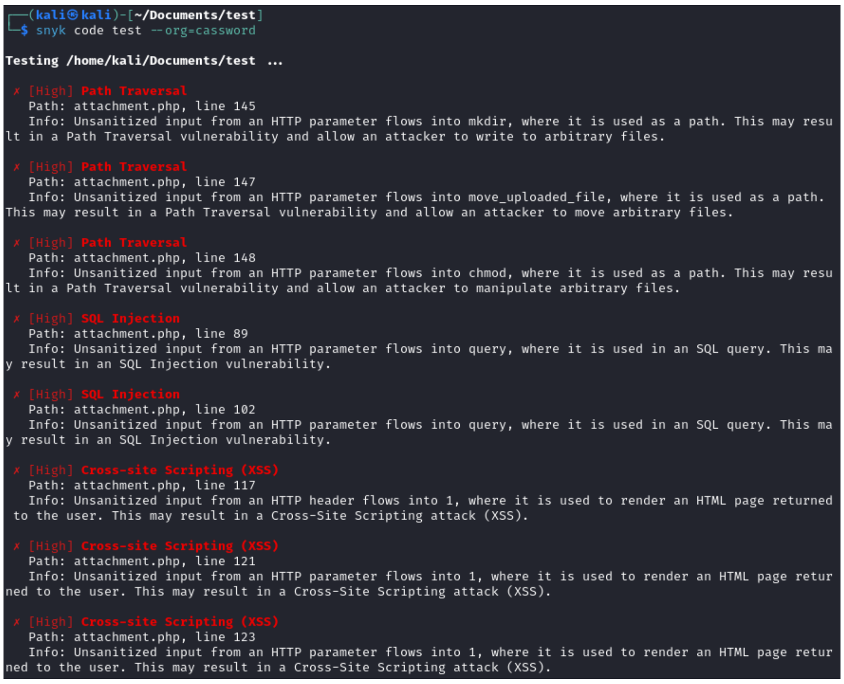
The Snyk CLI was installed with the following commands:

curl https://static.snyk.io/cli/latest/snyk-linux -o snyk
chmod +x ./snyk
mv ./snyk /usr/local/bin/
snyk auth
snyk code test --org=[USERNAME]

This was then run against the file containing the CVE SQL with the command:

```
snyk code test --org=[ORG ID]
```

Where "test" was the folder containing the vulnerable file. As shown below in Figure 3, this detected SQL injection as a potential vulnerability. It did so in less than a minute.



```
┌──(kali㉿kali)-[~/Documents/test]
└─$ snyk code test --org=cassword

Testing /home/kali/Documents/test ...

 ✗ [High] Path Traversal
   Path: attachment.php, line 145
   Info: Unsanitized input from an HTTP parameter flows into mkdir, where it is used as a path. This may resu
lt in a Path Traversal vulnerability and allow an attacker to write to arbitrary files.

 ✗ [High] Path Traversal
   Path: attachment.php, line 147
   Info: Unsanitized input from an HTTP parameter flows into move_uploaded_file, where it is used as a path.
This may result in a Path Traversal vulnerability and allow an attacker to move arbitrary files.

 ✗ [High] Path Traversal
   Path: attachment.php, line 148
   Info: Unsanitized input from an HTTP parameter flows into chmod, where it is used as a path. This may resu
lt in a Path Traversal vulnerability and allow an attacker to manipulate arbitrary files.

 ✗ [High] SQL Injection
   Path: attachment.php, line 89
   Info: Unsanitized input from an HTTP parameter flows into query, where it is used in an SQL query. This ma
y result in an SQL Injection vulnerability.

 ✗ [High] SQL Injection
   Path: attachment.php, line 102
   Info: Unsanitized input from an HTTP parameter flows into query, where it is used in an SQL query. This ma
y result in an SQL Injection vulnerability.

 ✗ [High] Cross-site Scripting (XSS)
   Path: attachment.php, line 117
   Info: Unsanitized input from an HTTP header flows into 1, where it is used to render an HTML page returned
 to the user. This may result in a Cross-Site Scripting attack (XSS).

 ✗ [High] Cross-site Scripting (XSS)
   Path: attachment.php, line 121
   Info: Unsanitized input from an HTTP parameter flows into 1, where it is used to render an HTML page retur
ned to the user. This may result in a Cross-Site Scripting attack (XSS).

 ✗ [High] Cross-site Scripting (XSS)
   Path: attachment.php, line 123
   Info: Unsanitized input from an HTTP parameter flows into 1, where it is used to render an HTML page retur
ned to the user. This may result in a Cross-Site Scripting attack (XSS).
```

Figure 3 - Snyk output when ran against vulnerable SQL

Which clearly indicated a successful detection of the SQL injection vulnerability, among others.

Overall, both static analysis tools succeeded in identifying the SQL vulnerability described in CVE-2006-6912. Based on the fact the result was demonstrated to be reproducible with two distinct static analysis tools, SQL injection vulnerabilities in webapps making use of PostgreSQL were demonstrated to be identifiable and thus preventable with static analysis. This can be applied to the infrastructure at Scottishglen with relative ease to the business, and will likely prove successful in mitigating attacks on the PostgresSQL system. Other secure software design precepts should be used concurrently such as dynamic analysis and secure code review in order to ensure maximum security, with the other systems facing similar scrutiny to ensure all vectors are appropriately defended.

References

CWE, 2025. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').* [Online]
Available at: https://cwe.mitre.org/data/definitions/89.html
[Accessed 25 February 2025].

MITRE, 2025. *Search Results - postgresql.* [Online]
Available at: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=postgresql
[Accessed 25 February 2025].

OWASP, 2024. *OWASP Top Ten.* [Online]
Available at: https://owasp.org/www-project-top-ten/
[Accessed 24 February 2025].

VMware, n.d. *SQL Injection Attack Definition.* [Online]
Available at: https://www.vmware.com/topics/sql-injection-attack
[Accessed 26 February 2025].