CMP417 Engineering Resilient Systems: Software Security

Mairi MacLeod
School of Design and Informatics
Abertay University
DUNDEE, DD1 1HG, UK

ABSTRACT

The company has asked the author to investigate potential security vulnerabilities in their currently deployed software and applications. Their PostgreSQL HR and finance databases were chosen for this investigation. The researcher has chosen to review SQL Injection vulnerabilities such as; CVE-2018-16850, CVE-2021-20229 and CVE-2019-10208. The prevention technique suggested in this report was to implement at least one input sanitation method to prevent a malicious actor from being able to use this exploitation technique. The input sanitation techniques demonstrated were; Input Filtering, Prepared Statements and Restricting Privleges. Of these remedial action has been demonstrated on a Flask application connected to a Docker Postgres container holding an example database. However none of these are limitation free and the author highlights the issues that implementing any of these requires additional efforts from a development team. Additionally there may still be unknown methods for malicious users to bypass them and exploit the database irregardless of what has been done to prevent it.

1. INTRODUCTION

This paper is written to discuss potential security flaws in this organisation's software systems and provide suggestions for mitigating against them. The author has chosen to research and discuss vulnerabilities within the company's PostgreSQL server. This server houses several databases which are used by the HR and finance sectors of the business.

Due to the nature of the databases it can be inferred that very sensitive data is stored within them, making it imperative that they be secure from malicious users. Databases can be seen as an obvious target for malicious hacking organisations and data breaches are becoming ever more popular in recent years. Also with the introduction of heavy government fines for private data being 'leaked', in accordance with the GDPR legislation, it can be a catastrophic blow both financially if any vulnerabilities are exploited.

2. BACKGROUND

2.1. PostgreSQL

PostgreSQL was developed in 1986 by researchers at University of California. It uses the language SQL and is used for creating object-relational databases. This form of database is comprised of tables that all hold a link with each other, in the form of shared columns and data *(PostgreSQL [1], 2021)*.

It is popular with businesses as it supports JSON, XML and can be used for long or complex queries. As it is open source this also means that developers can alter the source code in order to adapt it for whatever use case they need it for. In terms of security PostgreSQL supports basic encryption and verification of SSL certificates. These add a layer of security and can make it more difficult for malicious users to leak or use the data even when they have obtained it due to it not being in plaintext.

2.2. CVEs

Common Vulnerability and Exposures (CVE) is a project created by MITRE organisation to classify and organise vulnerabilities found in software. Each vulnerability is given a CVE number and a CVSS score, which explains how dangerous it is to a machine or application if exploited. The number consists of three parts, first 'CVE' then the year it was disclosed and the third part is assigned by a CNA or CVE Numbering Authority.

There are many vulnerability 'classes' and although there is a considerable amount of overlap in their results each can be exploited differently. In the case of a SQL-based database the main vulnerabilities to be aware of are; arbitrary code execution, memory disclosure, SQL injection and buffer overflows. These can cause Denial of Service (DoS) attacks, the tables to be outputted to the user or even the entire database to be deleted.

2.2.1. CVE-2018-16850

This vulnerability has a score of 7.5, making it fairly severe if exploited. It affects PostgreSQL databases pre 11.1 and is a flaw in the triggering of the pg_dump and pg_upgrade utilities. A malicious user can create a specific command that references either of these which will then allow them to execute arbitrary commands within the database with full privileges.

This vulnerability requires the CREATE privilege and the database to be running a version that is older than October 2019. Although this was patched over two years ago, this is only a few updates ago so if software is not regularly updated the tables can still be exploited. The CREATE privilege is a very basic one and for the case of this organisation, using their web application for HR, it is likely all basic SQL commands are permitted so anyone with access to the application could exploit this particular CVE *(NIST, 2018).*

2.2.2. CVE-2021-20229

In databases using versions of PostgreSQL before 13.2, a specific SELECT query can be inputted that returns the entire table. This is an issue for this business as a malicious user can exploit the CVE to get all of the financial and HR data in the databases. A table is only vulnerable to this if the database is using an older version of PostgreSQL but 13.2 was only released in September 2020 so it only needs to be out of date by one version for this to be exploitable. This also only has a CVSS score of 4.3 so if exploited it is likelyS that little to no real damage would be done to the companies software or data *(NIST, 2021).*

2.2.3. CVE-2019-10208

This vulnerability exists in PostgreSQL databases pre 11.5 and has a score of 8.8, making it very serious if exploited. It allows a person to exploit the SECURITY DEFINER function with specific SQL statement that will then allow them to execute code in the database.

If exploited a malicious user could use it to retrieve all of the information from the tables or even delete it all. By deleting all of the data in either the HR or financial tables the hacktivist group targeting the business would cause major disruption. This could cause the business to have to stop operating for a short period of time as without the tables it can prevent employees being paid correctly. As with CVE-2021-20229 this only affects software using the outdated version of PostgreSQL but this exploit is for even older versions so is more likely to already be mitigated against *(NIST, 2019).*

2.3. SQL Injection

All of the aforementioned CVEs can be exploited using SQLI or SQL injection. As the organisation likely interacts with their databases through a web client or their ASP.NET web application, it can be assumed that these may be exploitable by the malicious hacktivist group currently threatening them.

SQLI is when a user inputs a specifically crafted SQL statement to a database, usually through a web application that interfaces with it, that performs a malicious action. Common reasons this is done are; to retrieve important data from the tables, to delete all of the tables and to place code inside the database that may harm other applications *(Kao, 2018).* This is not an uncommon vulnerability exploit, according to research done in 2013 this particular vulnerability accounts for approximately a quarter of web-based attacks *(Demessie, 2019).*

There are a number of methods for performing SQLI and they are as follows; tautology, union query, logically incorrect, piggy backing, stored procedures and alternate encoding. Ones like tautology-based and logically incorrect attacks exploit the database's logic handling. Union query, piggy backed and stored procedure attacks are based on using permitted queries but also including malicious ones in the same statement. Alternate encoding is used to bypass input filters and encodes the statement so it appears harmless *(Kao, 2018).*

2.4. Input Handling

In order to mitigate an attack that uses SQLI the author suggests that all of the inputs are handled in a way that prevents the malicious hackers from gaining unwanted access to the database. This is also commonly referred to as 'input sanitisation'. In order to sanitise inputs a developer can remove potentially harmful characters from any inputs the user enters on the web application. An example of this would be if a user tried to enter `' OR 1=1; DROP TABLE PAY_INFO;` but the inputs were filtered to remove quotation marks then this command would not work and the table would be unaltered. Other filtering options, depending on the table, can be; removing certain words like 'DROP' or 'SELECT' and special characters like semi-colons or underscores.

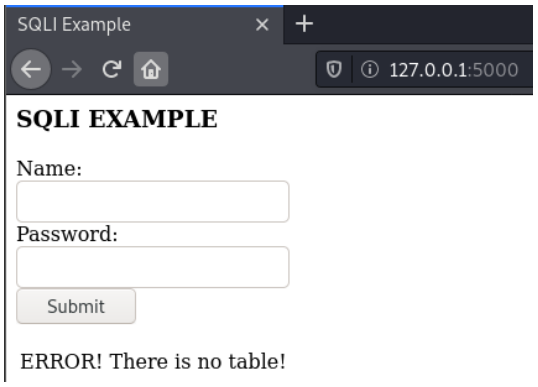
Prepared statements are a very powerful mechanism for disabling any potential SQLI attacks. By ensuring that all inputs are handled purely as data and all queries being handled in prewritten statements it is virtually impossible for a person to successfully achieve this attack through the web application.

A third way to mitigate SQLI is to restrict the privilege levels of any requests that are being made through the web application. For all CVEs that exploit this vulnerability they require a certain privilege level, from SELECT to SuperUser. By giving these requests the lowest privilege level possible the amount of attacks that can be performed is reduced drastically. Even if this attack is successfully performed by having very few privileges the amount of damage a malicious user can do is limited. *(EnterpriseDB, 2010).*

3. RECOMMENDATION

In order to demonstrate any recommendations made the author has created a Postgres Docker container and is interfacing with it using a Python container holding a very basic Flask web application. For the sake of brevity and the fact that it is not relevant to this report the set-up of these will not be discussed here.

The web application has two text input fields and when the 'submit' button is pressed it adds the data to the 'PAY_INFO' table and displays it neatly below. As the SQLI query being tested is `' OR 1=1; DROP TABLE PAY_INFO;` the author implemented a error catching system to display a message when no table is found after submitting the form *(Figure 1)*.



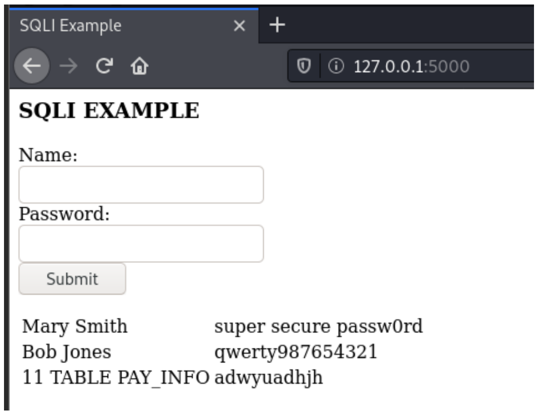
```
ERROR! There is no table!
```

Figure 1: Output After Successful DROP TABLE query

Although very basic the web application and database are designed as proof of concepts to demonstrate how the following recommendations work and their effectiveness against SQLI attacks. The author is aware that the organisation's web application uses ASP.NET not Flask but these examples should be adaptable to any other framework.

3.1. Input filtering

The first method is to filter the text being input into the application. This was demonstrated by creating a list of banned characters and phrases that if submitted will be removed by splitting the string, looking for forbidden characters, removing them and then joining it back together to submit into the database *(Figure 2)*. The malicious query is entered into text box is rendered useless as anything that could be actioned by the database is taken out.



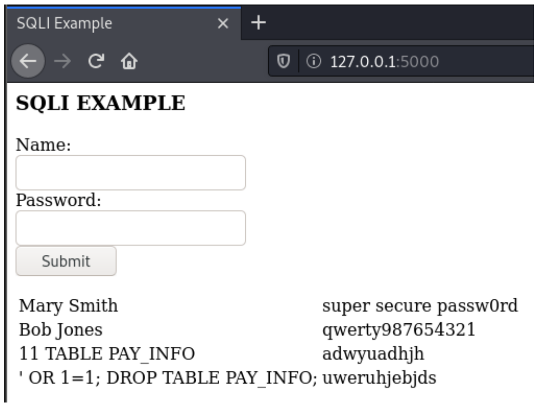
```
Mary Smith            super secure passw0rd
Bob Jones             qwerty987654321
11 TABLE PAY_INFO     adwyuadhjh
```

Figure 2: Output After Successful Input Filtering

Input filtering is arguably the quickest to implement as it is very simple. A banned character/word list can be created in a very small amount of time or can be found online meaning that making this change does not require a large investment on part of the organisation or their developers.

3.2. Prepared statements

In PostgreSQL prepared statements can be created using the PREPARE library. These statements treat all inputs as data to be used in a prewritten query and does not respond to any queries written in this data *(PostgreSQL [2], 2021)*. For this example the author used a Python PostgreSQL library, 'psycopg2' , and it's 'execute' function. This treats any inputs as purely values and since it only runs as a SELECT query any others inside the string will not be executed.



```
Mary Smith                          super secure passw0rd
Bob Jones                           qwerty987654321
11 TABLE PAY_INFO                   adwyuadhjh
' OR 1=1; DROP TABLE PAY_INFO;      uweruhjebjds
```

Figure 3: Output After Using a Prepared Statement

A reason that this would be an idea mitigation against possible SQLI attacks is that prepared statements are fairly easy to implement and if there are very limited queries being done this wont require too much of the developers time.

3.3. Restricting Privileges

Due to the nature of real application and the author not having access to it's source code this could not be demonstrated on the example Flask application. This is mainly due to the fact that what privileges to assign and how they are assigned depends greatly on how the database is being used. It is also technically not an input validation technique, which is the main recommendation being made here.

The benefits of implementing this recommendation is that even if a malicious user was able to send a valid SQL query it will not be executed if they are not allowed to. An example being; if a user only has INSERT privileges they cannot use SELECT or DROP. The latter two being very commonly used in exploiting databases and causing damage to an organisation's software.

4. LIMITATIONS

As mentioned in section 2.3 some hackers attempt to bypass input filtering by encoding their statements in a way that is not picked up by the filters. Examples of this are including the malicious query inside characters or words that are intended to be filtered out. Such as `' OSELECTR 1=1; SELECTDRSELECTOP TABLE PAY_INFO;` Another way is to use an encoding method such as using the hexadecimal values. Even with character filtration if the filter only makes one 'pass' of the string or does not catch encoded characters the query may still be run.

Even with restricted privileges or prepared statements it is still possible for an attacker to gain access to the database and it's contents by exploiting a privilege escalation vulnerability first. As the company is currently being targeted by a hactivist organisation it is likely that they will be willing to put in the effort in order to exploit as much as possible.

Irregardless of other vulnerabilities mitigating the aforementioned ones is still crucial. However altering the organisations software to fix these issues requires a sizeable investment of both time and money. The services will need to be taken offline while the work is being done and some features may have to be changed drastically in order to make them more secure.

5. REFERENCES

Demessie, S. (2019). *"Website Security vulnerability Detection and Prevention of SQLI and XSS Attack"*. Masters Thesis. Ethiopia: Adama Science and Technology University.

Douglas, K. and Douglas, S. (2003). *"PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases"*. Indiana: Sams Publishing. p. 728-743.

EnterpriseDB. (2010). *"Protecting PostgreSQL Against SQL Injection Attack"*. Corporate Whitepaper. Massachusetts: EnterpriseDB.

Kao, D. et al. (2018). *"A Framework for SQL Injection Investigations: Detection, Investigation, and Forensics"*. IEEE International Conference on Systems, Man, and Cybernetics (SMC), vol.28, pp.2838–2843.

NIST. (2018). *CVE-2018-16850*. [online]. Available at: https://nvd.nist.gov/vuln/detail/CVE-2018-16850 (Accessed: 8th Mar. 2021)

NIST. (2019). *CVE-2019-10208*. [online]. Available at: https://nvd.nist.gov/vuln/detail/CVE-2019-10208 (Accessed: 8th Mar. 2021)

NIST. (2021). *CVE-2021-20229*. [online]. Available at: https://nvd.nist.gov/vuln/detail/CVE-2021-20229 (Accessed: 8th Mar. 2021)

PostgreSQL [1]. (2021). *About*. [online]. Available at: https://www.postgresql.org/about/ (Accessed: 6th Mar. 2021)

PostgreSQL [2]. (2021). *PREPARE*. [online]. Available at: https://www.postgresql.org/docs/current/sql-prepare.html (Accessed: 12th Mar. 2021)