Table of contents

List of figures

1. Chapter 1

SQL Injections

1.1 Introduction

In today's interconnected digital landscape, web applications have become an integral part of our daily lives. They enable us to perform a wide range of tasks, from online shopping and banking to social networking and communication. However, the rise of web applications has also brought about a corresponding increase in security threats and vulnerabilities like gaining unauthorized access, stealing sensitive information, or disrupting services.

The following list shows the Top Five (05) most dangerous web application security risks published by OWASP Top 10 2021 project [1]:

Broken Access Control: refers to inadequate restrictions on what authenticated users can access or perform within an application. It can lead to unauthorized access, privilege escalation, or exposure of sensitive functionalities or data.

Cryptographic Failures: refer to vulnerabilities or weaknesses in the implementation or use of cryptographic techniques and algorithms. These failures can result in the compromise of encrypted data, unauthorized access, or the ability to tamper with cryptographic operations.

Injection Attacks: such as SQL, OS, or LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query, allowing an attacker to execute malicious code or unauthorized actions.

Insecure Design: refers to a fundamental flaw in the architecture or design of a system or application that compromises its security. It involves the failure to incorporate proper security mechanisms, access controls, or threat modeling during the design phase.
Security Misconfiguration: results from insecure configuration settings, default passwords, open cloud storage, or verbose error messages. Attackers exploit these misconfigurations to gain unauthorized access, extract sensitive information, or launch further attacks.

Focusing on SQL injection attacks, the impact of them on web applications can be severe. As an example, the attackers may have the ability to take control of the entire website or steal sensitive data like usernames, passwords, and credit card numbers. Furthermore, these attacks may result in data loss, website failures, and reputational harm to a company.

1.2 Understanding how web applications work

Web applications are defined as features that are most frequently seen on websites, such as shopping carts, product search and filtering, instant messaging, and social network newsfeeds. They enable users to access sophisticated functionality without having to install or set up software. The web application and users data are stored in a system called DBMS *"Database Management System"*, web apps that use these databases are called Database-driven Web applications.

Database-driven Web applications are very common in today's Web-enabled society. They normally consist of a back-end database with Web pages that contain server-side scripts written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions with the user. A database-driven Web application commonly has three tiers: Presentation tier (a Web browser or rendering engine, HTML, CSS, JS), logic tier (a programming language, such as C#, PHP, JSP, JS, etc.), storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.). Figure 1.1 Three-tier architecture. illustrates the simple three-tier example.

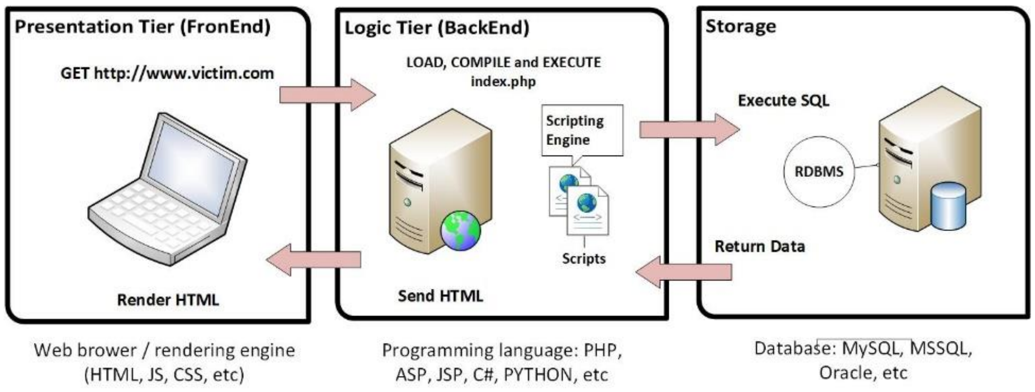

Presentation Tier (FronEnd)
GET http://www.victim.com
Render HTML
Logic Tier (BackEnd)
LOAD, COMPILE and EXECUTE index.php
Scripting Engine
Scripts
Send HTML
Storage
Execute SQL
RDBMS
Return Data
Web brower / rendering engine (HTML, JS, CSS, etc)
Programming language: PHP, ASP, JSP, C#, PYTHON, etc
Database: MySQL, MSSQL, Oracle, etc

Figure 1.1 Three-tier architecture.

The Web browser (the presentation tier, such as Internet Explorer, Safari, Firefox, etc.) sends requests to the middle tier (the logic tier), which services the requests by making queries and updates against the database (the storage tier).

A fundamental rule in a three-tier architecture is that the presentation tier never communicates directly with the data tier; in a three-tier model, all communication must pass through the middleware tier. Conceptually, the three-tier architecture is linear.

In Figure 1.1, the user fires up his Web browser and connects to http://www.victim.com. The Web server that resides in the logic tier loads the script from the file system and passes it through its scripting engine, where it is parsed and executed. The script opens a connection to the storage tier using a database connector and executes an SQL statement against the database. The database returns the data to the database connector, which is passed to the scripting engine within the logic tier. The logic tier then implements any application or business logic rules before returning a Web page in HTML format to the user's Web browser within the presentation tier. The user's Web browser renders the HTML and presents the user with a graphical representation of the code. All of this happens in a matter of seconds and is transparent to the user.

1.3 How SQL injections work

1.3.1 Definition

SQL injection attack consists of the insertion or "injection" of a SQL query via the input data from the client to the application, they are introduced when software developers create dynamic database queries constructed with string concatenation, which includes user-supplied input. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system [2].

Here are some facts regarding damages caused by SQL injection attacks:

1. In 2018, a SQL injection attack on a Canadian cryptocurrency exchange called MapleChange resulted in the loss of almost all of the company's cryptocurrency holdings, which were worth around $6 million.
   MapleChange hack: https://www.ccn.com/maplechange-hacked-all-funds-lost-in-cryptocurrency-exchange-hack/
2. In 2017, a SQL injection attack on Equifax, a major credit reporting agency in the United States, compromised the personal information of over 147 million people.

The breach resulted in a settlement of up to $700 million in damages, including compensation for victims and penalties.

Equifax breach settlement: https://www.npr.org/2019/07/22/744083587/equifax-reaches-700-million-settlement-over-2017-data-breach

3. In 2016, a SQL injection attack on the Philippine Commission on Elections exposed the personal information of over 55 million voters, including their names, addresses, and biometric data. The breach resulted in a class-action lawsuit and cost the commission over $2 million in damages.

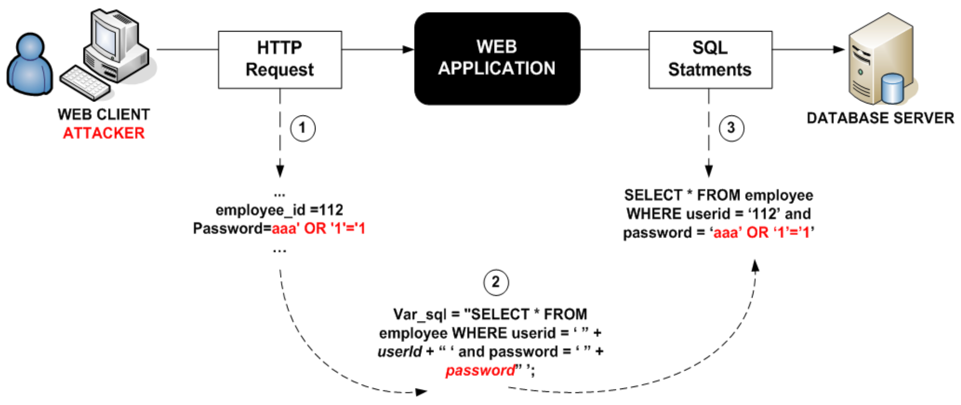
   Philippine Commission on Elections breach: https://www.reuters.com/article/us-philippines-election-cyber/philippines-election-commission-hit-by-record-data-breach-idUSKCN0XJ0BQ

In addition to financial damages, SQL injection attacks can also result in reputational damage, loss of customer trust, and legal liabilities. Companies that suffer from SQL injection attacks may face lawsuits, regulatory fines, and a loss of business due to damaged reputation.

Basically, SQL injection process is structured in Four (4) phases:

1. The attacker sends the malicious HTTP request to the web application
2. The malicious code concatenated with the developer's SQL statement
3. The system submits the SQL statement to the backend database.
4. The Database system returns the sensitive data for the Attacker to be exploited after.

As shown in Figure 1.2 describes a login by a malicious user exploiting SQL Injection **vulnerability**. The Administrator will be authenticated on the application after typing: employee id=112 and password=admin. The attacker is trying to find a SQL injection vulnerability to log in to the application by adding a tautology to the developer's SQL statement. [3]



```
...
employee_id =112
Password=aaa' OR '1'='1
...
```

```
Var_sql = "SELECT * FROM
employee WHERE userid = ' " +
userId + " ' and password = ' " +
password" ';
```

```
SELECT * FROM employee
WHERE userid = '112' and
password = 'aaa' OR '1'='1'
```

Figure 1.2 Example of a SQL injection attack.

The above SQL statement is always true because of the Boolean tautology that the attacker appended (OR 1=1) so, he will access the web application as an administrator without knowing the right password.

1.4 Techniques of SQL injections

1.4.1 Tautologies

This type of attack injects SQL tokens into the conditional query statement to be evaluated as always true. This type of attack is used to bypass authentication control and access to data by exploiting vulnerable input fields which use WHERE clause For example [5], consider the following SQL query used for user authentication:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password'
```

An attacker could inject a tautology into the password field, such as **' OR '1'='1'**

This would cause the query to become:

```
SELECT * FROM users WHERE username = 'admin' AND password = '' OR '1'='1'
```

Since the condition '1'='1' is always true, the query would return all users in the database, including the admin account. The attacker could then log in as the admin without knowing the correct password.

1.4.2 Error-based SQL injection

In error-based SQL injection, attackers can use SQL queries that trigger errors in order to learn about the structure and contents of a database. For example, an attacker could send an SQL query to the application that intentionally causes an error, and the error message returned by the database will contain information about the structure of the database or the content of the queried table.

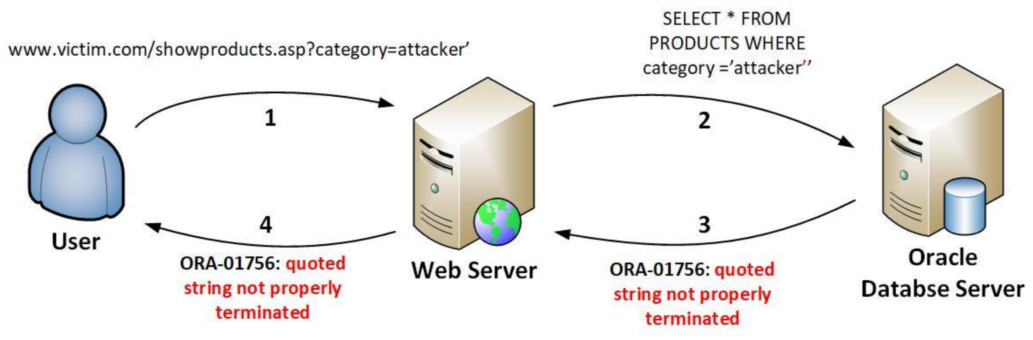

www.victim.com/showproducts.asp?category=attacker'
SELECT * FROM PRODUCTS WHERE category ='attacker''
1
2
User
4
3
Web Server
Oracle Databse Server
ORA-01756: quoted string not properly terminated
ORA-01756: quoted string not properly terminated

Figure 1.3 How Information flows during an SQL injection error.

According to Figure 1.3, the following occurs during a SQL injection error:

1. The user sends a request in an attempt to identify a SQL injection vulnerability. In this case, the user sends a value with a single quote appended to it.
2. The Web server retrieves user data and sends a SQL query to the database server. In this example, we can see that the SQL statement created by the Web server includes the user input and forms a syntactically incorrect query due to the two terminating quotes.
3. The database server receives the malformed SQL query and returns an error to the Web server.
4. The Web server receives the error from the database and sends an HTML response to the user. In this case, it sent the error message, but it is entirely up to the application how it presents any errors in the contents of the HTML response [4].

The following error is usually an indication of a MySQL injection vulnerability:

Warning: mysql_fetch_array(): supplied argument is not a valid MySQL result resource in /var/www/victim.com/showproduct.php on line 8

The preceding example illustrates the scenario of a request from the user which triggers an error in the database. Depending on how the application is coded, the response returned in step 4 will be constructed and handled as a result of one of the following:

- The SQL error is displayed on the page and is visible to the user from the Web browser.
- The SQL error is hidden in the source of the Web page for debugging purposes.
- Redirection to another page is used when an error is detected.
- An HTTP error code 500 (Internal Server Error) or HTTP redirection code 302 is returned.
- The application handles the error properly and simply shows no results, perhaps displaying a generic error page.

1.4.2.1 Mysql database errors

MySQL can be executed in many architectures and Operating systems. An Apache Web server running PHP on a Linux operating system forms a common configuration, but we can find it in many other scenarios as well.

In the example of my sql error shown in figured in Figure 1.3, the attacker injected a single quote in a GET parameter and the PHP page sent the SQL statement to the database. The following Figure 1.4 shows a fragment of PHP code that displays the vulnerability [4]:

```
   <?php

   //Connect to the database

   //Error checking in case the database is not accessible

   mysql_connect( “[database]”, “[user]”, “[password]”) or die(“Could
not connect:”. mysql_error() );

   //Select the database

   mysql_select_db (“[database_name]”);

   //We retrieve category value from the GET request

   $category = $_GET[“category”];

   //Create and execute the SQL statement

   $result = mysql_query(“SELECT * from products where
category=‘$category’”);

   //Loop on the results

   while ($row = mysql_fetch_array($result, MYSQL_NUM)) {printf(“ID: %s
Name: %s”, $row[0], $row[1]);}

   //Free result set

   mysql_free_result($result);

   ?>
```

Figure 1.4 SQL injection vulnerability in PHP code.

The code shows that the value retrieved from the GET variable is used in the SQL statement without sanitization. If an attacker injects a value with a single quote, the resultant SQL statement will be:

SELECT * FROM products WHERE category=‘attacker’’

The preceding SQL statement will fail and the mysql_query function will not return any value. Therefore, the *$result* variable will not be a valid MySQL result resource. In the following line of code, the *mysql_fetch_array($result, MYSQL_NUM)* function will fail and

PHP will show the warning message that indicates to an attacker that the SQL statement could not be executed.

In the preceding example, the application does not disclose details regarding the SQL error, and therefore the attacker will need to devote more effort in determining the correct way to exploit the vulnerability.

PHP has a built-in function called *mysql_error*, which provides information about the errors returned from the MySQL database during the execution of a SQL statement. In Figure 1.5, the following PHP code displays errors caused during the execution of the SQL query:

```
<?php
//Connect to the database
//Error checking in case the database is not accessible
mysql_connect("[database]", "[user]", "[password]") or die("Could not connect:". mysql_error());
//Select the database
mysql_select_db("[database_name]");
//We retrieve category value from the GET request
$category = $_GET["category"];
//Create and execute the SQL statement
$result = mysql_query("SELECT * from products where category='$category'");
if (!$result) { //If there is any error
//Error checking and display
die('<p>Error:'. mysql_error(). '</p>');
} else {
// Loop on the results
while ($row = mysql_fetch_array($result, MYSQL_NUM)) {printf("ID: %s Name:%s", $row[0], $row[1]);
}//Free result set
mysql_free_result($result);}
?>
```

Figure 1.5 handle query error with mysql_error() function in PHP.

When an application runs the preceding, the code that catches database errors and the SQL query fails, the returned HTML document will include the error returned by the database. If an attacker modifies a string, parameter by adding a single quote the server will return output similar to the following:

Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '''at line 1

The preceding output provides information regarding why the SQL query failed. If the injectable parameter is not a string and therefore is not enclosed between single quotes, the resultant output would be similar to this:

Error: Unknown column 'attacker' in 'where clause'

The behavior in MySQL server is identical to Microsoft SQL Server; because the value is not enclosed between quotes, MySQL treats it as a column name. The SQL statement executed was along these lines:

```
SELECT * FROM products WHERE idproduct=attacker
```

MySQL cannot find a column name called attacker and therefore returns an error.

This is the code snippet from the PHP script shown earlier in charge of error handling:

```
//If there is any error
//Error checking and display
    if (!$result) {
    die('<p>Error:'. mysql_error(). '</p>');}
```

In this example, the error is caught and then displayed using the *die()* function. The PHP *die()* function prints a message and gracefully exits the current script. Other options are available for the programmer, such as redirecting to another page:

```
//If there is any error
//Error checking and redirection
if (!$result) {
     header("Location:http://www.victim.com/error.php");}
```

1.4.3 Blind SQL Injection

Blind SQL injection is a technique of SQL Injection attack that asks the database true or false questions and determines the answer based on the application's response. This attack is often used when the web application is configured to show generic error messages but has not mitigated the code that is vulnerable to SQL injection.

When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query's syntax is incorrect. Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible [5].

An attacker may verify whether a sent request returned true or false in a few ways:

1.4.3.1 Content-based

Using a simple page, which displays an article with a given ID as the parameter, the attacker may perform a couple of simple tests to determine if the page is vulnerable to SQL Injection attacks.

Example URL: http://newspaper.com/items.php?id=2, this URL access sends the following query to the database:

SELECT title, description, body FROM items WHERE ID = 2

The attacker may then try to inject a query that returns *'false'*:

http://newspaper.com/items.php?id=2 and 1=2

Now the SQL query should look like this:

SELECT title, description, body FROM items WHERE ID = 2 and 1=2

If the web application is vulnerable to SQL Injection, then it probably will not return anything. To make sure, the attacker will inject a query that will return 'true':

http://newspaper.com/items.php?id=2 and 1=1

If the content of the page that returns 'true' is different than that of the page that returns 'false', then the attacker is able to distinguish when the executed query returns true or false.

Once this has been verified, the only limitations are privileges set up by the database administrator, different SQL syntax, and the attacker's imagination.

1.4.3.2 Time-based

This type of blind SQL injection relies on the database pausing for a specified amount of time, then returning the results, indicating successful SQL query execution. Using this method, an attacker enumerates each letter of the desired piece of data using the following logic:

- *If the first letter of the first database's name is an 'A', wait for 10 seconds.*
- *If the first letter of the first database's name is a 'B', wait for 20 seconds. etc.*

Using some time-taking operation e.g. *BENCHMARK()*, will delay server responses if the expression is True.

BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')) will execute the ENCODE function 5000000 times.

Depending on the database server's performance and load, it should take just a moment to finish this operation. The important thing is, from the attacker's point of view, to specify a high-enough number of BENCHMARK() function repetitions to affect the database response time in a noticeable way.

Here is an example combination of both queries:

```
1 UNION SELECT IF(SUBSTRING(user_password,1,1) =
CHAR(50),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null) FROM users
WHERE user_id = 1;
```

If the database response took a long time, we may expect that the first user password character with *user_id = 1* is character '2', *(CHAR(50) == '2')*

Using this method for the rest of the characters, it's possible to enumerate entire passwords stored in the database. This method works even when the attacker injects the SQL queries and the content of the vulnerable page doesn't change.

Obviously, in this example, the names of the tables and the number of columns were specified. However, it is possible to guess them or check with a trial and error method.

Conducting Blind SQL Injection attacks manually is very time-consuming, but there are a lot of tools that automate this process. One of them is SQLMap partly developed within the

OWASP grant program. On the other hand, tools of this kind are very sensitive to even small deviations from the rule. This includes:

- Scanning other website clusters, where clocks are not ideally synchronized,
- WWW services where the argument acquiring method was changed, e.g. from /index.php?ID=10 to /ID,10

If the attacker is able to determine when their query returns True or False, then they may fingerprint the RDBMS. This will make the whole attack much easier. If the time-based approach is used, this helps determine what type of database is in use. Another popular method to do this is to call functions, which will return the current date. MySQL, MSSQL, and Oracle have different functions for that, respectively *now()*, *getdate()*, and *sysdate()* [5].

1.4.4 Union-based SQL injections

The UNION operator is used in SQL to combine the results of two or more SELECT statements into a single result set. When a web application contains a SQL injection vulnerability that occurs in a SELECT statement, the attacker can often employ the UNION operator to perform a second, entirely separate query, and combine its results with those of the first.

All major DBMS products support UNION. It is the quickest way to retrieve arbitrary information from the database in situations where query results are returned directly [6]. Example of an application that enabled users to search for books based on author, title, publisher, and other criteria. Searching for books published by Wiley causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = ‘Wiley’
```

Suppose that this query returns the following set of results:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker’s Handbook	2005
Anley	The Shellcoder’s Handbook	2007

A far more interesting attack would be to use the *UNION* operator to inject a second SELECT query and append its results to those of the first. This second query can extract data from a different database table.

For example, entering the search term:

Wiley’ UNION SELECT username,password,uid FROM users-- causes the application to perform the following query:

SELECT author,title,year FROM books WHERE publisher = ‘Wiley’ UNION SELECT username ,password, uid FROM users--’

This returns the results of the original search followed by the contents of the users table:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker’s Handbook	2005
Anley	The Shellcoder’s Handbook	2007
admin	r00tr0x	0
cliff	Reboot	1

This simple example demonstrates the potentially huge power of the UNION operator when employed in a SQL injection attack. However, before it can be exploited in this way, two important provisos need to be considered [6]:

- When the results of two queries are combined using the UNION operator, the two result sets must have the same structure. In other words, they must contain the same number of columns, which have the same or compatible data types, appearing in the same order.
- To inject a second query that will return interesting results, the attacker needs to know the name of the database table that he wants to target, and the names of its relevant columns.

1.5 SQL Injection defense techniques

With user input channels being the main vector for such attacks, the best approach is controlling and vetting user input to watch for attack patterns by applying the following main prevention methods [7]:

1.5.1 Escaping

The developer must use always character-escaping functions for user-supplied input provided by each database management system (DBMS). This is done to make sure the DBMS never confuses it with the SQL statement provided by the developer.

For example, use the mysql_real_escape_string() in PHP to avoid characters that could lead to an unintended SQL command. A modified version for the login bypass scenario would look like the following in Figure 1.6:

```
  $db_connection = mysqli_connect("localhost", "user", "password", "db");

  $username          =          mysqli_real_escape_string($db_connection,
$_POST['username']);

  $password          =          mysqli_real_escape_string($db_connection,
$_POST['password']);

  $query = "SELECT * FROM users WHERE username = '" . $username. "' AND
password = '" . $password . "'";
```

Figure 1.6 Character-escaping in PHP code example.

Previously, the code would be vulnerable to adding an escape character (\) in front of the single quotes. However, having this small alteration will protect against an illegitimate user and mitigate SQL injection.

1.5.2 Input validation

The validation process is aimed at verifying whether or not the type of input submitted by a user is allowed. Input validation makes sure it is the accepted type, length, format, and so on. Only the value that passes the validation can be processed. It helps counteract any commands inserted in the input string. In a way, it is similar to looking to see who is knocking before opening the door.

Validation should not only be applied to fields that allow users to type in input, meaning developers should also take care of the following situations in equal measure:

- Use regular expressions as whitelists for structured data (such as name, age, income, survey response, zip code) to ensure strong input validation.
- In case of a fixed set of values (such as drop-down list, radio button), determine which value is returned. The input data should match one of the offered options exactly.

The below shows how to carry out table name validation.

```
switch ($tableName) {

    case 'fooTable': return true;

  case 'barTable': return true;

    default: return new BadMessageException('unexpected value provided as
table name');

}
```

The *$tableName* variable can then be directly appended—it is now widely known to be one of the legal and expected values for a table name.

In the case of a drop-down list, it's very easy to validate the data. Assuming the developers want a user to choose a rating from 1 to 5, change the PHP code to something like this:

```
<?php

if(isset($_POST["selRating"])) {

    $number = $_POST["selRating"];

      if( (is_numeric($number)) && ($number > 0) && ($number < 6) ){

          echo "Selected rating: " . $number;

      } else { echo "The rating has to be a number between 1 and 5!"; } ?>
```

The developers have added two simple checks:

1. It has to be a number (the is_numeric() function).
2. He requires that $number to be bigger than 0 and smaller than 6, which leaves him with a range of 1–5.

Data that is received from external parties has to be validated. This rule applies not only to the input provided by Internet users but also to suppliers, partners, vendors, or regulators. These vendors could be under attack and send malformed data even without their knowledge.

1.5.3 Parameterized queries

Parameterized queries are a means of pre-compiling an SQL statement so that you can then supply the parameters in order for the statement to be executed. This method makes it possible for the database to recognize the code and distinguish it from input data.

The user input is automatically quoted and the supplied input will not cause a change of the intent, so this coding style helps mitigate an SQL injection attack.

PHP 5.1 up versions present a better approach when working with databases: PHP Data Objects (PDO). PDO adopts methods that simplify the use of parameterized queries. Additionally, it makes the code easier to read and more portable since it operates on several databases, not just MySQL.

The code in Figure 1.7 uses PDO with parameterized queries to prevent the SQL injection vulnerability:

```
<?php

$id = $_GET['id'];

$db_connection = new PDO('mysql:host=localhost;dbname=sql_injection_example', 'dbuser', 'dbpasswd');

//preparing the query

$sql = "SELECT username FROM users WHERE id = :id";

$query = $db_connection->prepare($sql);

$query->bindParam(':id', $id);

$query->execute();

//getting the result

$query->setFetchMode(PDO::FETCH_ASSOC);

$result = $query->fetchColumn();

print(htmlentities($result)); ?>
```

Figure 1.7 Parameterized queries using PDO in PHP code example.

1.5.4 Web application firewalls WAF

One of the best practices to identify SQL injection attacks is having a web application firewall (WAF). A WAF operating in front of the web servers monitors the traffic which goes in and out of the web servers and identifies patterns that constitute a threat. Essentially, it is a barrier put between the web application and the Internet.

A WAF operates via defined customizable web security rules. These sets of policies inform the WAF what weaknesses and traffic behavior it should search for. So, based on that information, a WAF will keep monitoring the applications and the GET and POST requests it receives to find and block malicious traffic [7].

WAFs provide efficient protection from a number of malicious security attacks such as:

- SQL injection
- Cross-site scripting (XSS)
- Session hijacking
- Distributed denial of service (DDoS) attacks
- Cookie poisoning
- Parameter tampering

1.5.5 Detection using machine learning

Artificial **I**ntelligence can enhance the speed and efficiency of SQL injection detection. By automating the process of query analysis and classification, AI algorithms can rapidly scan vast amounts of incoming queries, flagging potential threats in real-time.

One of the many machine-learning algorithms that can be used for SQL injection detection as an example is SVM (Support Vector Machines).

SVM is a supervised machine-learning algorithm that is commonly used for binary classification tasks; it aims to find an optimal hyperplane that separates the legitimate queries from the malicious ones by maximizing the margin between them.

To train a model using SVM, a systematic process that include **Dataset Preparation**, **Feature Extraction**, **Dataset Splitting**, **and Model Training** must be followed. During the training point, the SVM model learns the decision boundary based on the provided features and their corresponding labels.

1.6 Conclusion

As web applications become increasingly complex, SQL injection attacks remain a persistent and evolving threat. However, various techniques and tools have been developed to detect and prevent SQL injection attacks like input validation and sanitization, parameterized queries, and security-focused coding practices.

Unfortunately, because of the large variation in the pattern of SQL injection attacks, it is often unable to protect databases. Therefore, it is recommended, to apply the above-mentioned techniques in combination with Machine Learning and AI tools.

2. Bibliography

[1] OWASP, "Top 10 Web Application Security Risks," 2021. [Online]. Available: https://owasp.org/www-project-top-ten/.

[2] OWASP, "SQL Injection," 2023. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection.

[3] Researchgate, "Web Application Security by SQL Injection DetectionTools," 2012.

[4] J. Clarke-Salt, SQL Injection Attacks and Defense, 2012.

[5] OWASP, «Blind SQL Injection,» 2023. [En ligne]. Available: https://owasp.org/www-community/attacks/Blind_SQL_Injection.

[6] S. M. P. Dafydd, The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2011.

[7] Positive Technologies, "how-to-prevent-sql-injection-attacks," Augest 2019. [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks.