Binghamton University, Watson School of Engineering

Final Report:

SQL Injection Offense, Detection, Prevention and Deception

Name: Rahul Verma

Science of Cyber Security – CS 559-01

Prof. Guanhua Yan

December 16, 2022

CONTENTS

I. SQL INJECTION OFFENSE

1. Intro to SQLI:

There are weaknesses in almost every web application and we make use of this weakness in SQL injection. In order to access the database, we use the code injection technique known as SQL injection and although, the user shouldn't have access to the database, this strategy allows them to do so. The most prominent and well-known attack used to alter already-existing data and the most common web hacking method used is SQL injection. By getting access and impersonating a specific user, one can also destroy the data for their own personal advantage. Additionally, the entire database can be accessed as the attacker gains admin privileges, which leaves it open to attack which the attacker can use to get a persistent backdoor in any organization's systems.

2. Vulnerability:

On servers that don't adequately validate their user input, SQL injection can be simply implemented. There are many different sorts of vulnerabilities present, which might allow the program to accept any data from an unreliable source. The input can be constructed in such a manner that it purposefully matches the SQL query that the web application needs to run in order to authenticate its users and gain access to their data. Consider the following query, which was created by concatenating the input string that the user is said to have entered: "SELECT * FROM users WHERE username = ' +username + ' AND password = ' +password+ ' ", would allow the user access if the password contained a single quoted character. The SQL command that may be created by altering the input in this example serves as the vulnerability, and the code used to send the SQL instructions to the program serves as the exploit.

3. Attack Surface:

The SQL Injection attack surface consists of the login page and database. In this case, SQL injection, which includes the execution of queries, aids in gaining unauthorized access to the user data. The login page contributes to the attack surface since it doesn't check user input. Additionally, database is used as the attack surface, because the attacker makes use of it as a means of access.

4. Attack Vector:

An attack vector is a way or a method for anyone to access the web server or the system. Here, SQL Injection itself is the attack vector, that uses malicious SQL code to manipulate backend databases and access data that was not meant to be shown. When the user enters username and password as (' or 1=1 --), then the SQL query becomes "SELECT * FROM userpass WHERE user = '$user' AND pass ='' or 1=1 –". The attacker gets access to the database since the logic statement "1=1" is always true.

5. How the attack occurs:

Bypassing the login procedure, the SQL Injection method is used to get access to the database. The functioning of a SQL Injection attack will be demonstrated using the example below. The user must input his login and password in this HTML website form in order to access his data. The user sees the message "Login Successful" if the username and password are accurate. In this example, we utilize HTTP POST but the request can be made using either the HTTP GET or HTTP POST protocols.

```
<form action="/cgi-bin/login" method=post>
    Username: <input type=text name=username>
    Password: <input type=password name=password>
    <input type=submit value=Login>
```

The web browser sends the entered username and password as a string to the web server once the user enters the necessary username and password in the form mentioned above and clicks the login button. All the user information required for login is on the web server. The string is included in the request's body as follows:

```
Username = enteredUser & password = enteredPassword
```

The program doesn't check the login procedure before granting the user immediate access without checking and using additional security measures like executing a SQL query.

```
SELECT * FROM userpass WHERE (username = "enteredUser" and password =
"enteredPassword");
```

The user's profile that contains the supplied username and password will be found using this SQL query. If the program does not employ any input validation techniques, it is susceptible to a SQL injection attack.

```
--Attackers Input to Access the database using SQLI:
    password = ' or 1=1 --

--SQL query in the source code gets manipulated to:
    select * from userpass where username = 'user' AND password = '' or 1=1 --

(This will always check the condition in username and password for 1=1 which
will always be true and give complete access to the database to the hacker.)
```

So, the fundamental concept is to provide a certain input that will modify the logic of SQL statements and therefore remove the password verification. The SQL query will treat the password field as commented out if it contains an empty string, skipping the password check and granting access of the database to the attacker.

6. Working of the attack:

SQL attack involves a query that serves as the attacker's entry point to the database. The SQL command grants the attacker access to the data which is susceptible. We do not need any extra tools for this kind of attack as the browser is sufficient for it. There are two possible attack scenarios: first, where the user must complete a php form before sending it to the server using the HTTP POST method and second, where the user may access the database by just entering their login and password.

7. Code Explanation:

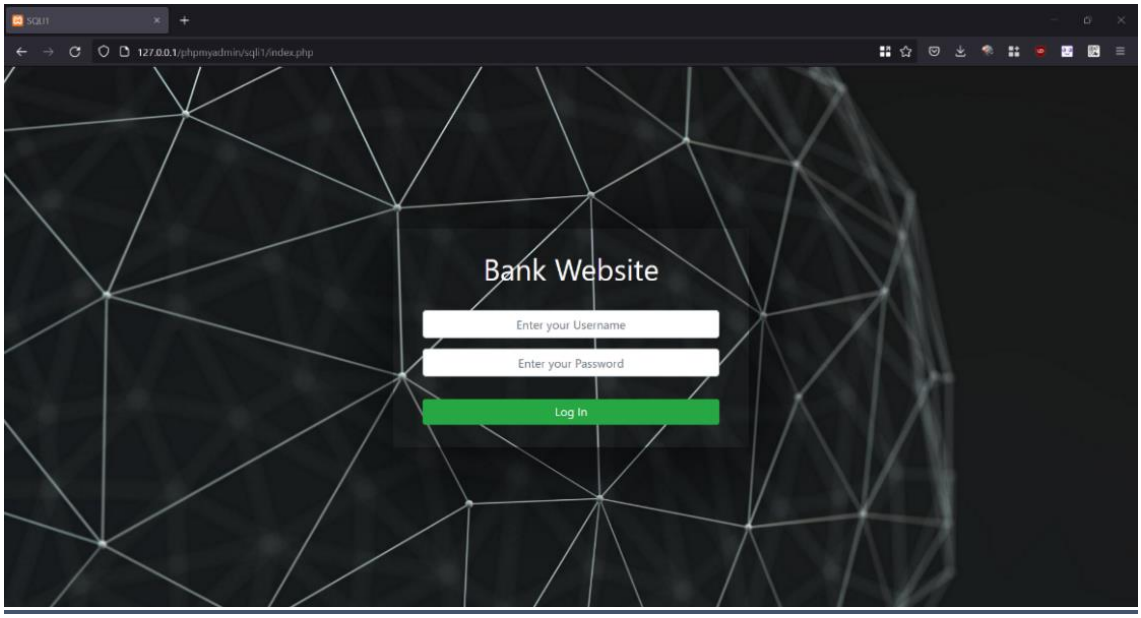
In "index.php" there are two links: one for bootstrap CSS and another for global CSS. These two files contain all of Bootstrap's stylesheets, so they're necessary in order to make the website look like an actual website instead of just some text on a white background There is a form container which consists of 3 input fields where user can enter their username and password and submit using the "Log In" button. Here, POST method is used to submit the data.

In "valid.php" when the correct username and password are entered, user will be able to see all the Account Details, along with "Login Successful" message on top, which includes the balance in the account, but this page will be accessible to the hacker using SQL Injection. There is a "Log Out" button at the bottom which will redirect to "index.php"

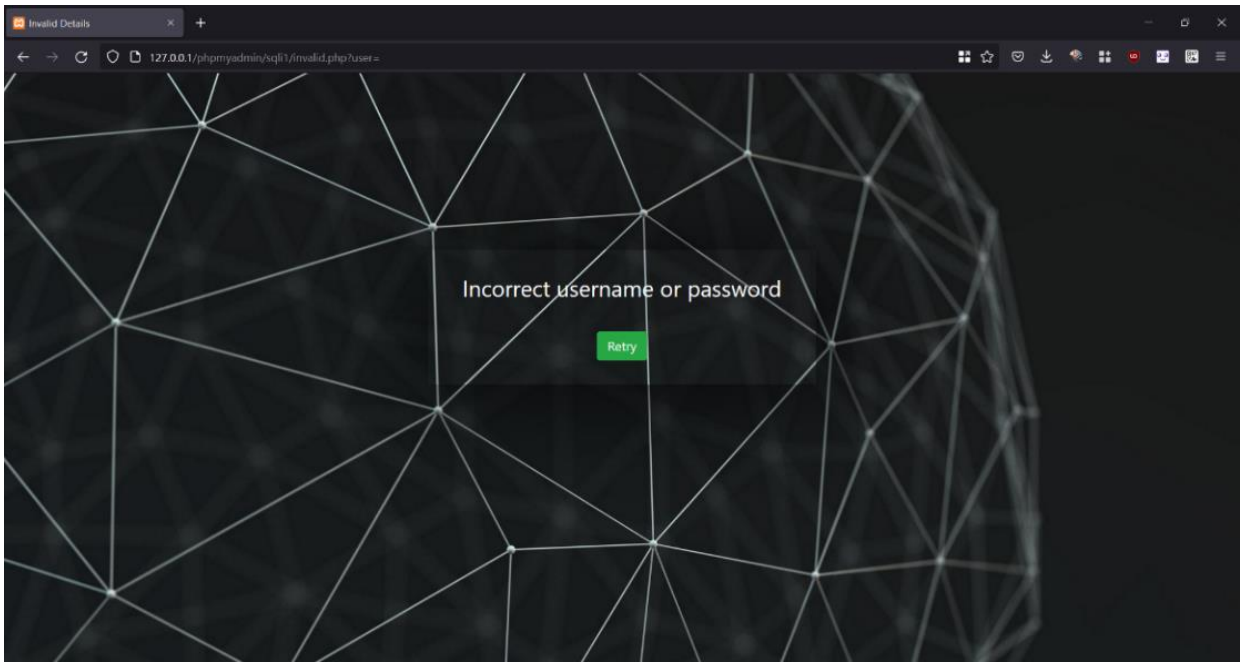
In “invalid.php” when the entered username and password is incorrect the page will show “Incorrect username or password” and a “Retry” button which redirects to “index.php” for the user to enter the correct username and password again.

The “login.php” file is used to check the inputs with the ones stored in the database. The created query has the following syntax: “SELECT * FROM userpass WHERE user=‘$user’ AND pass=‘$pass’ ”. The database is handled by the Xampp server. “user”, “pass” and “amount” are the respective columns in “userpass” table. The first row in the database is returned as the result when the user’s input is (1’ or ‘1’ = ‘1) on the website, which gives unauthorized access of user1’s data to the hacker, but we can see when entered in phpMyAdmin command line we get all the results including the first user. The row with the name user1 won’t be picked and the subsequent immediate user will be logged in if the input is of the type (1’ or ‘1’ = ‘1’ and user<>‘user1), we can see this in the phpMyAdmin screenshot where we are getting the output of all the users except the first user, we can modify this command such that it will return the third, fourth users and so on. This is the complete functioning of an SQL Injection Attack.

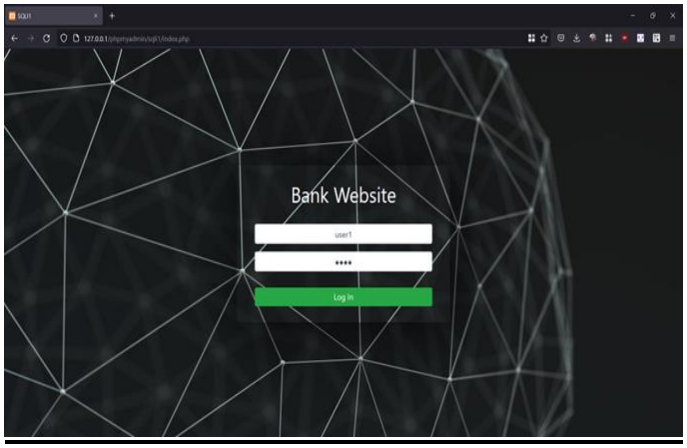
8. Screenshots:

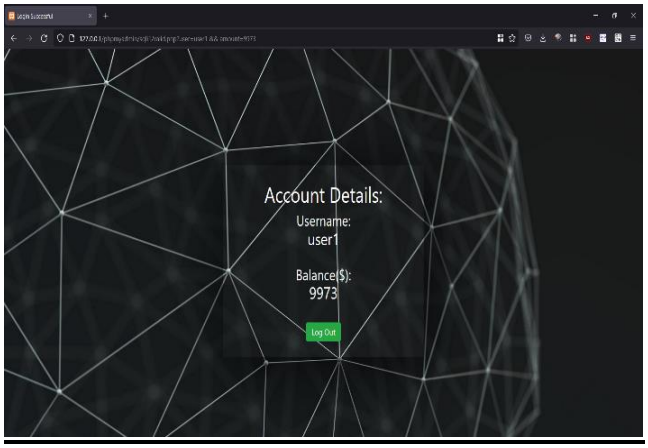


Login Page of the Website

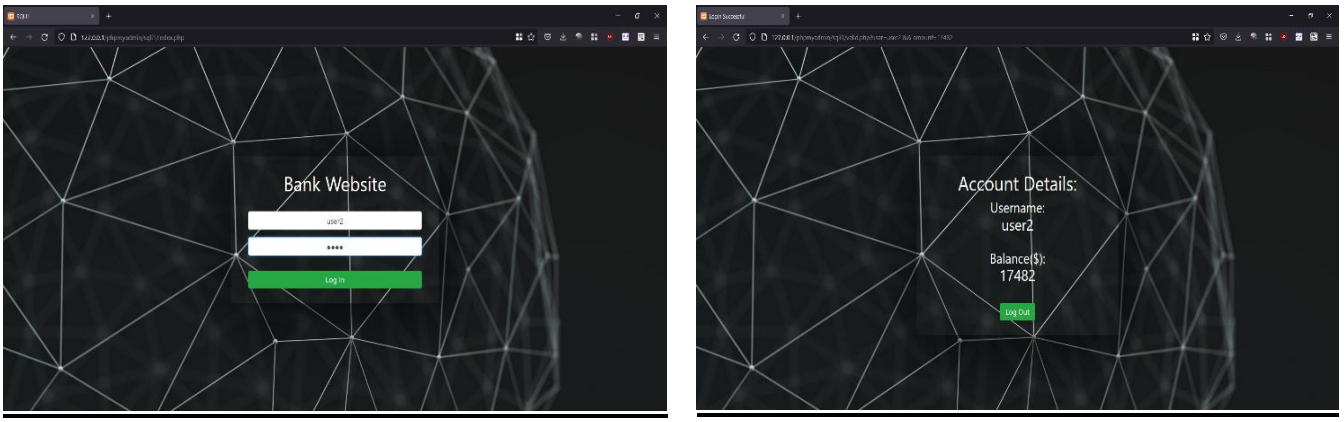


Incorrect username or password

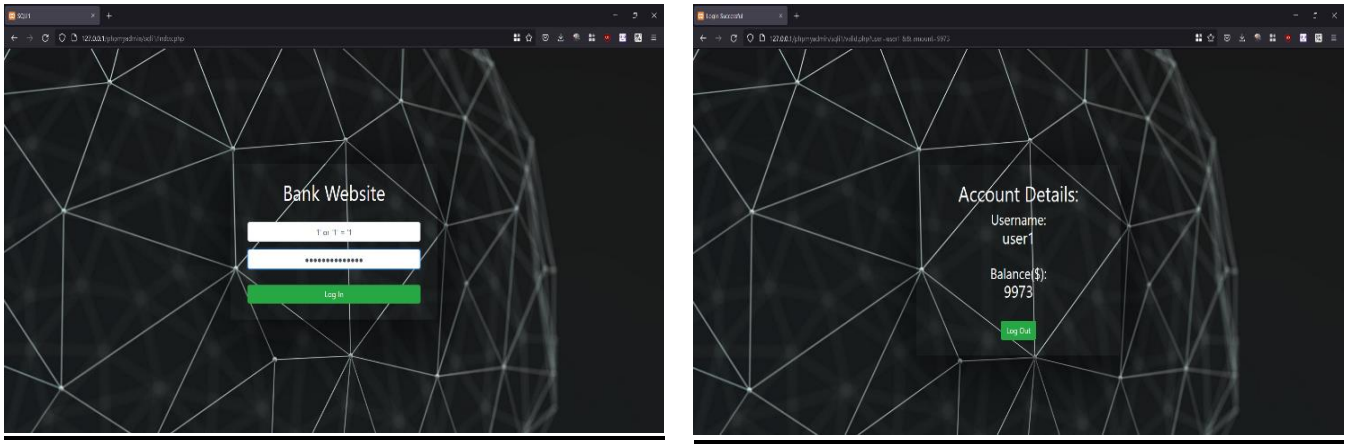




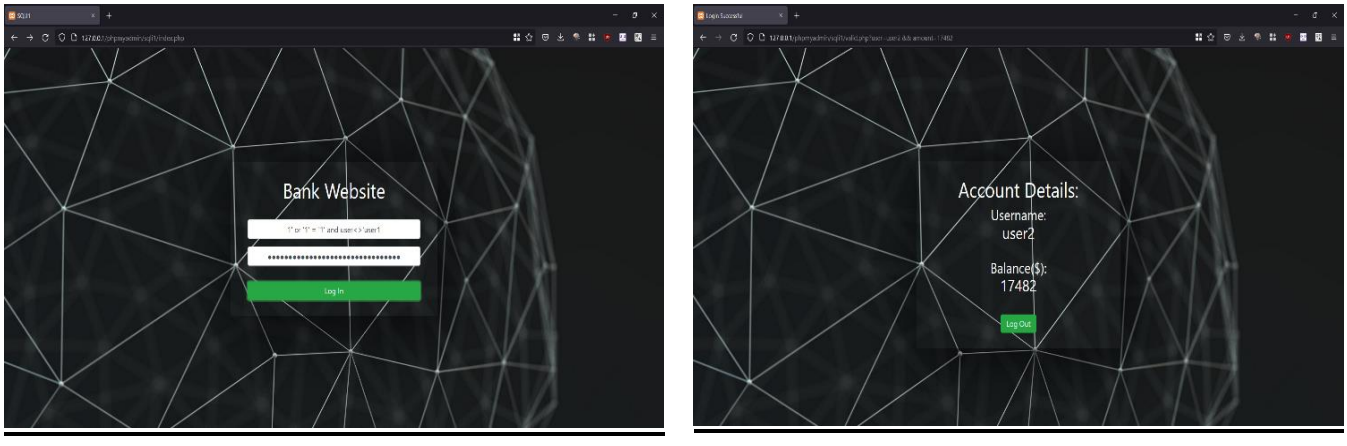
Accessing User1's account using correct username and password(username: user1 and password: 1111)



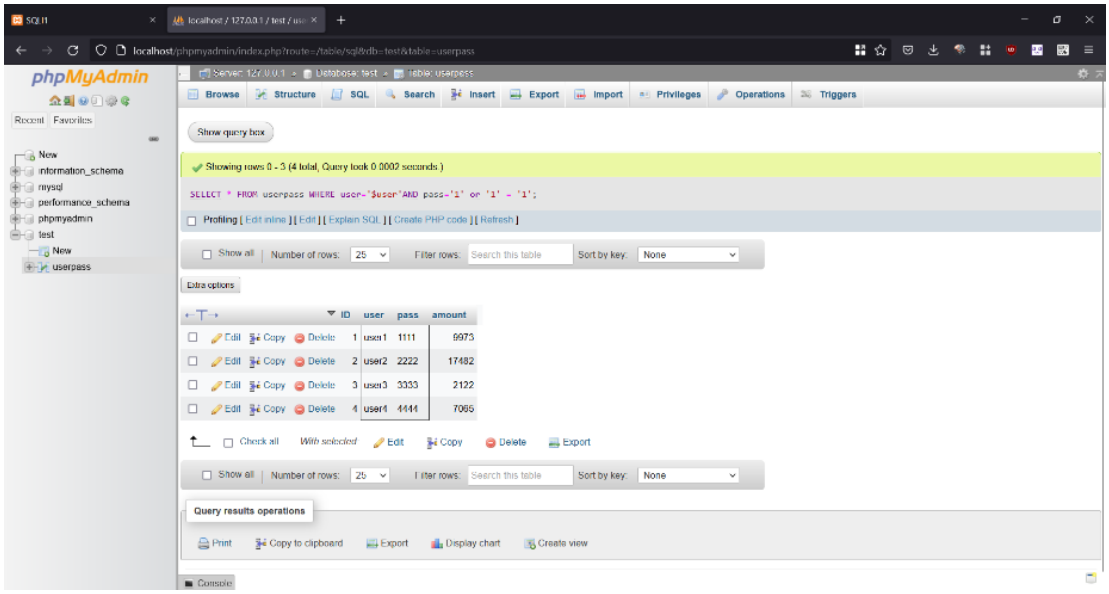
Accessing User2's account using correct username and password(username: user2 and password: 2222)



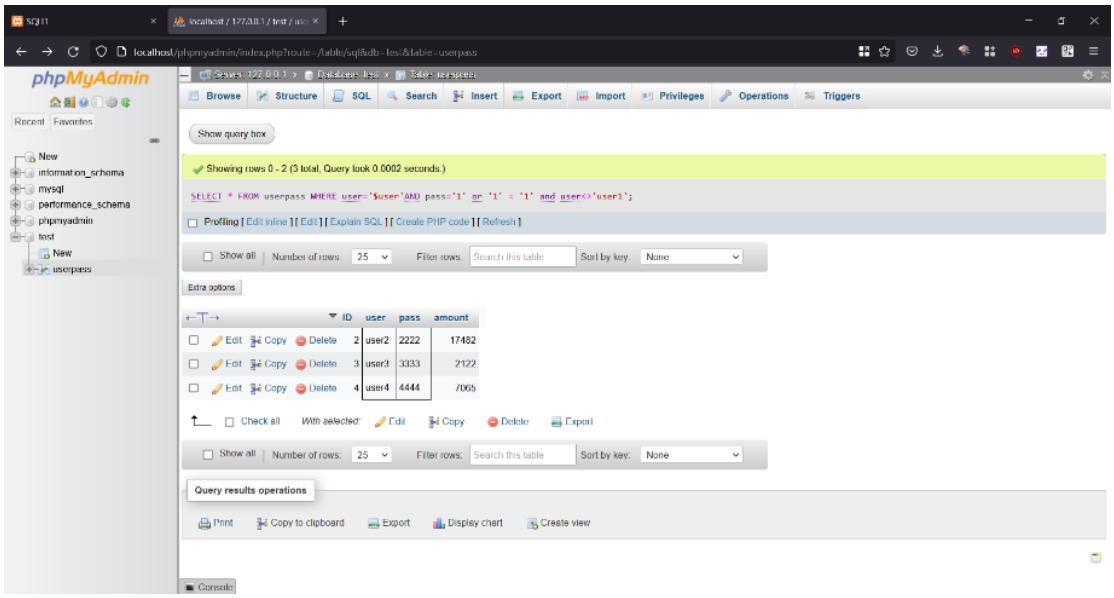
Unauthorized access to User1's account using SQL Injection(username and password: 1' or '1' = '1)



Unauthorized access to User2's account using SQL Injection(username and password: 1' or '1' = '1' and user<>'user1)



Using the SQLI query in phpMyAdmin returns all the users' data including user1



Using the SQLI query (<>user1) in phpMyAdmin returns all the users' data except user1

9. Bibliography:

- https://www.w3schools.com/sql/sql_injection.asp
- https://portswigger.net/web-security/sql-injection
- https://www.hacksplaining.com/exercises/sql-injection
- https://www.youtube.com/watch?v=ciNHn38EyRc
- https://www.youtube.com/watch?v=_jKylhJtPmI
- https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/
- https://en.wikipedia.org/wiki/SQL_injection
- http://testphp.vulnweb.com/ (sample SQLI vulnerable website)

II. SQL INJECTION DETECTION

10. Intro to SQLI detection:

As per the first project, the technique of SQL injection is used to insert harmful data like SQL query in the input as username and password to access the server. The fundamental working principle of the SQL injection approach is that the attacker may enter a statement that will always be true, giving him complete unauthorized control over the database and enable him to change or tamper the sensitive information stored. This is a very dangerous situation since it can result in significant loss for the person or an organization.

This detection method is machine learning based. A classifier will be used in this method to filter out the numerous dangerous spam keywords. The Naïve Bayes classifier is the one utilized in this project. We simply teach the algorithm to filter out the terms that we consider to be spam that might harm the application. The web application's source code isn't required to be changed in this method. Implementation is done separately and the source code of the web application is left as it is.

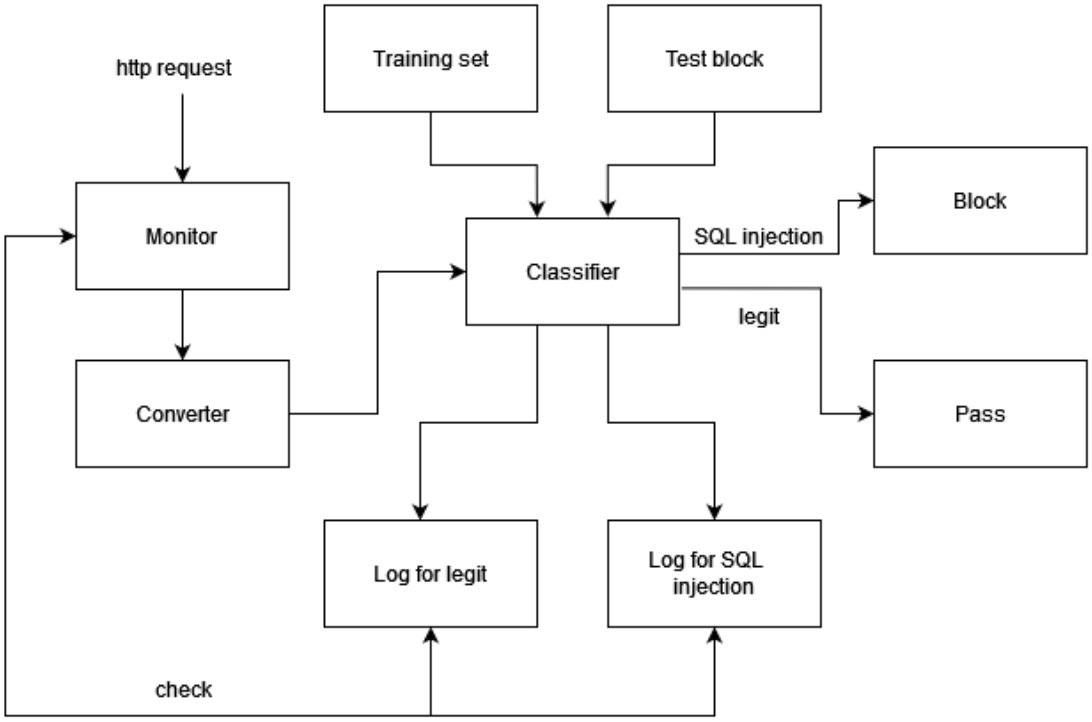
11. Type of detection method:

Detection using Naïve Bayes classifier comes under anomaly-based detection, as we are providing the training set to the classifier to categorise keywords into spam or legit category. Anomaly-based detection is basically a type of detection where the discovery of unexpected occurrences, observations, deviate considerably from the usual output, it is also known as outlier detection.

12. Naïve Bayes Classifier and its working:

The training set elements and the properties that we define here are crucial for identifying dangerous spam keywords. The monitor, classifier, and converter are the three main components utilized here. The fundamental operating principle is that we submit the HTTP request from the web app to the monitor first. It is sent to the converter via the monitor. The classifier then determines whether the data is legit or if it's an SQL injection attempt after receiving it from the converter. If it's a SQL injection attempt, it's detected, banned and access is denied or else the entry is allowed.

The classifier, which has a preset training set, determines whether the input is legit or if it's an SQL injection attempt. This collection includes the test samples that the developer has specified.



- **Monitor:**

The monitor is essentially the first stage that the data must go through. HTTP POST/HTTP GET methods are used to send the data. In this case, URL decoding is needed. What occurs in URL decoding is that it only permits a particular set of characters to pass when the user enters his credentials, which is significant to the server and authentication. In this instance, we ensure that all the letters are transformed to upper case after decoding. By doing this, the characters are all in the same format, making it impossible for an attacker to access the database by utilizing mixed-case characters.

- **Converter:**

HTTP parameters are the input the converter receives from the monitor. The parameters must be transformed into a numeric format so that the classifier can understand them. It is the classifier who determines if the data is legit or SQL injection. The pattern attributes and keyword set are the two primary attributes. The two most important factors in pattern characteristics seem to be length and the number of keywords. When it comes to SQL injection, the usage of special characters should be avoided since, when done correctly, they can make the query statement a true or false Boolean value, evading the security check and giving the attacker unauthorized access to all the information. Dollar signs($), single quotes(‘’), equal to signs(=), hashes(#), and other symbols are frequently used as characters. Additionally, terms like SELECT, FROM, DROP and spaces are utilized and are treated like characters, as well as there is a space requirement for every query. Consider the following example: SELECT * FROM users WHERE username = “xyz” etc. Here, there are three keywords i.e., SELECT, FROM, and WHERE and 43 characters.

- **Classifier:**

The converter provides the classifier with its input. The classifier determines whether a keyword qualifies as spam or not. The Naïve Bayes classifier is the one being utilized, where the Bayesian decision theory is used to build the classifier. Using the Bayes equation, the probability of classes is computed. The keyword with the highest value is determined to be a spam keyword based on probability.

In this case, X stands for the provided attributes. If there are 'n' patterns, then X is represented by:

X = {(X1, t1), (X2, t2) ... (Xn, tn)}

The collection of attributes for all patterns, Xn, is denoted by the symbol Xi = (x1, x2, ...)T

Ti is the set of all w’s such that: Ti = {w1, w2, ..., wm}

It is determined that attributes from the samples represent the functions' parameters. The probabilities based on the Bayes equation are calculated by this function alone and are written as follows: P(X|wi) P(wi).

- **Algorithm:**

The Naïve Bayes classifier's algorithm is as follows:

Input: X

Output: None

- Begin
- g1(X) = P(X|w1) (w1);
- g2(X) = P(X|w2) (w2);
- Set k as the maximum value between g1(X) and g2(X);
- If (wk == Injection pattern);
- Declare “SQL Injection”;
- Insert into SQL Injection Log;
- Execute Block Process;
- Else;
- Declare “Legit”;
- Insert into Legit Log;
- Execute Pass block;
- End.

Two probabilities are computed here as part of the algorithm's operation in this classifier. They are as follows: one for the SQL injection string and one for the legit string. The subsequent tasks are carried out based on the probability outcome. If there is a higher probability that the string is legit, it is run and a log entry is made in the legit log. The process is halted, the SQL injection string is executed, and logs are recorded in the SQL injection log if the probability of the SQL injection string is higher. This prevents access to the database.

13. Working of training set:

To carry out either one of the two options, the classifier must first be trained. For the algorithm to understand which strings are legit and which are not, predetermined sample sets are specified in the code. Here, we have two options: either manually enter a variety of strings that we believe to be spam, or just make a string pattern generator. For instance: String like (1’ or ‘1’ = ‘1) is a SQL injection string that an attacker can use as the username and password input to create a SQL query that will always return true. The algorithm may be trained to filter out such prohibited characters.

14. Performance of detection method:

As machine learning is based on the training set and requires some time to learn from the real-world inputs, it is impossible to get accurate results all the time, there is a chance for it to get false alarms, but as time progresses, we can make the system more reliable by letting it collect the input information from the usage and keep increasing its accuracy. As for the early usage of this system, it will show accurate detection results most of the time. After entering 5 different SQL injection queries, the classifier detected all of them.

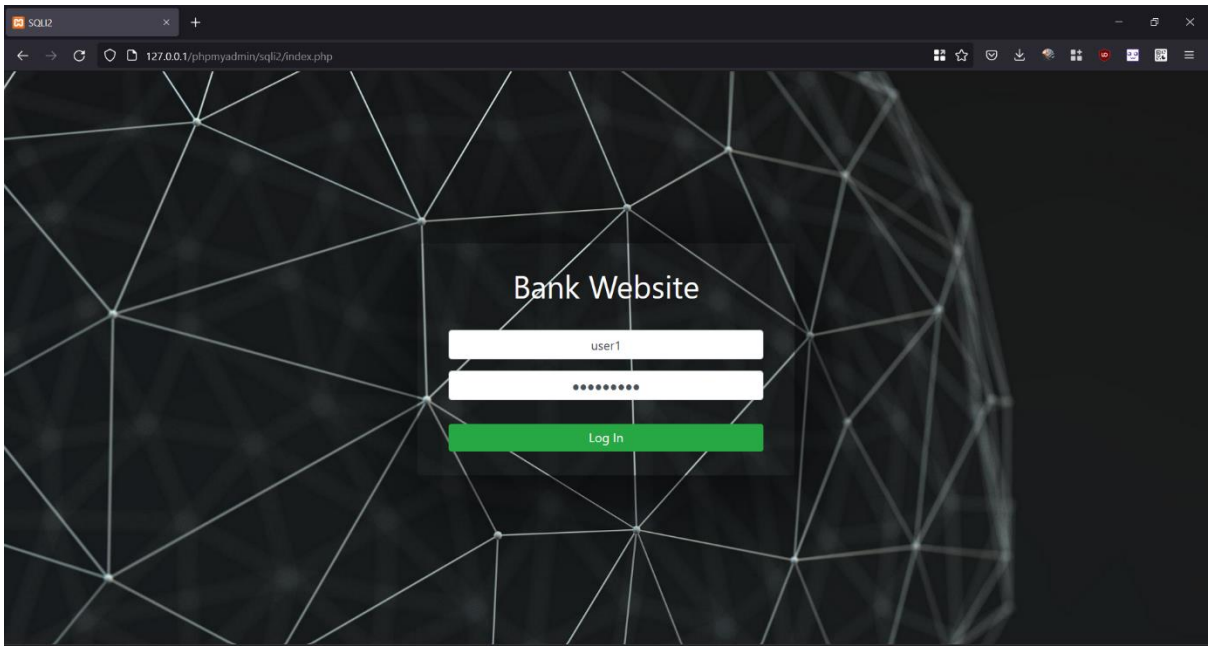
15. Screenshots:

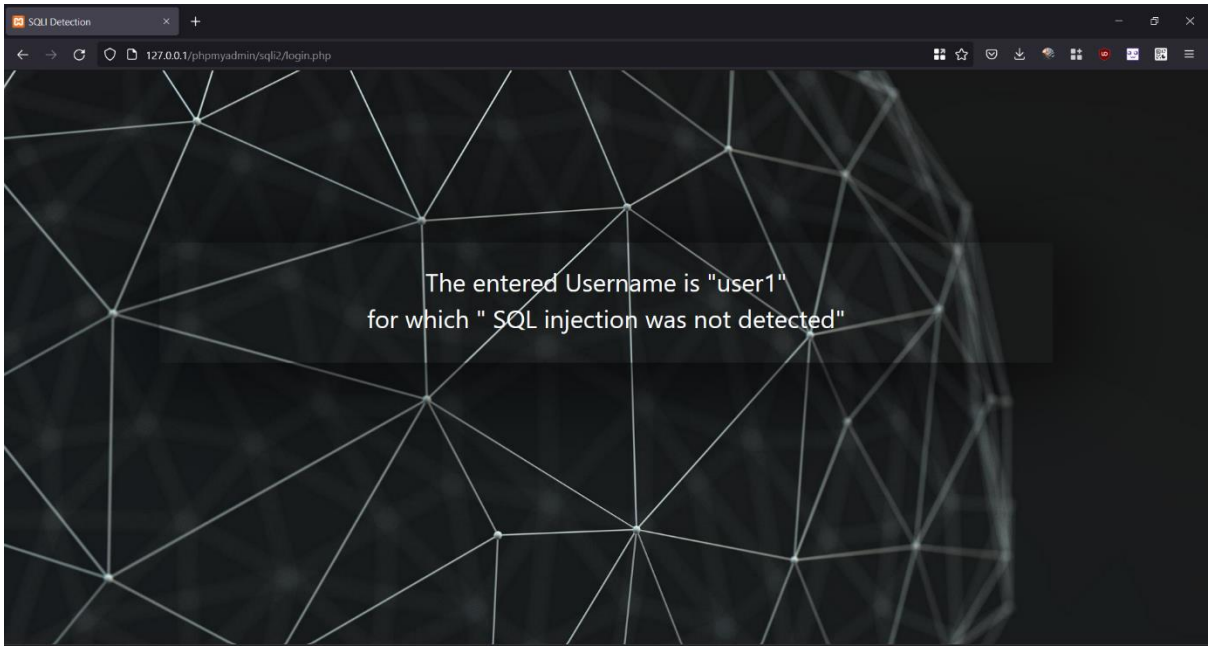
```
$classifier->train("$$, dollars", $spam);
$classifier->train("The possibility of it being an injection query increases with the length of the string", $spam);
$classifier->train("105; DROP TABLE Users", $spam);
$classifier->train("this is a spam username and password", $spam);
$classifier->train("drop * <>", $spam);
$classifier->train("drop * from database", $spam);
$classifier->train("drop * from database", $spam);
$classifier->train("select * from database", $spam);
$classifier->train("1 OR 1=1 and drop", $spam);
$classifier->train("1 or 1 = 1", $spam);
$classifier->train("1 or 1 = 1 and user <> rahul", $spam);
$classifier->train("select * FROM db", $spam);
$classifier->train("105; DROP TABLE Users", $spam);
$classifier->train("delete * from database", $spam);
$classifier->train("105; INSERT into TABLE Users", $spam);
$classifier->train("1 or 1 = 1 and user <> john", $spam);
```

Training the classifier for spam keywords

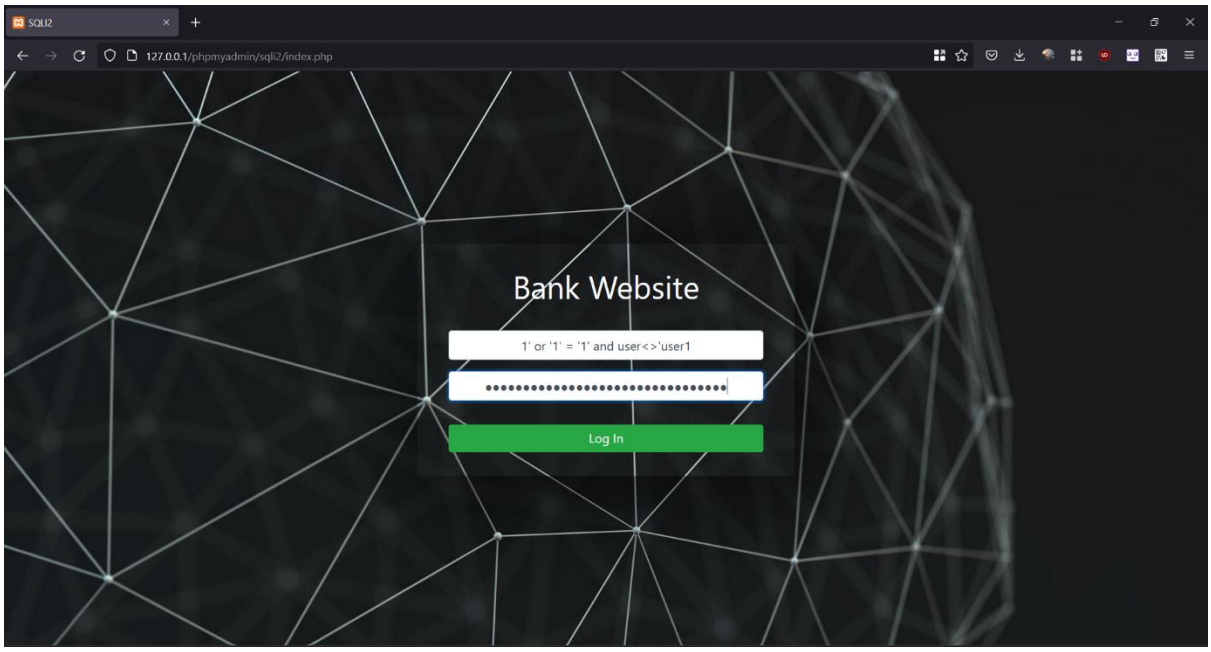
```
$classifier->train("rahul", $ham);
$classifier->train("john", $ham);
$classifier->train("rverma4", $ham);
$classifier->train("erenjaeger", $ham);
$classifier->train("killua@hxh", $ham);
$classifier->train("natusvincere", $ham);
$classifier->train("milesmorales", $ham);
$classifier->train("teamliquid.fc", $ham);
$classifier->train("johanliebert", $ham);
$classifier->train("Gojo666", $ham);
$classifier->train("Gon@hxh", $ham);
$classifier->train("Killua999", $ham);
$classifier->train("Goku111", $ham);
$classifier->train("freiza222", $ham);
$classifier->train("astralis.fc", $ham);
$classifier->train("hinatashoyo", $ham);
$classifier->train("kennys@cs", $ham);
$classifier->train("chainsawman", $ham);
$classifier->train("hyperbeast", $ham);
$classifier->train("doritojoe", $ham);
```

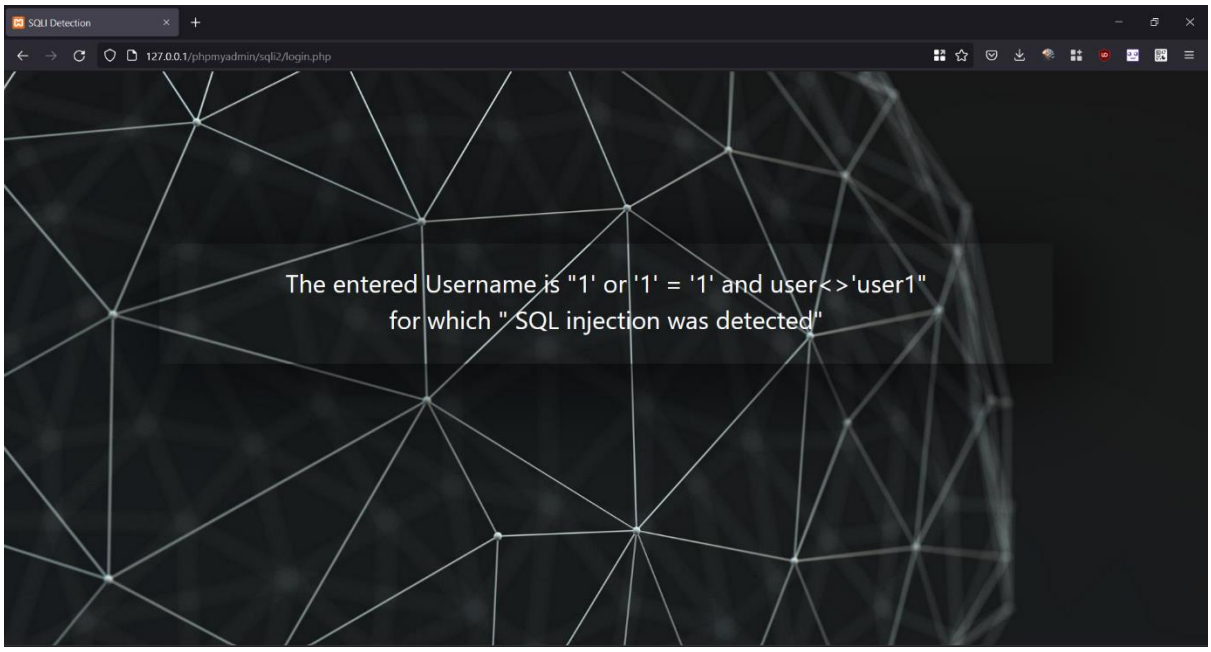
Training the classifier for legit keywords





Because it is a short string without any special characters and its probability is determined by the database count, the username “user1” that was entered above is secure from SQL injection. So, SQL injection was not detected.





The username “ 1’ or ‘1’ = ‘1’ and user<>‘user1 ” is an attempt at SQL injection with special characters that was used as an input to test the probability calculation of the classifier. Even though it is not specified in the training set in this case, the classifier classifies it as a vulnerable string. So, SQL injection is detected here.

16. Bibliography:

- https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c
- https://www.geeksforgeeks.org/naive-bayes-classifiers/
- https://en.wikipedia.org/wiki/Naive_Bayes_classifier
- https://www.sqlshack.com/sql-injection-detection-and-prevention/

III. SQL INJECTION PREVENTION

17. Previous work and intro to SQLI prevention:

In the first project, the approach used was vulnerable where the attacker can change the statement by entering a SQL injection query directly in the input username and password.

```
14   $query=mysqli_query($connection,"SELECT * FROM userpass WHERE user='$user'AND pass='$pass'");
```

Here, we can clearly see that the source code in the first project was vulnerable to manipulation. So, if the attacker inserted a vulnerable SQL query, the database information was easily accessible to him. So, when the attacker enters username and password as (' or 1=1 --), then the SQL query became "SELECT * FROM userpass WHERE user = '$user' AND pass ='' or 1=1--". The attacker gets access to the database since the logic statement "1=1" is always true and the rest of the statement is commented out.

In the second project, we detected SQL injection using Naïve Bayes Classifier, where the method of detection was based on machine learning. The Naïve Bayes classifier was used to filter out the numerous dangerous spam keywords using Bayesian decision theory, where we also teach the algorithm to filter out the terms that we consider to be spam and might harm the web application and so it keeps learning and making accurate decisions as time progresses. The web application's source code wasn't required to be changed in that method.

However, when the attacker tries brute forcing all the SQL injection queries, we need to prevent those by implementing prevention mechanism. Nowadays, we have advanced mechanisms to combat SQL injection attacks, using Prepared Statements in which before any parameters which are given by the user as an input are entered, the SQL statements are submitted to the SQL database server as well as compiled or converted by the SQL database server, MySQLI real escape string which is used to circumvent whatever special characters that an attacker may input in the username or password fields of login form on a website and Input Validation. Implementing these three mechanisms makes sure that the website will be completely safe against SQL injection attacks.

18. How to prevent against SQLI:

- To know the security threats that exist:
  The system's security is only at risk from user input. When performing a SQL Injection attack, the attacker can insert the queries as special characters, turning them into Boolean statements and gaining access to the database. The attacker can alter the data

by deleting, updating, or inserting fraudulent information after gaining access to the database. The HTML or the application's client side is where the input is provided.

- Vulnerabilities connected to the system development for the website:
  The client enters input username and password that are then cross-checked against the database to see if they match. If they do and pass, the client or user is allowed permission to access the database. Because of this, if the username and password entries are not properly checked, the attacker may exploit this and enter a customized query, giving them access to the database.

19. Prepared Statements:

Before any parameters which are given by the user as an input are entered, prepared statements, which are SQL statements are submitted to the SQL database server as well as compiled or converted by the SQL database server. We must build the query as well as the input separately for a prepared statement, sometimes referred to as a parameterized query. In this case, we specify all the SQL logic upfront and compile it, then enter parameters into the query immediately before execution. This is one of the methods which will help in completely safeguarding the website against any kind of SQL injection attacks, especially the one where hackers use brute force to keep injecting different SQL queries to gain access to the database and other private information of the users stored in the database.

Approach used in the first project which is vulnerable:

```
// Establish connection
$connection=mysqli_connect("localhost","root","");
$db=mysqli_select_db($connection,"test");

// Vulnerable approach
$query=mysqli_query($connection,"SELECT * FROM userpass WHERE user='$user'AND
pass='$pass'");
$row=mysqli_fetch_array($query);
```

Here, if we give the input as (1’ or ‘1’ = ‘1) the statement will become “ SELECT * FROM userpass WHERE user = ‘$user’ AND pass =‘1’ or ‘1’ = ‘1’ ” and the attacker gets access to the database since the logic statement “1=1” is always true. Due to the mixing of SQL code and user data, an attacker is now able to inject code, alter the SQL statement's structure, steal data, edit data, and even insert system instructions.

The following is an example of a safe approach using prepared statement:

```php
// Establish connection
$connection=mysqli_connect("localhost","root","");
$db=mysqli_select_db($connection,"test");

// Using "?" as placeholders
$stmt = $connection->prepare('SELECT * FROM table WHERE text = ? AND number = ?');

// Bind parameters to statement, here, "s" for string and "i" for integer
$stmt->bind_param("si", $text, $number);

// Final Step is to execute
$stmt->execute();
```

Here, we first define the structure of the query. We write the query without parameters and use "?" as placeholders. This string will be sent to the SQL server to be converted into SQL code. We can then send over the query's parameters separately. Here, "si" denotes that we are providing two parameters, the first of which is a string and the second of which is an integer. Finally, we execute the query.

20. Implementation and working of prepared statement:

```php
20  //SQLI prevention using prepared statement
21  $stmt = $connection->prepare('SELECT * FROM userpass WHERE user = ? AND pass = ?');
22  $stmt->bind_param('ss', $user, $pass);
23  $stmt->execute();
24  $result = $stmt->get_result();
```

As we can see in the "login.php" file we have successfully implemented prepared statement with the help of which we can make our website completely secure against SQL injection attacks consisting of brute force approach.

On line 21, almost all the SQL statement is the same as before, we create an object called $stmt and then we call the prepare method, here as we cannot use variable names directly and instead, we substitute "?" in the place of $user and $pass.

On line 22, we are going to execute bind parameters which has 2 different parameters which defines what the "?" means, the first parameter will be "ss" as both the variables are string, also we are binding the "?" to a variable, which here is $user and $pass.

On line 23, we execute the query and on line 24, we store the result in the object called $result.

21. MySQLI real escape string:

mysqli_real_escape_string() is used to circumvent whatever special characters that an attacker may input in the username or password fields of login form on a website as well as to insert input in a table in a database that contains special characters, circumventing the special characters totally shields the website from SQLI attacks, which often include manipulating the SQL query in the source code with special characters.

Syntax and example for the usage of MySQLI real escape string is shown below:

```
// mysqli_real_escape_string() syntax:
mysqli_real_escape_string(connection_var, string_var)

// mysqli_real_escape_string() example:
$example = mysqli_real_escape_string($connection, $example);
```

22. Implementation and working of MySQLI real escape string:

```
14   //SQLI prevention using mysqli_real_escape_string
15   $user=stripcslashes($user);
16   $pass=stripcslashes($pass);
17   $user=mysqli_real_escape_string($connection, $user);
18   $pass=mysqli_real_escape_string($connection, $pass);
```

As we can see in the "login.php" file we have successfully implemented mysqli_real_escape_string() with the help of which we can make the entered SQL query to be considered as a string. Thus, protecting against SQL injection attack.

On line 15 and 16 we use stripcslashes() to remove the back slashes in front of the provided words.

On line 17 and 18 we can see that we have used mysqli_real_escape_string() for $user and $pass which will take care of the username and password input if it contains any special characters used by hackers for SQL injection and make it completely considered like a string.

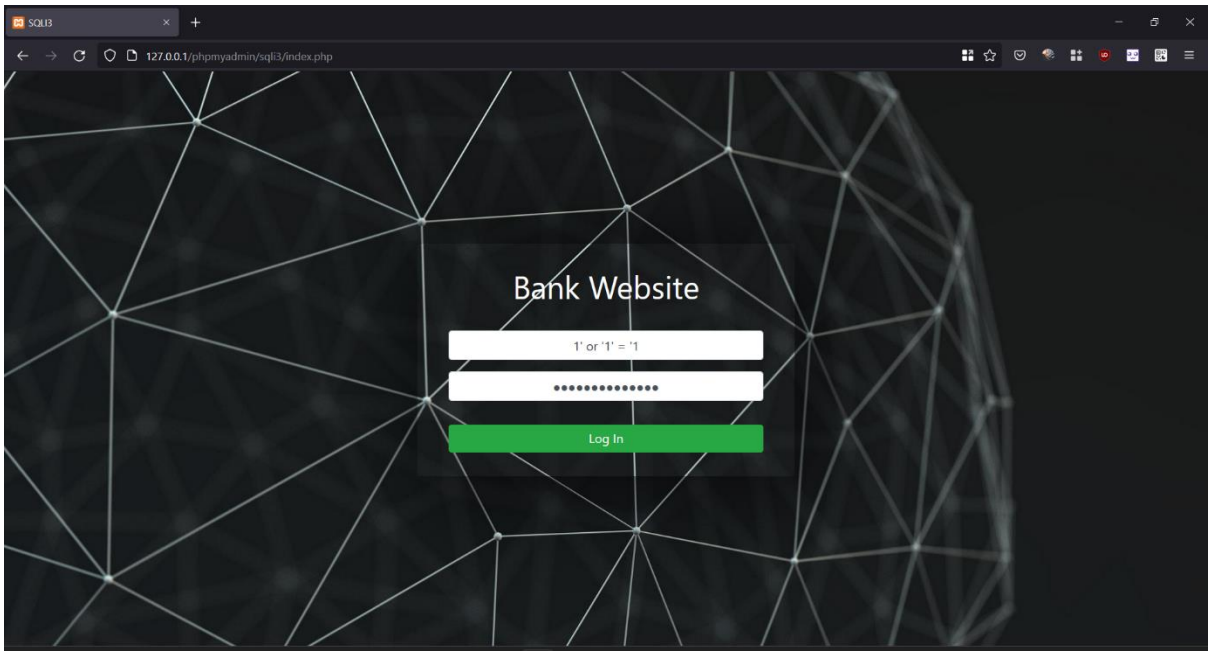
23. Input Validation implementation and working:

```
<!-- SQLI Prevention using input validation -->
<script>
    function validate() {
        var bool = /^[-\w\.\$@\*\!]{1,10}$/.test(document.getElementById("user").value);
        var next = /^[-\w\.\$@\*\!]{1,15}$/.test(document.getElementById("pass").value);
        if (bool == false || next == false) {
            document.getElementById("form").action = "invalid.php";
        }
    }
</script>
```

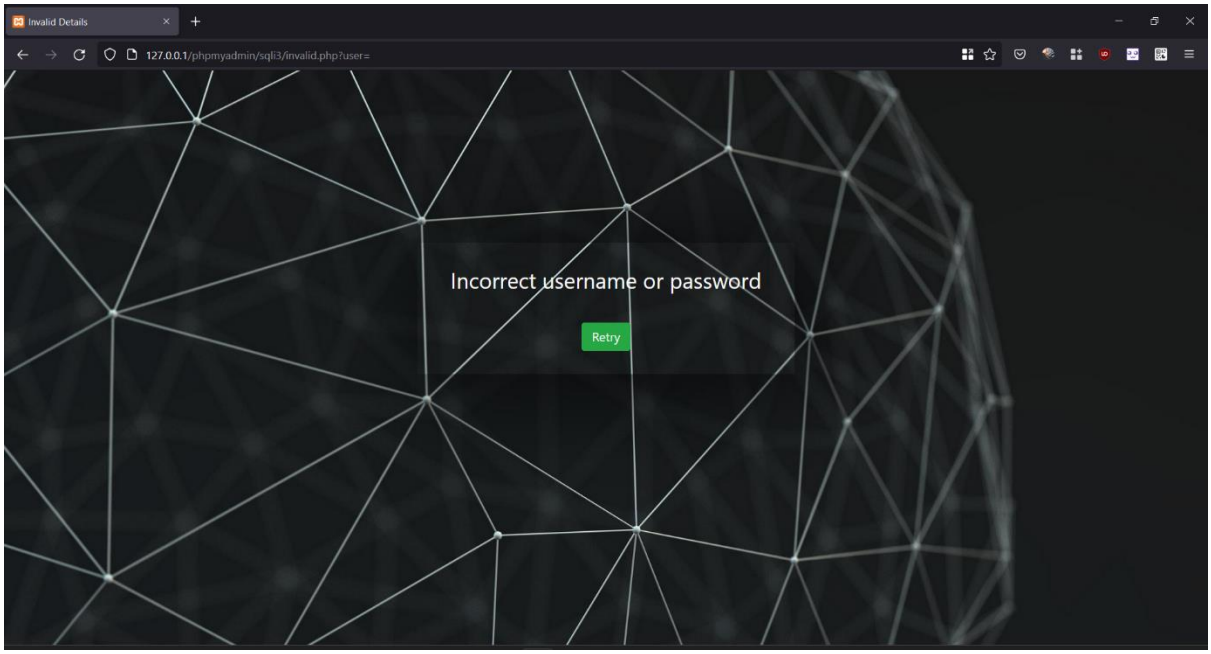
The validate function verifies the username and password. To match the inputs with the regular expression, we use this. The validation will fail even if one of both, user or pass, are false, preventing the attacker from accessing the database. Its username and password inputs are handled by bool and next, respectively. Only the special characters listed as being permitted are allowed.

An error will be raised and validation will fail if the attacker inputs something different from what is allowed, sending the user to "invalid.php" which is the error page. The password field allows for a maximum of 15 characters.

24. Screenshots:



The attacker should get access once the string for bypassing the security check is submitted. However, due to prepared statement in which before any parameters which are given by the user as an input are entered, the SQL statements are submitted to the SQL database server as well as compiled or converted by the SQL database server, mysqli_real_escape_string() where we circumvent whatever special characters that an attacker may input in the username or password fields of login form on a website and input validation, it doesn't happen.



Due to the special characters in the string that are not permitted, it displays an error and directs the attacker to the error page. This is how the prevention mechanism against SQL injection works using prepared statement, mysqli_real_escape_string() and input validation.

25. Bibliography:

- https://en.wikipedia.org/wiki/Prepared_statement
- https://www.php.net/manual/en/mysqli.quickstart.prepared-statements.php
- https://www.w3schools.com/php/php_mysql_prepared_statements.asp
- https://www.youtube.com/watch?v=_jKylhJtPmI
- https://www.php.net/manual/en/mysqli.real-escape-string.php
- https://www.w3schools.com/php/func_mysqli_real_escape_string.asp
- https://www.geeksforgeeks.org/php-mysqli_real_escape_string-function/
- https://logz.io/blog/defend-against-sql-injections/
- https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks/
- https://www.red-gate.com/simple-talk/databases/sql-server/learn/sql-injection-defense-in-depth/

IV. SQL INJECTION DECEPTION

26. Previous Work:

There are four projects, all of which center around the topic of SQL injection. The first project focuses on the offense demonstration component of the attack, the second on the detection of the SQLI attempt, the third on the implementation of prevention mechanisms to counter the SQLI attack, and this project, which is the fourth and final project, is based on deceiving the attacker rather than preventing the attack. The following is a summary of each project:

The first project consisted of creating a website that was vulnerable to SQLI for demonstration purposes. Here we came across the process of SQLI in action when the attacker enters the input username and password as (1' or '1' = '1' and user<>'user1). The SQL statement or query in the source code gets manipulated and becomes (SELECT * FROM userpass WHERE user='1' or '1' = '1' and user<>'user1'AND pass='1' or '1' = '1' and user<>'user1') and he gets unauthorized access to the database and all of its contents.

The second project consisted of creating a mechanism that detects the SQLI attempts even before the prevention mechanism is used. For this, we used Naïve Bayes Classifier, which was based on the Bayesian decision theory, and classified the input username or password as SQL injection string or legit string. We provided a training set for helping the classifier with the classification of the input string. In the end, it is displayed if SQLI was detected or not.

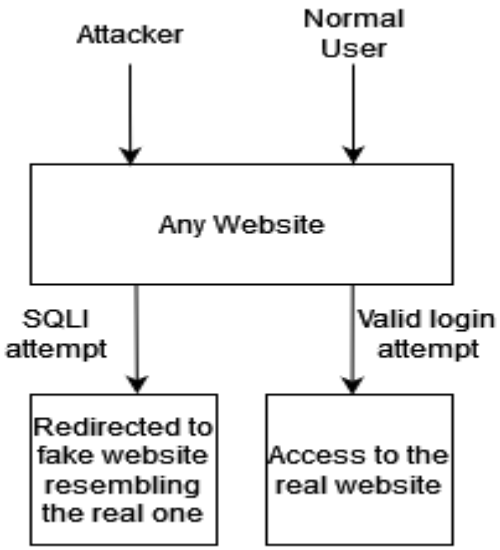
The third project consisted of creating three prevention mechanisms, namely: prepared statements, mysqli_real_escape_string() and input validation. In Prepared statements, we write queries without providing the parameters at the beginning, it is provided later. In mysqli_real_escape_string(), we circumvent the most used thing in SQLI which is special characters. In input validation, only special characters listed as being permitted are allowed.

In this project, which is the fourth and last project, we implement a deception mechanism where rather than blocking the attacker from accessing the database, we redirect him to a fake website that he believes to be legitimate but isn't. He thinks he has obtained access to the server, but all he is doing is visiting a fake website we set up so we can keep track of him all the time and monitor his actions. This mechanism is essential because an attacker who obtains access to susceptible data may disclose it online.

27. Introduction to Deception in SQLI:

Deception in SQLI is, tricking the attackers using lots of fake components throughout the website that resembles real components which here is the fake website("fakevalid.php") the attacker gets redirected to. We can keep track of the attacker and all the methods he used to get the access and take advantage of any vulnerabilities. We can then use all the analyzed information to make the system more secure and reduce the future risks of SQLI attacks.

28. Why is Deception required in SQLI:



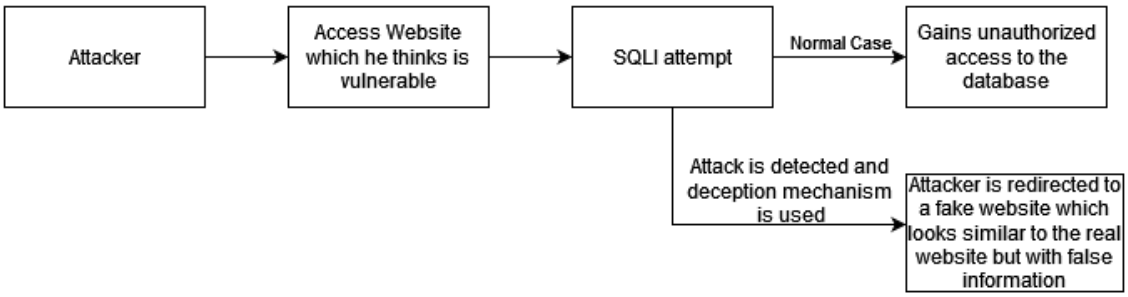
As we can see from the diagrammatic representation above, there are 2 cases:

The first case is the normal case where a user tries to login using the correct username and password in a login form on a particular website. On successful login, the user will be redirected to the account information webpage. Sometimes the user will not be able to enter the information correctly or may enter incorrect information in the login form. In that case, the user will be notified that the entered information is incorrect and is sent back to the login form to attempt the login again.

The second case is where the attacker will try to access the website thinking that it's vulnerable, but as we have already implemented the deception mechanism, this won't be the case. Here the attacker enters an SQL injection query in the login form so that he can access the database consisting of all the information of the users and will try to leak it online or sell the information. But the attacker is redirected to a fake website that resembles the real one.

We can do whatever we can to track the attacker, we can note down the IP Address of the attacker and we can notify the higher authorities to take the required action. This is done so that we can waste as much time of the attacker as possible in exploring the things that we have set up and in the meanwhile, we can analyze the information we have on the attacker and track him.

29. Steps involved in Deception and the end goal:



The following steps are involved in the process of Deception in SQLI:

Step 1: The attacker accesses the website.

In this step, the attacker tries to access the website and test out if it's vulnerable using his own methods. This can result in various kinds of attacks on the website, one of which can be SQLI attack.

Step 2: Tries to attempt SQLI.

The attacker then tries to attempt SQLI attack by entering various SQLI queries, which can manipulate the source code and give the attacker complete access to the database containing all the user account information.

Step 3: Thinks he has access to the database information.

Once the attacker attempts SQLI, he then gets access to all the information that he thinks is real but is in fact fake information kept in place.

Step 4: Gets redirected to the fake webpage.

The attacker here is redirected to a fake webpage where he is monitored continuously, and all the displayed information is fake and will result in no loss to any user or organization.

Step 5: Information is noted of the attacker.

We can take note of the attacker's IP Address, notify the higher authorities about the attack, monitor all the actions being performed by the attacker etc.

The end goal is to waste the time of the attacker as much as possible by letting him get fake information or data, improve the defense mechanism by analyzing the attacker's way of finding the vulnerability on the website by tracking the attacker's actions as soon as he is redirected to the fake website. The tracking data can also be utilized to determine the type of data the attacker is seeking.

30. Working and implementation of Deception mechanism in SQLI:

```php
<?php

$error = '';
session_start();
$user = isset($_POST['user']) ? $_POST['user'] : '';
$pass = isset($_POST['pass']) ? $_POST['pass'] : '';

require_once('NaiveBayesClassifier.php');
$classifier = new NaiveBayesClassifier();
$spam = Category::$SPAM;
$ham = Category::$HAM;

$categoryr = $classifier->classify($user);
$categoryp = $classifier->classify($password);

if($categoryr==$spam || $categoryp==$spam)
    header('Location: fakevalid.php');
else
{
$directurl="valid.php";
$nodirecturl="invalid.php";

if((preg_match("/^[-\w\.\$@\*\!]{1,10}$/",$user)) && (preg_match("/^[-\w\.\$@\*\!]{1,10}$/",$pass)) )
{
$connection=mysqli_connect("localhost","root","");
$db=mysqli_select_db($connection,"test");

$query=mysqli_query($connection,"SELECT * FROM userpass WHERE user='$user'AND pass='$pass'");
$row=mysqli_fetch_array($query);
$user1=$row['user'];
$amount=$row['amount'];
if($query){
    if($row!=""){
    header("location:$directurl?user=$user1 && amount=$amount");
    }
    else{
    header("location:$nodirecturl?user=$user1");
    }
    session_destroy();
}
else{$error='Error';}
    mysqli_close($connection);
}
else{
    header("location:$nodirecturl");
    session_destroy();
    }
}

?>
```

As shown above, we have successfully implemented the deception mechanism, the explanation is as follows:

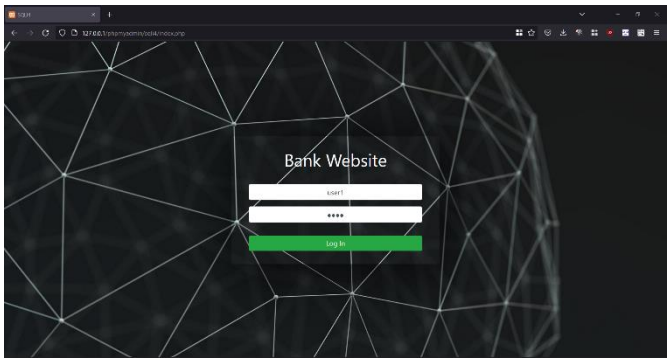
On line 8, we use “NaiveBayesClassifier.php” for the pre-existing implementation of the detection mechanism used in detecting the SQLI attempts, we use it on both the inputs, i.e., username and password.

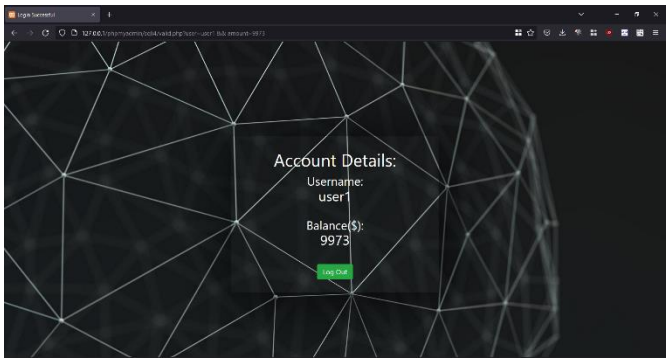
On lines 9, 10 and 11, we use Naïve Bayes Classifier by creating $classifer and for spam and legit keyword training set usage we create $spam and $ham respectively.

On lines 13 and 14, we create $categoryr and $categoryp and classify spam and legit keywords from input username and password respectively.

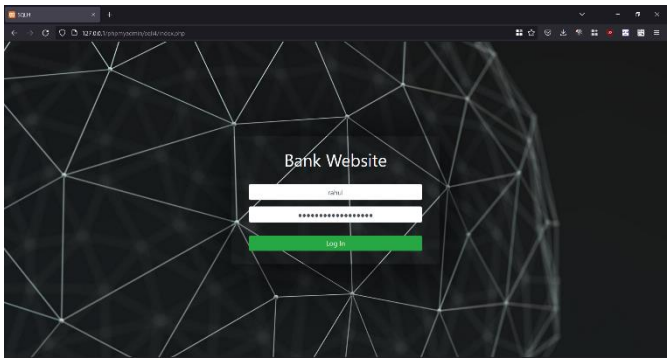
On line 16, as we can see, if any of the input username or password falls under the spam category, it will automatically redirect to fake website which is “fakevalid.php” here.

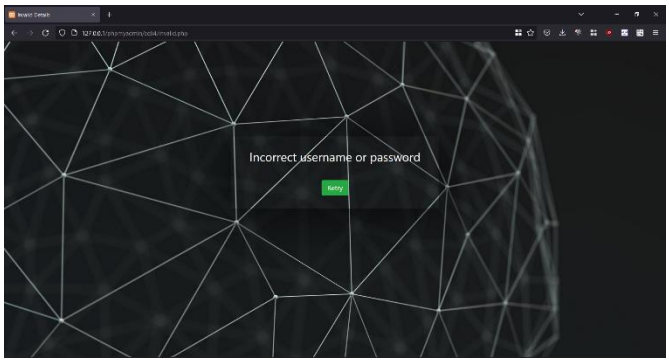
31. Screenshots:



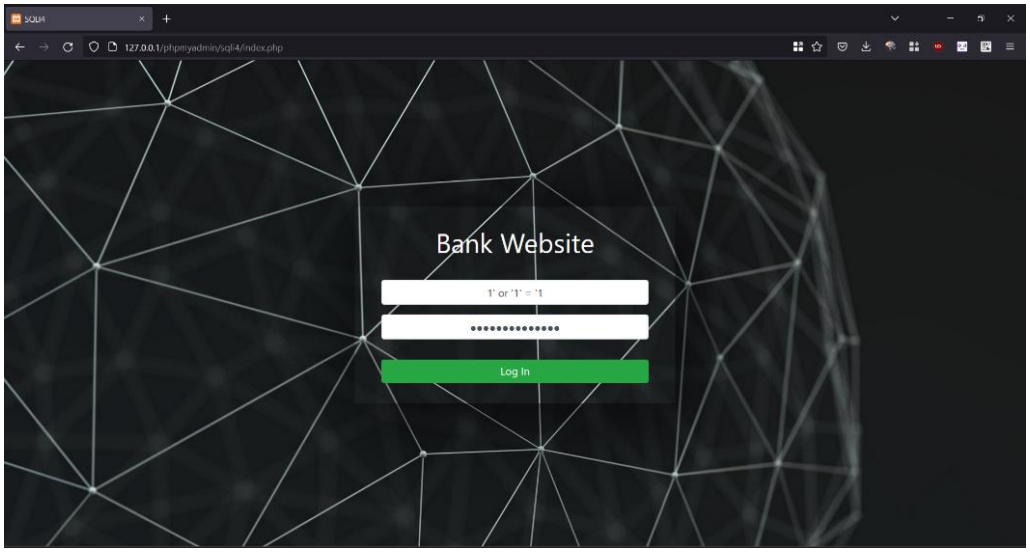


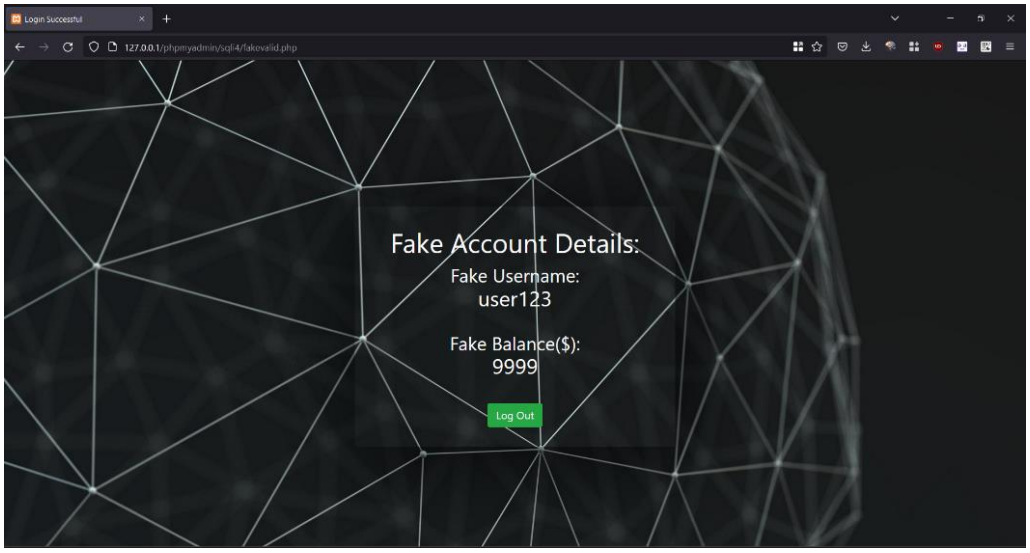
Entering the correct username and password as shown in the screenshots above, we get access to that particular user’s information and all the account details associated with it.





Entering an invalid username and password as shown in the screenshots above, we are displayed a message “Incorrect username or password”. We are shown a button to retry the login procedure again.





Here we can see that as soon as the attacker tries to attempt SQLI attack, it is detected by the classifier and is redirected to the fake website, with all the fake account information, so that no one is at a loss if he decides to leak the fake information online. The fake website accessed here is “fakevalid.php” which is accessed whenever the attacker attempts SQLI attack, thus deceiving the attacker.

32. Bibliography:

- https://www.forcepoint.com/cyber-edu/deception-technology
- https://www.techtarget.com/whatis/definition/deception-technology
- https://www.tutorialspoint.com/what-is-deception-technology-in-cybersecurity
- https://en.wikipedia.org/wiki/Deception_technology