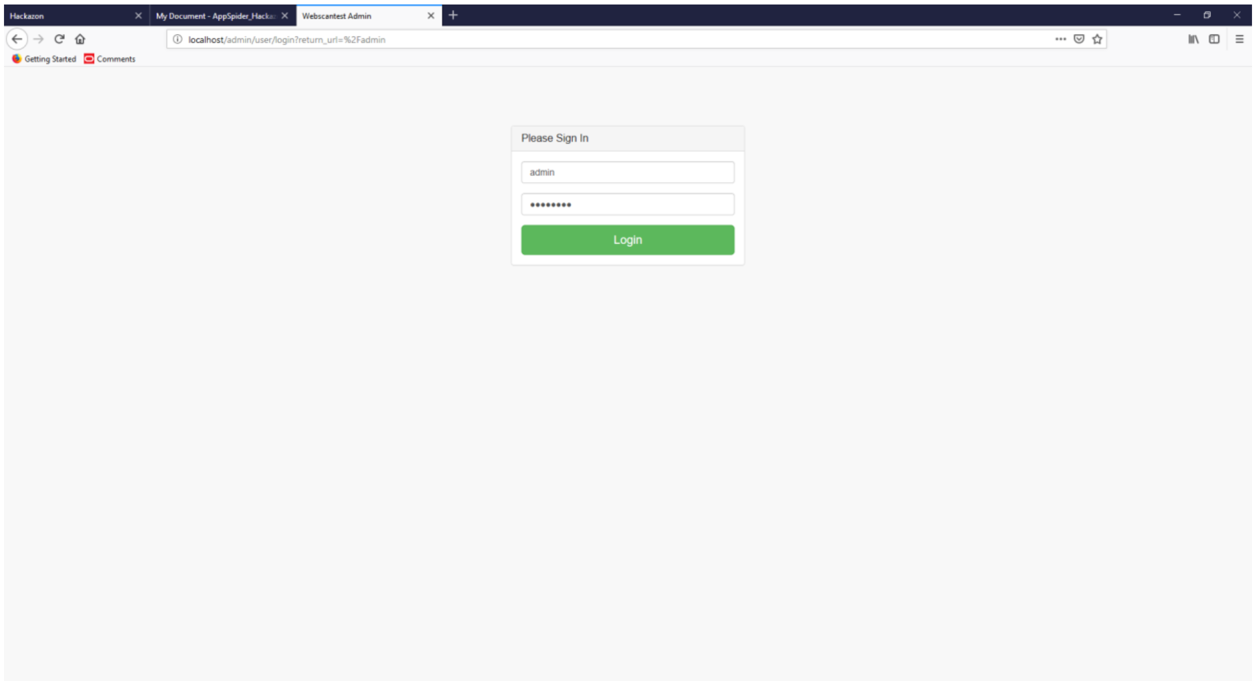
SQL Injection

What is an SQL Injection

An SQL injection is a code injection that manipulates the database to perform an alternative task than intended. Typically, an SQL injection is the placement of an SQL statement into a text field that will be read as a normal statement but when either filtered or accepted by the database, changes the SQL statement to perform some malicious activity.

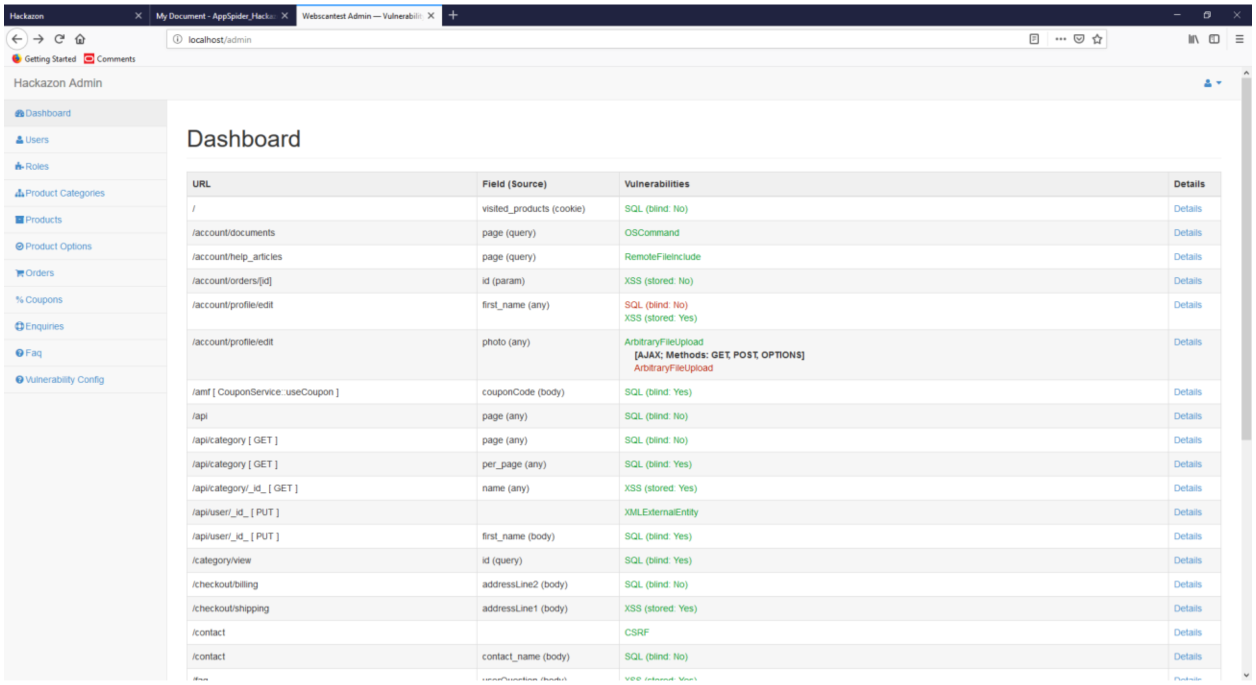
How to perform an SQL Injection

1. To perform an SQL injection against Hackazon, you need to first enable the SQL injection vulnerability. After the user has started up their local Hackazon website, the user must go to the Hackazon admin page at **/localhost/admin.**



2. Login in with the admin credentials that you created when installing Hackazon

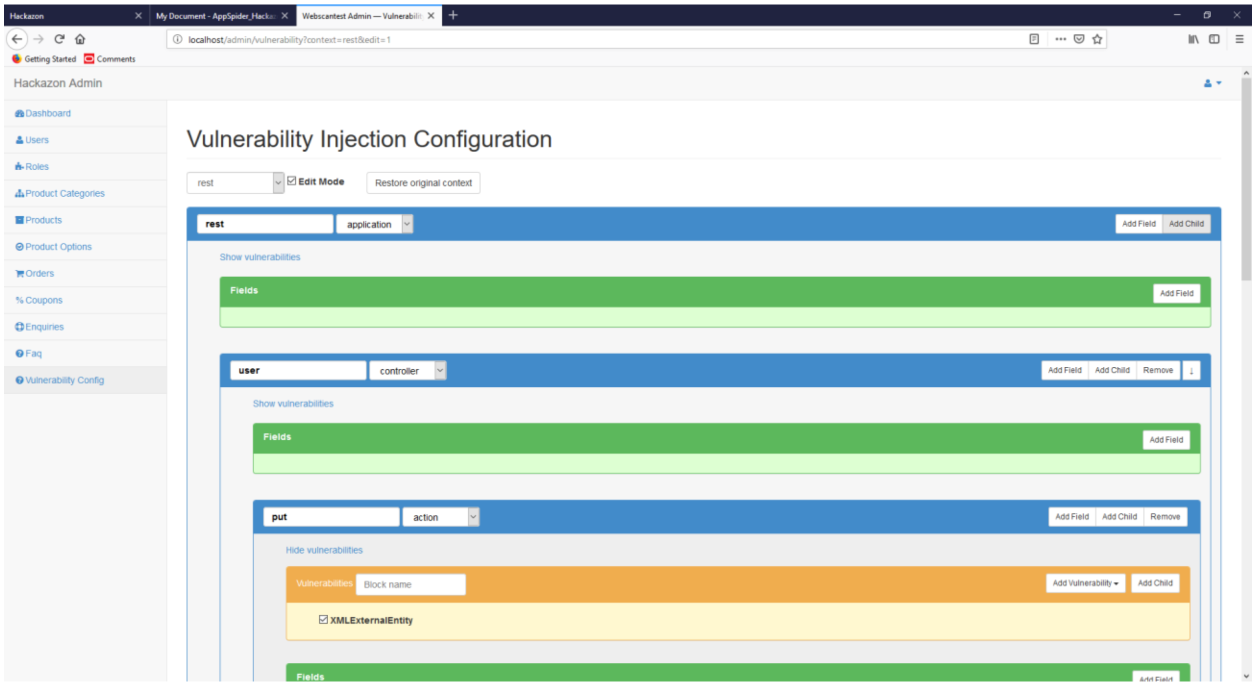
3. The first page that you will see will be a list of the vulnerabilities that Hackazon has and where they are. Click the **Vulnerability Config** option in the left side menu


Hackazon Admin

Dashboard

URL	Field (Source)	Vulnerabilities	Details
/	visited_products (cookie)	SQL (blind: No)	Details
/account/documents	page (query)	OSCommand	Details
/account/help_articles	page (query)	RemoteFileInclude	Details
/account/orders/[id]	id (param)	XSS (stored: No)	Details
/account/profile/edit	first_name (any)	SQL (blind: No) XSS (stored: Yes)	Details
/account/profile/edit	photo (any)	ArbitraryFileUpload **[AJAX; Methods: GET, POST, OPTIONS]** ArbitraryFileUpload	Details
/amf [CouponService::useCoupon]	couponCode (body)	SQL (blind: Yes)	Details
/api	page (any)	SQL (blind: No)	Details
/api/category [GET]	page (any)	SQL (blind: No)	Details
/api/category [GET]	per_page (any)	SQL (blind: Yes)	Details
/api/category/_id_ [GET]	name (any)	XSS (stored: Yes)	Details
/api/user/_id_ [PUT]		XMLExternalEntity	Details
/api/user/_id_ [PUT]	first_name (body)	SQL (blind: Yes)	Details
/category/view	id (query)	SQL (blind: Yes)	Details
/checkout/billing	addressLine2 (body)	SQL (blind: No)	Details
/checkout/shipping	addressLine1 (body)	XSS (stored: Yes)	Details
/contact		CSRF	Details
/contact	contact_name (body)	SQL (blind: No)	Details

4. Select **rest** from the drop-down menu and select the **Edit Mode** check box


Hackazon Admin

Vulnerability Injection Configuration

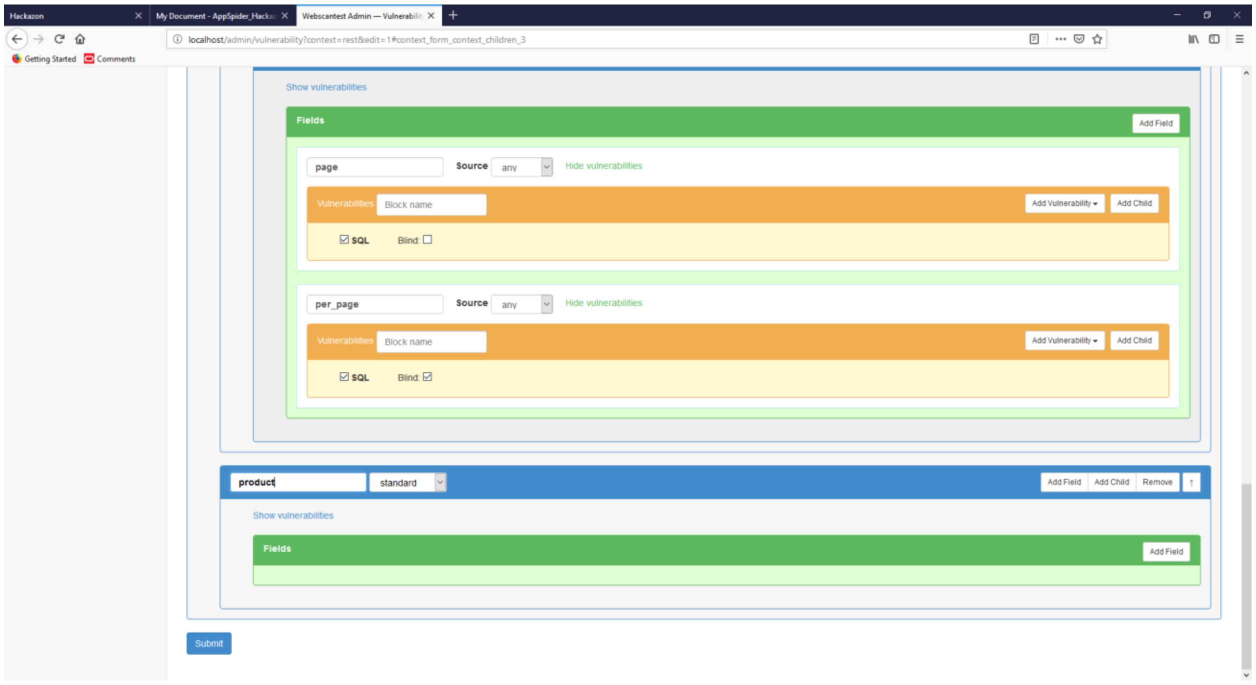
rest ☑ Edit Mode | Restore original context

rest — application
Show vulnerabilities
Fields

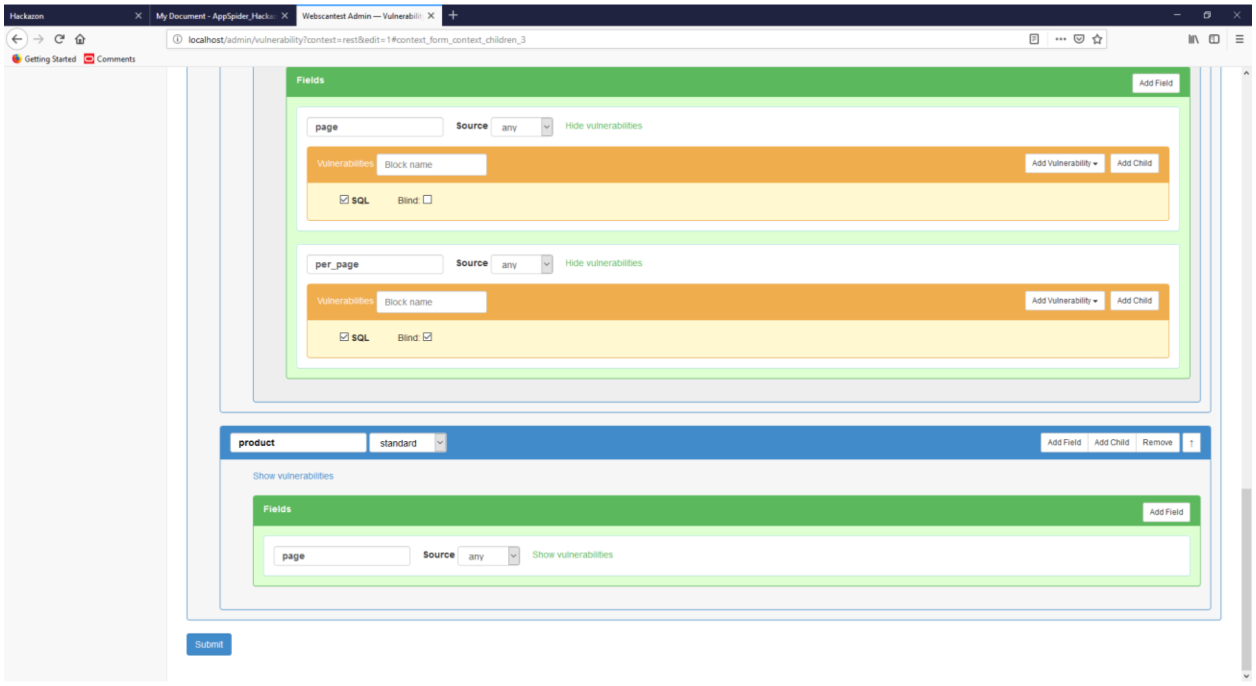
user — controller
Show vulnerabilities
Fields

put — action
Hide vulnerabilities
Vulnerabilities — Block name
☑ XMLExternalEntity
Fields

5. Click the **Add Child** button in the rest vulnerability header. The application will generate an empty child box. Name this child **product**

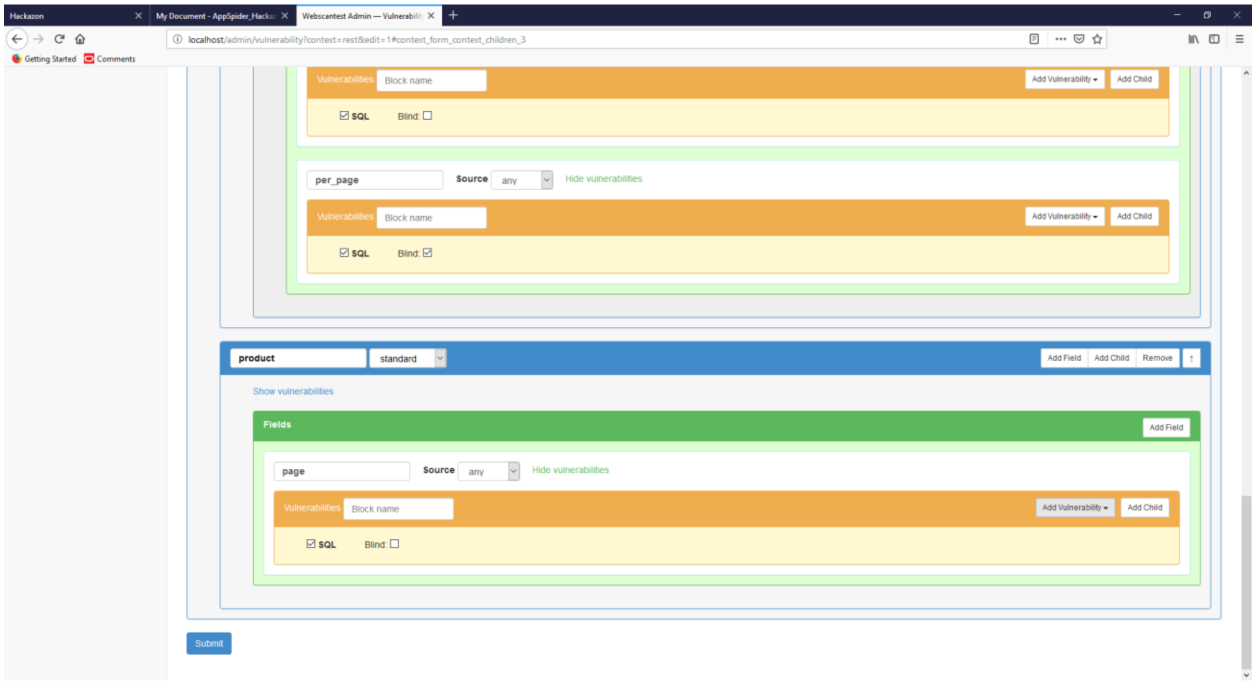


6. Click **Add Field** and name the field **page**



7. Click the **Show vulnerabilities** button and click the **Add Vulnerability** drop-down menu and choose SQL

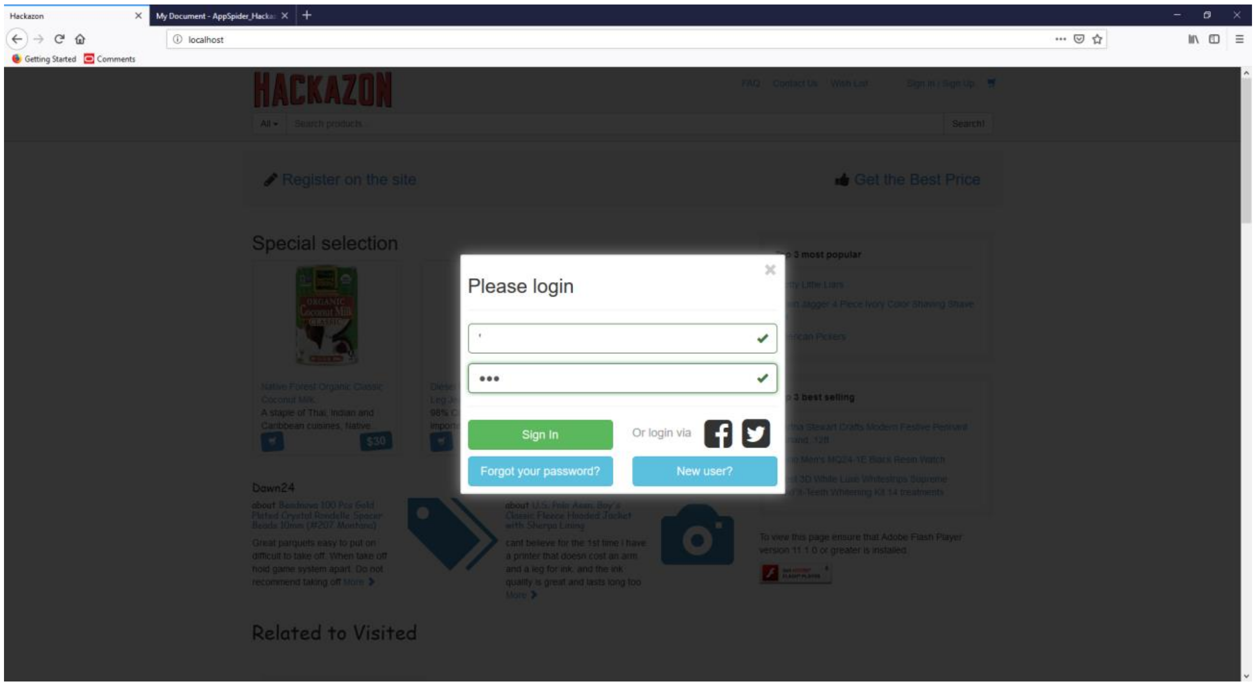
8. Select the **SQL** check box to enable the SQL Injection



localhost/admin/vulnerability?context=rest&edit=1#context_form_context_children_3

Vulnerabilities	Block name	☑ SQL	Blind: ☐
per_page	Source: any	Hide vulnerabilities	
Vulnerabilities	Block name	☑ SQL	Blind: ☑
product	standard	Show vulnerabilities	
Fields	page	Source: any	Hide vulnerabilities
Vulnerabilities	Block name	☑ SQL	Blind: ☐

9. Next is to find the fields that are vulnerable to an SQL injection. Refer to the admin dashboard to find the exact fields and where they are. You can test the fields by inputting a quotation mark (‘). If you receive any kind of error page, that means this field is vulnerable. This is because the quotation mark is an exceptional input. So, the website deals with it differently from the other considered normal input.



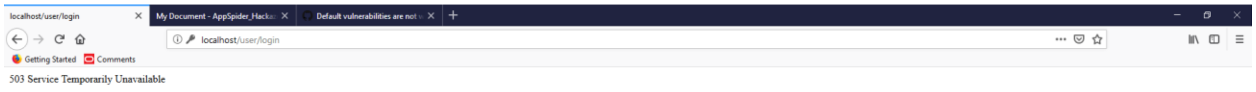
localhost

Please login

'

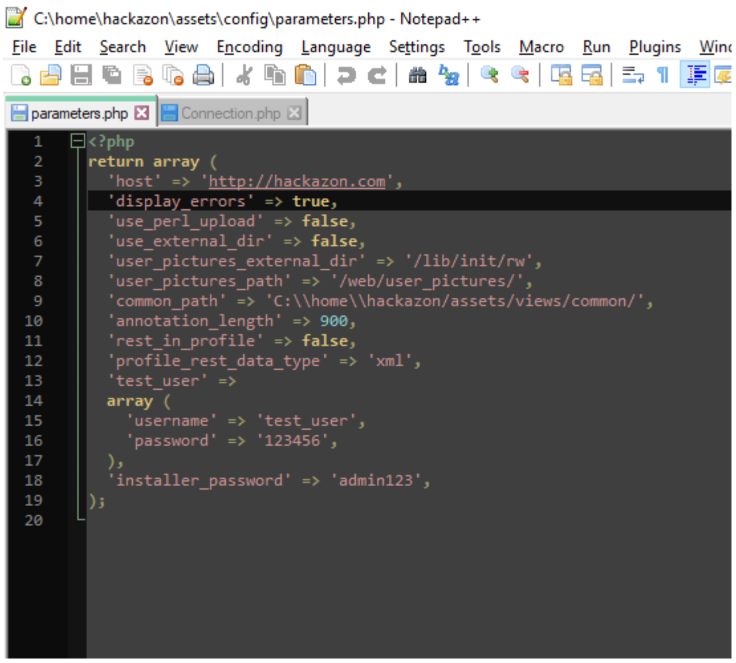
•••

10. When you first input a quotation mark (‘) into the username field, the website will return an error page with no other information. This is the page that indicates that this input is exceptional and is being dealt with differently.

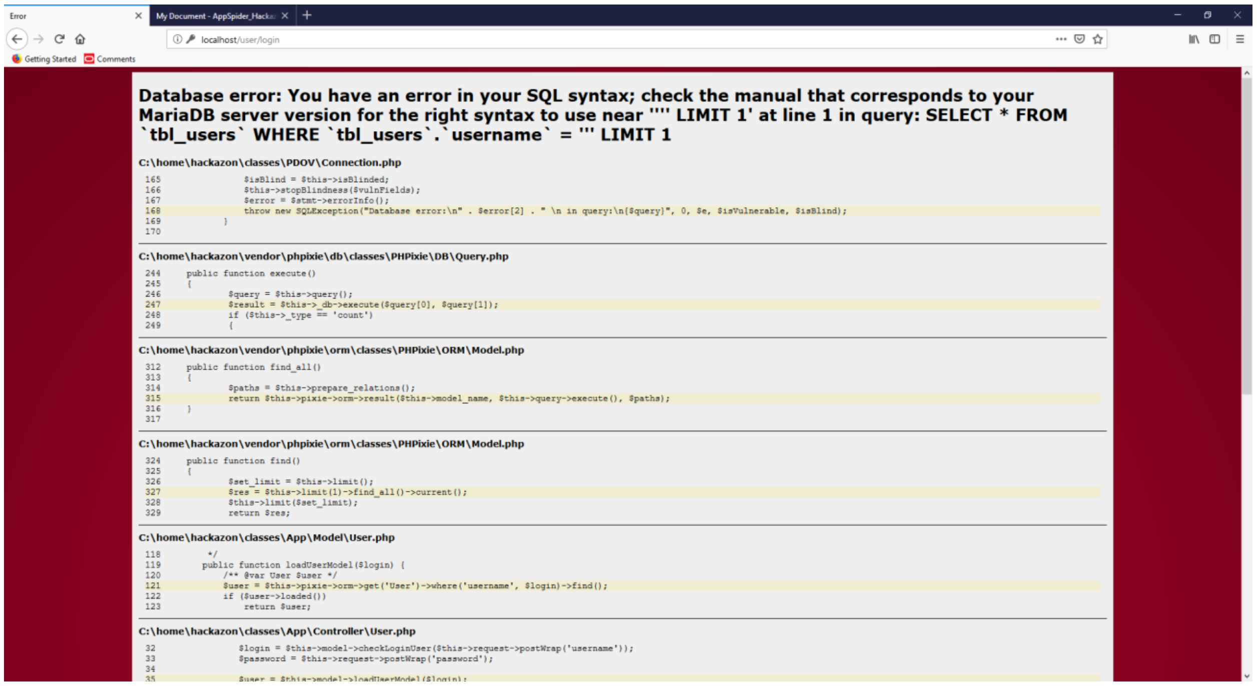


503 Service Temporarily Unavailable

11. To change this so you get an informative error page, which gives you a little more insight to what is happening. Modify the config file in **/assets/config/parameters.php** and set **display_errors => true.**


C:\home\hackazon\assets\config\parameters.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Wind
parameters.php
Connection.php

```php
<?php
return array (
  'host' => 'http://hackazon.com',
  'display_errors' => true,
  'use_perl_upload' => false,
  'use_external_dir' => false,
  'user_pictures_external_dir' => '/lib/init/rw',
  'user_pictures_path' => '/web/user_pictures/',
  'common_path' => 'C:\\home\\hackazon/assets/views/common/',
  'annotation_length' => 900,
  'rest_in_profile' => false,
  'profile_rest_data_type' => 'xml',
  'test_user' =>
  array (
    'username' => 'test_user',
    'password' => '123456',
  ),
  'installer_password' => 'admin123',
);
```



Database error: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '''' LIMIT 1' at line 1 in query: SELECT * FROM `tbl_users` WHERE `tbl_users`.`username` = ''' LIMIT 1

C:\home\hackazon\classes\PDOV\Connection.php

```
            $isBlind = $this->isBlinded;
            $this->stopBlindness($vulnFields);
            $error = $stmt->errorInfo();
            throw new SQLException("Database error:\n" . $error[2] . " \n in query:\n{$query}", 0, $e, $isVulnerable, $isBlind);
        }

```

C:\home\hackazon\vendor\phpixie\db\classes\PHPixie\DB\Query.php

```
    public function execute()
    {
        $query = $this->query();
        $result = $this->_db->execute($query[0], $query[1]);
        if ($this->_type == 'count')
        {
```

C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php

```
    public function find_all()
    {
        $paths = $this->prepare_relations();
        return $this->pixie->orm->result($this->model_name, $this->query->execute(), $paths);
    }

```

C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php

```
    public function find()
    {
        $set_limit = $this->limit();
        $res = $this->limit(1)->find_all()->current();
        $this->limit($set_limit);
        return $res;
```

C:\home\hackazon\classes\App\Model\User.php

```
     */
    public function loadUserModel($login) {
        /** @var User $user */
        $user = $this->pixie->orm->get('User')->where('username', $login)->find();
        if ($user->loaded())
            return $user;
```

C:\home\hackazon\classes\App\Controller\User.php

```
            $login = $this->model->checkLoginUser($this->request->postWrap('username'));
            $password = $this->request->postWrap('password');

            $user = $this->model->loadUserModel($login);
```

12. We will be using the user login field for our injection

a. Now we just input our SQL injection command. Input the following command into the username field, **' OR 1=1 #** **or you can input** **' OR 'a'='a**

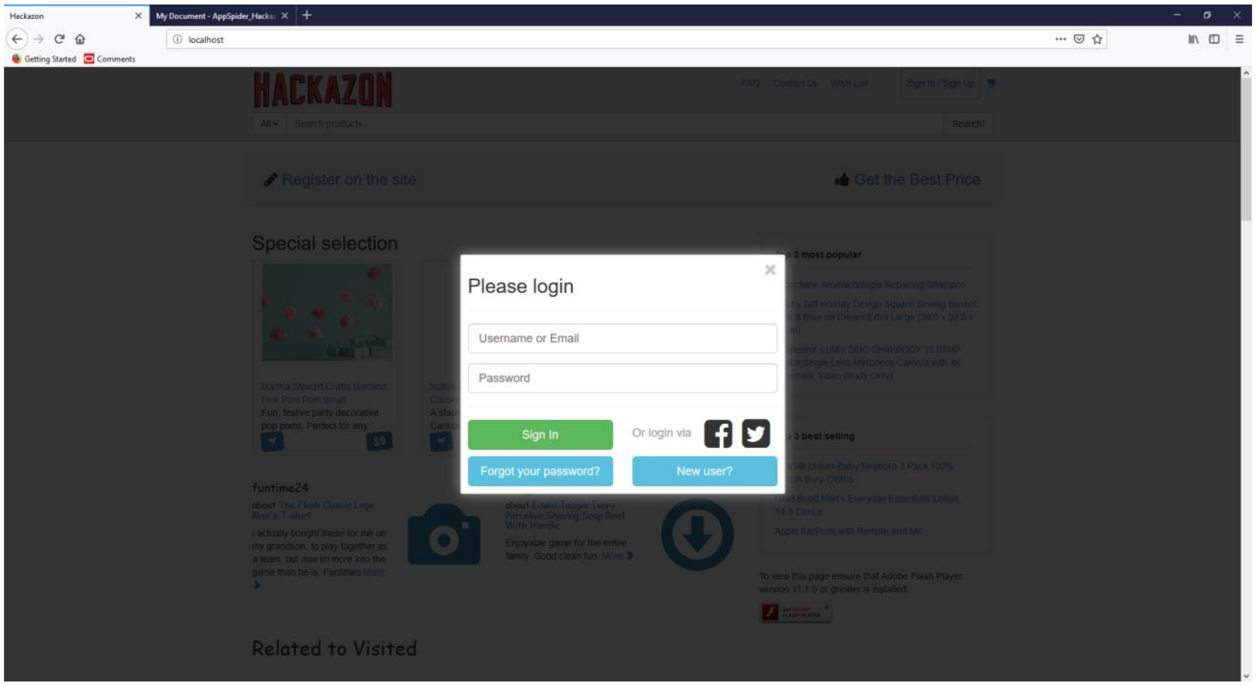
- The rest of the statement will proceed as normal. Which means the query will look through all users in the database until it finds a match, or a true value is returned.

- The quotation mark (‘) is what ends the SQL statement.

- The OR will ensure that a true Boolean value is returned. Meaning that either a username will be returned if found, or true will be returned

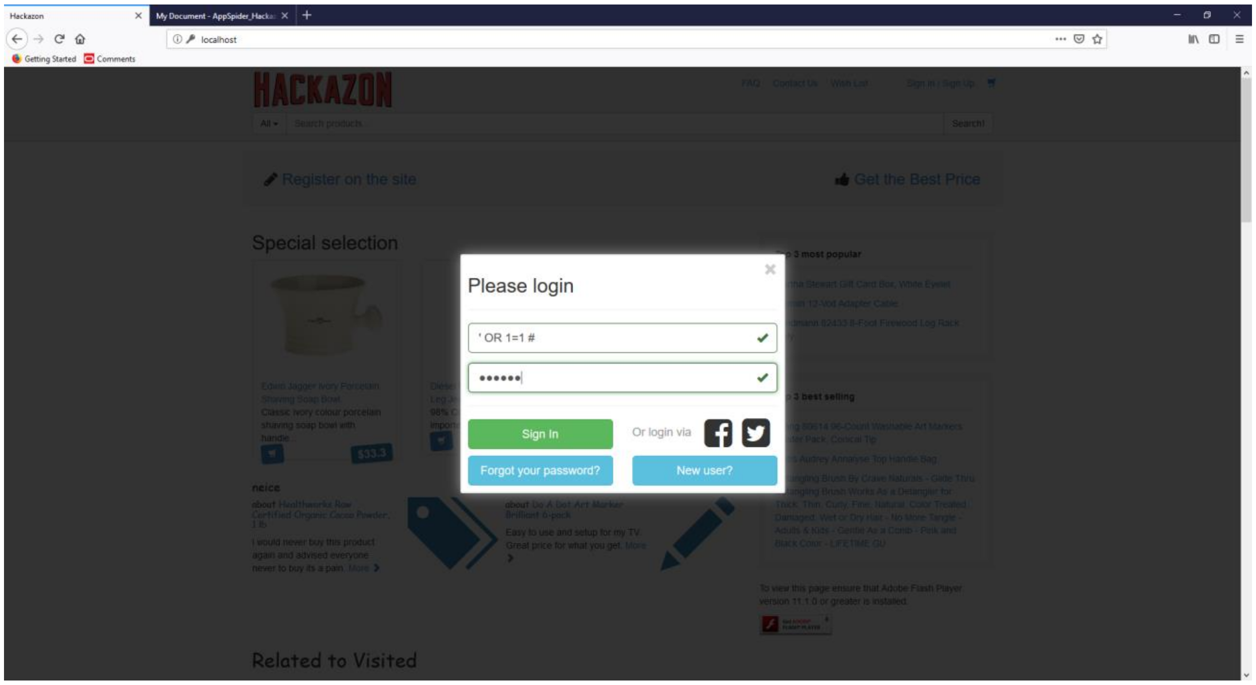
- 1=1 will always return true, meaning that no matter if no username is found, the database will return a value that indicates a successful login.

- The # makes the rest of the statement a comment. So, the purpose of this is to cancel out the password query. Thus, rendering that the statement does not need a password.



13. For the password field enter a valid password for a user.

a. We had problems with any other account information, but this seemed to work. with the test_user only. So input **123456** as the password

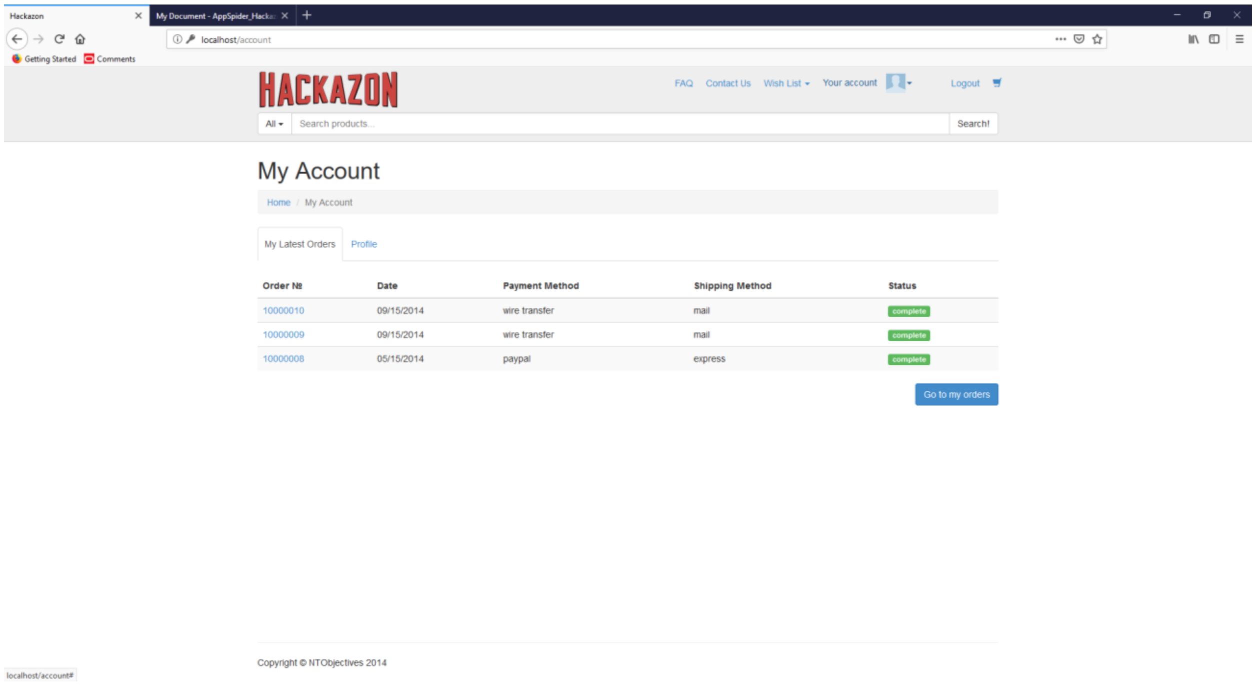


Please login

' OR 1=1 #

••••••

14. If the command was correctly inputted, the user will be logged in.



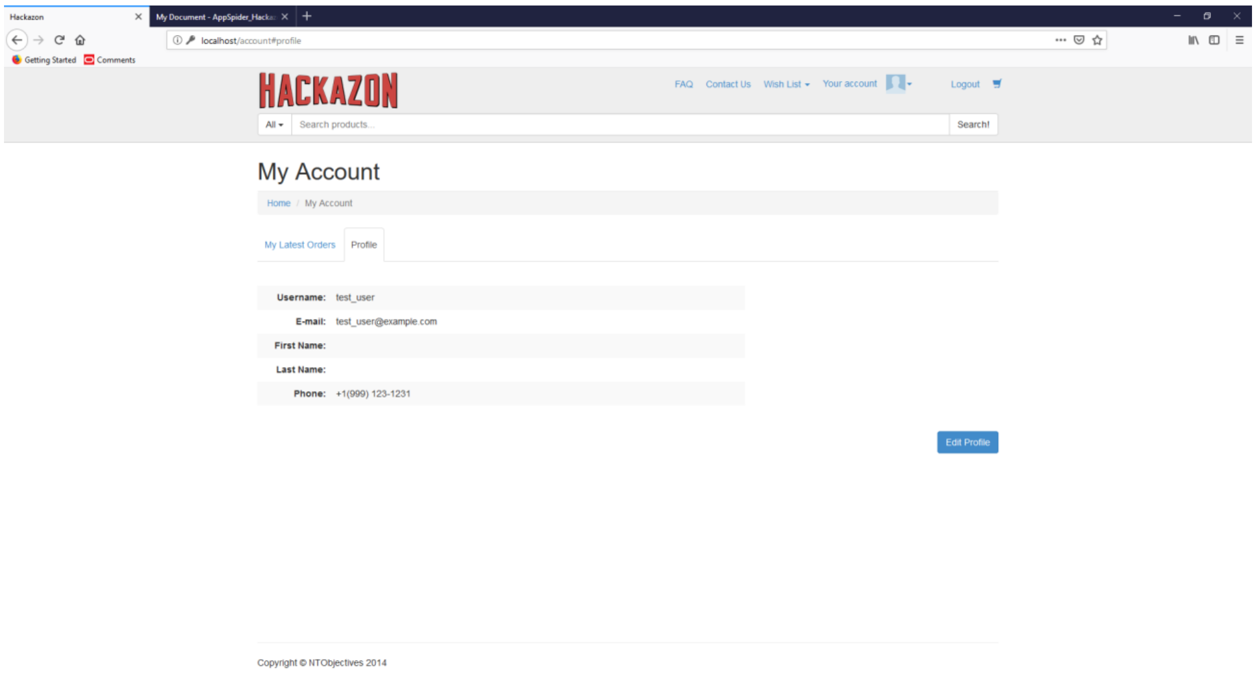
localhost/account

My Account

Home / My Account

Order №	Date	Payment Method	Shipping Method	Status
10000010	09/15/2014	wire transfer	mail	complete
10000009	09/15/2014	wire transfer	mail	complete
10000008	05/15/2014	paypal	express	complete

Copyright © NTObjectives 2014



My Account

Home / My Account

My Latest Orders | Profile

Username:	test_user
E-mail:	test_user@example.com
First Name:	
Last Name:	
Phone:	+1(999) 123-1231

Edit Profile

Copyright © NTObjectives 2014

How to fix an SQL Injection Vulnerability

Through my research, a good way to prevent an SQL injection is through prepared statements in PDO or MYSQLi for its queries. This will check and filter the SQL statement before it reaches or executes in the database. But Hackazon appears to already have implemented prepared SQL statements. On top of the prepared statements, we added a black list function to remove special characters from the $query string. We added a conditional statement before the end of the query function. This conditional statement will remove any (#) or (‘) remaining in the string.

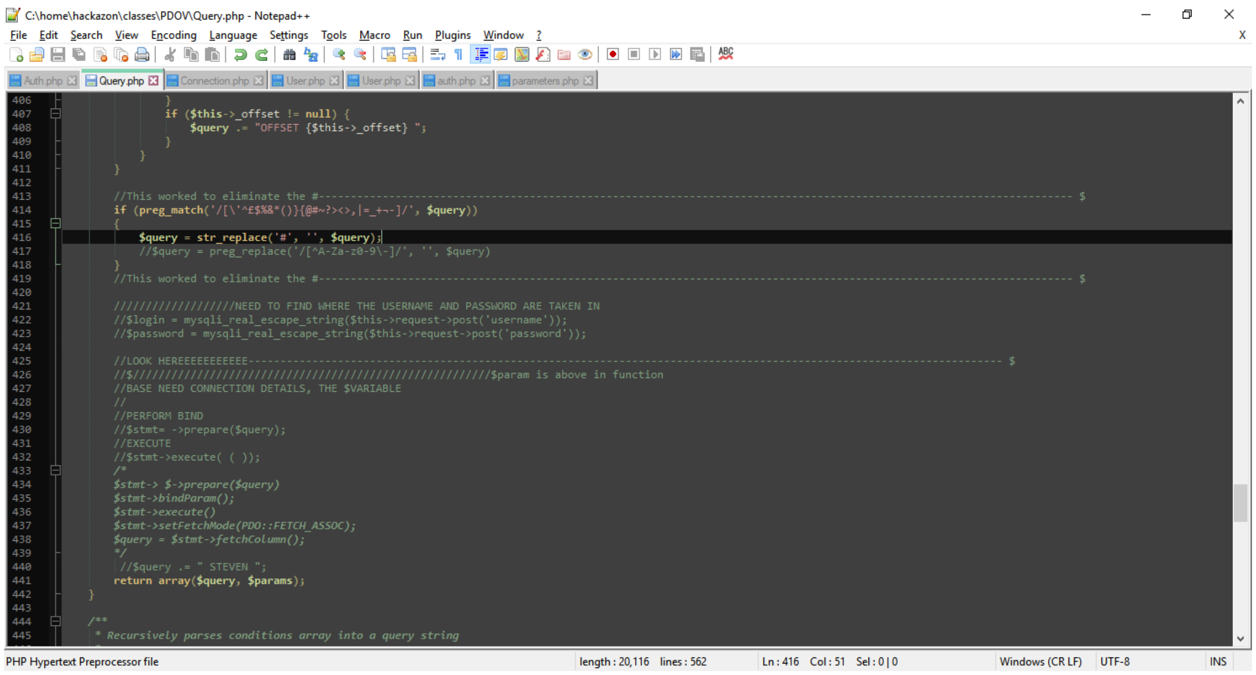
1. You will need to open the **Query.php** file in C:\home\hackazon\classes\PDOV\Query.php

2. Scroll to the end of the function named *query* and above the function *get_condition_query*

3. Add the following code just prior to the *return array($query, $params);*

```
if (preg_match('/[\'^£$%&*()}{@#~?><>,|=_+¬-]/', $query))
```

```
{       $query = str_replace('#', '', $query);     $query = preg_replace("/'/", "", $query);     }
```



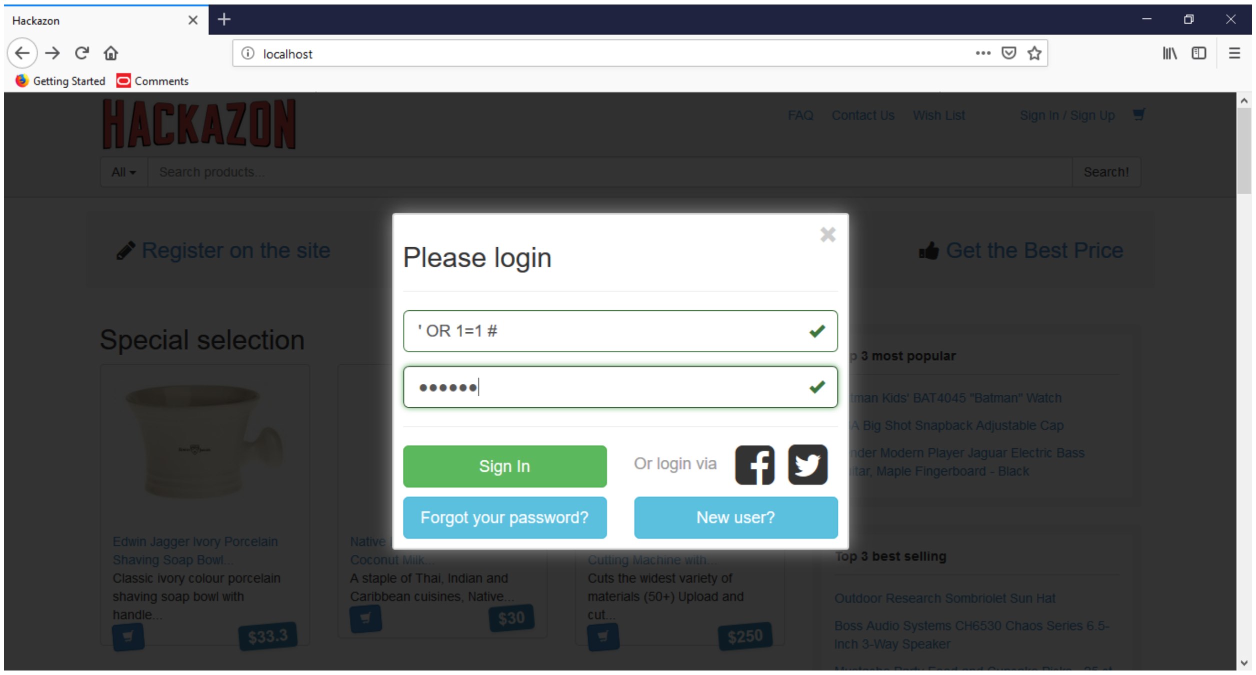
C:\home\hackazon\classes\PDOV\Query.php - Notepad++

```
            }
            if ($this->_offset != null) {
                $query .= "OFFSET {$this->_offset} ";
            }
        }
    }

    //This worked to eliminate the #------------------------------------------------------------------------------------------------------------------ $
    if (preg_match('/[\'^£$%&*()}{@#~?><>,|=_+¬-]/', $query))
    {
        $query = str_replace('#', '', $query);
        //$query = preg_replace('/[^A-Za-z0-9\-]/', '', $query)
    }
    //This worked to eliminate the #------------------------------------------------------------------------------------------------------------------ $

    ///////////////////NEED TO FIND WHERE THE USERNAME AND PASSWORD ARE TAKEN IN
    //$login = mysqli_real_escape_string($this->request->post('username'));
    //$password = mysqli_real_escape_string($this->request->post('password'));

    //LOOK HEREEEEEEEEEEEE------------------------------------------------------------------------------------------------------------ $
    //$///////////////////////////////////////////////////////////$param is above in function
    //BASE NEED CONNECTION DETAILS, THE $VARIABLE
    //
    //PERFORM BIND
    //$stmt= ->prepare($query);
    //EXECUTE
    //$stmt->execute( ( ));
    /*
    $stmt-> $->prepare($query)
    $stmt->bindParam();
    $stmt->execute()
    $stmt->setFetchMode(PDO::FETCH_ASSOC);
    $query = $stmt->fetchColumn();
    */
     //$query .= " STEVEN ";
    return array($query, $params);
}

/**
 * Recursively parses conditions array into a query string
```

PHP Hypertext Preprocessor file   length : 20,116   lines : 562   Ln : 416   Col : 51   Sel : 0 | 0   Windows (CR LF)   UTF-8   INS

4. This code will look for any special characters in $query. The $query string is what is holding the SQL statement. The if statement will check the string for any special characters and it will essentially remove them and store the new filtered string back into $query.

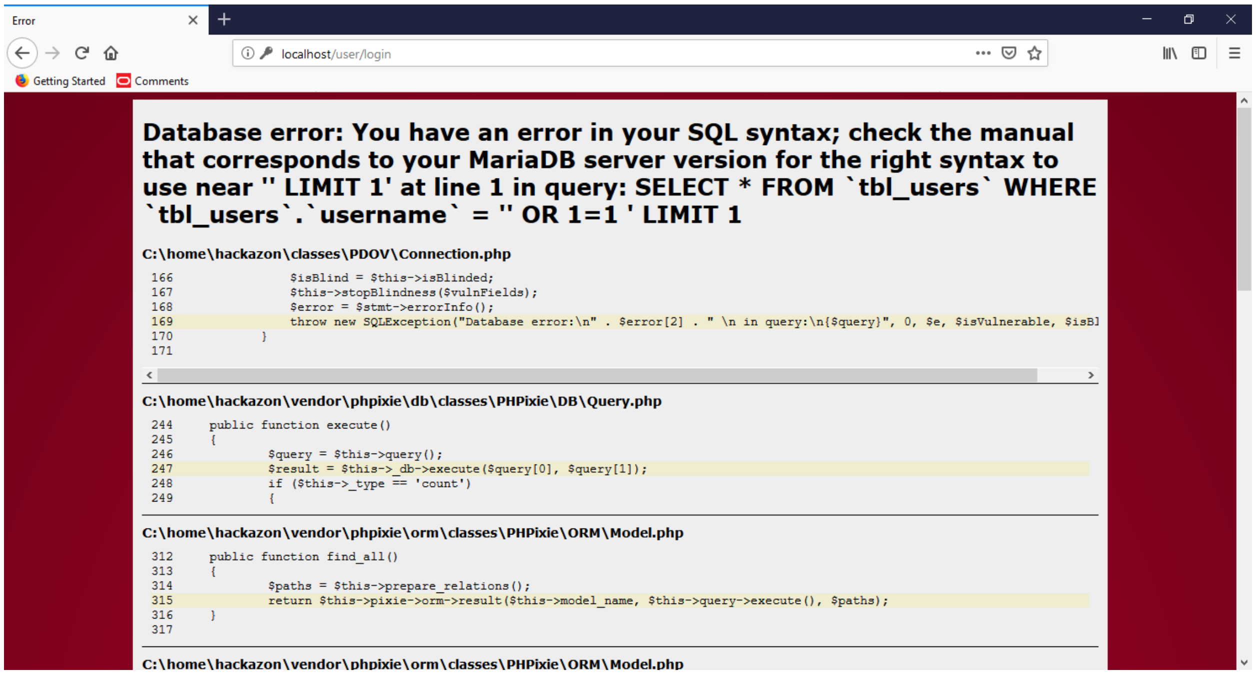
5. So now when you enter the same attack, an error will appear. To further note, if you look at the string submitted, you will notice that your “#” has been removed. %23 will not be accepted either.



Please login

' OR 1=1 #

••••••



Database error: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' LIMIT 1' at line 1 in query: SELECT * FROM `tbl_users` WHERE `tbl_users`.`username` = '' OR 1=1 ' LIMIT 1

C:\home\hackazon\classes\PDOV\Connection.php

```
                $isBlind = $this->isBlinded;
                $this->stopBlindness($vulnFields);
                $error = $stmt->errorInfo();
                throw new SQLException("Database error:\n" . $error[2] . " \n in query:\n{$query}", 0, $e, $isVulnerable, $isBl
            }

```

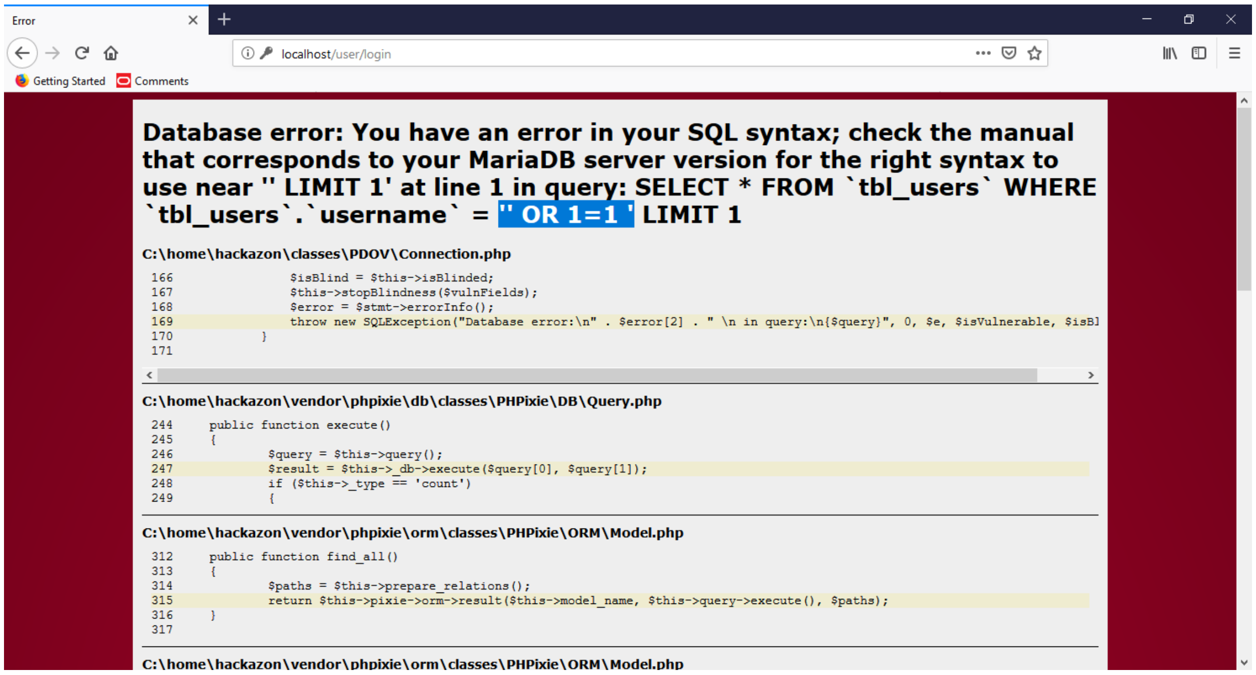
C:\home\hackazon\vendor\phpixie\db\classes\PHPixie\DB\Query.php

```
    public function execute()
    {
            $query = $this->query();
            $result = $this->_db->execute($query[0], $query[1]);
            if ($this->_type == 'count')
            {
```

C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php

```
    public function find_all()
    {
            $paths = $this->prepare_relations();
            return $this->pixie->orm->result($this->model_name, $this->query->execute(), $paths);
    }

```

C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php



```
Database error: You have an error in your SQL syntax; check the manual
that corresponds to your MariaDB server version for the right syntax to
use near '' LIMIT 1' at line 1 in query: SELECT * FROM `tbl_users` WHERE
`tbl_users`.`username` = '' OR 1=1 ' LIMIT 1

C:\home\hackazon\classes\PDOV\Connection.php

                $isBlind = $this->isBlinded;
                $this->stopBlindness($vulnFields);
                $error = $stmt->errorInfo();
                throw new SQLException("Database error:\n" . $error[2] . " \n in query:\n{$query}", 0, $e, $isVulnerable, $isBl
            }


C:\home\hackazon\vendor\phpixie\db\classes\PHPixie\DB\Query.php

    public function execute()
    {
            $query = $this->query();
            $result = $this->_db->execute($query[0], $query[1]);
            if ($this->_type == 'count')
            {

C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php

    public function find_all()
    {
            $paths = $this->prepare_relations();
            return $this->pixie->orm->result($this->model_name, $this->query->execute(), $paths);
    }


C:\home\hackazon\vendor\phpixie\orm\classes\PHPixie\ORM\Model.php
```

How to Prevent against SQL Injection Attacks

An organization can adopt the following policy to protect itself against SQL Injection attacks.

- **User input should never be trusted -** It must always be sanitized before it is used in dynamic SQL statements.
- **Stored procedures –** these can encapsulate the SQL statements and treat all input as parameters.
- **Prepared statements –**prepared statements to work by creating the SQL statement first then treating all submitted user data as parameters. This has no effect on the syntax of the SQL statement.
- **Regular expressions –**these can be used to detect potential harmful code and remove it before executing the SQL statements.

References:

https://www.w3schools.com/sql/sql_injection.asp

https://www.rapid7.com/docs/download/AppSpider_Hackazon_User_Guide.pdf

https://stackoverflow.com/questions/3938021/how-to-check-for-special-characters-php

https://www.sitepoint.com/how-to-protect-your-website-against-sql-injection-attacks/

https://www.veracode.com/security/sql-injection

https://www.guru99.com/learn-sql-injection-with-practical-example.html

https://www.lionblogger.com/ways-to-prevent-sql-injection-attacks-in-php/