SEcuring Web Applications

Jordon Coady - 20096529

SECURE PROGRAMMING & SCRIPTING

Contents

[Part 1 – Processing HTML Form Data with PHP and HTTP Requests 2](#_Toc132565456)

[1.A – Form Processing Using POST 2](#_Toc132565457)

[1.B – Form Processing Using GET 2](#_Toc132565458)

[1.C – Cross Site Scripting Attack Demo 3](#_Toc132565459)

[Part 2 – Databases Query & SQL Injection 3](#_Toc132565460)

[2.A – Database Query 3](#_Toc132565461)

[2.B – SQLi Demo 3](#_Toc132565462)

[Part 3 - HTML Forms Using Basic Protections 4](#_Toc132565463)

[Part 4 - PHP validation 5](#_Toc132565464)

[Part 5 – Cookies 6](#_Toc132565465)

[Part 6 – sessions 7](#_Toc132565466)

[Hijacking A Session 7](#_Toc132565467)

# Part 1 – Processing HTML Form Data with PHP and HTTP Requests

## 1.A – Form Processing Using POST

Relevant Files – 1aPost.php

The code in this file includes an HTML form for users to input their name, email address, and a message. When the user submits the form, the PHP script at the top of the file is called using the action attribute in the form. When the form data is submitted the action element is set to **<?php $\_SERVER["PHP\_SELF"];?>">** when means that the from data will be submitted back to the same page. **$\_SERVER["PHP\_SELF"]** is a supergobal variable.

Once the user submits the form, the PHP script at the top of the page checks if the request method is POST which is set using the method attribute in the form, using **$\_SERVER["REQUEST\_METHOD"] == "POST"**. If the condition is true, the script assigns the values from the form inputs to the $name, $email, and $message variables, and then echoes a personalized message back to the user using the $name variable.

I have left this script vulnerable to Cross Site Scripting (XSS) attacks. The reason this script is vulnerable is because it does not properly sanitize user input before displaying it back to the user in the $name variable. This means that an attacker could inject malicious code into the input field, such as a script that steals the user's session cookie or redirects them to a phishing site. To demonstrate this vulnerability, I have included a button that injects a simple XSS attack script into the name field when clicked.

## 1.B – Form Processing Using GET

Relevant Files – 1bGet.php

One of the differences between using a HTTP POST request and a HTTP GET request is the way that the form data is submitted to the server. With POST, the data is sent in the request body, while with GET the data is sent in the URL query string. This means that with GET, the form data can be seen in the URL, which can be a problem if sensitive information is being submitted. Another difference is that HTTP POST requests are typically more difficult to tamper with than HTTP GET requests. With GET, it is easier for an attacker to manipulate the parameters in the URL to try and exploit vulnerabilities. In contrast with POST, the data is not visible in the URL, making it harder to manipulate.

Using HTTP POST is better and more secure than using HTTP GET, because the form data is not visible in the URL and is therefore less susceptible to certain types of attacks like Cross-Site Request Forgery (CSRF).

## 1.C – Cross Site Scripting Attack Demo

Relevant Files – 1aPost.php

When the Demo XSS button is pressed the name field is filled with **<script>alert('XSS Attack!');</script>** and because thename field is echoed back to the screen when the user submits the form the script will be executed on the browser. To prevent this type of attack, the name field should be properly sanitized and validated, PHP's htmlspecialchars() function could be used to convert special characters thus preventing the browsing from executing the script.

# Part 2 – Databases Query & SQL Injection

## 2.A – Database Query

Relevant file – 2aSignInForm.php

In the relevant file when a user submits their username and password using the form the PHP script checks if the HTTP request method is POST. If it is the server name, username, and password are set to variables which will be used to access my database. It then creates a new PDO object using the above variables and sets its attributes for error handling. If the connection is successful, the submitted username and password is retrieved from the form using the $\_POST global variable.

The code then forms a database query using the retrieved username and password to select all the rows in the "2a\_userSignIn" table where the username and password match the submitted values. It then executes the query using the PDO query method and stores the result in the $result variable. Below is the query that is formed from the users submitted username and password.

**$query = "SELECT \* FROM 2a\_userSignIn WHERE username='$username' AND password='$password'";**

The SQL statement checks if both the username and password are found in table. If the query returns one or more rows, a foreach loop is used to iterate over the results and print the username, password, and a message indicating that the login was successful. If the query returns no results, it prints a message indicating that the submitted username or password was incorrect. The connection is then closed.

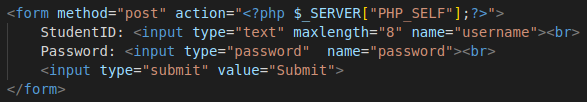
## 2.B – SQLi Demo

Relevant file – 2aSignInForm.php

In the relevant file there are two buttons that demonstrates two different methods of SQL injection. The first uses **' or '' ='**. The resulting SQL query will look like this **Query: SELECT \* FROM 2a\_userSignIn WHERE username='' or '' ='' AND password='' or '' =''**. This query will return all rows from the 2a\_userSignIn table because **' or '' ='** will always be true. Because the above query needs both the username and password to match **' or '' ='** is entered into both form fields.

The next method of SQLi used is **' or '1' = '1**. When both form fields have **' or '1' = '1** entered into them the SQL query looks like this **Query: SELECT \* FROM 2a\_userSignIn WHERE username='' or '1' = '1' AND password='' or '1' = '1'**. This SQLi functions the same as the pervious SQLi in terms of 1=1 will always equate to true so when the database is queried using the above statement every row is returned.

# Part 3 - HTML Forms Using Basic Protections



1. "maxlength" attribute: This attribute is used to restrict the maximum length of the input text in the "StudentID" field. In this case, the maximum length is set to 8 characters.
2. "password" type: This attribute is used to hide the characters entered in the "Password" field.

These restrictions offer no real protection against attacks. For example, an attacker can still perform a brute-force attack to guess the correct StudentID and Password combinations. Also, an attacker can use various attack techniques, such as cross-site scripting (XSS), SQL injection, or phishing attacks, to steal the credentials entered in the form fields.

Another method an attacker could use is to intercept the HTTP request before it gets to the server and bypass the HTML form restrictions. I will demonstrate this using burp suite. I entered my student ID and a password which is password. Then the I intercepted the HTTP request in burp suite.

Table

Description automatically generated with low confidence

At line 22 we can see the form values submitted by the user and the limit of 8 characters on the student ID field won’t apply here which could result in an attacker injecting malicious into that field. So, the form protections don’t offer any real protections for web applications.

Here I changed the value of studentID to **<script>alert('XSS Attack!');</script>** and when the altered HTTP request is sent, the alert pops up on the target browser.

Graphical user interface, text, application, email

Description automatically generated

# Part 4 - PHP validation

Relevant file – 4PHPValidation.php

Validating and sanitizing form inputs are critical steps in ensuring the security and integrity of any web application. Without proper validation, attackers can exploit vulnerabilities in the application by submitting malicious inputs that can lead to various security risks, such as SQL injection and cross-site scripting (XSS) which was demonstrated in pervious sections of this report. Such attacks can result in data breaches, unauthorized access, and other security threats that can cause significant damage to a website or web application. Below is a list of PHP functions I used to validate and sanitise data submitted through a form.

1. Empty(): This function returns a Boolean value depending on whether the variable is empty or not. If the variable is empty, the function returns true otherwise it returns false. I used the empty() function in conjunction with every other PHP validation function to ensure the user filled in every field, if the field is empty the user will be told to enter the required information into the filed.
2. Is\_numeric(): This function checks whether a given variable is numeric or not. The function returns true if the variable is numeric, including integers, floats, and numeric strings; otherwise, it returns false. I used this function to ensure the user submitted a student ID that only contains numbers.
3. Filter\_var(): This function takes two arguments: the first is the variable to be filtered, and the second is the filter to be applied. The filter defines the validation or sanitization behaviour. Some of the available filters include FILTER\_VALIDATE\_EMAIL, FILTER\_SANITIZE\_STRING, FILTER\_VALIDATE\_INT and FILTER\_SANITIZE\_URL. I used the filter\_var() function to validate the email address provided in the **$\_POST["email"]** variable. The FILTER\_VALIDATE\_EMAIL filter is applied to this variable, ensuring that it is a valid email address. If the email address is not valid, the script outputs an error message, prompting the user to enter a valid email address.
4. Preg\_match(): This function takes three arguments: the regular expression pattern to match, the string to match against, and an optional array variable to store the matched groups which I didn’t use. The function returns 1 if the string matches the pattern and 0 if it does not match. I used the preg\_match() function to validate the date of birth provided in the **$\_POST['dob']** variable. The regular expression pattern /^[0-9]{4}-[0-1][0-9]-[0-3][0-9]$/ is used to match the date format "YYYY-MM-DD". If the string matches the pattern, the script outputs a message indicating that it is a valid date. If the string does not match the pattern, the script outputs an error message, prompting the user to enter a valid date.
5. Strip\_tags(): This function takes in two arguments: the first is the string to be sanitized, and the second is an optional parameter that allows certain HTML tags and attributes to be preserved. By default, the function removes all HTML tags and attributes from the string. I used the strip\_tags() function is used to sanitize the **$\_POST['tags']** variable by removing any HTML tags that may have been entered by the user. If the variable is not empty, the function removes any HTML tags from the string and outputs the sanitized string. If the variable is empty, the script prompts the user to enter HTML tags.

# Part 5 – Cookies

Relevant file – 5cookie.php

When creating a cookie in PHP, there are several security method parameters that can be set to help reduce attacks involving cookies. Below is a list of some of the options available when creating a cookie.

1. secure: If this parameter is set to true, the cookie can only be sent over HTTPS. This can help prevent man-in-the-middle attacks.
2. httponly: If this parameter is set to true, the cookie can only be accessed through HTTP or HTTPS requests and cannot be accessed through client-side scripts. This can help prevent cross-site scripting attacks.
3. samesite: This parameter can be set to Strict, Lax, or None. It determines whether the cookie can be sent in cross-site requests. Strict prevents the cookie from being sent in any cross-site requests, while Lax allows the cookie to be sent in some cross-site requests (e.g., when the user clicks a link), but not others (e.g., when the request is initiated by a script). None allows the cookie to be sent in all cross-site requests but requires that the secure parameter is also set to true.
4. expires: This parameter sets the expiration date of the cookie as a UNIX timestamp. If set to a specific timestamp (e.g., 1647548400), the cookie will expire at that time. This can also help prevent cookie theft by limiting the amount of time that the cookie can be used.
5. domain: This parameter specifies the domain for which the cookie is valid. If set to a specific domain (e.g., example.com), the cookie will only be sent to that domain and its subdomains. This can help prevent cookie theft by limiting the scope of the cookie.

# Part 6 – sessions

Relevant files – 6SessionVar.php

In the relevant file the php script starts by calling **session\_start();** which will start a new session or will resume an existing one. Then this if statement **if(isset($\_SESSION["name"]) && isset($\_SESSION["pass"]))** is used to check if the session variables "name" and "pass" are set. If they are set, it means that the user has already logged in and the code redirects them to the login screen by calling **header("Location: loginScreen.php");** and then exits.

If the session variables are not set, it means the user is not yet logged in. Once the user submits the form with the correct login details the code stores the values in the session variables "name" and "pass" by calling **$\_SESSION["name"] = "bob"; and $\_SESSION["pass"] = "pass"**. After storing the session variables, the code redirects the user to the login screen by calling **header("Location: loginScreen.php");** and exits.

## Hijacking A Session

The first step in successfully hijacking a session is to obtain the session ID. This can be achieved through various methods, such as by exploiting a cross-site scripting vulnerability using a script like this:

**<script>**

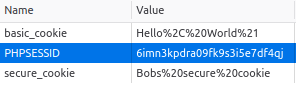
**VarSessionID= document.cookie.replace(/(?:(?:^|.\*;\s\*)PHPSESSID\s\*\=\s\*([^;]\*).\*$)|^.\*$/, "$1");**

**alert("Session ID: " + sessionID);**

**</script>**

When this script is injected into a form field, it retrieves the session ID and displays it in an alert box. An attacker would ideally inject this script into a website and have it save to a database. Then, whenever another user browses to the infected page, the script will be executed on their client-side browser. Instead of displaying the session ID in an alert box, the script could be modified to send the session ID to the attacker. In this way, the attacker can obtain the session ID and use it to hijack the user's session.

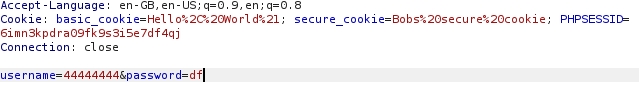
I will now demonstrate how session hijack using burp suite. Below is the session ID of a user who has successfully logged into the page relating to the file 6sessionVar.php.

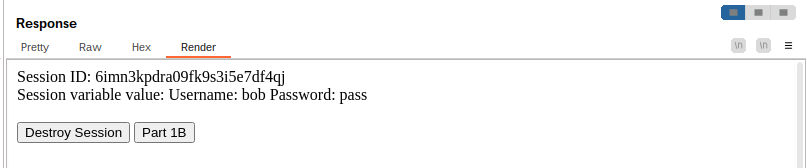
Graphical user interface, text, application

Description automatically generated



Above is an image of HTTP request which shows the attacker trying to sign in with the wrong username and password. I will now change the PHPSESSID variable to the session ID of the person who successful logged in. This can be seen below.



Now that the session ID has been changed to match the user who successfully signed in, I can now send this HTTP request a follow the redirection and render the response in a different browser and you can see we have picked up where the previous user has left off.