Project 4:

Web Security Report Entry

Spring 2022

Task 1 – Warm Up Exercises

Activity 1 - The Inspector & Console tabs

1. What is the value of the 'CanYouSeeMe' input? *Do not include quotes in your answer.*

   A_Value_Between_One_And_Ten

2. The page references a single JavaScript file in a script tag. Name this file including the file extension. *Do not include the path, just the file and extension. Ex: "ajavascriptfile.js".*

   project4.js

3. The script file has a JavaScript function named 'runme'. Use the console to execute this function. What is the output that shows up in the console?

   There are 42 bugs on the wall.

Activity 2 - Network Tab

1. What request method (http verb) was used in the request to the server?

   POST

2. What status code did the server return?. *Ex: "200"*

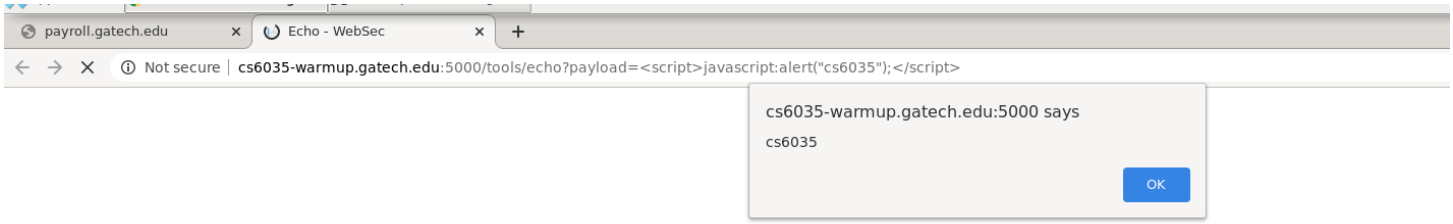
   499

3. The server returned a cookie named 'Samy' for the browser to store. What is the value of this cookie? *Do not include quotes in your answer.*

   but_most_of_all_samy_is_my_hero

Activity 3 - Built-in browser protections

1. You can do more than just echo back text. Construct a URL such that a JavaScript alert dialog appears with the text cs6035 on the screen. Upload **activity3.html** and paste in a screenshot of the page with the dialog as your answer below. Be sure to include the URL of the browser in your screenshot.



Activity 4 - Submitting forms

1. Copy and paste below the entire output message you see and submit that as your answer to this activity. Upload **activity4.html** which is the form that you constructed.

   Congratulations!, you've successfully finished this activity. The answer is Stuxnet (2010)

Activity 5 - Accessing the DOM with JavaScript

1. Upload **activity5.html** which is the form that you constructed. No other answers are required for this activity.

Task 5 – Epilogue Questions

Target 1 -- Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.

   For this attack, the *account.php* page needs to be changed to fix the XSRF vulnerability. The code block that is causing this vulnerability are lines 27-29.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of XSRF is not what we're looking for.

   One of the main vulnerabilities with CSRF is having a lack of identity protection when a user submits a form. This can be in the form of a session cookie, IP address, username/password combination, etc. Within the *account.php* code, there is a protection check on lines 27-29, that confirms if the *response* value input equals the "expected" value calculated from the *account*, *routing*, and *challenge (which is a generated csrf_token)* inputs submitted onto the server (lines 24-26). If this check fails, a detailed CSRF prevention message is displayed. Typically, the *response* value is calculated based on the user's existing *account* and *routing* numbers via a hash script ran on page load. Funny enough

though, the vulnerability comes in to play when the CSRF prevention method is actually triggered. When the error message is displayed, both the current calculated *response* and the expected *response* values are displayed. All the attacker needs to do is submit a form with an empty *response* input (triggering the error message), grab the expected value from the error message, and update the malicious *response* input value accordingly. Once this is done, the prevention check is satisfied, and the attack is successful.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!

There are a couple different ways to fix this code to defend against CSRF attacks. The first thing that needs to be done is making the CSRF prevention message that's displayed a lot vaguer. There's no need to include both the current and expected *response* values in the error message for a potential attacker to see. Another way would be to implement a SameSite cookie check within the code. By having this session cookie attribute set to *Strict*, the browser will not include the set cookie in any requests that originate from another site. This means, the cookie will only be set if the user goes from the *index.php* page to the *account.php* page. However, if the attacker goes from their malicious site to the *account.php* page, then the cookie does not get set, hence preventing the CSRF attack.

```
Set-Cookie: SessionId=sYMnfCUrAlmqVVZn9dqevxyFpKZt30NN; SameSite=Strict;
```

Target 2 Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.

   For this attack, the *index.php* page needs to be changed to fix the XSS vulnerability. The main line that causes this vulnerability is 34.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of XSS is not what we're looking for.

   By looking at the form input values within the *index.php* code, the *login* input has a default value of *<?php echo @$_POST['login'] ?>*. This means that the field automatically gets set by the *login* value that's submitted onto the server. By using a reflected XSS attack, the attacker just needs to submit their malicious script injection as the *login* input value from their page to the server. This script is then displayed in the *login* field on the *index.php* page because of the echo command mentioned above. Due to how the script injection is formatted, the attack is able to appear hidden when looking at the page, but in reality, the malicious script is displayed right beneath line 34.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
   a. Be careful with your explanation here. There are wrong ways to fix this vulnerability. Hint: Never write your own crypto

algorithms. This concept extends to XSS sanitization.

b. Warning: Removing site functionality will not be accepted here.

One of the best ways that can prevent an XSS attack without removing site functionality would be to add an encode sanitizing function for the echo command in line 34. This can be achieved by validating each character passed from *POST['login']* that is a known HTML element such as: <, >, /, etc. There's also a built-in PHP function that will encode htmlentities to their UTF-8 counterpart.

```
<?php echo htmlentities($input, ENT_QUOTES, 'UTF-8');?>
```

The reason this sanitization approach works is because when the input is encoded, the input value is unable to be escapable by formatting the malicious injection a specific way. Since the input value would now be an encoded string, the input value cannot be escaped, thus the script will not be injected successfully. Even if the attack is sent encoded, this will also fail since the webpage would not automictically decode it.

Target 3 Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.
   For this attack, the *auth.php* page needs to be changed to fix the SQLi vulnerability. The lines that are causing this vulnerability are 30-53.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of SQL Injection is not what we're looking

for.

Within the *auth.php*, there is already a filter in place that removes popular SQLi commands like: AND, OR, DROP, etc, from the *username* input. The filter does a relatively good job covering all possible keywords needed to achieve an SQLi attack, but it's not perfect. The main issue with this is that the *str_replace* is case sensitive, while SQL commands are not. So even though two of the filters can catch the "OR" and "or" keywords, if an attacker uses "Or" in their script instead, the input will both beat the filter and inject successfully.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
    a. Be careful with your explanation here. There are wrong ways to fix this vulnerability. Hint: Never write your own crypto algorithms. This concept extends to SQL sanitization.

Since there is already an SQLi function in place, the main way to fix this vulnerability would be to further strengthen this function. This can be done by upgrading to *str_ireplace* to check for keywords in a case insensitive manner. This would also reduce the lines of code needed to cover these potential SQLi injection keywords, making the function more efficient. Another way to fix this code would be to use a function instead that utilizes strongly type parameterized queries, like *bindParam()*. This is a built in PHP function that will prevent attackers from achieving string manipulation and trick the input from running an extra command, such as: `tom' or '1'='1`
Keep in mind though, this does impact performance.

Additional Targets

1. Describe any two additional issues (they need not be code issues) that create security holes in the site.

   The other two additional security issues that I noticed was that the payroll site is HTTP protocol as opposed to HTTPS, and overly detailed error handling within the webpage.

2. Provide an explanation of how to safely fix the identified issues. Feel free to include snippets and examples. Be detailed!

   One of the first things I noticed when working through each target was that the payroll site is under a HTTP protocol instead of HTTPS. HTTP is used for transferring data over a network in the form of request and response methods. This protocol is responsible for various communications (*POST*, *GET*, *etc.*) between the webpage and API. These messages are sent over the internet in plaintext, where anyone is able to see PII data such as: passwords, banking information, and any other sensitive form data. This is where HTTPS comes into play. The HTTPS protocol uses SSL encryption to secure HTTP requests and responses. So instead of plaintext data, an attacker trying to spy on transferred data would see the same information, but with random characters, making it hard to decipher. By making the simple switch over to a HTTPS protocol, several security holes with the payroll webpage will be plugged.

   Another huge security hole I noticed was that the error handling messages displayed throughout the webpage are way too detailed.

From telling the user if the username or password they've entered is invalid (depending on which input is incorrect), to displaying the expected *response* value from a CSRF attack. The more detail that an error message has, the easier it is for an attacker to obtain much needed information to perform an attack. In some cases, attackers will purposely send a payload they know will throw an error, just to see what the error handler returns. By making the error message vaguer such as: *"Invalid credentials"* or *"CSRF attempt prevented, cannot change information at this time"* makes it increasingly harder for attackers to gather information.

Works Cited

1. Arampatzis, A. (2020, December 10). *What Are the Differences Between HTTP & HTTPS?* Venafi. https://www.venafi.com/blog/what-are-differences-between-http-https-0
2. CheatSheets Series Team. (2021). *SQL Injection Prevention - OWASP Cheat Sheet Series*. Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
3. PortSwigger Ltd. (2022). *Web Application Security, Testing, & Scanning - PortSwigger*. PortSwigger. https://portswigger.net/
4. SecurityScorecard. (2021, March 25). *41 Common Web Application Vulnerabilities Explained*. https://securityscorecard.com/blog/common-web-application-vulnerabilities-explained

5. W3Schools. (2022). *W3Schools Free Online Web Tutorials*. https://www.w3schools.com/

When complete, please save this form as a PDF and submit with your HTML files as "report.pdf". Do not zip up anything!