Learning goals: In this homework, you will learn (1) the basics of web app security and privacy, (2) how to attack a web app by exploiting commonly occurring vulnerabilities, and (3) how to avoid and mitigate those vulnerabilities. You will also learn the basics of building a web app with the classic LAMP stack and deploying it. This homework is also intended to provide you with ideas for your projects.

Problem 1.

a.In general, should validation of a user's input (e.g., submitting form content) be based on allowlisting or denylisting?

Allowlisting: This is because the approach defines a list of acceptable inputs and rejects everything else. Allowlisting is more secure because it only permits known safe inputs and blocks any unexpected or potentially malicious data. Denylisting (blacklisting), on the other hand, tries to block known bad inputs, but it can be bypassed if the attacker uses an unknown or new malicious input.

(b) What effect does it have if you set HTTP cookies in your web app with the Secure attribute?

Setting HTTP cookies with the Secure attribute ensures that the cookies are only sent over secure connections, such as HTTPS. This means the cookie will not be transmitted over an unencrypted HTTP connection, reducing the risk of the cookie being exposed to attackers through man-in-the-middle (MITM) attacks or eavesdropping. It helps protect sensitive information, such as session identifiers, from being intercepted when data is transmitted over the network

(c) Can you use one or more of the following to mitigate clickjacking? Please briefly explain. • HTTPS Connection • X-Frame-Options HTTP Header • Content-Security-Policy HTTP Header • Secure Cookie Attribute • All • None

X-Frame-Options HTTP Header: This header helps prevent clickjacking by controlling whether a browser is allowed to render a page in an <iframe>. For example, using X-Frame-Options: DENY or X-Frame-Options: SAMEORIGIN ensures that your site cannot be embedded in an iframe by external sites, preventing attackers from overlaying deceptive content

Content-Security-Policy HTTP Header: You can use the frame-ancestors directive in the Content Security Policy (CSP) header to specify which sources are allowed to embed your content using <iframe>, <embed>, or <object> tags. For example, Content-Security-Policy: frame-ancestors 'self' will only allow your content to be framed by pages from your domain, mitigating clickjacking.

(d) What are the two fundamental mechanisms to defend against cross-site scripting attacks?

-Input Sanitization and Validation: This mechanism involves cleaning and validating all user inputs to ensure they do not contain any potentially malicious scripts.

-Output Encoding/Escaping: This mechanism involves encoding or escaping data before it is rendered in the browser. For example, when displaying user-generated content, all special characters (e.g., <, >, &, " etc.) should be encoded so that they are not interpreted as HTML or JavaScript. This ensures that any malicious input is displayed as text rather than being executed as code.

e.True or false? Please explain. Escaping all HTML-specific characters, e.g., & , will prevent all cross-site scripting attacks.

-It is helpful but it will not prevent all cross-site scripting attacks due to:
Context-Specific Escaping Required: XSS attacks can occur in different contexts, such as HTML, JavaScript, URLs, and CSS. Escaping for HTML-specific characters might prevent XSS in HTML contexts but not in JavaScript or CSS contexts. Each context requires its own form of escaping. For example, JavaScript-specific characters like single quotes (') and double quotes (") may also need to be escaped in script contexts. Even with HTML escaping, an attacker might still inject malicious code using event handlers (e.g., onclick="alert(1)") or other attributes if the application is not properly validating and sanitizing inputs. Advanced attackers might find ways to bypass escaping using encoding tricks, nested scripts, or other browser quirks that standard HTML escaping might not account for.

f.You would like to monetize your web app and integrate an ad network in your front-end code. The ad network code loads JavaScript code from the ad network server into an iframe on your website. Is it possible that cross-site scripting attacks can occur due to the ad network integration on your site?

Yes, it is possible that cross-site scripting (XSS) attacks can occur due to the ad network integration on the site. This is because the ads displayed in the <iframe> could contain malicious scripts that try to interact with your website or your users' browsers. This includes untrusted Content(Ad networks often serve dynamic and third-party content. If the ad network does not properly vet or sanitize the ads, an attacker could inject malicious scripts into the ads, which could be displayed in your site's <iframe>).Malicious Ads (Malvertising)(Attackers can exploit ad networks by injecting "malvertising" (malicious advertising) into the ads that run on your site. If the <iframe> is not properly sandboxed or the content security policies are not restrictive, these malicious ads can cause harm, such as redirecting users, stealing cookies, or displaying harmful content.). Cross-Frame Scripting Attacks(Although JavaScript in one domain cannot usually access content from another domain due to the Same-Origin Policy, an attacker might exploit vulnerabilities to bypass these restrictions, such as using postMessage to communicate between the iframe and the parent page.)

g.Would it help to avoid cross-site scripting attacks if a user sets their browser such that it would (i) prohibit third parties from running JavaScript or (ii) turn off JavaScript completely? For each, why or why not?

(i) Prohibiting Third Parties from Running JavaScript:

Yes, this would help mitigate cross-site scripting (XSS) attacks from third-party sources. By preventing third-party scripts from running, the browser blocks any scripts that come from domains other than the primary site's domain. This means that if an attacker tries to inject malicious JavaScript from a third-party domain (e.g., through an ad network or a third-party widget), those scripts would not execute, reducing the likelihood of an XSS attack. However, this protection only addresses third-party scripts. If an attacker manages to inject malicious JavaScript directly within the same domain (e.g., using a form input that's vulnerable to XSS), prohibiting third-party JavaScript alone will not stop the attack.

(ii) Turning Off JavaScript Completely:

Yes, turning off JavaScript completely would prevent all JavaScript-based XSS attacks, because without JavaScript, no script can execute—malicious or otherwise. This means that any injected script, regardless of its origin, will not run, rendering XSS attacks ineffective. The possible downside with this is it breaks JavaScript functionality as many websites rely heavily on JavaScript for essential features and this might not be the most practical solution

h. To keep a user authenticated and memorize their state in a web app, apps assign users a session ID in an HTTP cookie or a token that is assigned at session creation time. It is exchanged between the user and the web app for the duration of the session and sent on every HTTP request. The session ID is a name=value pair. When is it necessary to renew a session ID? Choose one and briefly explain. • At the expiration of any fixed period in time (e.g., every 60 seconds) • At any change of access privileges • When a user logs out

At any change of access privileges: It is necessary to renew a session ID at any change of access privileges to prevent session fixation attacks. When a user's access privileges change (e.g., after logging in, changing roles, or gaining admin access), renewing the session ID ensures that an attacker cannot use a previously valid session ID to gain unauthorized access to elevated privileges. Renewing the session ID in such scenarios helps maintain security by creating a new session with the updated permissions and invalidating any existing session that might have been hijacked.

i)Your web app is using a self-signed certificate. Under which circumstance is the communication between your server and the clients, i.e., web browsers, secure?

The communication between your server and the clients (i.e., web browsers) using a self-signed certificate is secure only if the clients trust the self-signed certificate. This means that the client (browser) has explicitly accepted the self-signed certificate: This usually requires the user to manually add the certificate to their browser's list of trusted certificates. If the certificate is not trusted, the browser will show a security warning indicating that the connection is not secure, and the data exchanged may be at risk. Because self-signed certificates are not verified by a trusted Certificate Authority, an attacker could potentially present a different self-signed certificate, intercept the communication, and decrypt or modify the data. Therefore, using a self-signed certificate is secure only if the client can confirm that the certificate truly belongs to the server they are trying to reach (e.g., by manually verifying the certificate's fingerprint).

j.Your web app is using HTTPS. How can you make sure a client will not accidentally request a page over a non-secure HTTP connection? Choose one and briefly explain.

• Requests for port 80 should all be redirected to port 443 • Close port 80 completely • Use the HTTP Strict-Transport-Security Header

The best way to ensure that a client will not accidentally request a page over a non-secure HTTP connection is to use the HTTP Strict-Transport-Security header. The HSTS header tells browsers to only communicate with the server over HTTPS and never to use HTTP, even if the user tries to access the site via an HTTP URL or clicks on a non-secure link. Once the HSTS policy is set and recognized by the browser, any attempt to access the site over HTTP will automatically be redirected to HTTPS by the browser without ever touching the server. This ensures a secure connection and eliminates the risk of a downgrade attack where an attacker tries to force the connection to use HTTP instead of HTTPS.

(a) OnePhoto is vulnerable to session hijacking. If you are able to spoof the OnePhoto HTTP cookie for another user, you can hijack that user's session. As it happens, you were recently at the airport to intercept network traffic from the unsecured network. You observed that the following cookie value was issued for a user while they were logging into OnePhoto: 17206828aaf634d81578ef30db7198a1 You also observe that the user logs in on the following days: - September 19, 2024, 10 to 11 am ET - September 20, 2024, 11 am ET to noon - September 23 to 27, 2024, 11 am ET to noon

Can you log into OnePhoto as that user? If you managed to do so, please explain how you did it, why your technique worked, and give the name of the user as whom you masqueraded. Please also give the name of the user that you have legitimately created. Hint: You need to find a way to edit cookies in your browser; maybe, a browser extension to "edit this cookie"? Important: Please do not log out of OnePhoto as that would log out all users with the above cookie. If you do accidentally log out, please contact me as soon as possible at szimmeck@wesleyan.edu. If you want to log out, change your cookie value beforehand.

Yes. I opened a different Chrome browser and logged into OnePhot using Username: try1. I used a cookie editor to inspect the phpsessid. I then opened the OnePhoto website again on a different device. I realized OnePhoto does not enforce HTTPS, making it vulnerable to session hijacking. I used a browser extension called **Cookie Editor** to modify my session cookie for the OnePhoto website and replaced it with one belonging to the 'try1' user. I did this by navigating to the OnePhoto site and opened the Cookie Editor. Initially, the extension did not have permission to access the cookies for the site, so I clicked **"This site"** to grant permission. After granting permission, I accessed the list of cookies associated with the OnePhoto site. I located the session cookie and replaced its value with the intercepted cookie value. I then refreshed the page and that is how i was able to log in.

(b) OnePhoto is vulnerable to SQL injection. Can you log in without a username and password? If you managed to do so, please explain how you did it and give the code that you used, if any. Hint: As a starting point, look at the SQL query that is executed in line 23 of login.php of OnePhoto.

Yes. The input fields for username and password are not sanitized in login.php, which allows for malicious SQL code to be injected directly into the query. I entered ' OR '1'='1 in the username field. Since ‘1’=’1 is always true, the query returns a valid result, bypassing the authentication check and allowing me to log in without a username or password. The resulting query was:

SELECT * FROM users_table WHERE username = '' OR '1'='1' AND password = '';

There was no need to write any code for this attack and I performed by manually entering the payload into the input fields on the login page.

(c) If you log into OnePhoto (as any user), you will find one photo. Where was this photo taken? Give the exact location and closest street address. Please explain how you arrived at your findings. What should the developer of the OnePhoto app do to avoid revealing photo locations? Hint: Learn about photo metadata.

The picture was taken at 52°22'12.41"N 9°44'48.56"E. Along Warmbuchenstrabe Street, Hanover, Germany.

I logged into OnePhoto and saved the photo displayed on the site. I used an online EXIF viewer to view the metadata of the downloaded image.I found that the metadata contained GPS coordinates in the fields labeled **GPS Latitude** and **GPS Longitude**. I entered these coordinates into Google Maps, which showed that the photo was taken

(d) When visiting http://comp360.lovestoblog.com/login.php, the browser navigation displays the warning “Not Secure” as shown below. Briefly explain what this warning means and what the owner of the app should do to make it more secure.

The “Not Secure” warning in the browser navigation bar indicates that the website is not using HTTPS to encrypt data transmitted between the client and server. Instead, it is using HTTP, which does not provide encryption, making data vulnerable to interception and attacks. The owner should obtain and install an SSL/TLS certificate from a trusted Certificate Authority (CA) and then configure the webserver to use HTTPS and redirect all HTTP traffic to HTTPS. They should also Enable HTTP Strict Transport Security (HSTS) by adding the header Strict-Transport-Security: max-age=31536000; includeSubDomains; preload. Lastly, they should update all internal links and resources to use HTTPS instead of HTTP

At this point, it is assumed that you deployed OnePhoto and created the SQL database. Now, mitigate the following vulnerabilities of OnePhoto. If you write new code, highlight in a comment where you revised the code with a *Revised* tag and explain in your comment how your revision mitigates the vulnerability.

To address the "Not Secure" warning, the site should use HTTPS. InfinityFree offers free SSL certificates that can be configured via the control panel. After obtaining and installing an SSL certificate, the following `.htaccess` rule should be added to force HTTPS

[Other solutions added as comments in code]

iii. To address the "Not Secure" warning, the site should use HTTPS. InfinityFree offers free SSL certificates that can be configured via the control panel. After obtaining and installing an SSL certificate, the following .htaccess rule should be added to force HTTPS:

```
# Force HTTPS for all requests

RewriteEngine On

RewriteCond %{HTTPS} off

RewriteRule ^(.*)$ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

Iv. The password is stored in plaintext format in the database, meaning anyone with access to the database can see the exact password for this user. Passwords should always be stored in a hashed format rather than plaintext. This means that even if an attacker gains access to the database, they will not be able to read the passwords without performing an expensive operation to crack the hashes.I used the PHP function password_hash() to securely hash passwords before storing them in the database