Week 2

Creating Engaging Classrooms

Facilitation:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, Word

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Tech Training 2:

Cookies:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface

Description automatically generated

* Stateless: The server doesnt know that uve had a connection
  + Server thinks your a new client every time

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

* Session management: we want retention so that the user doesn't have to be signing in every time the page refreshes or they open a different page on the domain
  + Cookies are stored in the browser (user end) not by the website. Browsers may "permanently" store your cookies
  + Cross-site cookies are vulnerable
  + server gives cookie id which is stored on the browser

Text

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, diagram

Description automatically generated

Text

Description automatically generated with low confidence

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application, Word

Description automatically generated

CSRF 2:

<form name="csrfForm" action="https://security.codepath.com/user/csrfchallengetwo/plusplus" method="POST">

<input type="text" name="userId" value="" />

<input type="submit"/>

</form>

<https://docs.github.com/en/pages>

<form name="csrfForm" action="https://security.codepath.com/user/csrfchallengetwo/plusplus" method="POST">

<input type="text" name="userId" value="9deb285472d34e4f0be9ab2b26c1075ca03e4f27" />

<input type="submit"/>

</form>

Notes:

* **Faked Requests:**
  + Requests to webservers can be faked.
  + nothing unique about a request; it is simply a string of formatted data.
  + An attacker can manufacture a request
  + This is also known as "request spoofing"
  + Faked requests can even duplicate forms and send cookie data
    - Forms can be duplicated and altered
    - Values in hidden input fields can be altered
    - Cookie data can be modified
    - Any client-side JavaScript validations can be bypassed
  + HTTP referer: An optional HTTP header field that identifies the address of the webpage (i.e., the URI or IRI), which is linked to the resource being requested. By checking the referrer, the new webpage can see where the request originated.
  + Uniform Resource Identifier (URI): a unique sequence of characters that identifies a logical or physical resource used by web technologies. URIs may be used to identify anything, including real-world objects, such as people and places, concepts, or information resources such as web pages and books. Some URIs provide a means of locating and retrieving information resources on a network (either on the Internet or on another private network, such as a computer filesystem or an Intranet), these are Uniform Resource Locators (URLs)
  + Internationalized Resource Identifier (IRI): an internet protocol standard which builds on the Uniform Resource Identifier (URI) protocol by greatly expanding the set of permitted characters.
  + User Agent: any software, acting on behalf of a user, which "retrieves, renders and facilitates end-user interaction with Web content." A user agent is therefore a special kind of software agent e.g. web browsers or email readers
  + The most commonly spoofed parts of a request are:
    - Referer
    - X-Forwarded-For
    - User-Agent
    - Cookie data
    - Form data
  + cURL: command line tool and library for transferring data with URLs. curl is used in command lines or scripts to transfer data. curl is also used in cars, television sets, routers, printers, audio equipment, mobile phones, tablets, settop boxes, media players and is the Internet transfer engine for thousands of software applications in over ten billion installations.
  + Protection from Faked Requests:
    - From the perspective of a server-side application, is not possible to prevent incoming faked requests.
    - not different enough from regular requests to make them easily identifiable.
    - The primary defense is simply to always assume these values from incoming requests are unreliable
    - One simple protection is to check the HTTP Referer in order to enforce same-domain form submissions.
    - Defense in Depth:
      * having redundant security measures in place e.g. backup parachute
      * It is too important to risk everything on one point of failure
      * essential to have a secondary plan in place in case the primary plan fails. Secure systems avoid single points of failure.
      * multiple layers of security e.g. castles
      * In security, the layers of defense are commonly grouped into three main categories: People, Technology, Process.
      * People:
        + write security policies
        + follow best practices
        + assign responsibilities
        + perform system administration
      * Technology:
        + firewalls
        + Intrusion detection
        + web server, code, database, installed technologies
        + encryption
        + access controls
      * Process:
        + technology evaluation, acquisition, and maintenance
        + data handling procedures
        + security monitoring
        + security responses
        + security reviews
      * Each category needs to be strong and each category should itself have multiple layers so there is no single failure point.
    - Another way to secure forms is to use the same protections as for Cross-Site Request Forgery (CSRF).
    - CSRF protections can prevent a user from un-knowingly submitting a form, but they can also be used to prevent an attacker from submitting a form as a faked request.

* **Cross Site Request Forgery:**
  + an attack in which a user is tricked into performing actions on another site by inadvertently clicking a link or a submitting a form. It often called CSRF, or sometimes XSRF, for short.
    - "Cross-Site": originates on one site but performs an action on another
    - "Request Forgery": it is not a genuine user request
  + especially powerful if the target site has previously authenticated the user's browser i.e. user is already logged into the target site through the same browser that subsequently loads the attacking site.
  + Whenever any request is sent to a website, the browser sends all stored cookies for that website along with the request.
  + The exploit takes advantage of the fact that the server trusts the user's browser.
  + Typically attackers use CSRF attacks to make state-changing requests. The request performs some action, but the attacker will not be able to see the results of the request since it is still a request by the user's browser.
  + CSRF GET Request:
    - The simplest CSRF attack is simply to trick a user into making a GET request to a specific URL
    - This can done by putting the URL into a deceptively named link. The link could be put in a blog comment (lots of WordPress exploits have used this technique), a post on a web forum, or in a phishing email.
      * e.g. <a href="http://hackermag.com/best\_hacker/vote/48576">View PDF</a>
    - The link hides its true action but still requires user input
    - More sophisticated version is one that avoids user input all together e.g. the image source attribute of an image tag.
      * <img src="http://hackermag.com/best\_hacker/vote/48576" />
      * As the page loads the browser makes individual HTML requests to retrieve all images. This would automatically load the malicious link without the users knowledge
  + CSRF GET Request with Authentication:
    - Biggest risk comes from sites that require authentication and save the users authenticated state through cookies
    - The attacker takes advantage of this
    - Imagine that a user logs in to their bank website. The bank website sets a cookie in the user's browser to indicate that their access has been authorized. Future requests to any page on the bank's website will include that cookie as proof of authorization.
    - If the user closes the page without logging out, the cookies remain on the browser
    - The attacker can then trick the user to load a the bank page again using image tag CSRF and since the cookies are preserved, the request will be authenticated.
      * e.g. <img src="https://bank.com/transfer?amount=10000&to\_account=2468013579" />
      * If this link is loaded it will trigger a get request. An unauthorised user will be directed to the banks login page but if they never logged out and the cookies remain, the request will be accepted
    - There are many actions that a CSRF attack can take. Some of the more common ones are:
      * Change password
      * Change email address
      * Login to a site
      * Transfer funds
      * Download malware
  + Get Request Preventions:
    - The best defense against CSRF attacks which take the form of GET requests is to disallow GET requests for key actions, especially actions which "change state" in some way.
    - It is considered a best practice to only use GET requests for retrieving data, not for actions which make changes. Instead, use POST requests (such as form submissions) for actions which make changes.
    - Links and image tag sources are always GET request. Making all GET requests harmless removes a major pathway for CSRF attacks.
  + CSRF POST Request Attack:
    - GET requests are not the only way to trigger a CSRF attack. An attacker can forge a form.
    - If an attacker forges a legit looking form, they can then put that HTML into another page and then trick a user into submitting the form.
    - This would pass any "allow POST requests only" protections and the request would send any authorization cookie, just like a GET request does.
    - The attacker only needs to find a way to trick a user into submitting this form. They can do that by using CSS to hide the form, and then using JavaScript to submit the form when the page loads.
      * <html>

<head>

<title>Fake Form</title>

</head>

<body onload="document.bank\_form.submit()">

<form action="http://bank.com/transfer" method="POST" name="bank\_form" style="display: none;">

<input type="text" name="amount" value="10000" />

<input type="text" name="to\_account" value="2468013579" />

</form>

</body>

</html>

* onload submits the form immediately without need for user input and the style attribute hides it from the user
* The user never had to do a thing. Convincing a user to visit this page is enough to trigger the request. And that is easily done, such as with a fake link in a phishing email.
* There is only one weakness with this attack so far; The form will submit and then the user will see the results of the submission. If a user saw that, they would be suspicious and call their bank to void the transaction.
* Therefore an attacker also needs to hide the results:
  + <html>

<head>

<title>Fake Form</title>

</head>

<body onload="document.bank\_form.submit()">

<form action="http://bank.com/transfer" method="POST" name="bank\_form" style="display: none;" target="hidden\_results" >

<input type="text" name="amount" value="10000" />

<input type="text" name="to\_account" value="2468013579" />

</form>

<iframe name="hidden\_results" style="display: none;"></iframe>

</body>

</html>

* Forms allow sending their results to an iframe using a target attribute.
* If the attacker hides an iframe and sends the form results there, the user will have no hint that the CSRF attack has occurred.
* CSRF POST Request Defenses:
  + One easy defense is to verify that the HTTP referer in the POST request is originating from the correct domain. Any legitimate form submission should have a referer and that referer should be the expected one for the form page.
  + The referer is spoofable, but it adds another challenge for an attacker to overcome. And each layer of defense adds to a Defense in depth strategy.
  + The strongest defense against CSRF attacks is the use of "CSRF Tokens" which is also known as the "synchronizer token pattern".
  + The synchronizer token pattern can be implemented on forms using the following steps:
    1. Generate a large, unique, random string for use as a token.
       1. return bin2hex(random\_bytes(64));
    2. Add token to user's session data
       1. $token = csrf\_token();
    3. Add token to the form data.
       1. return '<input type="hidden" name="csrf\_token" value="' . $token . '">';
    4. When the form is submitted, compare the form token with the session token.
       1. return ($\_POST['csrf\_token'] === $\_SESSION['csrf\_token']);
    5. If the tokens match then the form is valid. If not, it is invalid.
  + Generally, the attacker should not be able to see the session token given to the user, nor should they be able to set it. Unless the attacker also can use Cross-Site Scripting (XSS) to gain access to the user's browser cookies.
  + When putting CSRF protections in place, it is equally important to make sure that XSS protections are in place.
  + This example of CSRF tokens is very simple. More sophisticated examples include:
    1. tokens valid for limited time (5-10 mins)
    2. unique for each form on the site
    3. session token and the form token could be complimentary values e.g. encrypting one string could return the other string.
  + CSRF tokens are the best protection, but there are a few more which can further strengthen application defenses.
  + For sensitive actions the site could require an addition user action or confirmation in addition to the form submission e.g.
    1. a second "please confirm this action" page
       1. A CSRF attack would not be able to react to the second page
    2. a CAPTCHA image to confirm that there is a user behind an action
    3. require a user to re-authenticate by providing their password again
       1. A CSRF attack would not be able to respond to the second request, nor would the attacker know the correct password

* **PHP Cookies and Sessions:**
  + PHP Cookies:
    - Cookies, or browser cookies, are small pieces of data which the web server asks the client's web browser to store. Each request back to the server will include these pieces of data. The data is organized as key/value pairs.
    - A cookie can be set using PHP's setcookie()
      * <?php  
        setcookie('language', 'english');  
        ?>
    - On future requests, the cookie key/value pairs will assigned to the $\_COOKIE superglobal
      * <?php  
        echo $\_COOKIE['language'];  
        *// english*?>
    - In addition to the $name and $value arguments, setcookie() also accepts many other arguments for configuration
      * <?php  
        $name = 'language';  
        $value = 'english';  
        $expire = time() + 60\*60\*24\*3; *// 3 days from now*$path = '/blog';  
        $domain = '[www.mysite.com](http://www.mysite.com)';  
        $secure = isset($\_SERVER['HTTPS']); *// or use true/false*$httponly = true;

setcookie($name, $value, $expire, $path, $domain, $secure, $httponly);  
**?>**

* Many of these configuration arguments are important for preventing attacks such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Cookie Theft and Manipulation, Session Hijacking, and Session Fixation.
* PHP Sessions:
  + Sessions are an alternative to cookies. A session is usually a file or database record on the server side which contains the small pieces of data which the server wants to store for each user.
  + Instead of sending key/value pairs to the browser, these values are stored on the server, and only a reference identifier ("session ID") is sent to the user's browser as a cookie.
  + This session ID is a long and unique string.
  + On each future request, the browser will send the session ID as a cookie and the server will locate the corresponding session to allow access to the stored user data.
  + In PHP it is important to always initialize sessions using session\_start(). After being initialized, session values can be set and retrieved using the $\_SESSION superglobal.
    - **<?php**  
      session\_start();  
      $\_SESSION['user\_id'] **=** 42;  
      **echo** $\_SESSION['user\_id'];  
      *// 42***?>**
  + A session can also be unset and destroyed when expired or no longer needed. If not unset/destroyed, then the session file and session data will remain on the server unless the file or database storage for the session is deleted.
    - **<?php**  
       *// use both unset and destroy for compatibility* *// with all browsers and all versions of PHP* session\_unset();  
       session\_destroy();  
      **?>**
  + There are several configurations for PHP sessions which can be set in the php.ini file.
    - session.use\_only\_cookies = 1

session.cookie\_lifetime = 0 // '0' = expire when browser closes

session.cookie\_secure = 1

session.cookie\_httponly = 1

* In PHP 7 or greater, it is also possible to set these values when the session is started.
  + **<?php**  
     session\_start([  
     'use\_only\_cookies' **=>** 1,  
     'cookie\_lifetime' **=>** 0,  
     'cookie\_secure' **=>** 1,  
     'cookie\_httponly' **=>** 1  
     ]);  
    **?>**
* Many of these configuration arguments are important for preventing attacks such as Cookie Theft and Manipulation, Session Hijacking, and Session Fixation.

* Cookie Theft:
  + Browser cookies are very visible and can easily stolen or manipulated.
  + Cookies are usually visible in browsers however it has become more common place to hide them even though they are still easily visible to an attacker.
  + Stored cookies can also be stolen using Cross-Site Scripting (XSS).
  + Cookie data is also visible in transit. sent in plain text in the header of every request, an attacker observing network traffic will be able to see it
  + This is especially easy to do on an open WiFi network such as those commonly found at coffee shops and other businesses.
  + Imagine a website which makes the terrible security choice to store the user's login state in a cookie as plain text.
    - **<?php**  
       setcookie('user\_id', 42);  
       setcookie('logged\_in', **true**);  
      **?>**
  + The response to the user will be in plain text
    - HTTP/1.0 200 OK  
      Content-type: text/html  
      Set-Cookie: user\_id=42  
      Set-Cookie: logged\_in=true
  + After that every request back to the webserver will display those cookie values in plain text
    - GET /any\_page.php HTTP/1.1  
      Host: 55.66.77.88  
      Cookie: user\_id=42; logged\_in=true
  + If an attacker can see cookie data, then it is easy for them to "steal" it. They can forge a request and include the cookie data as if it were their own. An attacker could set their own cookies to those values or forge new requests which include "user\_id=42; logged\_in=true". Alternatively, an attacker could modify the cookie values. In this example, they might try "user\_id=1; logged\_in=true" to see if that granted access to a different account.
  + Cookie Theft and Manipulation Preventions:

HW Notes:

* BSM: change header to lessonComplete
* SMC1: change checksum to administrator
* SMC2: enter admin in username box which will return zoidberg22@shepherd.com
  + click forgot password and enter discovered email
  + open burp and run the forgot password command in repeater which will return password
* SMC3: enter admin in username box = Error
  + press toggle button and set password
  + open burp and run toggle request. change current to admin and run
  + admin password should now be changed to new password
* CSRF: Click on given Get request and open it in a new page. copy that url, paste in image box and replace userid with your id
* CSRF1: Using the same link as the one above, switch /root/ to the new request url /user/and plug in ID
  + Another account will then need to load the page to get the key
* CTF2.01: load in burp and change csrf\_token to admin
* 2.02: inspect the page. Find HTTP method and change to POST. change csrf\_token to admin. press update
* 2.03: change session id and csrf\_token to admin
* 2.04: change csrf\_token in body to decoded try\_this\_token
* 2.05: change get to post
* 2.06: Change get to post and change referer to page url
* 2.07: inspect csrf token replace body token with header token
* 2.08: change get to post and change csrf token to admin in body
* 2.09: Change post to get and hit update
* 2.10: Change Accept Encoding to identity
  + The "identity" content-coding is always acceptable, unless specifically refused because the Accept-Encoding field includes "identity;q=0", or because the field includes "\*;q=0" and does not explicitly include the "identity" content-coding. If the Accept-Encoding field-value is empty, then only the "identity" encoding is acceptable.
* 2.11: Gives CTF
* 2.12: edit url to access enumerated pages:
  + index.php?id=13
* 2.13: again, edit URL:
  + ?flag=true
* 2.14: Change sessionid to abcsession123
* 2.15:
* Change GET to POST. make sure there's a referer
* For earlier challenges, make sure to hit update button if applicable to send POST
* for 2.08 we need to change the get request to a post. It is also missing a bunch of headers which we can find in other post requests (2.04). Easy way is to just use 2.04 request and change everything that says 04 to 08 since they have the same URL. Also need to change privilege to admin
* make sure parameters are in the body not in the head