SECURE CODING

HEADLINES

- Input Validation and errors
- Authentication and Logins
- Authorization best practices
- Managing your sessions
- More Exercises

Session

A session is defined as **a series of related browser requests that come from the same client during a certain time period**.

Weakness of session identifier

Most web applications offer a login page and need to **keep track of the user session**. The main goal is to avoid users from logging in each time they visit the web application.

The **session identifier** is the unique key that identifies a user session within a database of sessions.

It is **critical to the session management mechanism** because, if an attacker manages to obtain it, he can **impersonate (or "ride") another user's session**.

Session Hijacking

Session Hijacking refers to the exploitation of a valid session assigned to a user. The attacker can get the victim's session identifier using a few different methods.

The attacker's goal is to find the session identifier used by the victim.

Remember that in most web applications, session IDs are typically carried back and forth between client web browser and server by using:

- Cookies
- URLs

http://www.bank.com?error=<script>CODE_STOLE_USER_CREDINTAL</script>

Session Hijack Prevention

- HTTPS: The use of HTTPS ensures that there is SSL/TLS encryption throughout the session traffic. Attackers will be unable to intercept the plaintext session ID, even if the victim's traffic was monitored. It is advised to use HSTS (HTTP Strict Transport Security) to guarantee complete encryption.

- Phishing Scam: Avoiding falling for phishing attacks. Only click on links in an email that you have verified to have been sent from a legitimate sender.

- Token encryption:

```
$token = md5(uniqid(rand(), TRUE));

$_SESSION['token'] = $token;
```

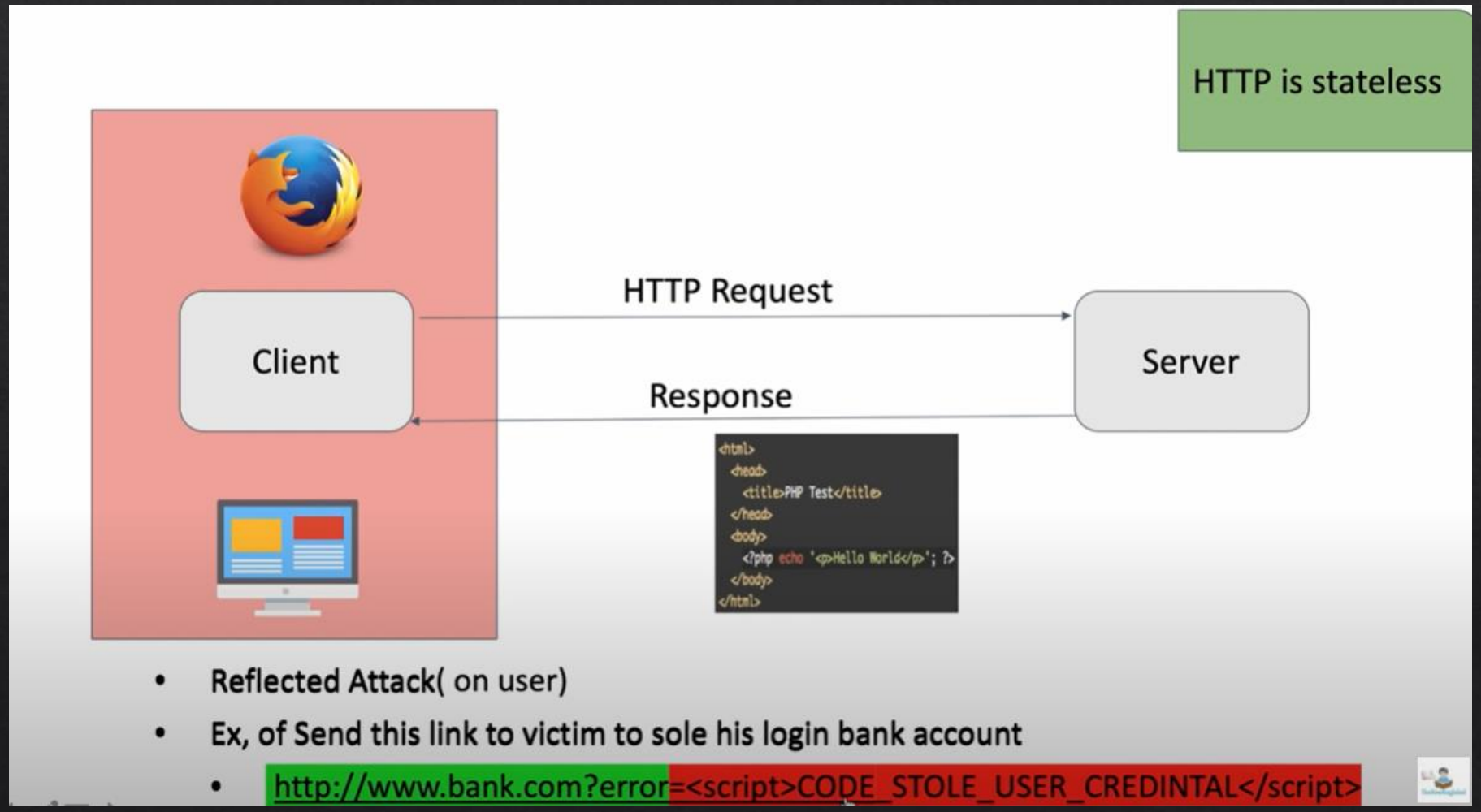
Cross-site scripting (XSS)

Cross-site scripting (XSS) is a type of security vulnerability that can be found in some web applications. XSS attacks enable attackers to inject client-side scripts into web pages viewed by other users.

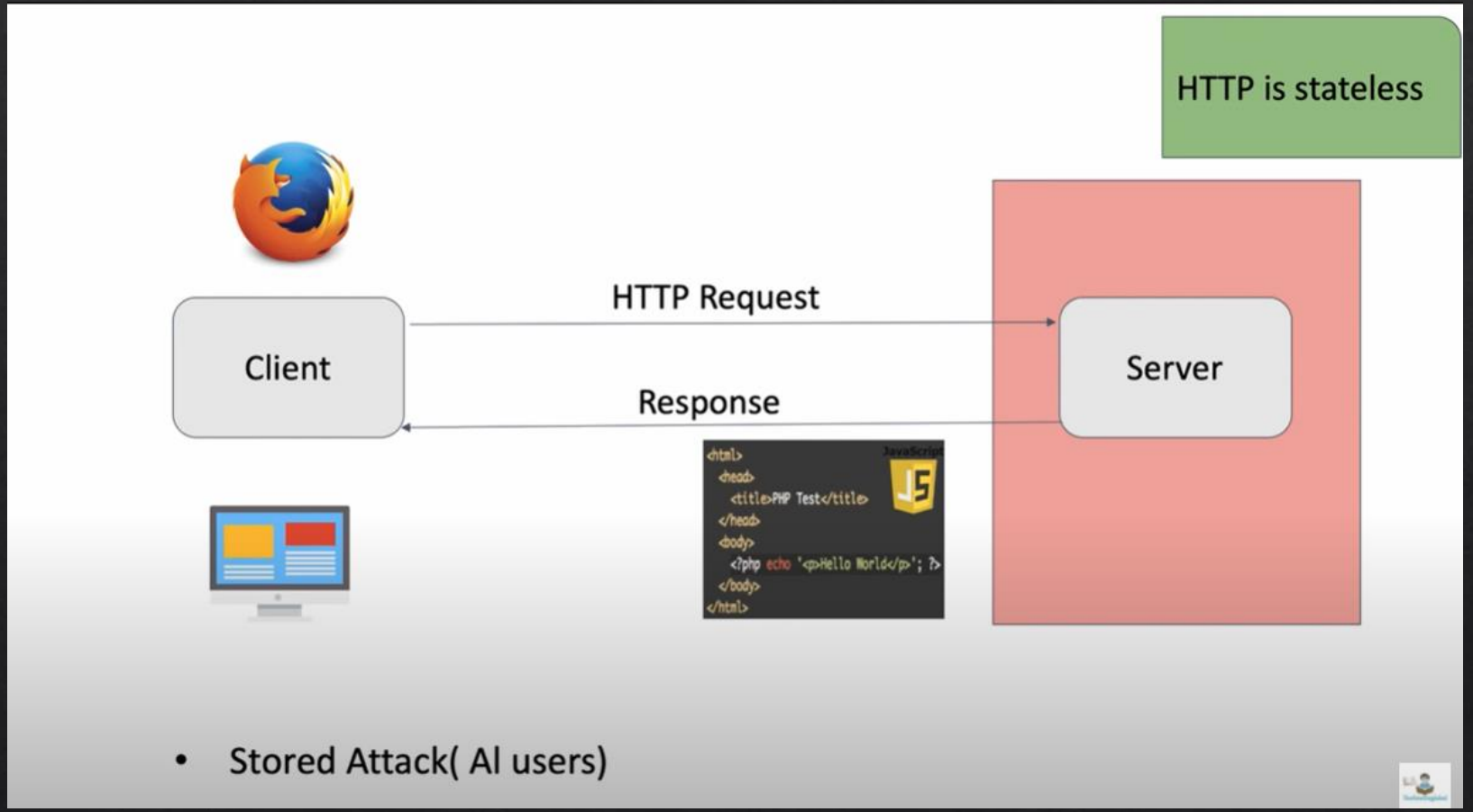
Types:

- Reflected XSS
- Stored attack

XSS Stored vs Reflected



XSS Stored vs Reflected



- Stored Attack(Al users)

INPUT VALIDATION AND ERRORS

- Don't have trust in input
- Don't depend on client-side validation.
- Constrain, reject and sanitize input whereas always validate for kind, length, format as well as range.

```php
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    // Check database
    $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
    $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error())

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Get values
        $first = $row["first_name"];
        $last  = $row["last_name"];

        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    mysqli_close($GLOBALS["___mysqli_ston"]);
}

?>
```

```php
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Get input
    $id = $_GET[ 'id' ];

    // Check database
    $getid  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
    $result = mysqli_query($GLOBALS["___mysqli_ston"],  $getid ); // Removed 'or die' to suppress mysql errors

    // Get results
    $num = @mysqli_num_rows( $result ); // The '@' character suppresses errors
    if( $num > 0 ) {
        // Feedback for end user
        echo '<pre>User ID exists in the database.</pre>';
    }
    else {
        // User wasn't found, so the page wasn't!
        header( $_SERVER[ 'SERVER_PROTOCOL' ] . ' 404 Not Found' );

        // Feedback for end user
        echo '<pre>User ID is MISSING from the database.</pre>';
    }

    ((is_null($___mysqli_res = mysqli_close($GLOBALS["___mysqli_ston"]))) ? false : $___mysqli_res);
}

?>
```

```php
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

Is it vulnerable !!?

```php
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $id = $_GET[ 'id' ];

    // Was a number entered?
    if(is_numeric( $id )) {
        // Check the database
        $data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE user_id = (:id) LIMIT 1;' );
        $data->bindParam( ':id', $id, PDO::PARAM_INT );
        $data->execute();
        $row = $data->fetch();

        // Make sure only 1 result is returned
        if( $data->rowCount() == 1 ) {
            // Get values
            $first = $row[ 'first_name' ];
            $last  = $row[ 'last_name' ];

            // Feedback for end user
            echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
        }
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

AUTHENTICATION AND LOGINS

- Only over the HTTPS connections pass the forms authentication cookies.
- Always encrypt communication channels in order to protect authentication tokens.
- Use strong passcodes.

AUTHORIZATION BEST PRACTICES

- Restrict user access towards system-level resources.
- Carry API validation.
- List down all allowable methods.
- Give protection to privilege actions and sensitive resource collections.
- Protect against all cross-site resource forgery (CSRF).

Cross site request forgery

CSRF or **XSRF** attacks (also pronounced **Sea-Surf attacks**) are something that every web app tester should both know and master since automated scanning tools cannot easily find the vulnerabilities.

CSRF exploits a feature of internet browsing, instead of a specific vulnerability.

CSRF is a vulnerability where a third-party web application can perform an action on the user's behalf. It is based on the fact that web applications can send requests to other web applications, without showing the response.

Cross site request forgery – Cont'd

1. **Hannora (victim)** visits ***amazon.com***, logs in, then leaves the site without logging out.
2. **Hannora** then visits **egybest.com** (malicious website) which executes a request to **amazon.com** from **Hannora's** browser (such as buy a book).
3. The victim browser sends this request, along with all the victim cookies. The request seems legit to amazon.com.

Cross site request forgery – Cont'd

Since Hannora is still logged in on Amazon, the request goes through, and the money is withdrawn from his account for the purchase of the book.

Whatever is in a webpage, Hannora's browser parses it and requests it; if an image with the URL amazon.com/buy/book is present in the page, the web browser will **silently issue a request** to Amazon.com, thus buying the book.

This is because Hannora already has an authenticated session open on Amazon.

Cross site request forgery – Cont'd

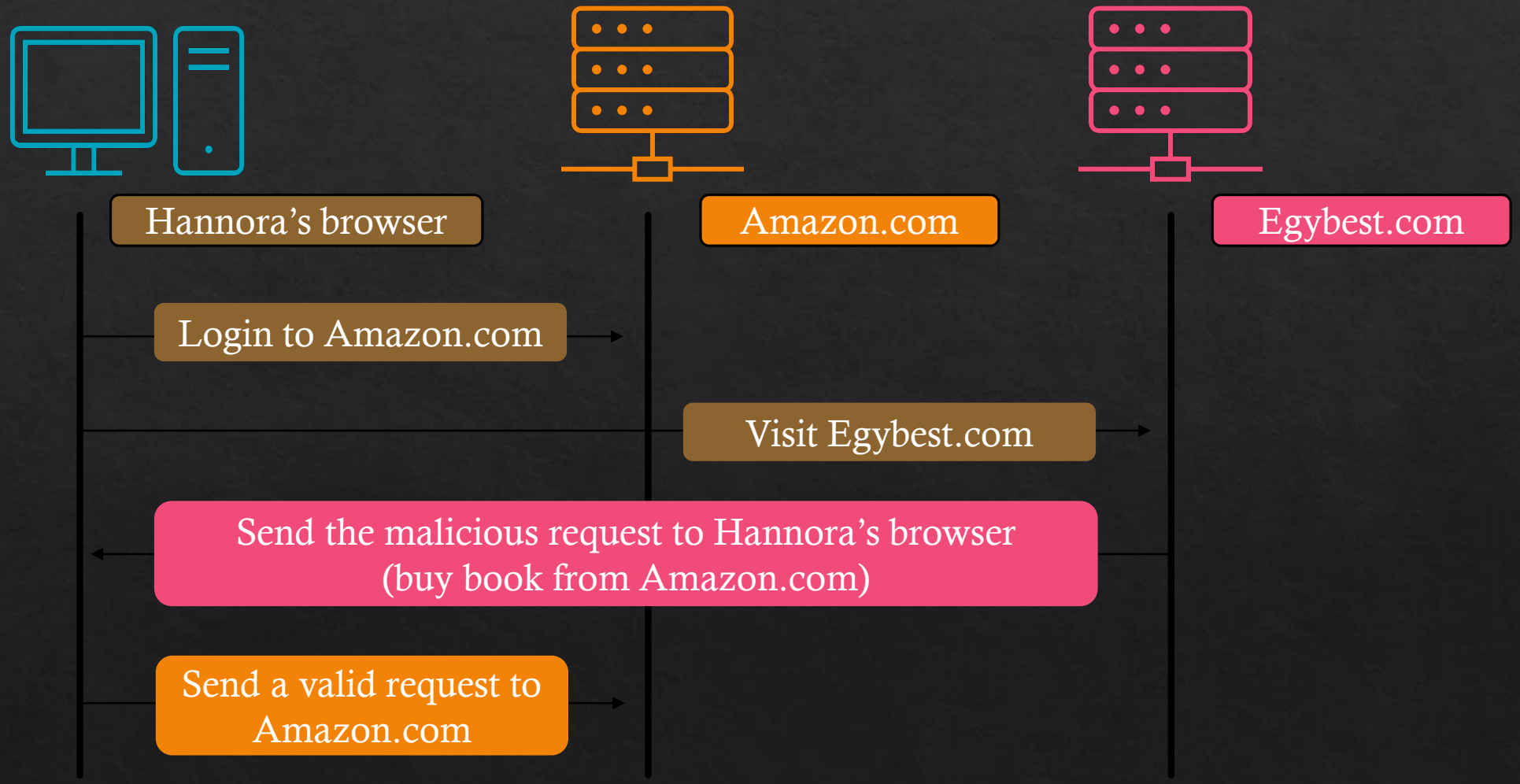
Google, Amazon, eBay and other major websites used to be vulnerable to CSRF, but many smaller websites and third-party application scripts remain vulnerable.

All requests to a web application that do not implement an Anti-CSRF mechanism are automatically vulnerable.

When a web application stores session information in cookies, these cookies are sent with every request to that web application.

This may sound odd but storing session tokens in cookies enables CSRF exploitability (while, of course, storing session tokens into URLs enable other kind of exploits).

Cross site request forgery



Prevent CSRF

The most common protection mechanism against CSRF exploit is the **token**.

The token is a nonce (a number used once and discarded) and makes part of the request required to perform a given action unpredictable for an attacker.

In a real-world scenario, the Buy Book includes a hidden input that requires a token.

This token can be implemented as MD5 hash (or stronger) of some randomly-generated string

Prevent CSRF (cont.)

While rendering the form to the user, the following steps are taken by the web application to enforce protection against CSRF exploits:

- Generate a token
- Include the token as hidden input on the form
- Save the token in the session variables

Prevent CSRF (cont.)

An attacker able to force the victim to request the forged URL, must provide a valid token.

Two conditions may occur :

- The victim has not visited the page and, as such, has no token set in session variables at all
- The victim has visited the page and has a token:

  In this case, the attacker must guess the correct token.

  That is impossible if token was randomly generated and so it is not easily predictable.

It is critical that the token must be random, unpredictable and change at least for every session

Conclusion

- To prevent CSRF, one must implement a random token for every request

MANAGING YOUR SESSIONS

- End session with logoff.
- Create a new session on re-authentication.
- Set secure attribute for cookies transmitted over TLS.

MORE EXERCISES

Which of the following code snippets **PREVENTS** a Command Injection attack?

```
String updateServer = request.getParameter("updateServer");
List<String> commandArgs = new ArrayList<String>();
commandArgs.add("ping");
commandArgs.add(updateServer);
ProcessBuilder build = new ProcessBuilder(commandArgs);
```

```
String updateServer = request.getParameter("updateServer");
String cmdProcessor = Utils.isWindows() ? "cmd" : "/bin/sh";
String command = cmdProcessor + "-c ping " + updateServer;

Process p = Runtime.getRuntime().exec(command);
```

Solution

- The code sample is preventing a vulnerability by using a parameterized command object.

```
String updateServer = request.getParameter("updateServer");
List<String> commandArgs = new ArrayList<String>();
commandArgs.add("ping");
commandArgs.add(updateServer);
ProcessBuilder build = new ProcessBuilder(commandArgs);
```

MORE EXERCISES

Which of the following code snippets **PREVENTS** a Buffe Overflow vulnerability?

```
printf("Enter the password:\n");
fgets(userPass,9,stdin);

if(strncmp(userPass,PASSWORD,9)==0){
    printf("PASSWORD VERIFIED\n");
}
```

```
printf("Enter the password:\n");
gets(userPass);

if(strncmp(userPass,PASSWORD,9)==0){
    printf("PASSWORD VERIFIED\n");
}
```

Solution

- The code sample is preventing a vulnerability by using a function that controls the size of the input ('fgets').

```
printf("Enter the password:\n");
fgets(userPass,9,stdin);

if(strncmp(userPass,PASSWORD,9)==0){
    printf("PASSWORD VERIFIED\n");
}
```

MORE EXERCISES

Which of the following code snippets **PREVENTS** XSS?

```
<div class="form-group">
    <label for="search">Search:</label>
    <input type="text" class="form-control" id="search" name="search">

    <input type="submit" id="submit" class="btn" value="Search">
    <div class="alert alert-danger <%=alertVisibility%>">
        Cannot find <%=StringEscapeUtils.escapeHtml4(request.getParameter("search"))%>
    </div>
</div>
```

```
<div class="form-group">
    <label for="search">Search:</label>
    <input type="text" class="form-control" id="search" name="search">

    <input type="submit" id="submit" class="btn" value="Search">
    <div class="alert alert-danger <%=alertVisibility%>">
        Cannot find <%=request.getParameter("search")%>
    </div>
</div>
```

Solution

- The code sample is neutralizing HTML markup.

```
<div class="form-group">
    <label for="search">Search:</label>
    <input type="text" class="form-control" id="search" name="search">

    <input type="submit" id="submit" class="btn" value="Search">
    <div class="alert alert-danger <%=alertVisibility%>">
        Cannot find <%=StringEscapeUtils.escapeHtml4(request.getParameter("search"))%>
    </div>
</div>
```