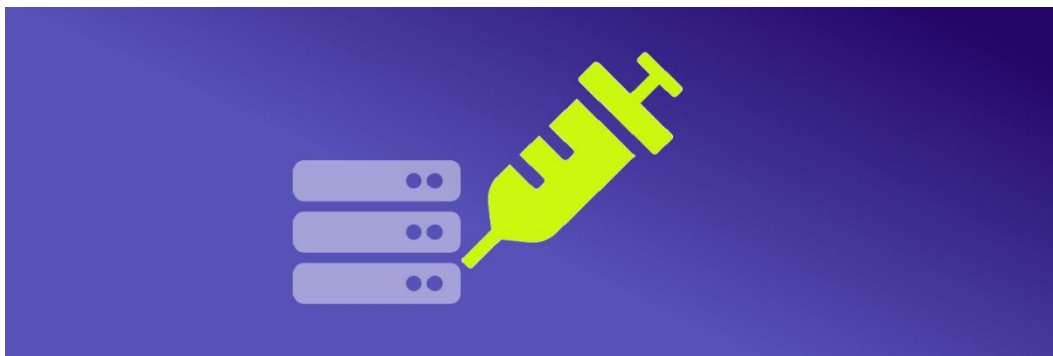


29 MARS 2020



TP2 REMOTE

SYNTHESE

TRISTAN GUERIN

ENSIBS

Cybersécurité du Logiciel A2

Description	3
Base de données	3
Application	5
Injections & Patch	17
Scan d'application	18
Patch	21
Vol de cookie	22
Remarques	24
Annexes	25
Questions	25
Sources	27

Description

Ce TP est la suite du *TP Remote 1* dans lequel nous avons créé une application web affichant le contenu d'une table dans une base de données, et où certaines pages étaient sensibles aux injections SQL et où d'autres pages patchaient ce problème.

Ici nous allons repartir sur la même base et nous allons rajouter des pages sensibles aux injections XSS et des pages patch.

Nous allons aussi aborder la sécurité des données en transit et au repos.

Base de données

On crée tout d'abord, comme pour le TP précédent, une base de données nommée selon notre nom de famille. (Ici base de données 'GUERIN' dans le SGBD 'MySQL')

On peut ensuite créer une table 'user' comportant six champs différents ('id', 'name', 'password', 'salary', 'age' et 'role').

Contrairement au précédent TP, le champ 'password' ne contiendra pas directement le mot de passe mais son empreinte en *SHA512*.

On crée aussi un utilisateur 'GUERIN' ayant accès qu'à cette table (il ne possède pas de mot de passe).

On insère aussi plusieurs utilisateurs dont un nommé « manager » ayant comme mot de passe « manager1 ».

(Toutes les commandes nécessaires sont dans le fichier 'scriptSQL.sql' ci-joint à ce rapport)

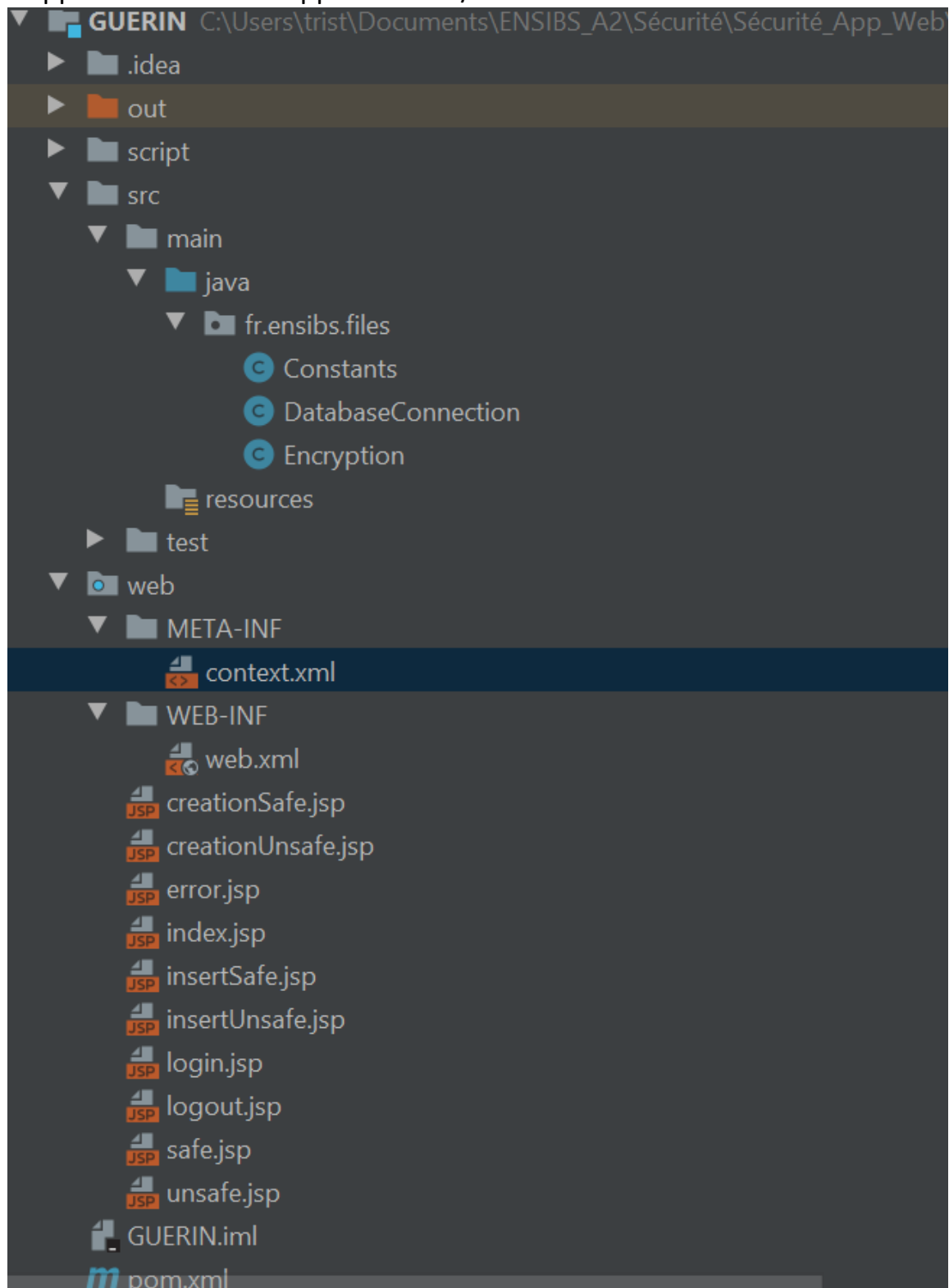
```
mysql> show columns from GUERIN.user;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| name  | varchar(50) | YES | | NULL | |
| password | varchar(150) | YES | | NULL | |
| salary | bigint(20) | NO | | NULL | |
| age   | int(10) unsigned | NO | | NULL | |
| role  | varchar(50) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.03 sec)

mysql> select * from GUERIN.user;
+-----+-----+-----+-----+-----+-----+
| id | name | password | salary | age | role |
+-----+-----+-----+-----+-----+-----+
| 1 | manager | 92a881051a0d26ba0fe4a65cb1039c10e18718c68591efb6afb883b672a328bc8ba8c13fdaa90eedc018c280782cbbd2a842acbd9a5f3b8965012a1ba489234 | 1234 | 22 | manager |
| 2 | tristan | b16ed7d24b3ecbd4164dcdad374e08c0ab7518aa07f9d3683f34c2b3c67a15830268cb4a56c1ff6f54c8e54a795f5b87c08668b51f82d0093f7baee7d2981181 | 2500 | 21 | manager |
| 3 | antoine | 6d201beefb589b08ef0672dac82353d0cbd9ad99e1642c83a1601f3d647bcca003257b5e8f31bdc1d73fbec84fb085c79d6e2677b7ff927e823a54e789140d9 | 2400 | 27 | manager |
| 4 | maxime | cb872de2b8d2509c54344435ce9cb43b4faa27f97d486ff4de35af03e4919fb4ec53267caf8def06ef177d69fe0abab3c12fbdc2f267d895fd07c36a62bff4bf | 3900 | 18 | manager |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

On voit ici les colonnes de la table, ainsi que la première entrée

Application

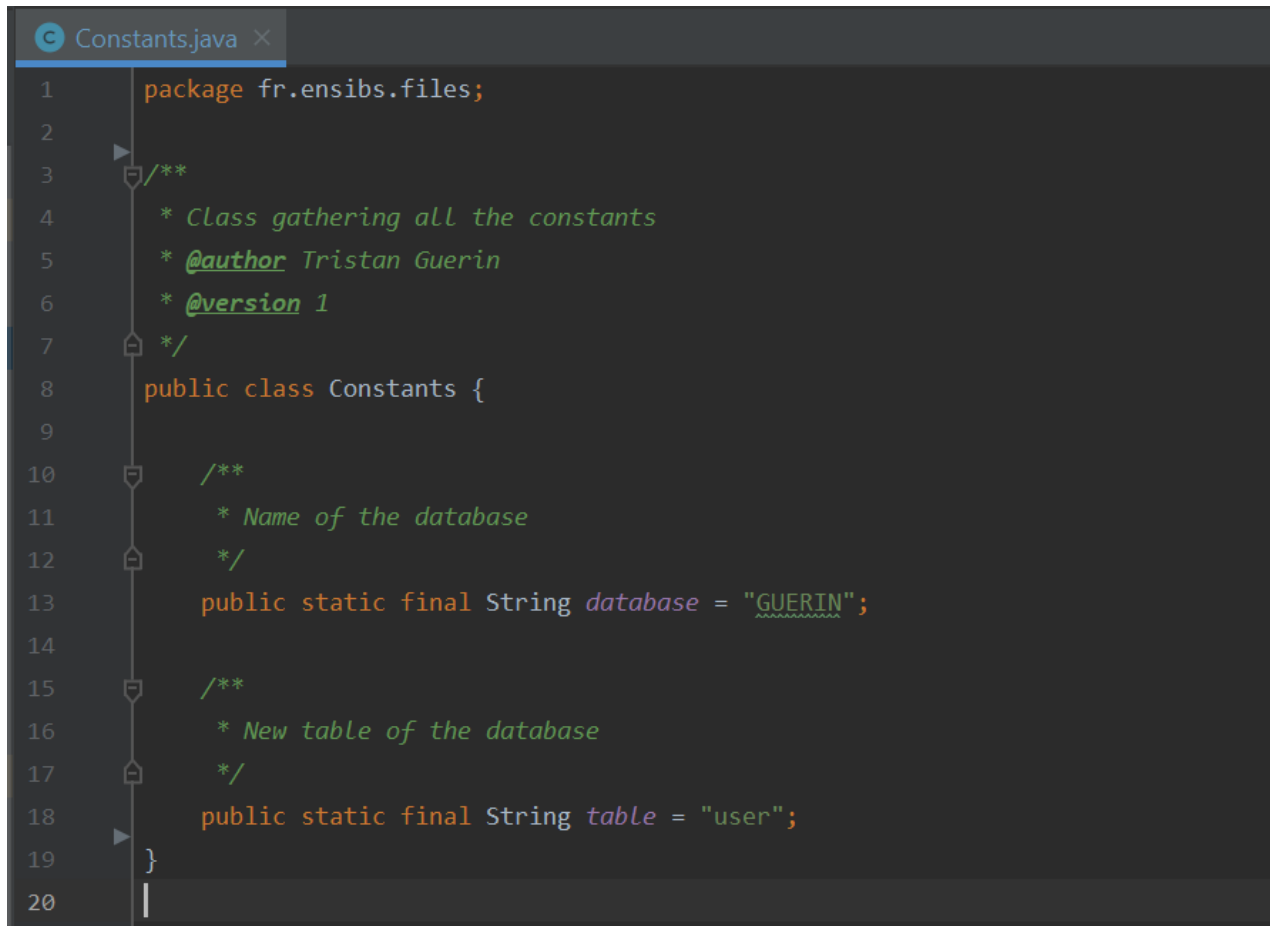
L'application est développée en Java/JSP et utilise 'Tomcat 9.0.30'.



On retrouve différents fichiers *.java*, servant à contenir les constantes nécessaires, à établir la connexion avec la base de données et à hasher les mots

de passe. On va aussi avoir des fichiers de configuration (*context.xml* et *web.xml*), ainsi que des fichiers *.jsp*.

La classe Java *Constants* va contenir toutes les constantes nécessaires au lancement de l'application web.



```
1 package fr.ensibs.files;
2
3 /**
4  * Class gathering all the constants
5  * @author Tristan Guerin
6  * @version 1
7  */
8 public class Constants {
9
10     /**
11      * Name of the database
12      */
13     public static final String database = "GUERIN";
14
15     /**
16      * New table of the database
17      */
18     public static final String table = "user";
19 }
20 |
```

Elle contient le nom de la base de données et de sa table contenant les informations sur les utilisateurs.

La classe *DatabaseConnection* va permettre de se connecter avec le SGBD MySQL.

```
/* Class having a connection to the database */
public class DatabaseConnection {

    public static Connection getConnection(){
        Connection connection = null;
        try {
            Context context = new InitialContext();
            DataSource ds = (DataSource)context.lookup( name: "java:comp/env/jdbc/user");
            connection = ds.getConnection();
        } catch (NamingException | SQLException e) {
            e.printStackTrace();
        }
        return connection;
    }
}
```

Il va pour cela utiliser le fichier de contexte de pool de connexion *context.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/">
    <Resource
        name="jdbc/user"
        auth="Container"
        type="javax.sql.DataSource"
        maxAxtive="100"
        maxIdle="30"
        maxWait="10000"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/GUERIN?allowMultiQueries=true"
        username="GUERIN"
        password=""
    />
</Context>
```

Ce fichier va référencer quel type de ressource pour le contexte utiliser (ici *javax.sql.DataSource*) ainsi que le *driver* nécessaire. On y indique également l'url de connexion à la base de données ainsi que les *credentials* pour établir la connexion (ici les *credentials* du premier utilisateur créé via *scriptSQL.sql*).

Enfin la classe *Encryption* va proposer une méthode pour hasher les mots en passe suivant la fonction *SHA512*.

```
Encryption.java x
5  import java.security.NoSuchAlgorithmException;
6
7  /**
8   * Class providing a method to hash a password
9   * @author Tristan Guerin
10  * @version 2
11  */
12  public class Encryption {
13
14
15  @ public static String hashPassword(String input) {
16      try {
17          MessageDigest md = MessageDigest.getInstance("SHA-512");
18          byte[] messageDigest = md.digest(input.getBytes());
19          BigInteger no = new BigInteger( signum: 1, messageDigest);
20          String hashtext = no.toString( radix: 16);
21
22          while (hashtext.length() < 32) {
23              hashtext = "0" + hashtext;
24          }
25
26          // return the HashText
27          return hashtext;
28      } catch (NoSuchAlgorithmException e) {
29          e.printStackTrace();
30      }
31      return null;
32  }
```

Les fichiers *index.jsp*, *unsafe.jsp* et *safe.jsp* sont issus du premier TP et servent respectivement à afficher des liens de détails des entrées de la table, à accéder aux détails d'une entrée en étant sensible aux SQLI et à accéder aux détails d'une entrée en patchant la sensibilité aux SQLI. Ces fichiers ont subi de légères modifications, dû aux changements de structure. Ces changements ne seront pas détaillés ici car vraiment légers.


```
JSP error.jsp x
1  <%--
2      Created by IntelliJ IDEA.
3      User: trist
4      Date: 24/03/2020
5      Time: 17:29
6      To change this template use File | Settings | File Templates.
7  --%>
8  <%@ page contentType="text/html; charset=UTF-8" language="java" %>
9  <html>
10 <head>
11     <title>Error</title>
12 </head>
13 <body>
14     <br/>
15     <a href= "logout.jsp">Logout</a>
16     Your request has lead to an error.
17 </body>
18 </html>
```

La page *error.jsp* est la page vers laquelle on sera automatiquement redirigé si une erreur intervient sur une autre page.

La page *login.jsp* va permettre l'authentification par formulaire sur le serveur *Tomcat*.

```
<html>
<head>
  <title>Login Page</title>
</head>
<body>

<form method="post" action="j_security_check">
  <table border = "0">
    <tr>
      <td>Login</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td>Password</td>
      <td><input type="password" name="j_password"></td>
    </tr>
  </table>
  <input type="submit" value="Login">
</form>

</body>
</html>
```

C'est une page de formulaire *POST* ayant comme action *j_security_check*, action inhérente à Tomcat.

Afin d'établir l'authentification par formulaire, il est nécessaire de modifier certains fichiers de *conf* de *Tomcat*.

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
  driverName="com.mysql.jdbc.Driver"
  connectionURL="jdbc:mysql://localhost:3306/GUERIN"
  connectionName="GUERIN"
  connectionPassword=""
  userTable="user"
  userNameCol="name"
  userCredCol="password"
  userRoleTable="user"
  roleNameCol="role">
  <CredentialHandler className="org.apache.catalina.realm.MessageDigestCredentialHandler" algorithm="sha-512" />
</Realm>
```

Ainsi, dans le fichier *server.xml* du dossier *apache-tomcat*, on va ajouter un *JDBCRealm*, qui va alors établir un lien avec la table *user* de notre base de données *GUERIN*.

On y indique la table des utilisateurs et celle des rôles (ici la même table fait les deux), ainsi que le nom des colonnes nécessaires.

On indique aussi le fait que le mot de passe envoyé par le formulaire doit être hashé en *SHA512* avant comparaison.

Enfin, on va rajouter dans le fichier *web.xml* les balises suivantes :

```
<security-constraint>
  <!-- web resources that are protected -->
  <web-resource-collection>
    <web-resource-name>Protected Page</web-resource-name>
    <!-- Change the url-pattern to /* to enable or /somethingelse to disable-->
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method><http-method>HEAD</http-method>
    <http-method>POST</http-method><http-method>PATCH</http-method>
    <http-method>OPTION</http-method><http-method>CONNECT</http-method>
    <http-method>TRACE</http-method><http-method>PUT</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>
```

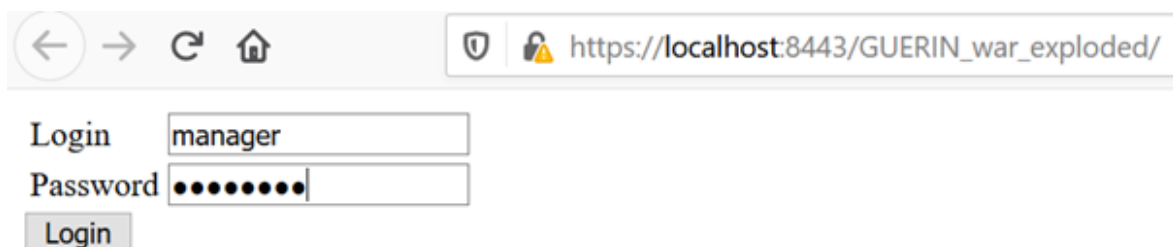
On va alors indiquer quels types de méthode et quelles URLs vont être soumis à une authentification (ici toutes les pages, via le pattern */**). On précise aussi le rôle d'authentification *manager*.



On précise aussi la méthode d'authentification (ici par formulaire, 'FORM'), ainsi que les pages de login et d'erreur.

```
<security-role>
  <role-name>manager</role-name>
</security-role>

<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Dès lors que l'on lancera le serveur *Tomcat*, on sera automatiquement vers la page *login.jsp* afin de s'authentifier.



← → ↻ 🏠   https://localhost:8443/GUERIN_war_exploded/

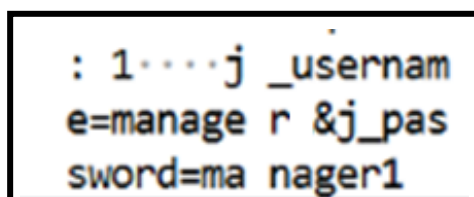
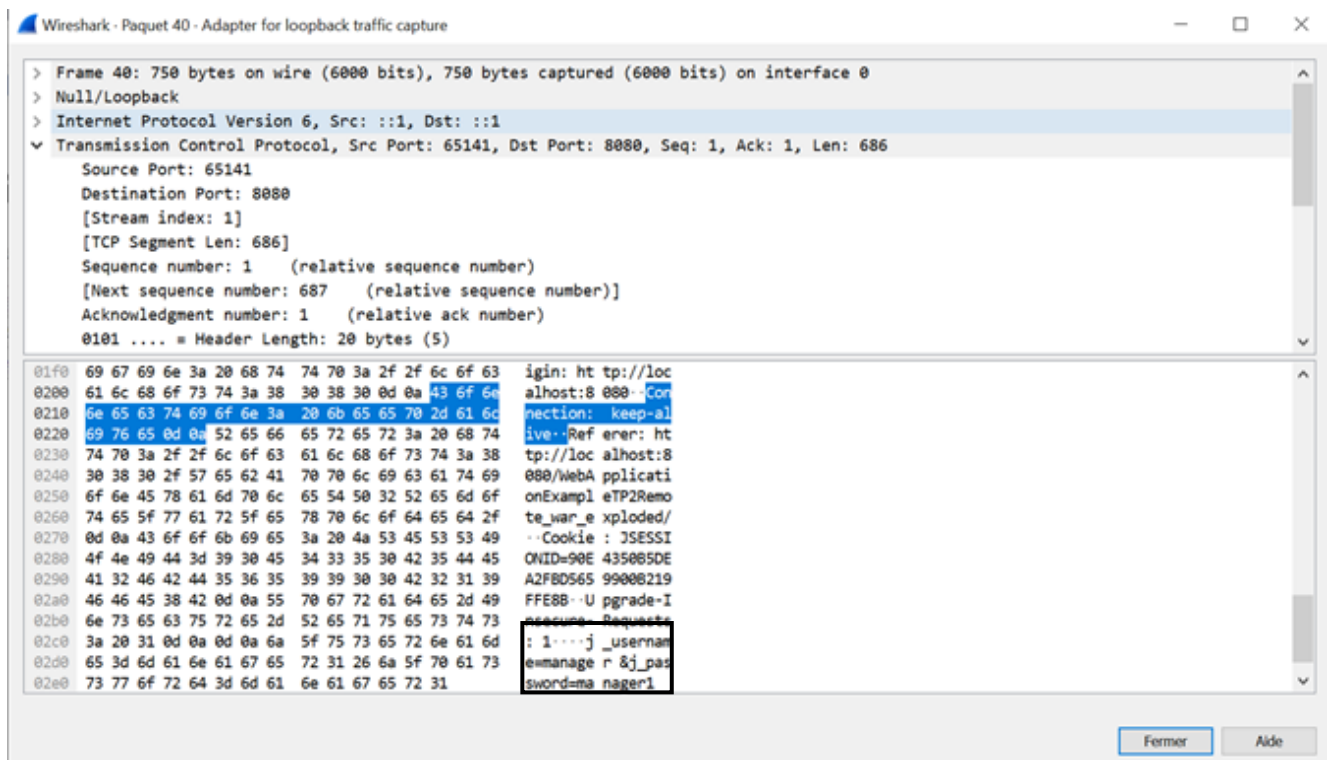
Login

Password

Login

Après avoir mis en place une authentification avec les utilisateurs de la table dans la base de données, il est intéressant de se pencher sur la sécurisation des données en transit. En effet, lorsque l'on va par exemple s'authentifier sur la page de *login*, le site étant initialement en *http*, les données vont circuler en clair.

L'image ci-dessous est un extrait de capture *WireShark* lors de l'authentification :



Les *credentials* sont ici en clair. Un attaquant peut alors facilement les récupérer pour se connecter à la plateforme.

C'est pourquoi nous allons implémenter le protocole *https* afin de chiffrer ces données.

On va tout d'abord avoir besoin de générer une clé SSL via la commande :

```
keytool -genkey -alias tomcat -keyalg RSA -keystore apache-tomcat-9.0.30/conf/key
```

Une fois cette clé générée et placée dans le dossier *conf* d'*apache-tomcat*, il va ensuite être nécessaire d'indiquer au fichier *server.xml* le port et le type de connexion de la plateforme.

On va pouvoir rajouter la balise *Connector* suivante contenant les différents paramètres de mise en place :

```
<Connector SSLEnabled="true" acceptCount="100" clientAuth="false"
disableUploadTimeout="true" enableLookups="false" maxThreads="25"
port="8443" keystoreFile="conf/key" keystorePass="manager1"
protocol="org.apache.coyote.http11.Http11NioProtocol" scheme="https"
secure="true" sslProtocol="TLS" />
```

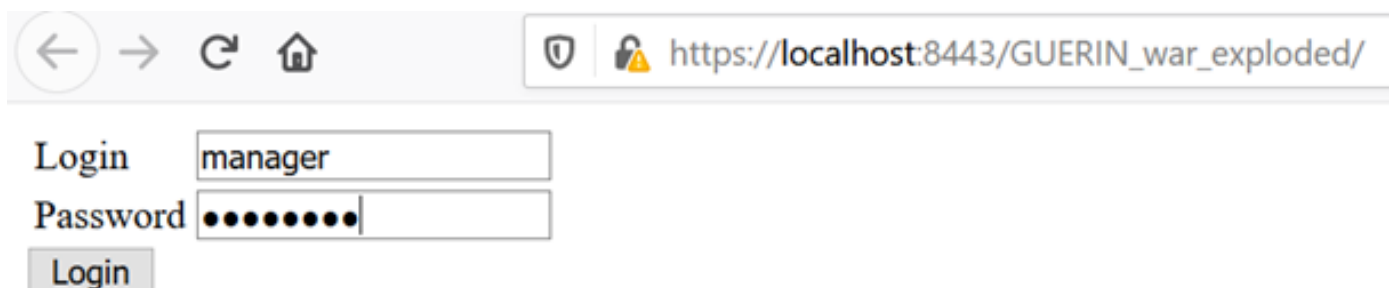
On indique le port, où trouver le fichier contenant la clé, quel est le mot de passe pour la clé (ici on a choisi *manager1*), le protocole SSL, ...

Enfin dans le fichier *web.xml* de notre application, on va rajouter dans la balise *security-constraint* le type de garantie de transport :

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

On choisit *CONFIDENTIAL* car on souhaite empêcher la lecture des données transmises par d'autres entités.

Après avoir lancé le serveur *Tomcat*, on va pouvoir se connecter à l'application avec l'adresse suivante : https://localhost:8443/GUERIN_war_exploded/



The screenshot shows a web browser window with the address bar displaying https://localhost:8443/GUERIN_war_exploded/. Below the address bar, there is a login form. It consists of a 'Login' label followed by a text input field containing the text 'manager'. Below this is a 'Password' label followed by a password input field filled with 10 dots. At the bottom left of the form is a 'Login' button.

On utilise bien le protocole *https* et lorsque l'on capture un extrait du flux réseau via *Wireshark* lors de l'authentification, on remarque les *credentials* sont bien chiffrés :

No.	Time	Source	Destination	Protocol	Length	Info
494	2020-03-26 23:59:02,099945	127.0.0.1	127.0.0.1	TCP	44	8443 → 56
2426	2020-03-27 00:00:00,697893	127.0.0.1	127.0.0.1	TCP	56	51006 → 8
2427	2020-03-27 00:00:00,697951	127.0.0.1	127.0.0.1	TCP	56	8443 → 51
2428	2020-03-27 00:00:00,697996	127.0.0.1	127.0.0.1	TCP	44	51006 → 8
2431	2020-03-27 00:00:00,699715	127.0.0.1	127.0.0.1	TLSv1...	561	Client He
2432	2020-03-27 00:00:00,699756	127.0.0.1	127.0.0.1	TCP	44	8443 → 51
2439	2020-03-27 00:00:00,717618	127.0.0.1	127.0.0.1	TLSv1...	177	Server He
2440	2020-03-27 00:00:00,717667	127.0.0.1	127.0.0.1	TCP	44	51006 → 8
2441	2020-03-27 00:00:00,718622	127.0.0.1	127.0.0.1	TLSv1...	50	Change Ci
2442	2020-03-27 00:00:00,718715	127.0.0.1	127.0.0.1	TCP	44	51006 → 8

> Frame 2431: 561 bytes on wire (4488 bits), 561 bytes captured (4488 bits) on interface 0						
> Null/Loopback						
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1						
> Transmission Control Protocol, Src Port: 51006, Dst Port: 8443, Seq: 1, Ack: 1, Len: 517						
> Transport Layer Security						

0090	c0 13 c0 14 00 33 00 39 00 2f 00 35 00 0a 01 003-9 ./-5....
00a0	01 8f 00 00 00 0e 00 0c 00 00 09 6c 6f 63 61 6clocal
00b0	68 6f 73 74 00 17 00 00 ff 01 00 01 00 00 0a 00	host.....
00c0	0e 00 0c 00 1d 00 17 00 18 00 19 01 00 01 01 00
00d0	0b 00 02 01 00 00 10 00 0e 00 0c 02 68 32 08 68h2-h
00e0	74 74 70 2f 31 2e 31 00 05 00 05 01 00 00 00 00	ttp/1.1.....
00f0	00 33 00 6b 00 69 00 1d 00 20 4c d3 58 8b 6a d6	3-k-i...L-X-j
0100	24 a5 ba 4a 41 e5 69 29 b0 a2 63 28 2d 99 b4 01	\$-JA-i) -c(-...
0110	ae 6d 7e 15 e6 bf 58 8b 47 56 00 17 00 41 04 5f	m~...X- GV...A-
0120	9a 3a 36 a6 5e 3c 21 2d 0b 75 70 1c 3a 47 e0 14	..6.^< - up.:G..
0130	87 3d a5 f1 8f 7d 2c 41 eb ce 7a 49 04 0f 16 34	==...},A ..zI...4
0140	83 af 2c 5f 33 f0 b7 87 19 45 b7 95 4b bc 0c 6e	..,_3...E-K..n
0150	92 ba d0 1c c3 e7 aa 28 65 1a 59 ef d9 78 9a 00(e-Y..x..
0160	2b 00 09 08 03 04 03 03 03 02 03 01 00 0d 00 18	+.....
0170	00 16 04 03 05 03 06 03 08 04 08 05 08 06 04 01	

Comme nous avons mis en place une authentification par formulaire, il est aussi nécessaire de pouvoir se déconnecter.

Ainsi une page *logout.jsp* va tout simplement invalider la session en la retirant du registre.

```
logout.jsp ×
1  <!--
2      Created by IntelliJ IDEA.
3      User: trist
4      Date: 27/03/2020
5      Time: 23:31
6      To change this template use File | Settings | File Templates.
7  -->
8  <%@ page contentType="text/html; charset=UTF-8" language="java" %>
9  <html>
10 <head>
11     <title>Logout page</title>
12 </head>
13 <body>
14 <%
15     session.invalidate();
16     response.sendRedirect("index.jsp");
17 %>
18 </body>
19 </html>
```

Il faudra dès lors se réauthentifier par formulaire.

Injectons & Patch

Après avoir abordé la sécurité des données en transit, et l'authentification par formulaire, on va pouvoir s'attaquer aux exercices d'injections XSS.

L'application possède deux pages *creationUnsafe.jsp* et *creationSafe.jsp*, qui sont toutes deux des pages de formulaire *POST* qui vont envoyer respectivement des requêtes aux pages *insertUnsafe.jsp* et *insertSafe.jsp* afin d'insérer une nouvelle entrée dans la base de données.

```
<body>
<form action="insertUnsafe.jsp" method="post" class="form-example">
  <div class="form-example">
    <label>Enter your name: </label>
    <input type="text" name="name"><br/>
    <label>Enter your password: </label>
    <input type="password" name="password"><br/>
    <label>Enter your salary: </label>
    <input type="text" name="salary"><br/>
    <label>Enter your age: </label>
    <input type="text" name="age"><br/>
    <input type="submit" value="Register">
  </div>
</form>
```

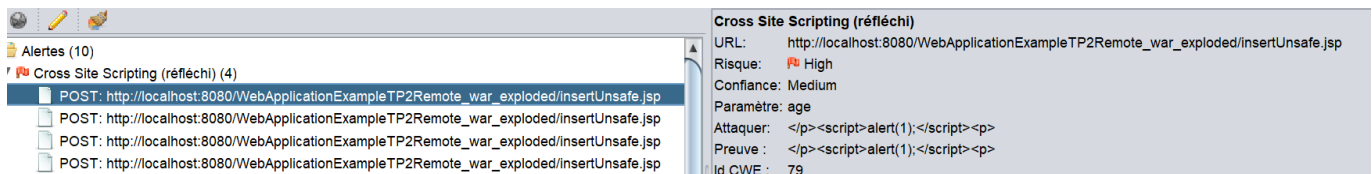
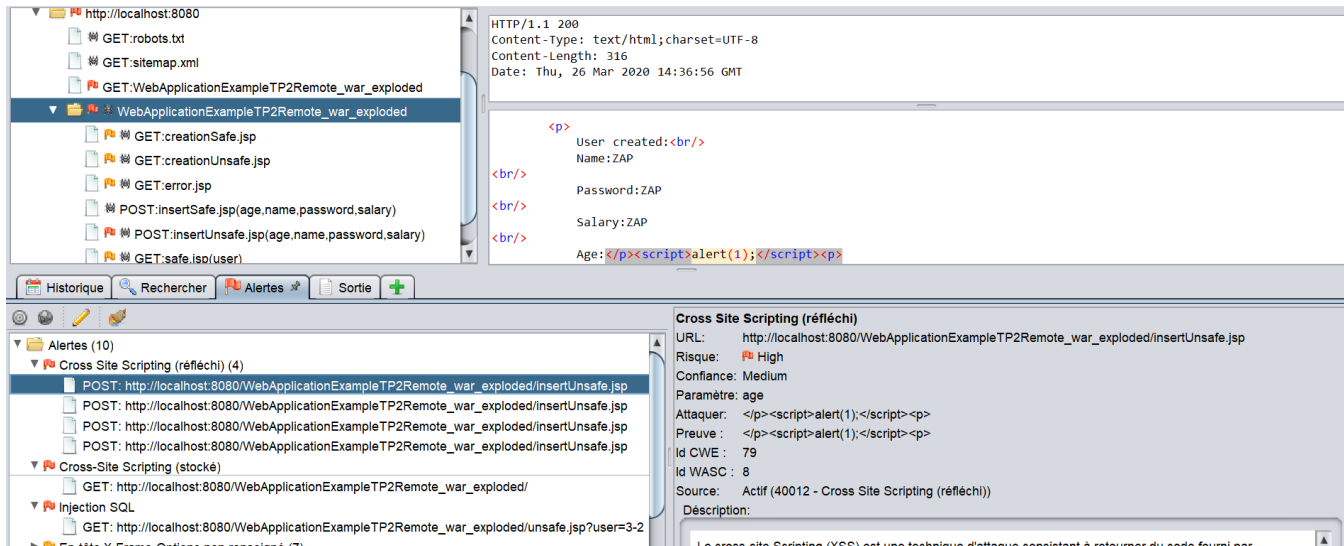
Formulaire *creationUnsafe.jsp*

```
<body>
<form action="insertSafe.jsp" method="post" class="form-example">
  <div class="form-example">
    <label>Enter your name: </label>
    <input type="text" name="name"><br/>
    <label>Enter your password: </label>
    <input type="password" name="password"><br/>
    <label>Enter your salary: </label>
    <input type="text" name="salary"><br/>
    <label>Enter your age: </label>
    <input type="text" name="age"><br/>
    <input type="submit" value="Register">
  </div>
</form>
```

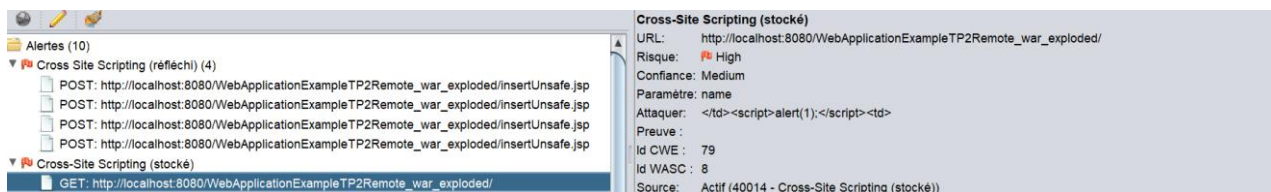
Formulaire *creationSafe.jsp*

Scan d'application

On peut effectuer un scan d'application avec *OWASP ZAP* afin de détecter les vulnérabilités. (*WebApplicationExampleTP2Remote_war_exploded* est l'ancien nom de *GUERIN_war_exploded*)



On voit que plusieurs XSS réfléchis sont possibles sur la page *insertUnsafe.jsp*, ainsi qu'une XSS permanente sur la page *index.jsp*, ce qui est logique car cette dernière va afficher le contenu de la table, donc si une entrée de la table est une injection, alors la page va l'exécuter.



La page *insertUnsafe.jsp* se contente de récupérer ce que le formulaire de *creationUnsafe.jsp* lui envoie, sans vérification, et la page va ensuite insérer ces données après les avoir chiffrées dans la table, et va aussi afficher les détails de l'utilisateur créé :

```
String db = Constants.database;
String table = Constants.table;
Connection connection = null;

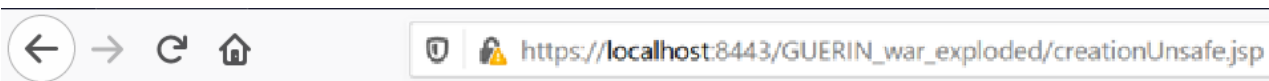
try {
    String nameForm = request.getParameter("name");
    String passwordForm = request.getParameter("password");
    String salaryForm = request.getParameter("salary");
    String ageForm = request.getParameter("age");
    String roleForm = request.getParameter("role");

    if(connection == null || connection.isClosed()){
        connection = DatabaseConnection.getConnection();
    }

    String sqlStatement = "INSERT INTO "+db+"."+table+"(name,password,salary,age,role) values(?, ?, ?, ?, ?)";
    PreparedStatement preparedStatement = connection.prepareStatement(sqlStatement);
    try{preparedStatement.setString(1,nameForm);}catch (Exception e){preparedStatement.setString(1,"");}
    try{preparedStatement.setString(2,Encryption.hashPassword(passwordForm));}catch (Exception e){preparedStatement.setString(2,"");}
    try{preparedStatement.setInt(3, Integer.parseInt(salaryForm));}catch (Exception e){preparedStatement.setInt(3,0);}
    try{preparedStatement.setInt(4, Integer.parseInt(ageForm));}catch (Exception e){preparedStatement.setInt(4,0);}
    try{preparedStatement.setString(5,roleForm);}catch (Exception e){preparedStatement.setString(5,"");}
    int result = preparedStatement.executeUpdate();
    preparedStatement.close();
}
```

```
<p>
  User created:<br/>
  Name:<%out.println(nameForm);%><br/>
  Password:<%out.println(passwordForm);%><br/>
  Salary:<%out.println(salaryForm);%><br/>
  Age:<%out.println(ageForm);%><br/>
  Role:<%out.println(roleForm);%><br/>
</p>
```

Ainsi lorsque l'on se retrouve sur la page *creationUnsafe.jsp* et que l'on remplit le formulaire en mettant dans l'un des champs l'injection suivante par exemple : `<script>alert('bonjour')</script>`, cela va ouvrir une fenêtre indiquant 'bonjour', prouvant ainsi l'injection XSS réfléchi :



Logout

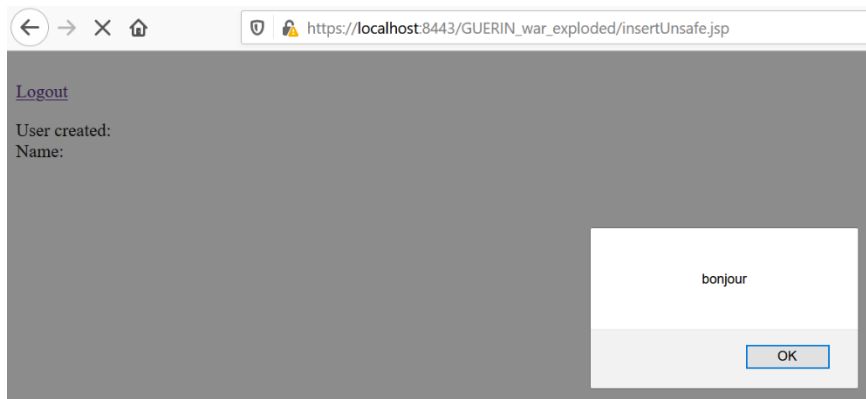
Enter your name:

Enter your password:

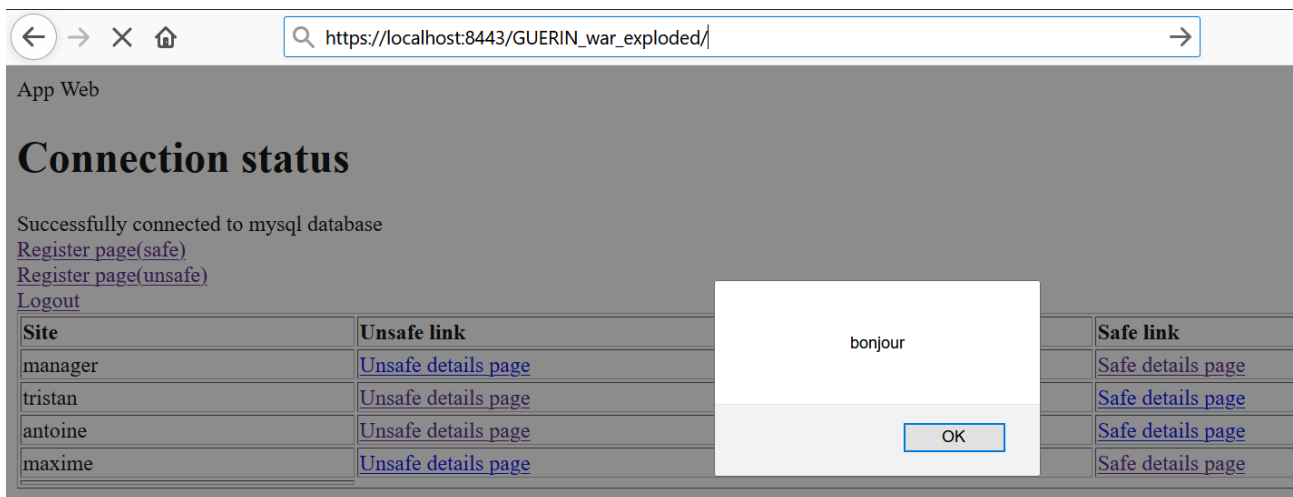
Enter your salary:

Enter your age:

Enter your role:



Lorsque l'on retourne sur l'*index.jsp*, on voit même que la fenêtre est de retour, prouvant ici l'injection permanente, car les données ont été stockées en base de données.



Patch

La page *insertSafe.jsp* patche cette vulnérabilité car, à la suite de l'envoi du formulaire, cette dernière utilise les *regex* afin de vérifier que les données entrées dans les champs ne contiennent des injections :

```
String nameForm = request.getParameter("name");
String passwordForm = request.getParameter("password");
String salaryForm = request.getParameter("salary");
String ageForm = request.getParameter("age");
String roleForm = request.getParameter("role");

String regexName = "[a-zA-Z]*";
if(!Pattern.matches(regexName,nameForm)){
    throw new Exception("Regex error on the name in the form");
}
String regexPassword = "(.)*(<(.)*>){1,}(.)*";
if(Pattern.matches(regexPassword,passwordForm)){
    throw new Exception("Regex error on the password in the form");
}
String regexInt = "[0-9]{1,}";
if(!Pattern.matches(regexInt,salaryForm)){
    throw new Exception("Regex error on the salary in the form");
}
if(!Pattern.matches(regexInt,ageForm)){
    throw new Exception("Regex error on the age in the form");
}
if(!Pattern.matches(regexName,roleForm)){
    throw new Exception("Regex error on the role in the form");
}
```

Par exemple, le nom et le rôle d'un nouvel utilisateur ne doit contenir que des lettres de l'alphabet, son salaire et son âge que des nombres, son mot de passe ne peut pas être validé si des chevrons ouvrants et fermants s'y trouvent.

Ainsi, lors d'une tentative d'injection, on sera redirigé vers une page d'erreur.

The image shows two screenshots of a web browser. The top screenshot shows a registration form at the URL `https://localhost:8443/GUERIN_war_exploded/creationSafe.jsp`. The form has fields for 'name', 'password', 'salary', 'age', and 'role', and a 'Register' button. The 'name' field contains the payload `alert('bonjour')</script>`. The bottom screenshot shows an error page at the URL `https://localhost:8443/GUERIN_war_exploded/error.jsp` with the message 'Your request has lead to an error.'

[Logout](#)
Enter your name:
Enter your password:
Enter your salary:
Enter your age:
Enter your role:

[Logout](#) Your request has lead to an error.

Vol de cookie

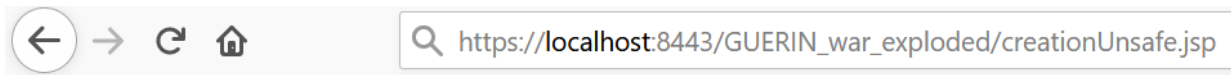
On va enfin voir si l'injection XSS permet de récupérer le cookie et ainsi le vol de session.

Afin de pouvoir récupérer le cookie `JSessionID`, il est nécessaire de rajouter le paramètre `useHttpOnly="false"` au `Connector` dans le fichier `conf/context.xml` du dossier `apache-tomcat`.

```
<Context useHttpOnly="false">
  <!-- Default set of monitored resources. If one of these changes, the -->
  <!-- web application will be reloaded. -->
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <WatchedResource>WEB-INF/tomcat-web.xml</WatchedResource>
  <WatchedResource>${catalina.base}/conf/web.xml</WatchedResource>

  <!-- Uncomment this to disable session persistence across Tomcat restarts -->
  <!--
  <Manager pathname="" />
  -->
</Context>
```

Sur la page *creationUnsafe.jsp*, on voit qu'en injectant la commande `<script>alert(document.cookie)</script>` dans le champ *name*, on récupère le jeton de session.



Logout

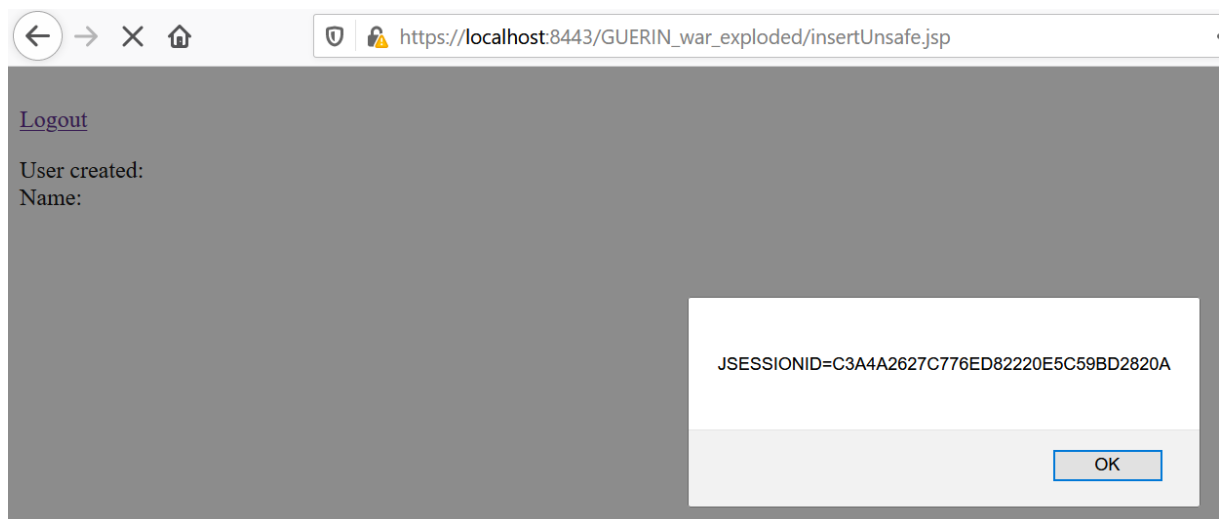
Enter your name:

Enter your password:

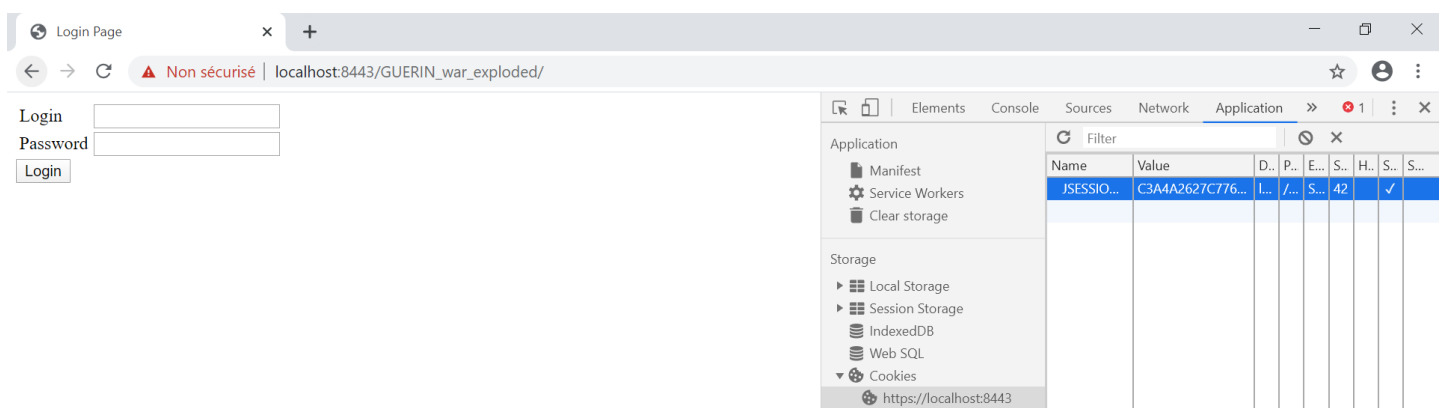
Enter your salary:

Enter your age:

Enter your role:



En récupérant la valeur de ce jeton (C3A4A2627C776ED82220E5C59BD2820A) et en utilisant un autre navigateur, on voit qu'il est possible de récupérer la session de l'utilisateur.



Après rechargement de la page :

The screenshot shows a web browser window with the URL `https://localhost:8443/GUERIN_war_exploded/`. The page title is "List of users". The main content area displays "Connection status" and a message "Successfully connected to mysql database". Below this, there are three links: "Register page(safe)", "Register page(unsafe)", and "Logout". A table with three columns: "Site", "Unsafe link", and "Safe link" is shown. The table contains five rows of data for users: manager, tristan, antoine, maxime, and an empty row. Each row has an "Unsafe link" (e.g., "Unsafe details page") and a "Safe link" (e.g., "Safe details page"). The right side of the screenshot shows the Chrome DevTools "Application" tab. The "Storage" section is expanded, showing "Local Storage", "Session Storage", "IndexedDB", "Web SQL", and "Cookies". The "Cookies" section is further expanded, showing a cookie for "https://localhost:8443" with a value of "C3A4A2627C776..." and a domain of "localhost".

Site	Unsafe link	Safe link
manager	Unsafe details page	Safe details page
tristan	Unsafe details page	Safe details page
antoine	Unsafe details page	Safe details page
maxime	Unsafe details page	Safe details page
	Unsafe details page	Safe details page

On constate que l'on a pu voler sans problème la session.

On pourra imaginer une injection XSS plus poussée qui enverrait la valeur du jeton vers un site, ensuite un attaquant récupérerait cette valeur afin de se recréer un jeton sur un poste tiers et se connecterait directement sans passer par l'authentification à la plateforme.

Remarques

Le chiffrement des données au repos est possible en activant le plugin de chiffrement du *SGBD*, il permet ensuite de choisir quelles tables seront chiffrées.

Dès lors que le service du *SGBD* sera éteint, un attaquant récupérant les données sera alors confronté à des données chiffrées auxquelles il n'aura pas la clé nécessaire pour les déchiffrer.

J'ai pour ma part tenté d'implémenter ce plugin, mais *MySql* me lançait plusieurs erreurs que je n'arrivais pas à corriger, donc j'ai abandonné cette partie.

Annexes

Le fichier script pour la base de données est ci-joint de ce rapport, ainsi que le rapport ZAP sur l'application web. Les différents fichiers qui ont été modifiés durant l'implémentation des aspects sécurité (*https*, authentification, ...) sont aussi donnés pour les remplacer dans le dossier *apache-tomcat* du serveur *Tomcat*. Les fichiers de contexte de pool de connexion seront notamment dans le dossier *afterToUpdate*.

Veuillez suivre le *README* aussi en annexe afin de voir comment lancer l'application web dans un container docker. (La mise en place du *SGBD* dans un volume ou container n'est cependant pas précisée)

Questions

Quelle est la faiblesse d'authentification « basic » ?

La faiblesse de l'authentification « basic » repose notamment sur le fait que les *credentials* sont encodés en base64, ce qui est facilement réversible. Ainsi un attaquant récupérant les *creds* en base64 peut passer par de simples plateformes telles que <https://www.base64decode.org/> afin de retrouver le login et le mot de passe initial.

Les *credentials* sont aussi passés en clair à chaque requête sur le serveur qui les nécessite. C'est pourquoi il est intéressant de lier une authentification à une plateforme et l'utilisation de *https/TLS* afin de chiffrer les données en transit.

Une autre faiblesse est le fait qu'il n'y a pas à proprement parler de « logging out » avec l'authentification « basic », le serveur va stocker les *credentials* en interne pour les renvoyer à chaque requête. Si un utilisateur veut protéger sa session, il doit impérativement quitter son navigateur Internet (il est même préférable qu'il retourne sur la page d'authentification afin de s'assurer de sa « déconnexion »).

Quelle est la faiblesse du chiffrement applicatif ?

Le chiffrement applicatif implique que ce n'est pas le *SGBD* qui chiffre ses données mais le serveur qui va donner des données chiffrées directement au *SGBD* qui va l'insérer dans ses tables.

Cela implique donc que le serveur contient en brute la clé secrète (si on utilise un protocole symétrique style *AES*) ou la paire de clé publique/privée (si on utilise un protocole asymétrique style *RSA*). Cela pose problème car un attaquant analysant le serveur et récupérant la/les clé(s) (par exemple via *WireShark* si le *https* n'est pas utilisé), peut ensuite *dumper* les tables de la *SGBD* et déchiffrer les valeurs derrière.

Opter pour le chiffrement natif, afin que les données soient chiffrées au repos, résout ce problème.

Comment protéger le compte de connexion à la base de données ?

Le compte de connexion peut être protégé en choisissant particulièrement robuste (au moins 12 caractères avec majuscules, minuscules, ...) et le changer régulièrement (tous les trois mois environ) (Recommandations ANSSI).

Il peut être aussi important de chiffrer la connexion entre le serveur web et la *SGBD* via *SSL* afin que les *credentials* de la *SGBD* ne transitent pas en clair entre les deux.

On peut aussi envisager une authentification à deux facteurs afin d'augmenter la protection.

Comment prendre en charge le point 10 du Top 10 OWASP (Insufficient Logging & Monitoring) ?

La surveillance et la journalisation sont des aspects importants de la sécurisation d'une application web. Elles permettent d'anticiper et de comprendre les erreurs/attaques survenues sur l'application.

Ainsi, il est important pour un serveur (style *apache*) d'établir ses propres règles de connexion à l'application afin de par exemple autoriser un nombre maximal de tentatives de connexion par session, de bloquer certaines IPs temporairement ou indéfiniment. Cela permettra notamment de se protéger d'attaques DDOS ou encore de tentatives de brute-force.

Sources

<https://beuss.developpez.com/tutoriels/tomcat/authentification/formulaire/>

<https://docs.oracle.com/cd/E19226-01/820-7627/bncbk/index.html>

<http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html>

https://www.youtube.com/watch?v=RaEG_DOpNPc

https://www.youtube.com/watch?v=ke1SgU_HY80

<https://www.youtube.com/watch?v=AYwNU1Zzdr4>

<https://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html#Introduction>

<https://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html>

<https://javaee.github.io/tutorial/security-webtier002.html#BNCBM>

<https://stackoverflow.com/questions/33412/how-do-you-configure-httponly-cookies-in-tomcat-java-webapps>

<https://security.stackexchange.com/questions/988/is-basic-auth-secure-if-done-over-https>

<https://www.base64decode.org/>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication>

<https://security.stackexchange.com/questions/67427/what-are-the-disadvantages-of-implementing-http-authentication-in-a-web-applicat>

<https://www.avajava.com/tutorials/lessons/how-do-i-log-out-of-an-application-that-uses-form-authentication.html?page=2>

<https://www.concretepage.com/java-ee/jsp-servlet/form-based-authentication-in-jsp-using-tomcat>

<https://www.ssi.gouv.fr/guide/mot-de-passe/>

<https://stackoverflow.com/questions/24677949/why-session-is-not-null-after-session-invalidate-in-java>

https://miro.medium.com/max/2844/1*qWAFJ0WnyExJw37sQcR3xQ.png

http://linux-sxs.org/internet_serving/c619.html

<https://www.avajava.com/tutorials/lessons/how-do-i-use-a-jdbc-realm-with-tomcat-and-mysql.html>

<https://stackoverflow.com/questions/39967289/how-to-use-digest-authentication-in-tomcat-8-5>