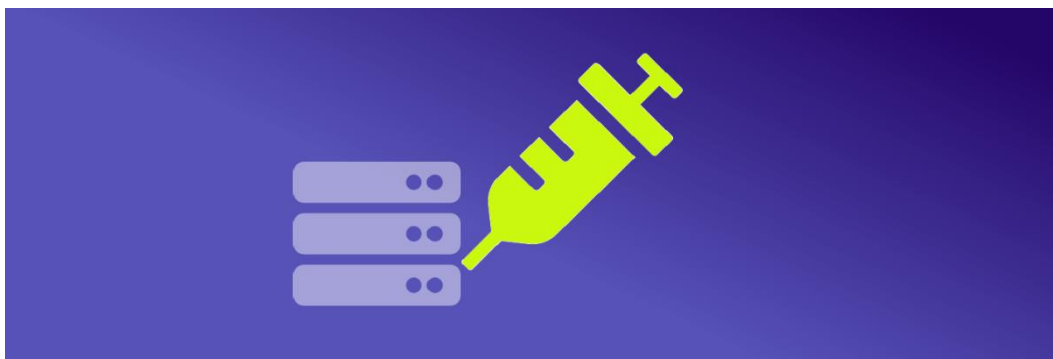


27 MARS 2020



## TP2 REMOTE

SYNTHESE

TRISTAN GUERIN

ENSIBS

Cybersécurité du Logiciel A2

Description	3
Base de données	3
Application	5
Injections & Patch	16
Scan d'application	17
Patch	20
Vol de cookie	21
Annexes	23
Questions	23
Sources	25

## 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 cinq champs différents ('id', 'name', 'password', 'salary' et 'age').

Les noms des champs sont les mêmes que pour le premier TP, mais cette fois-ci, tous les champs (hormis 'id') sont de type *chaîne de caractères* car nous allons chiffrer les valeurs à insérer dans la table via le protocole AES.

En effet, la base de données représente les *données au repos*, ainsi, en chiffrant au préalable les entrées, un attaquant réussissant à récupérer la table ne pourra pas la déchiffrer tant qu'il n'aura pas la clé secrète.

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

(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(250)  | YES  |     | NULL    |                |
| password | varchar(250) | YES  |     | NULL    |                |
| salary | varchar(250)  | YES  |     | NULL    |                |
| age   | varchar(250)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.02 sec)

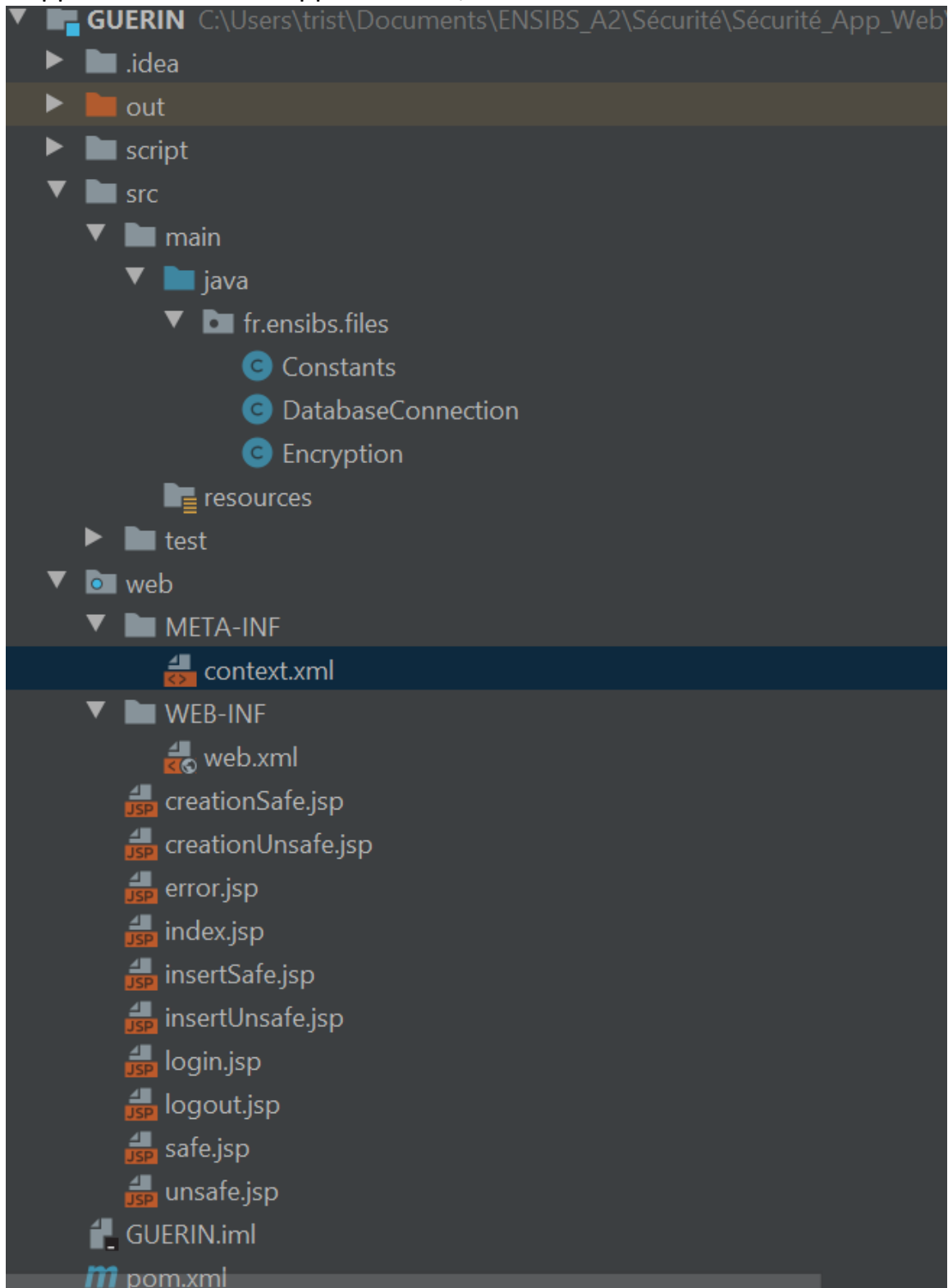
mysql> select * from GUERIN.user;
+-----+-----+-----+-----+-----+
| id | name | password | salary | age |
+-----+-----+-----+-----+-----+
| 1 | NGyDPZYcEkiM5OMjgLvVwA== | sXtiUP5u9L3s9YR/S7GiuQ== | 1pEvz9fSWD1gSEwjL3p8uA== | r8LS43ENuBL1mtS7ch03jw== |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

*On voit ici que les champs 'name', 'password', 'salary' et 'age' sont de type chaîne de caractères*

*On a aussi un exemple d'une entrée dans la table : chaque valeur initiale des champs a été chiffré par AES*

## 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

chiffrer/déchiffrer les entrées. 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.

```
/**
 * Class gathering all the constants
 * @author Tristan Guerin
 * @version 1
 */
public class Constants {

    /**
     * Name of the database
     */
    public static final String database = "GUERIN";

    /**
     * New table of the database
     */
    public static final String tableEncrypted = "user";

    /**
     * Password of the AES algorithm to encrypt/decrypt the database's table
     */
    public static final String AESpassword = "sqlfdgjiwolalolawolalolasqlfdgji";

    /**
     * AES key
     */
    public static final SecretKeySpec AESKey = new SecretKeySpec(Constants.AESpassword.getBytes(), algorithm: "AES");
```

Elle contient notamment le nom de la base de données et de sa table contenant les informations chiffrées, ainsi que le mot de passe AES servant à chiffrer/déchiffrer ces données.

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* de l'utilisateur créé via *scriptSQL.sql*).

Enfin, la classe *Encryption* va proposer des méthodes de chiffrement/déchiffrement selon le protocole symétrique AES.

```
public static String encrypt(String strToEncrypt, SecretKeySpec secretKeySpec){
    try{
        byte[] iv = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
        IvParameterSpec ivspec = new IvParameterSpec(iv);

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivspec);
        return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets.UTF_8)));
    }catch (Exception e){
        System.out.println("Error while encrypting: " + e.toString());
        e.printStackTrace();
    }return null;
}
```

```
public static String decrypt(String strToDecrypt, SecretKeySpec secretKeySpec){
    try{
        byte[] iv = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
        IvParameterSpec ivspec = new IvParameterSpec(iv);

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
        cipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivspec);
        return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));
    }catch (Exception e){
        System.out.println("Error while decrypting: " + e.toString());
        e.printStackTrace();
    }return null;
}
```

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.

Etant donné que la table *user* ne contient plus directement les valeurs mais leurs chiffrements via AES, les trois fichiers *.jsp* ont subi quelques modifications.



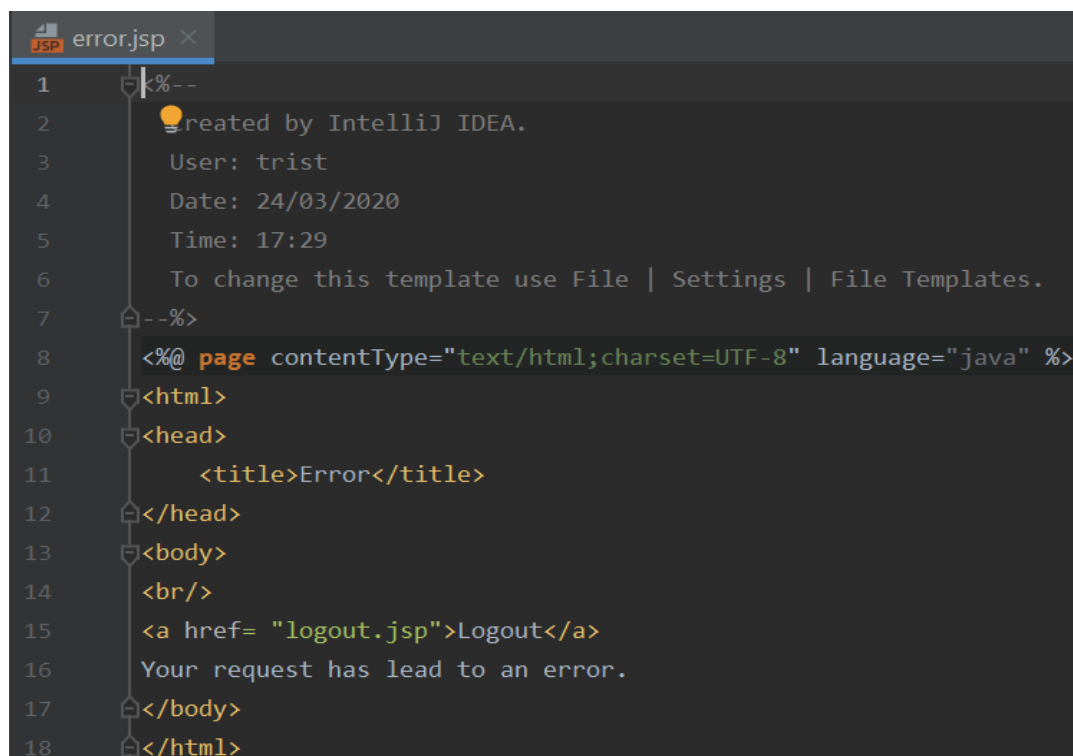
```

<% String id = request.getParameter("user");
String table = Constants.tableEncrypted;
ResultSet resultSet1= null;
Connection connection = null;
try {
    if(connection == null || connection.isClosed()){
        connection = DatabaseConnection.getConnection();
    }
    String sqlStatement = "select * from "+table+" where id = ?";
    PreparedStatement preparedStatement = connection.prepareStatement(sqlStatement);
    preparedStatement.setInt(1,Integer.parseInt(id));
    resultSet1 = preparedStatement.executeQuery();

    while (resultSet1.next()){
        out.println("<div><a>Details :</a>" +
            "<div><a>ID :"+ resultSet1.getString("id")+ "</a></div>" +
            "<div><a>Name :"+Encryption.decrypt(resultSet1.getString("name"),Constants.AESKey)+"</a></div>" +
            "<div><a>Password :"+Encryption.decrypt(resultSet1.getString("password"),Constants.AESKey)+"</a></div>" +
            "<div><a>Salary :"+Encryption.decrypt(resultSet1.getString("salary"),Constants.AESKey)+"</a></div>" +
            "<div><a>Age :"+Encryption.decrypt(resultSet1.getString("age"),Constants.AESKey)+"</a></div>" +
            "</div>");
    }
} catch (Exception e) {
    e.printStackTrace();
    response.sendRedirect("error.jsp");
}

```

Sur cet extrait du fichier *safe.jsp*, on voit que l'on affiche plus les valeurs récupérées via la requête à la BDD mais leurs déchiffrements avec la clé secrète AES



```

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*.

Ainsi dans le fichier *conf/tomcat-users.xml* du dossier *apache-tomcat*, il faut rajouter un rôle et un utilisateur qui seront nécessaire à l'authentification.

On rajoute ici le rôle *manager* et l'utilisateur *manager1* ayant comme mot de passe *manager1*.

```
<role rolename="manager"/>
<user username="manager1" password="manager1" roles="manager"/>
```

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 /* or /wallah-->
    <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>
```

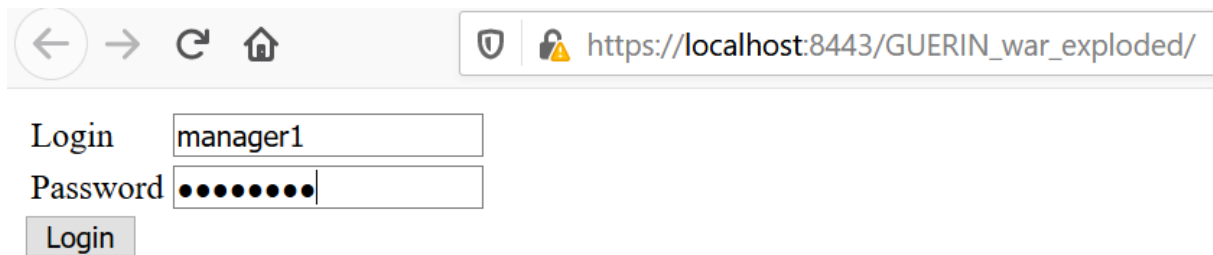
On va alors indiquer quels types de méthode et quelles URLs vont être soumis à une authentification (ici toutes les pages, via le pattern */\**). On précise aussi le rôle d'authentification *manager*.

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/

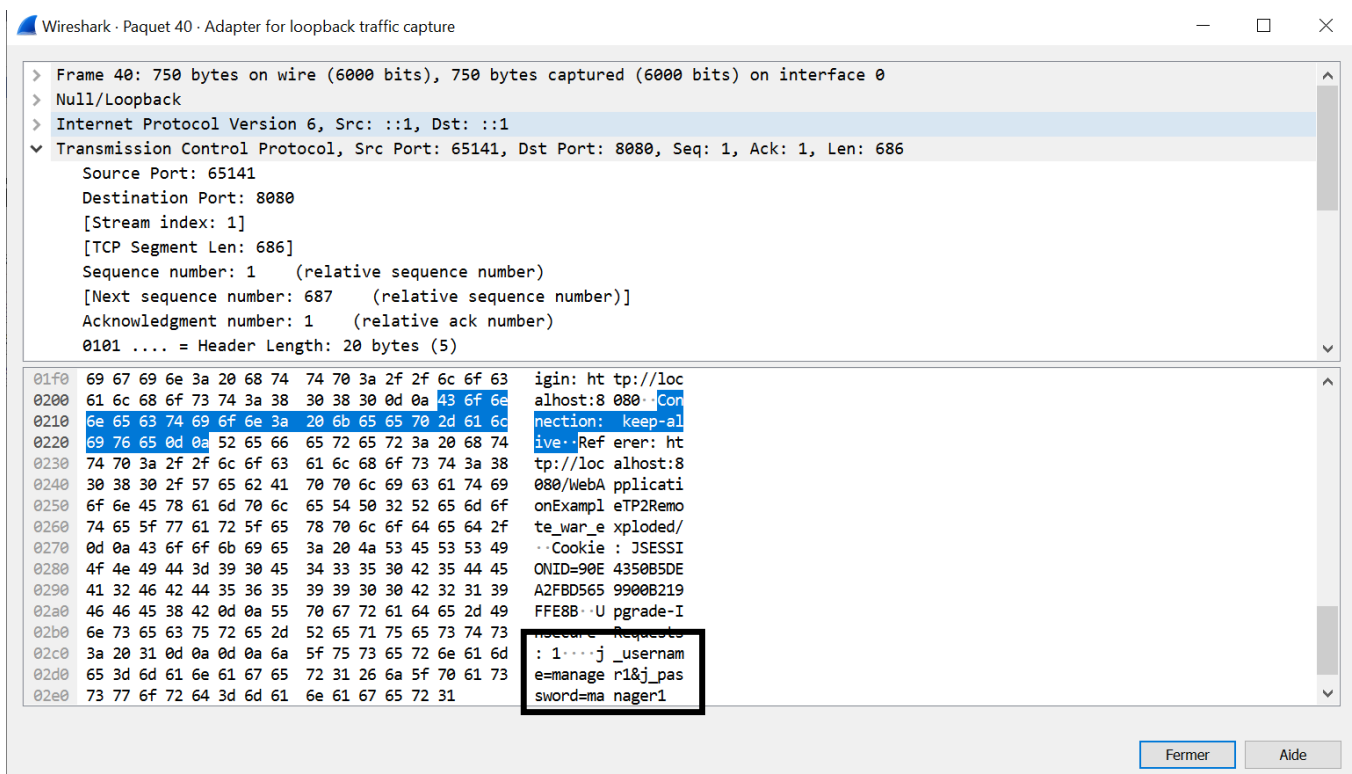
Login

Password

Login

Après avoir mis en place une base de données avec des données en repos chiffrées, 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 :



```
: 1....j _usernam  
e=manage r1&j_pas  
sword=ma nager1
```

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" />
```

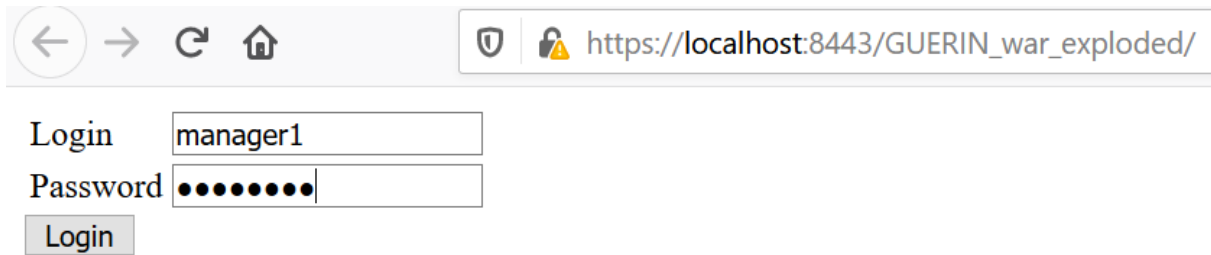
On indique le port, où trouver le fichier contenant la clé, quel est le mot de passe pour la clé (ici on a choisi *manager1*), le protocole SSL, ...

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\\_explored/](https://localhost:8443/GUERIN_war_explored/)



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 → 5000
2426	2020-03-27 00:00:00,697893	127.0.0.1	127.0.0.1	TCP	56	51006 → 8443
2427	2020-03-27 00:00:00,697951	127.0.0.1	127.0.0.1	TCP	56	8443 → 51006
2428	2020-03-27 00:00:00,697996	127.0.0.1	127.0.0.1	TCP	44	51006 → 8443
2431	2020-03-27 00:00:00,699715	127.0.0.1	127.0.0.1	TLSv1...	561	Client Hello
2432	2020-03-27 00:00:00,699756	127.0.0.1	127.0.0.1	TCP	44	8443 → 51006
2439	2020-03-27 00:00:00,717618	127.0.0.1	127.0.0.1	TLSv1...	177	Server Hello
2440	2020-03-27 00:00:00,717667	127.0.0.1	127.0.0.1	TCP	44	51006 → 8443
2441	2020-03-27 00:00:00,718622	127.0.0.1	127.0.0.1	TLSv1...	50	Change Cipher
2442	2020-03-27 00:00:00,718715	127.0.0.1	127.0.0.1	TCP	44	51006 → 8443

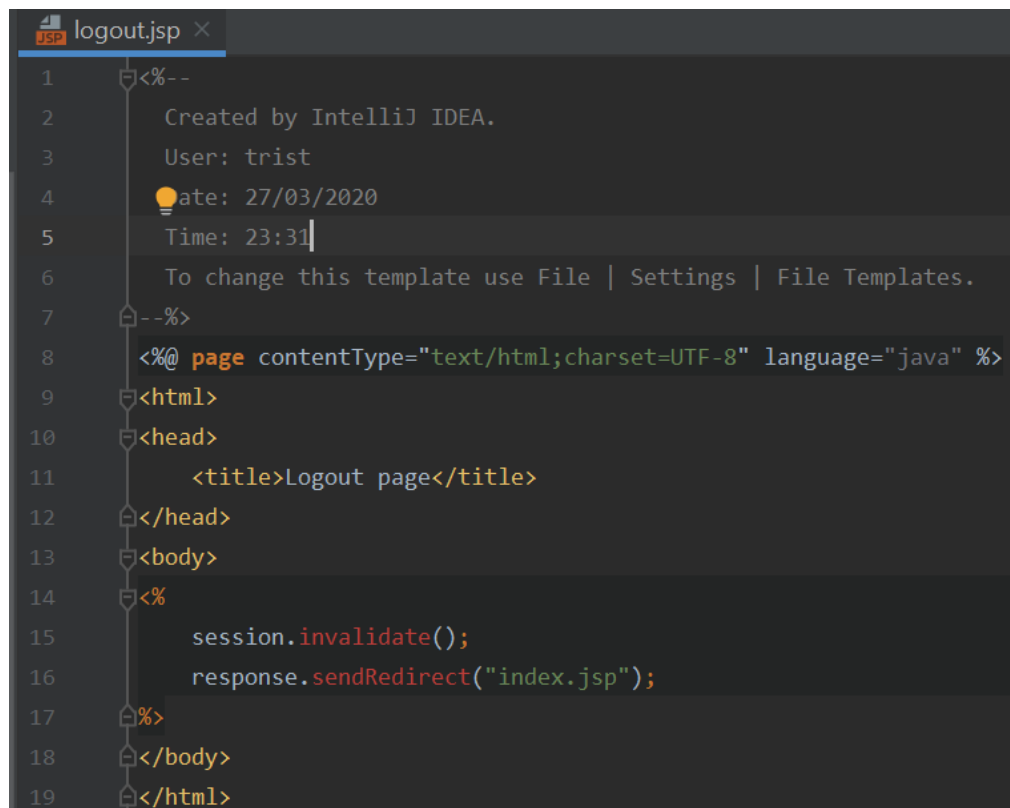
>	Frame 2431: 561 bytes on wire (4488 bits), 561 bytes captured (4488 bits) on interface 0
>	Null/Loopback
>	Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
>	Transmission Control Protocol, Src Port: 51006, Dst Port: 8443, Seq: 1, Ack: 1, Len: 517
>	Transport Layer Security

0090	c0 13 c0 14 00 33 00 39 00 2f 00 35 00 0a 01 00	.....3.9 ./5....
00a0	01 8f 00 00 00 0e 00 0c 00 00 09 6c 6f 63 61 6c	.....local
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 68	.....h2.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.

A screenshot of an IDE window titled 'logout.jsp'. The code is as follows:

```
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.

## Injections & Patch

Après avoir abordé la sécurité des données en repos et 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*

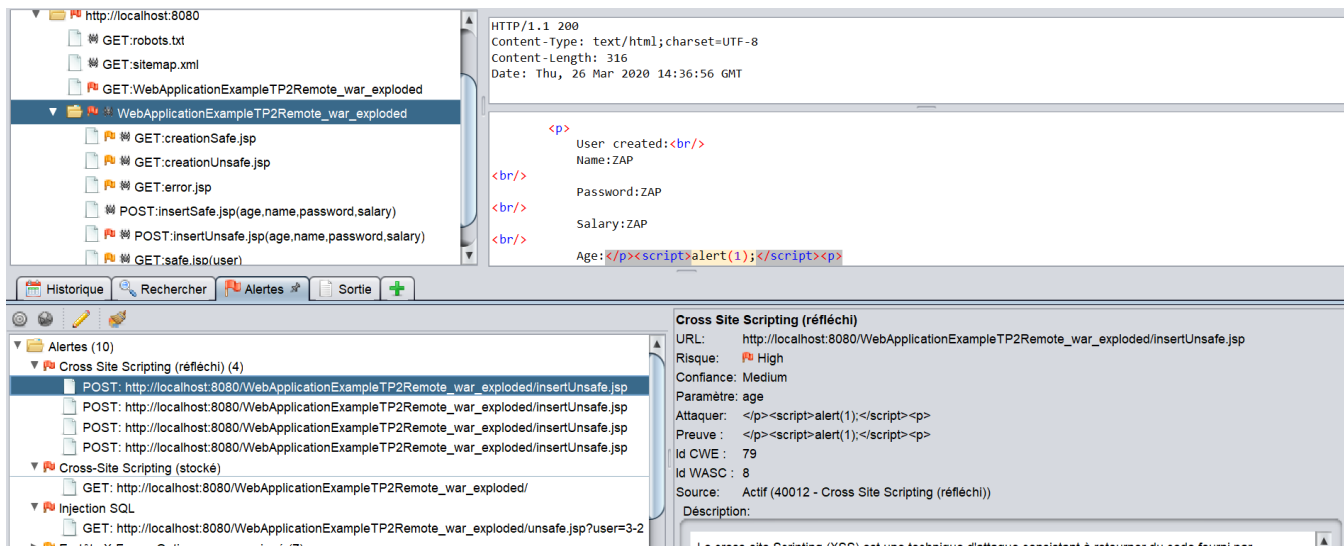
```
<body>
<form action="insertSafe.jsp" method="post" class="form-example">
  <div class="form-example">
    <label>Enter your name: </label>
    <input type="text" name="name"><br/>
    <label>Enter your password: </label>
    <input type="password" name="password"><br/>
    <label>Enter your salary: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

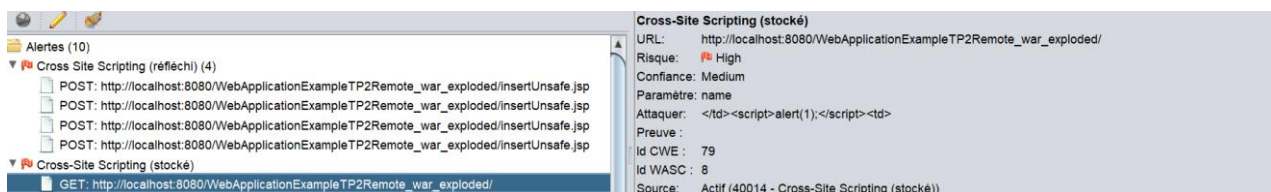


## Scan d'application

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 refléchi sont possibles sur la page *insertUnsafe.jsp*, ainsi qu'une XSS permanent 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.tableEncrypted;
Connection connection = null;

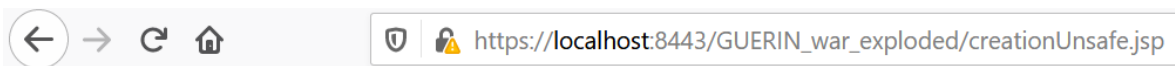
try {
    String nameForm = request.getParameter("name");
    String passwordForm = request.getParameter("password");
    String salaryForm = request.getParameter("salary");
    String ageForm = request.getParameter("age");

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

    String sqlStatement = "INSERT INTO "+db+"."+table+" (name,password,salary,age) values(?, ?, ?, ?)";
    PreparedStatement preparedStatement = connection.prepareStatement(sqlStatement);
    try{preparedStatement.setString(1,Encryption.encrypt(nameForm,Constants.AESKey));}catch (Exception e){preparedStatement.setString(1, Encryption.encrypt("",Constants.AESKey));}
    try{preparedStatement.setString(2,Encryption.encrypt(passwordForm,Constants.AESKey));}catch (Exception e){preparedStatement.setString(2,Encryption.encrypt("",Constants.AESKey));}
    try{preparedStatement.setString(3,Encryption.encrypt(salaryForm,Constants.AESKey));}catch (Exception e){preparedStatement.setString(3,Encryption.encrypt("0",Constants.AESKey));}
    try{preparedStatement.setString(4,Encryption.encrypt(ageForm,Constants.AESKey));}catch (Exception e){preparedStatement.setString(4,Encryption.encrypt("0",Constants.AESKey));}
    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/>
</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 :



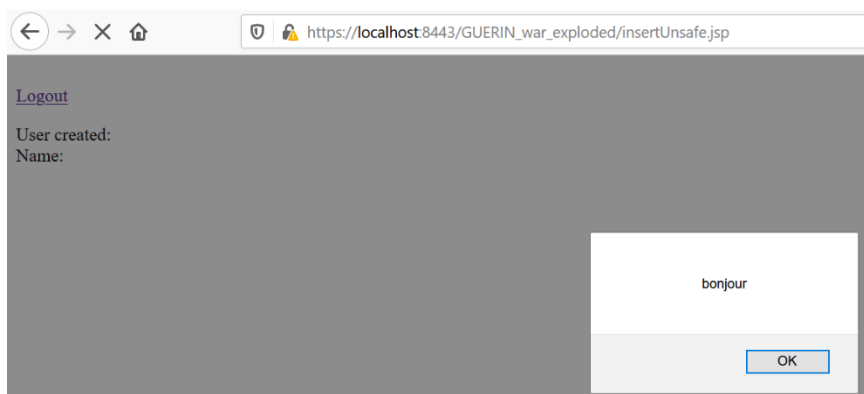
### Logout

Enter your name:

Enter your password:

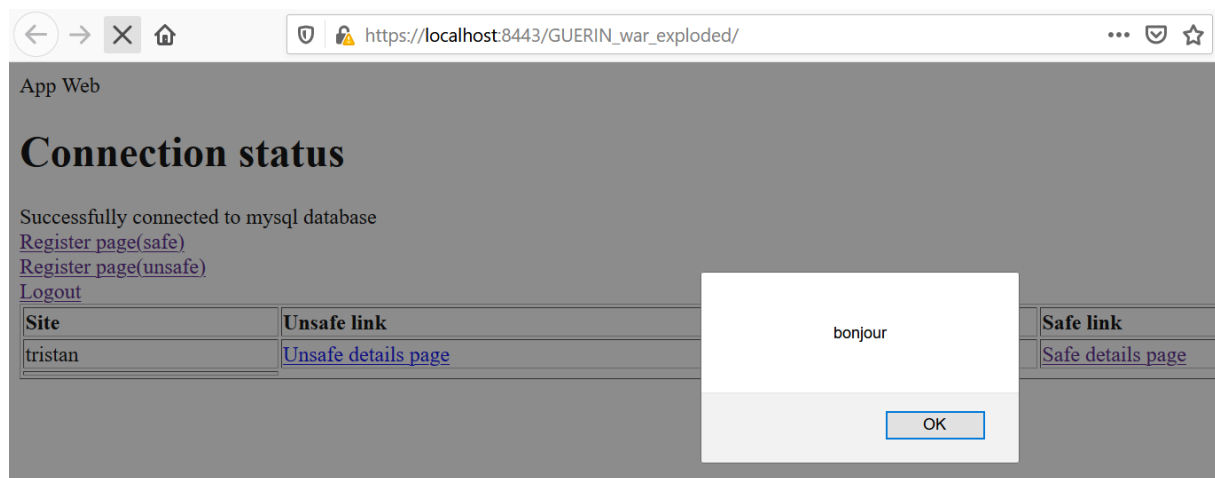
Enter your salary:

Enter your age:



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

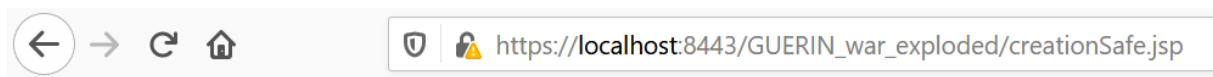
La page *insertSafe.jsp* patche cette vulnérabilité car, à la suite de l'envoi du formulaire, cette dernière utilise les *regex* afin de vérifier que les données entrées dans les champs ne contiennent des injections :

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

String 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");
}
```

Par exemple, le nom 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.



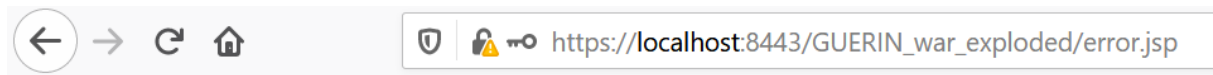
[Logout](#)

Enter your name:

Enter your password:

Enter your salary:

Enter your age:



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

## Vol de cookie

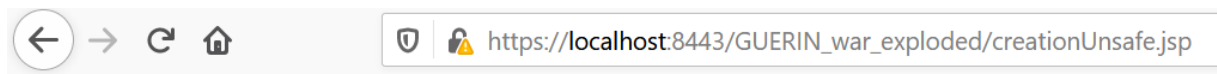
On va enfin voir si l'injection XSS permet de récupérer le cookie et ainsi le vol de session.

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 injecter la commande `<script>alert(document.cookie)</script>` dans le champ *name*, on récupère le jeton de session.



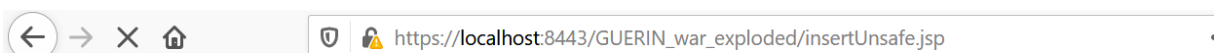
[Logout](#)

Enter your name:

Enter your password:

Enter your salary:

Enter your age:

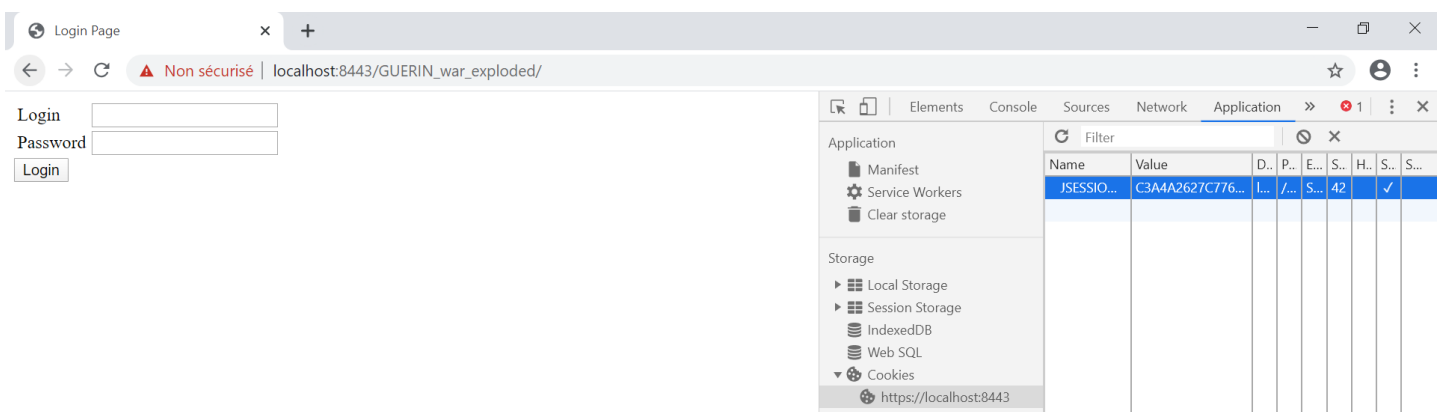


[Logout](#)

User created:

Name:

JSESSIONID=C3A4A2627C776ED82220E5C59BD2820A



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 at `localhost:8443/GUERIN_war_explored/`. The page title is "List of users". The main content area shows "Connection status" with the message "Successfully connected to mysql database". Below this are three links: "Register page(safe)", "Register page(unsafe)", and "Logout". A table with three columns is displayed:

Site	Unsafe link	Safe link
tristan	<a href="#">Unsafe details page</a>	<a href="#">Safe details page</a>
Ã€Ã€Ã€	<a href="#">Unsafe details page</a>	<a href="#">Safe details page</a>
Ã€Ã€Ã€	<a href="#">Unsafe details page</a>	<a href="#">Safe details page</a>

The browser's developer tools are open, showing the Application tab. The session storage contains a JSESSIONID with the value C3A4A2627C776...

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.

## 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*.

## 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 ?

### 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/authentication/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=RaEG_DOpNPc)

[https://www.youtube.com/watch?v=ke1SgU\\_HY80](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](https://miro.medium.com/max/2844/1*qWAFJ0WnyExJw37sQcR3xQ.png)