

Department of Computer Architecture (DAC)

Universitat Politècnica de Catalunya (UPC)

Security in Information Systems (SSI)

Laboratory Booklet

Manuel García-Cervigón, Jordi Nin and
Sergio Ricciardi

November, 2013



Vulnerabilities in web applications

Contents

1.1	Objective	1
1.2	Start the LiveCD	2
1.3	Exercises	6
1.3.1	Parameter validation	6
1.3.2	Session administration and authentication	12
1.3.3	Cross Site Scripting	15
1.3.4	SQL Injection	17
1.4	References	19

1.1 Objective

Vulnerabilities in web applications are responsible for most of the security violations in computer networks. Every time more often, the attacks are addressed to applications such as Internet shopping, web forms, as well as the authentication and access points to protected web pages and dynamic contents from linked databases with transactions and information requests.

When we talk about web application vulnerabilities we are not talking about operating system or http server vulnerabilities (version update, patches, etc) but about the vulnerabilities of the software on top of them. Such vulnerabilities are directly related to the logic, code scripting and content of the web application.

Being able to detect such vulnerabilities provides us with more security as well as to be able to provide more control and quality to our software products.

The objective of this session is to study some of the main vulnerabilities found in web applications, study some basic ways to perform attacks and understand the origin of such vulnerabilities and how to be able to avoid them.

We will use the following applications for this session:

WebGoat is a J2EE application developed by OWASP (The Open Web Application Security Project) and based on Tomcat. It is an insecure application and it is basically its purpose. The objective is to use it as an introduction to different attacks directed to web applications (test environment). It has different lessons that provide us with help and information to understand and to be able to overcome them.

WebScarab is a framework to analyze web applications developed by OWASP. It uses http and https and it can be used as a proxy to study a web page requests and responses, review and modify them before they get to the client or the server.

Both applications can be found on a LiveCD named OWASP Live CD Education Project (LabRat), that we are going to use. This Live CD can be downloaded from <http://appseclive.org/content/downloads>. However, for this lab, we will download the images directly from the FIB racó.

The vulnerabilities that we are going to see are:

Hidden field authentication: how to obtain additional information from web applications and modify the client's generated requests or server responses to be able to perform the attack.

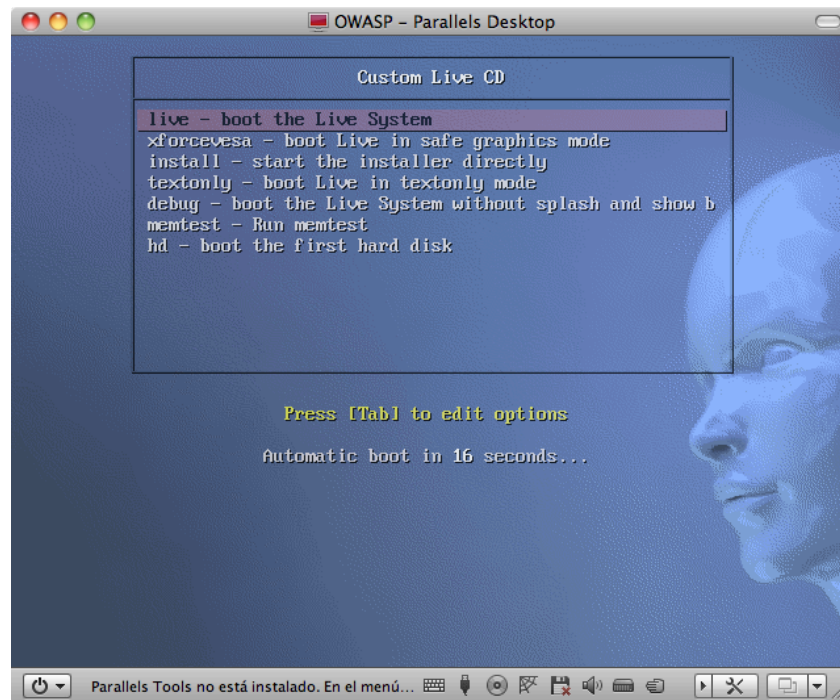
Weak session identification: we will see the dangers of a weak authentication, and in this case, how to impersonate another user by means of a session cookie.

Cross-Site Scripting (XSS): is an attack based in the vulnerabilities exploit of the embedded HTML validation. It takes advantage on the lack of filtering mechanisms of the input fields, allowing the data input and transfer without any validation, being able to generate malicious command sequences or scripts.

SQL Injection: it is a vulnerability found at the input data validation of a database associated to a web application. The origin is the incorrect filtering of variables used in the application code that perform SQL sentences.

1.2 Start the LiveCD



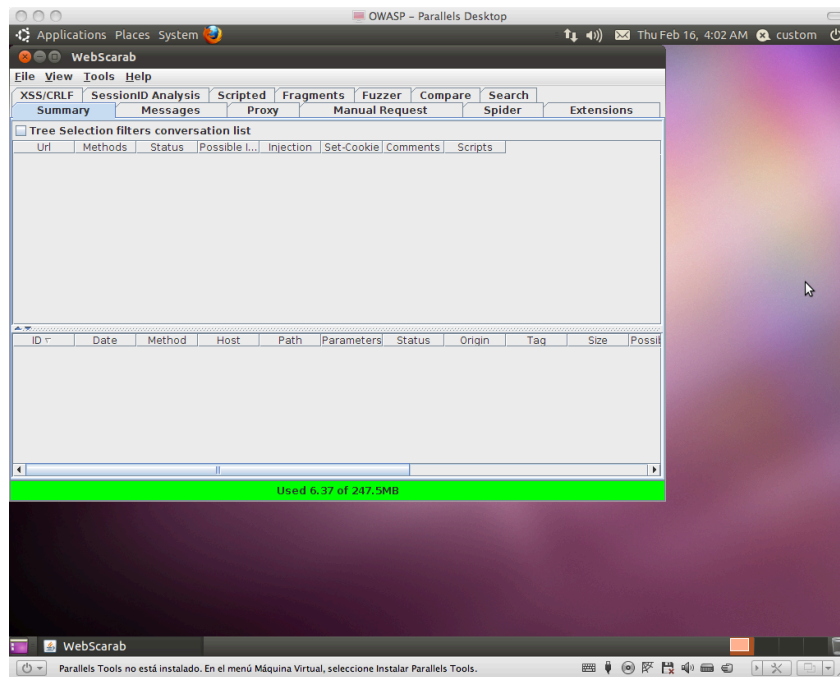


1. **Keyboard layout:** 



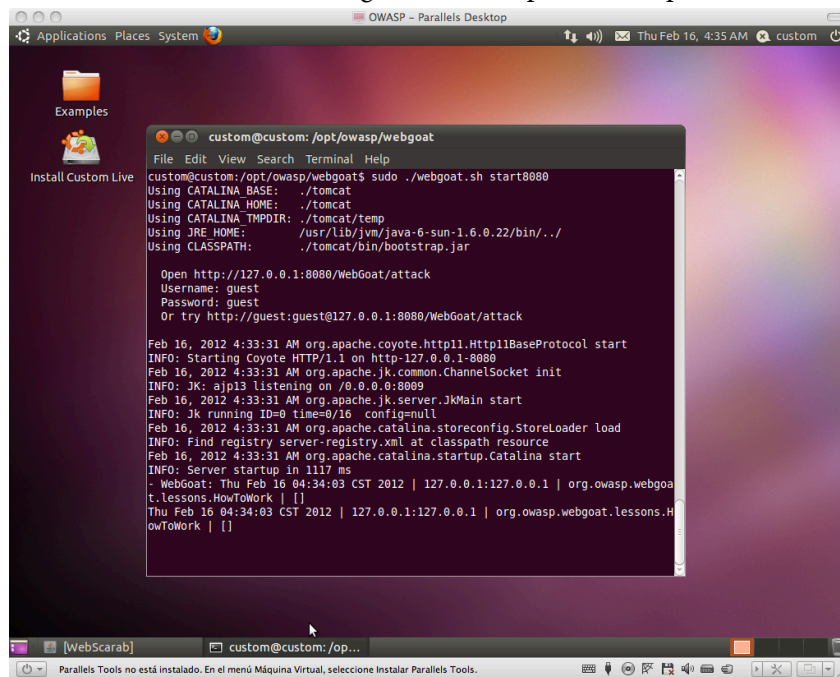

2. **Screen resolution:** The default resolution is quite low; increase it in System -> Preferences -> Monitors

3. **Start the WebScarab application:** When starting the graphic interface go to the OWASP -> Proxies -> WebScarab and click to see the following screen:



4. Start WebGoat application:

If error: Do this before Starting Webscarab (previous step).



- Open a shell
- Change to the directory where the WebGoat application is installed

```
# cd /opt/owasp/webgoat
```

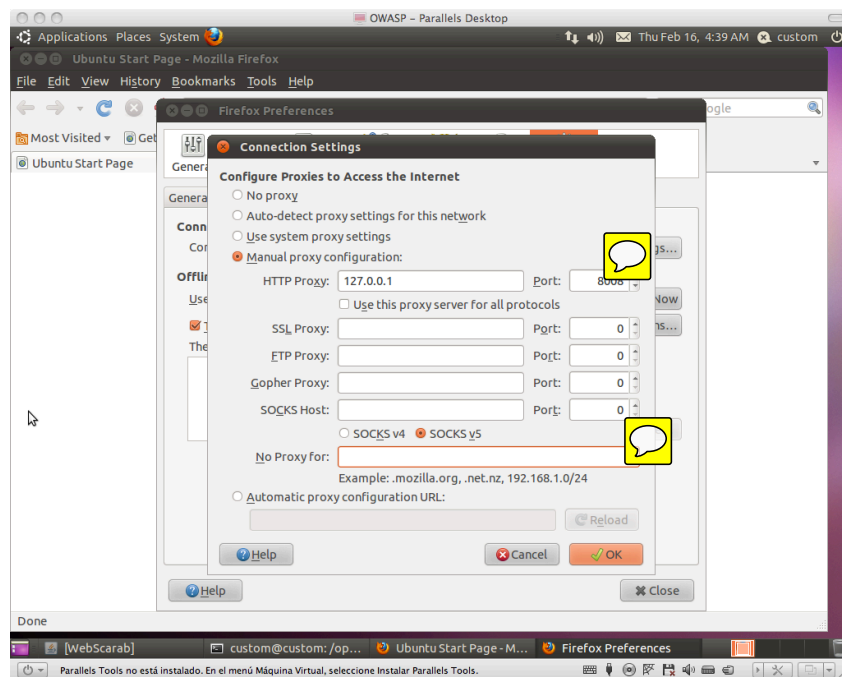
- Start the application on port 8080

```
#sudo ./webgoat.sh start8080 &
```

- Watch the messages appearing while the application starts. When the following message appears the application will be running

```
INFO: Server startup in 4350 ms
```

5. **Web browser configuration:** Open the web browser and configure it to use a local proxy, in our case WebScarab, using port 8008 by default. Go to 'Edit -> Preferences -> Advanced -> Network -> Settings...' (may change depending on the Firefox version)



Configure manually the proxy option as shown in the figure.

Warning! observe that the field 'No Proxy for' doesn't have neither '127.0.0.1' nor 'localhost'.

Disable Firefox automatic checks for updates in 'Edit -> Preferences -> Advanced -> Update'.

At the browser address bar type `http://127.0.0.1:8080/webgoat/attack`

Remember: *user: guest password: guest*

Click on the 'Start WebGoat' button.

1.3 Exercises

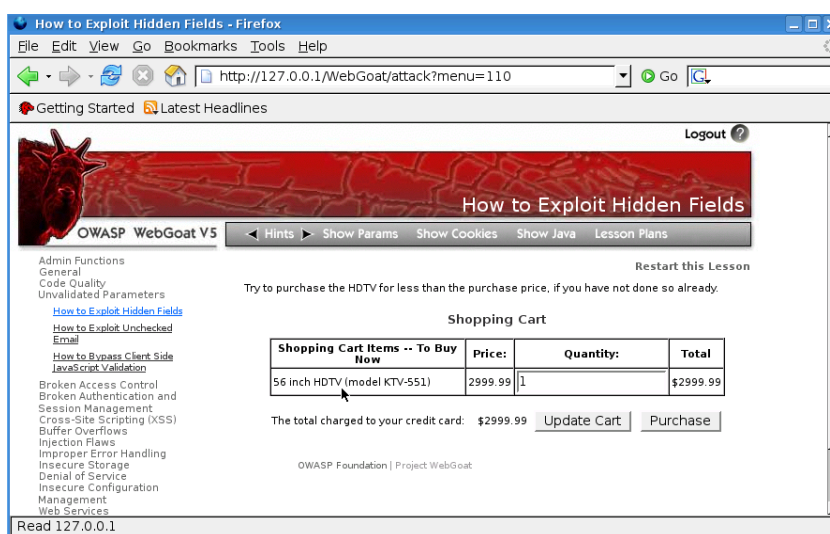
1.3.1 Parameter validation

We will see the danger of not validating input parameters on a Web application or doing a poor or incorrect validation. On 'Parameters tampering' we will find three exercises:

1.3.1.1 Hidden fields

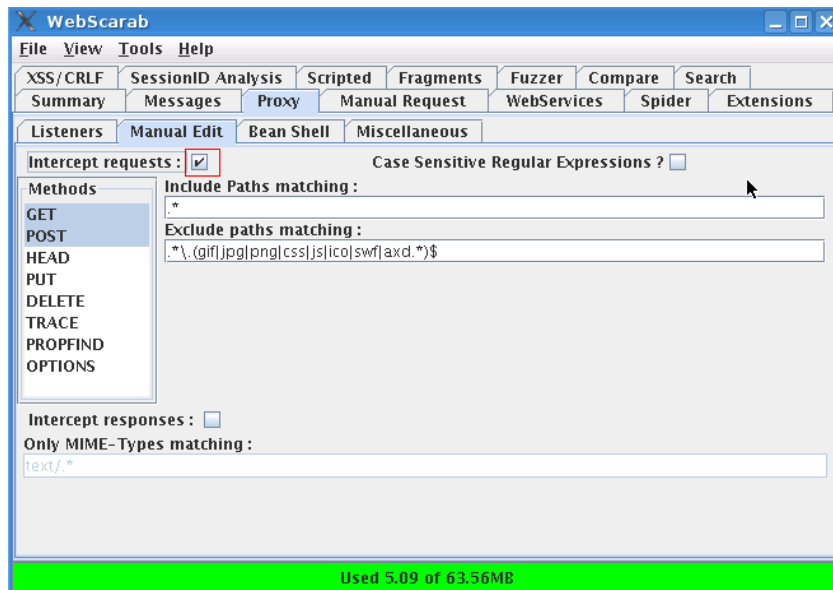
Access to WebGoat's lesson 'Exploit Hidden Fields' in 'Parameter Tampering'. Its goal is to buy from a web page for a lower price.

Observing the web page we can see that the field 'Price' can't be modified. Try several times 'Update cart' or 'Purchase' or observing the code clicking 'Show Java' to modify the product's price. Have you found anything?

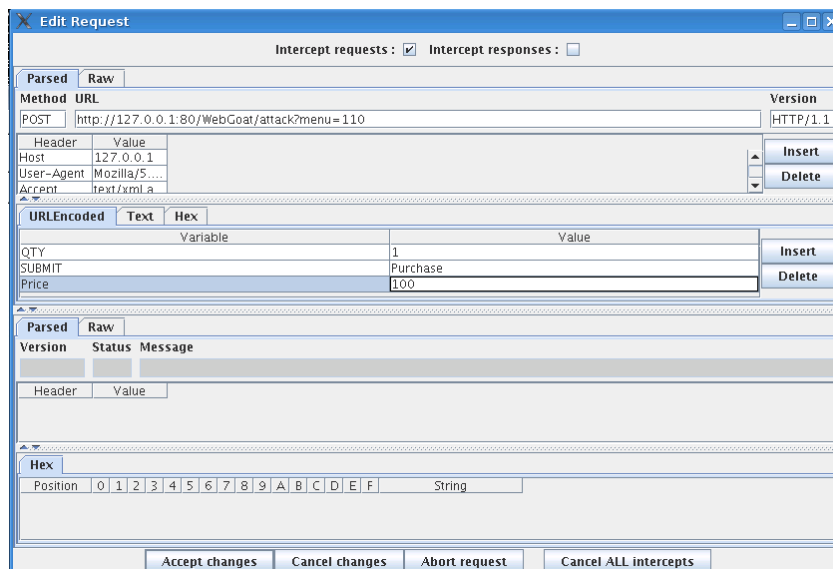


Let's now watch the request sent to the server when we try to buy. Follow the steps:

- Go to WebScarab and tick 'Intercept Requests' inside 'Proxy->Manual Edit' tab:

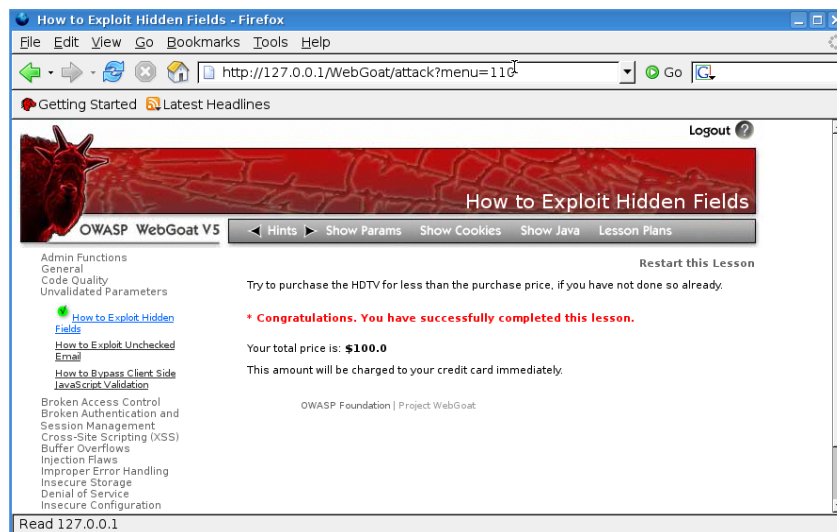


- Once the requests reception has been activated, go back to WebGoat and try to buy clicking on 'Purchase'.



- You will then see a new WebScarab window popping up with the intercepted request. If we take a look at tab 'URLEncoded' we'll find variables (QTY, SUBMIT, Price). One of the variables refers to the purchase price, try to modify the price at column 'Value', disable the requests interception (clicking again on 'Intercept Requests') and click on 'Accept Changes'.

We have been able to modify the purchasing price.



1.3.1.2 e-mail not validated

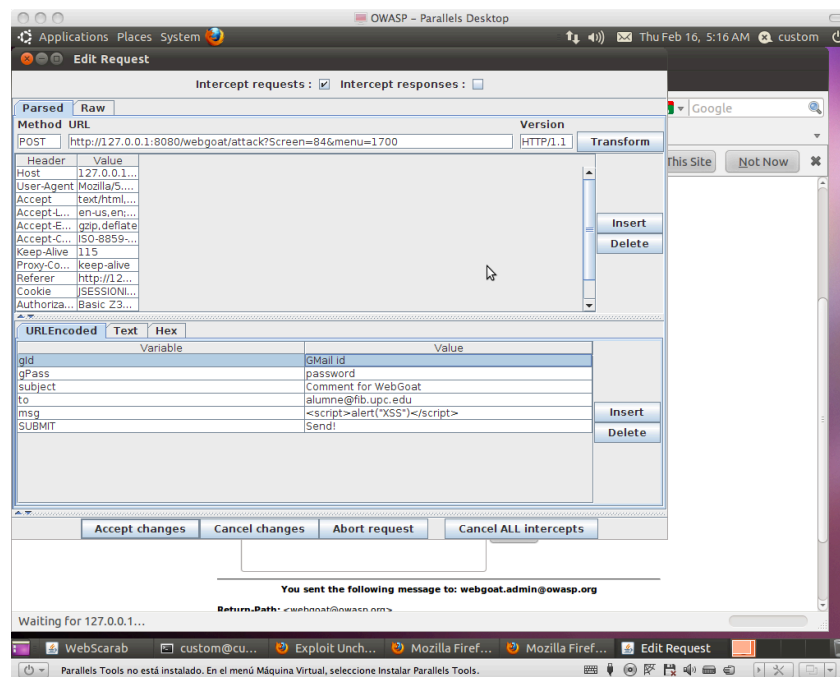
Go to WebGoat lesson 'Exploit unchecked mail' in 'Parameter Tampering'. Now the goal is to be able to change the e-mail address where the comments typed at the web page are sent. Enter some comments and see the code by clicking on 'Show Java' to be able to modify the e-mail address. Have you found anything?

Type a malicious script like:

```
<script>alert ("XSS")</script>
```

into the comments field and send. Observe that you are able to add your own script and execute whatever you want.

Now, let's change the e-mail address field. This can be accomplished by intercepting the request with webScarab and changing the hidden field "to" from webgoat.admin@owasp.org to alumne@fib.upc.edu (don't worry, no email is actually sent during this test).

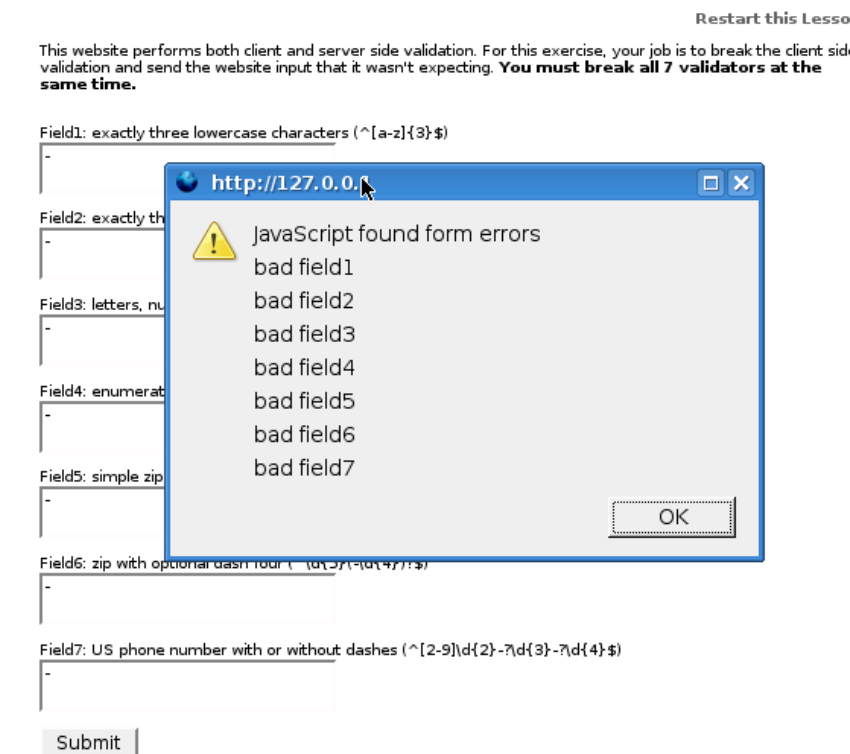


1.3.1.3 Avoid validations on the client side

Go to WebGoat lesson 'Bypass Client Side JavaScript Validation' in 'Parameter Tampering'. The goal is to avoid validation implemented on the client side of the application.

This web page sends seven values to the web server that need to match regular expressions validated locally. Try introducing correct and incorrect values on every field and send them clicking on 'Submit' or observing the code clicking on 'Show Java' to be able to find the code implementing the fields validation. Have you found anything?

What would happen if we sent incorrect values in all the fields? For instance a dash (-)



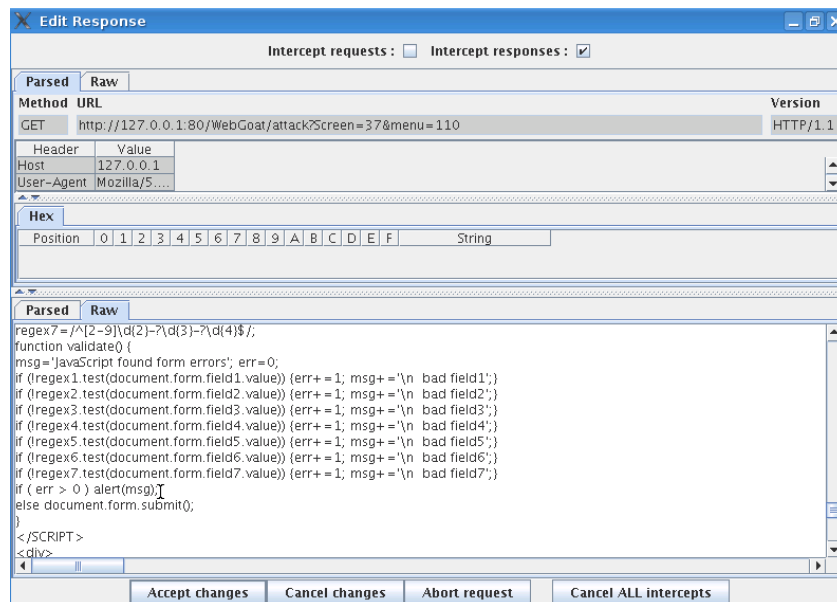
Data is being validated at the client side and we can't send it to the server. Looking at the code you can find the section implementing the validation:

```
if (!pattern1.matcher(param1).matches()){
    err++;
    msg += "<BR>Server side validation violation:
           You succeeded for Field1.";
}
if (!pattern2.matcher(param2).matches()){
    err++;
    msg += "<BR>Server side validation violation:
           You succeeded for Field2.";
}
...
if (err > 0){
    s.setMessage(msg);
}
```

This code is downloaded when requesting the web page. Let's try to skip the validation:

We need to modify WebScarab configuration to be able to intercept server's responses. On tab 'Proxy', check 'Intercept Responses' and verify that 'Intercept Requests' is disabled.

Now reload the browser's page to issue a new request. You will see a new WebScarab window with the intercepted response. If we click on 'Raw' tab from the lower half window we'll be able to see the server's response and there we will find the code to validate the fields that we want to avoid.



Edit the code erasing the validation code: lines between “*msg='JavaScript found form errors'*” and “*(if (err>0 alert(msg); else)*”.

Uncheck 'Intercept responses', click on 'Accept Changes', go back to the web page and click 'Submit' to send incorrect data to the server.

We have been able to avoid the client's validation and to send incorrect information to the server.

[Restart this Lesson](#)

This website performs both client and server side validation. For this exercise, your job is to break the client side validation and send the website input that it wasn't expecting. **You must break all 7 validators at the same time.**

*** Server side validation violation: You succeeded for Field1.
 Server side validation violation: You succeeded for Field2.
 Server side validation violation: You succeeded for Field3.
 Server side validation violation: You succeeded for Field4.
 Server side validation violation: You succeeded for Field5.
 Server side validation violation: You succeeded for Field6.
 Server side validation violation: You succeeded for Field7.
 * Congratulations. You have successfully completed this lesson.**

Field1: exactly three lowercase characters (^[a-z]{3}\$)

Field2: exactly three digits (^[0-9]{3}\$)

Field3: letters, numbers, and space only (^[a-zA-Z0-9]*\$)

Field4: enumeration of numbers (^(one|two|three|four|five|six|seven|eight|nine)\$)

Field5: simple zip code (^[d]{5}\$)

Field6: zip with optional dash four (^[d]{5}(-[d]{4})?\$)

1.3.1.4 Fail Open Authentication Scheme

Go to WebGoat lesson 'Fail Open Authentication Scheme' in 'Improper Error Handling'. Try solving the lesson generating an uncaught error in the server.

Due to an error handling problem in the authentication mechanism, it is possible to authenticate as the 'webgoat' user without entering a password. Try to login as the webgoat user without specifying a password.

Sign In

Please sign in to your account. See the OWASP admin if you do not have an account.

*Required Fields

*User Name:

*Password:

OWASP Foundation | Project WebGoat

1.3.2 Session administration and authentication

1.3.2.1 Authentication using cookies

We'll see now how applications use cookies to maintain session information and how that information can be used to establish a session for a different user without having its credentials.

Go to WebGoat lesson 'Spoof an Authentication Cookie' in 'Session Management Flaws'. The goal is to be able to establish a session as user 'Alice' without having her credentials.

Try authenticating as 'webgoat/webgoat' and 'aspect/aspect' reloading the screen to observe how the web page uses the cookie to validate and maintain the session. Watch the code clicking on 'Show Java' to be able to establish a session as user 'Alice'. Have you found anything?

Let's take a look at the mechanism used to authenticate using the cookie:

- Log in as 'webgoat'.
- Check the WebScarab 'Intercept Request' box and click 'Refresh'. You will see a new window with the request; observe the cookie's value.

Parsed Raw			
Method	URL	Version	
GET	http://127.0.0.1:80/WebGoat/attack?	HTTP/1.1	
Header		Value	
Host		127.0.0.1	
User-Agent		Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.7) Gecko/20060830 Firefox/1.5.0.7...	
Accept		text/xml,application/xml,application/xhtml+xml;text/html;q=0.9;text/plain;q=0.8;im...	
Accept-Language		en-us,en;q=0.5	
Accept-Encoding		gzip,deflate	
Accept-Charset		ISO-8859-1,utf-8;q=0.7,*;q=0.7	
Keep-Alive		300	
Proxy-Connection		keep-alive	
Referer		http://127.0.0.1/WebGoat/attack?	
Cookie		AuthCookie=65432ubphcfx; JSESSIONID=1530C35D8E2616D908E8E71D9E1D3B40	
Authorization		Basic Z3Vic3Q6Z3Vic3Q=	

- Uncheck 'Intercept Requests' and click 'Accept Changes'.
- Do the same for user 'aspect' observing the value of the cookie; compare it with the one for user 'webgoat'.
- Repeat the same process several times, observing the value of the cookie for each user.

Parsed Raw			
Method	URL	Version	
GET	http://127.0.0.1:80/WebGoat/attack?	HTTP/1.1	
Header		Value	
Host		127.0.0.1	
User-Agent		Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.7) Gecko/20060830 Firefox/1.5.0.7...	
Accept		text/xml,application/xml,application/xhtml+xml;text/html;q=0.9;text/plain;q=0.8;im...	
Accept-Language		en-us,en;q=0.5	
Accept-Encoding		gzip,deflate	
Accept-Charset		ISO-8859-1,utf-8;q=0.7,*;q=0.7	
Keep-Alive		300	
Proxy-Connection		keep-alive	
Referer		http://127.0.0.1/WebGoat/attack?	
Cookie		AuthCookie=65432udfttb; JSESSIONID=1530C35D8E2616D908E8E71D9E1D3B40	
Authorization		Basic Z3Vic3Q6Z3Vic3Q=	

The value of variable 'AuthCookie' for each user is always the same. Therefore we can think that if we find the logic that generates that value we can try to modify the cookie to simulate a session for another user.

Let's study each value:

User	AuthCookie
webgoat	65432ubphcfx
aspect	65432udfqtb
Alice	65432?????

We can see that variable 'AuthCookie' always begins with '65432', we'll then concentrate on the variable part.

It looks like the number of characters of the username is directly related to the generated code

- webgoat 7 characters -> upbhcfx 7 characters
- aspect 6 characters -> udfqtb 6 characters
- alice 5 characters ... we may then think that the code will have 5 characters

Let's now try finding some relationship amongst the characters:

The code generated for both users begins with 'u' but the username doesn't begin with the same character. But we see that both end in 't'. Reversing the order:

User	AuthCookie
taogbew	ubphcfx
tcepsa	udfqtb

Now, we only need to watch a bit longer the characters to discover that, each character corresponds to one of the characters of the username in the 'AuthCookie' alphabet. That's a variant of the Caesar enciphering, classic and very basic method where a character is replaced by another one, with a bijective correspondence.

t->u, a->b, o->p, g->h, b->c, e->f, w->x

t->u, c->d, e->f, p->q, s->t, a->b

Therefore to generate the code for user 'alice':

User	AuthCookie
webgoat	65432ubphcfx
aspect	65432udfqtb
alice	65432fdjmb

Now, let's send the server the modified value of 'AuthCookie': start a session with for user 'webgoat' or 'aspect', check WebScarab's 'Intercept Requests' and click 'Refresh'.

On the new WebScarab window containing the request, modify the value for 'AuthCookie' using the one we have calculated, uncheck 'Intercept Requests' and click on 'Accept Changes'.

*** Congratulations. You have successfully completed this lesson.**

Welcome, alice

You have been authenticated with COOKIE

[Logout](#)

[Refresh](#)

You can now see a page that is greeting user 'alice'.

1.3.3 Cross Site Scripting

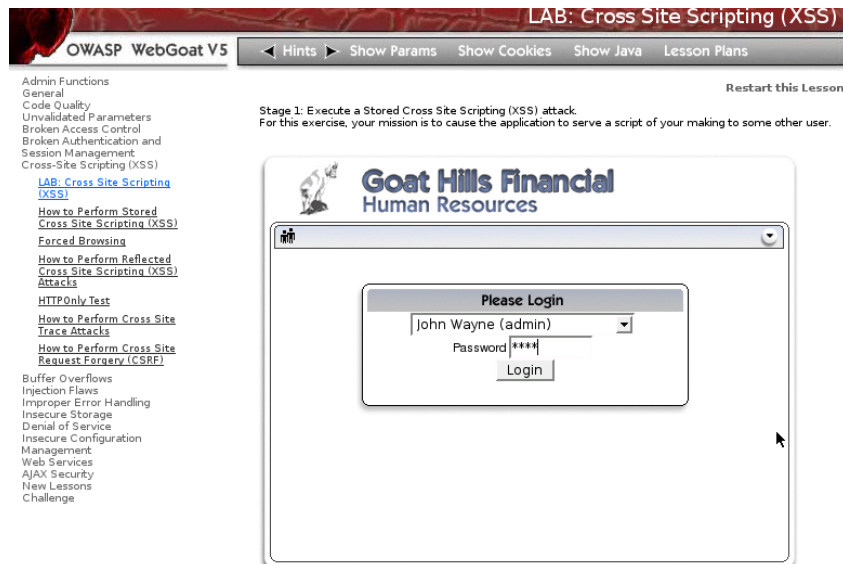
We will see now how to take advantage on a form vulnerability using what we have already learnt.

Go to WebGoat lesson 'LAB: Cross Site Scripting (XSS)' in 'Cross-Site Scripting (XSS)'. The goal of that lesson is that the application serves a script developed by us or any other user.

Try authenticating against the web page using different users (the password is the same as the username) observing what features are available and looking for a way to execute our own script when a user logs in. Any idea?

Basically this is a human resources application where every user can access to his related information. Administrative users can query and update other users.

Authenticate as an administrative user i.e. 'John Wayne (admin)' password 'john'.



Select an employee from the list and click on 'ViewProfile'.

We will now modify one of the fields of that user adding a script that will be executed when the user logs in to check his data.

Add to the field 'Street':

```
<script>alert("Stolen session" + document.cookie)</script>
```

Click on 'UpdateProfile'; a pop-up alert should appear since the form field is processed without validation. Click on 'Logout'.



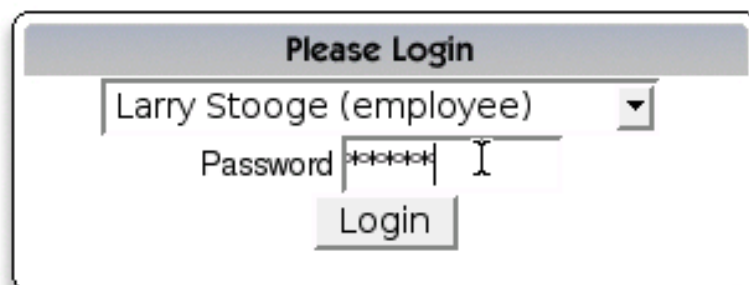
Goat Hills Financial
Human Resources

Welcome Back **John**

First Name:	Larry	Last Name:	Stooge
Street:	ent.cookie)</script>	City/State:	New York, NY
Phone:	443-689-0192	Start Date:	01012000
SSN:	386-09-5451	Salary:	55000
Credit Card:	2578546969853547	Credit Card Limit:	5000
Comments:	Does not work well wi	Manager:	Larry Stooge
Disciplinary Explanation:	Constantly harrassing coworkers	Disciplinary Action Dates:	010106

[ViewProfile](#) [UpdateProfile](#) [Logout](#)

To check whether our scripts executes also for the target user, authenticate as the modified user, select its name in the left form and then click on 'View Profile'.

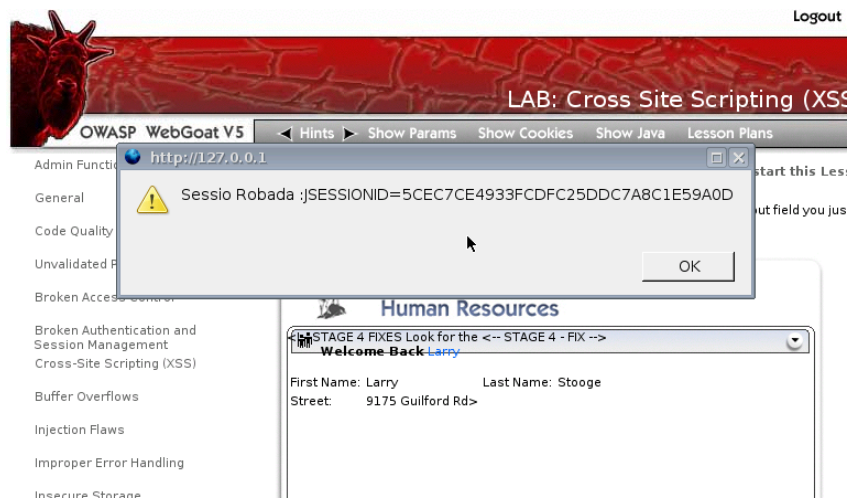


Please Login

Larry Stooge (employee) ▼

Password [masked]

[Login](#)



See that the application has served that script to another user, therefore we have achieved our goal.

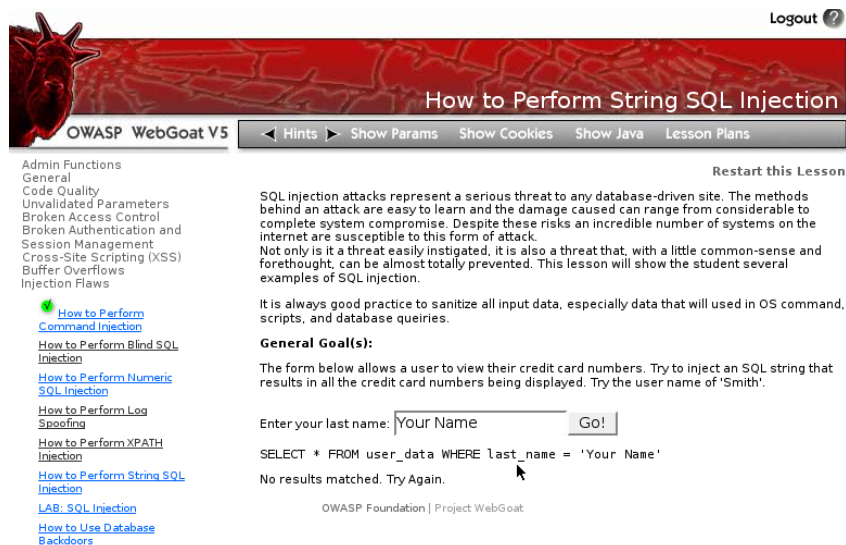
1.3.4 SQL Injection

We are going to study how to insert SQL sentences inside of a previously written query in order to manipulate the correct procedures of a given application.

In 'Injection Flaws', go to WebGoat lesson 'String SQL Injection' (not 'LAB: SQL Injection'). Its goal is to obtain a listing of the credit cards stored in a database.

Type 'Smith' as a parameter and try other values. Observe the results and study how the SQL sentence providing the listing is modified.

See that the query is waiting for a value to be entered and then used.



General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

Enter your last name:

SELECT * FROM user_data WHERE last_name = 'Smith'

userid	first_name	last_name	cc_number	cc_type	cookie	login_count
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

What if we type two quotes without any value?

General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

Enter your last name:

SELECT * FROM user_data WHERE last_name = ''

No results matched. Try Again.



OWASP Foundation | Project WebGoat

The SQL sequence is correct and returns no value. What if we type just one quote?

General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

Enter your last name:

SELECT * FROM user_data WHERE last_name = ''

SELECT * FROM user_data WHERE last_name = '' Don't understand SQL after: "WHERE"
Expected: "(" found: "="

Now the SQL syntax is not correct: it requires a quote at the beginning of a string and another one at the end. We must then find a correct SQL sentence that is always true Enter this text for 'last_name' and execute the query clicking on 'Go!'

FIB' or '1'='1

We are closing the first quote with any value and then add an expression that always is true ('1'='1'). We need to remember that a quote is added at the end. The Boolean OR operator will make the trick returning all the values of the table.

General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

*** Congratulations. You have successfully completed this lesson.**
*** Bet you can't do it again! This lesson has detected your successful attack and has now switch to a defensive mode. Try again to attack a parameterized query.**

Enter your last name:

SELECT * FROM user_data WHERE last_name = 'FIB' OR '1'='1'

userid	first_name	last_name	cc_number	cc_type	cookie	login_count
101	Joe	Snow	987654321	VISA		0
101	Joe	Snow	2234200065411	MC		0
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
103	Jane	Plane	123456789	MC		0
103	Jane	Plane	333498703333	AMEX		0
10312	Jolly	Hershey	176896789	MC		0
10312	Jolly	Hershey	333300003333	AMEX		0
10323	Grumpy	White	673834489	MC		0
10323	Grumpy	White	33413003333	AMEX		0
15603	Peter	Sand	123609789	MC		0
15603	Peter	Sand	338893453333	AMEX		0
15613	Joeph	Something	33843453533	AMEX		0

We have finally achieved to list all the values of the customer's credit cards stored in the database.

1.4 References

- OWASP Project , <http://www.owasp.org>
- OWASP Project at Sourceforge, <http://sourceforge.net/projects/owasp>
- Web Application Security Consortium (WASC), <http://www.webappsec.org>
- Common Vulnerabilities and Exposures (CVE), <http://www.cve.mitre.org>
 - <http://www.faqs.org/rfcs/rfc2660.html>
 - <http://www.faqs.org/rfcs/rfc2616.html>
 - <http://www.faqs.org/rfcs/rfc1945.html>
- Secure Coding: Principles & Practices, <http://www.securecoding.org/>