**Local Testing (Validation) - Unexpected Input**

With two shortlisted functions, we can start testing them individually, starting with processPayment() from **payment-controllers.js**. The main thing we are looking for is weakness in type safety, and potential cases where we may be able to use an unexpected type that passes the validation filter.

**processPayment()**

From a first look, we can see that this function is longer than average, which means we may have a larger attack surface within the function. As mentioned in the previous section, the function has two direct user inputs we can control; cardId and items.

We see that cardId gets used to retrieve the user's payment card with PaymentCard.findOne. However, this is also coupled with the userId from the auth token, so it is unlikely that we would be able to trick it to somehow use another user's payment card, even if we knew their card's ID. If a user doesn't have a card, or provides an incorrect cardId, we get an error, as we can notice in the front-end cart.

Code: js

try {

card = await PaymentCard.findOne({

userId,

\_id: cardId,

});

if (!card) {

throw new Error();

}

} catch (err) {

return next({

message: "Could not find a card with this id for this user.",

statusCode: 404,

});

}

**Exercise:** Try to confirm the validity of the above test, by sending a request to the charge endpoint with any string as the cardId. You can use an empty array as the value of items.

So, this leaves us with items. We see that the function loops over each item of the items we sent. We do not, however, see anywhere that the function validates that the items parameter is actually an array. So, what would happen if we do send a number instead of an array?

Code: js

// validate items + get prices

try {

for (const item of items) {

const { name, category, price, amount } = item;

<SNIP>

}

}

**Note:** We chose to send a number instead of a string, because JavaScript sees a string as an array as well, and can loop over it. If we send an empty string, it would be the same as sending an empty array.

Let's try to send a request with RapidAPI, as we did previously. Backtracking processPayment() in /routes, we see that it needs a POST request to /api/payment/charge.

**Challenge:** Try to see how we get the API for this function, by searching for processPayment in the /routes directory, and linking that with the routes defined in app.js.

**Obtaining cardId**

As shown earlier, the /charge API endpoint requires us to send a valid cardId; otherwise, the function would stop executing before we reach the part we are testing. If we log in with the provided user credentials, the pre-added user has a payment card added to their name, so we'll try to grab their ID. If this were another production application, we would probably just add a payment card and grab its ID, but here, the process of adding a card is irrelevant, so we pre-added it to the DB.

To grab the cardId, we have two options: Monitoring requests on the front-end, or reviewing code on the back-end. We'll go with the "more fun" route, and will review back-end code. To do so, we can utilize the other API endpoint we see in payment-routes.js, which is getUserCards():

Code: js

router.use(verifyToken);

router.get("/cards", getUserCards);

router.post("/charge", processPayment);

If we CMD/CTRL click on it, we can read the function, and will see that it does not take any input, and basically uses our userId from our auth token to obtain our payment cards, and then return them to us:

Code: js

const userId = req.user?.id;

let userCards = null;

try {

userCards = await PaymentCard.find({

userId,

});

<SNIP>

}

So we only need to send a GET request to /api/payment/cards along with our auth token, and we should get the cardId. Once we send the request, we simply get a list of our cards, which includes each card's id:

Screens screenshot of a computer screen

Description automatically generated

This is likely used to avoid sending sensitive card details over the network (e.g. full card number), and rather use a reference to obtain it from the back-end. With cardId at hand, we can proceed with our attempt at tampering with sending an unexpected type for the items parameter.

**Note:** As mentioned earlier, we could have reached the same conclusion by monitoring the requests sent by the front-end at the checkout page "on the cart". However, unlike the previous demo, we are trying here to stick to back-end analysis only in order to learn a different approach of code review. For the sake of thoroughness, we will also demonstrate the other approach in a bit.

**Expected Arrays vs Unexpected Numbers**

We can prepare a POST request to /api/payment/charge as planned earlier, and include our auth token in Bearer, as shown previously. For the JSON body, we will add the cardId we just obtained, along with the items key with any number for its value, like the following:

Code: json

{

"cardId": "64b85d58cabeffbc46ce76c9",

"items": 0

}

Once we send the request, we see that it takes quite a while, and then responds with Could not find prices for all items:

A screenshot of a computer

Description automatically generated

Checking the code, we see that this is within the catch block in the function, as the function must have failed trying to loop over a number "instead of an array". We can confirm this by watching the items variable and setting a breakpoint within the try block, then repeating the above request. After that, we can step into it the block to better understand what happened exactly and where it broke.

A screen shot of a computer

Description automatically generated

As we can see, the specific error we got was (TypeError: items not iterable). This confirms that this, indeed, is an unexpected input bug, albeit harmless as far as we can tell. The try/catch block, in this case, saved the function from continuing with a faulty items value, though the function should have validated that the items value is indeed an array and is not empty; otherwise, it would not land in the for loop and reach the validateCartItemDetails() validation within.

We can consider this a false positive and move on with reviewing the function code. This is normal in any code review practice, as not everything we test will be useful or vulnerable. Most things we test will be false alerts, especially with securely coded applications. Still, the function filtering we did in the previous section should reduce the count of false alerts, and hopefully get us to test the most promising functions of the application.

**validateCartItemDetails()**

Now we get to the interesting part that caught our eyes in the previous section, which is the function that validates our user input. The function starts by looping over the items array, as we have established already, and then does the following for each item:

Code: js

const { name, category, price, amount } = item;

// validate CartItemType array

const errors = await validateCartItemDetails({

name,

category,

price,

amount,

});

if (errors) {

return next(errors);

}

We see that it obtains four variables from each item, and then validates them with validateCartItemDetails(). We can once again CMD/CTRL click on it to go through what it is doing, and we see that it is running a yup validation test based on the CartItemSchema, which we can find right below it:

Code: js

export const CartItemSchema = yup

.object({

name: yup.string().required(),

category: yup.mixed().oneOf(["subscription", "exam", "cubes"]).required(),

price: yup.number().required(), // in usd

amount: yup.number().required(), // item count

})

.required();

As we have mentioned in the previous section, any code reviewers need to be able to validate the solidity of any validation test of any form, so let's go through this schema and try to pick it apart.

**CartItemSchema**

The first thing we notice is that every item in the schema is marked as required(), along with the entire object being required. So, if the item object is entirely missing, or if any of the 4 keys are missing, then it would error out.

Apart from that, it starts by ensuring that the name value is a string, which appears to be acceptable, as this is the expected format. For category, it ensures that it has to be one of the three allowed values (subscription, exam, cubes), which also appears to be acceptable as it is quite restrictive. As for price and amount, we see that it ensures that their type is a number, but it provides no further validation on top of that, even though these are the main elements in this JSON object, and are very sensitive to the processPayment function.

What is missing from the price/amount validation? First of all, it does not ensure that the number is a positive number, so we can provide a negative amount or price, which can lead to various issues, as we will see. Furthermore, it does not set a minimum amount for either, so we can use 0 for the amount, which does not make any sense from a purchase processing point of view, but the function does not validate that as well. So, the validation test is clearly flawed.

So, in the next section, we will see if we can take advantage of this logic bug, and if we can validate that, we will move to create a proof of concept.