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

Now that we have solid evidence suggesting that the validation test is flawed in CartItemSchema, let's continue our testing to see if we can take advantage of this. Our goal is to prove that we can impact the total price we would pay by sending unexpected input, which negatively affects the total price calculations and alters the intended logic.

**Manipulating Prices**

Let's start with the easier option, which is to provide a price of 0 for any item we purchase, which is allowed by the schema since the only condition for the parameter is to be a number. This may potentially allow us to buy any item completely free of charge.

We can start with a dynamic test, and send a charge request to buy some cubes with the price of 0. Once again, we can simply copy the request sent by the front-end, but we'll try to stick to the back-end, and will create our own JSON object. We can reuse the previous JSON object that we sent, but instead of a 0 for the items value, we will use an object with the 4 elements shown earlier, and set "price" to 0, as follows:

Code: json

{

"cardId": "64b85d58cabeffbc46ce76c9",

"items": [

{

"name": "1000",

"category": "cubes",

"price": 0,

"amount": 1

}

]

}

**Note:** By reviewing the code, we understand that the name value is meant to match the subscription or exam names, but what about cubes? If we read the function's code, we see that it parses it as a number and uses it as the number of cubes in (parseInt(name)), so we will enter the number of cubes we want to get "hopefully for free!". Once again, we could have easily used the front-end to obtain this information, but we chose the code review route.

Unfortunately, if we send the request, we get an error message saying Insufficient funds:

Screens screenshot of a computer

Description automatically generated

How can we possibly not have enough funds, when the price we set is 0? If we continue reading the remaining part of the for loop, we will see that the function is not using the prices we sent, and instead either manually calculating them or pulling them from the DB:

Code: js

case "cubes":

total += (parseInt(name) \* amount) / 10;

break;

So, with the price value not being used, it would appear that it does not cause a logic bug, even though it was not being properly validated.

**Challenge:** As the function parses the name value as an integer, what would happen if we send a value that cannot be parsed? Does the code validate this information as well? Try and see what happens, and try to find out why it is happening.

**Manipulating Items**

Even though the price value is not being used, the amount value is definitely being used and does indeed affect both the total price and the later item processing function:

Code: js

switch (item.category) {

case "cubes":

total += (parseInt(name) \* amount) / 10;

break;

<SNIP>

}

// process items

try {

for (const item of items) {

// repeat by the amount

for (let i = 0; i < item.amount; i++) {

switch (item.category) {

// if coupon.type is cubes, then buy cubes for user with item.name

case "cubes":

await buyCubes(userId, parseInt(item.name), next);

break;

<SNIP>

}

}

}

}

With amount not being properly validated, can we manipulate it by sending an unexpected value to take advantage of this flaw? Let's review the rest of the code to see where and how it is being used. Within the above process items for loop, we see that for each item, it uses a switch statement to execute the proper type of calculation depending on the specified category. For cubes, it simply parses the name into an integer, and then multiplying that by the amount to calculate the total price:

Code: js

case "cubes":

total += (parseInt(name) \* amount) / 10;

break;

As we asked earlier, what would happen if we use a string that cannot be parsed into an integer, like "test"? If we do so, e see that the request passes all tests, and even returns a successful purchase, as shown below:

A screenshot of a computer

Description automatically generated

We can set watchers to the total and name variables, and set a breakpoint at the beginning of the items loop, and then follow the function execution, as shown below:

A screen shot of a computer

Description automatically generated

We will see that when JavaScript tries to parse the string "test" into an integer, it fails and returns NaN instead, but it does not throw an error or stop execution! This is an odd behavior in JavaScript, and will be discussed further in the upcoming sections. The key learning point is that JavaScript's dynamic typing can proceed with the execution, even if the values are completely off (i.e. null or NaN).

So, in this case, it corrupted the total to NaN, which passed the payment card balance test. When it tried to buy cubes, it did the same thing, and tried to adjust our cube count by NaN, which may potentially corrupt our data in the database. Finally, it tried to adjust our card's balance with the total (i.e. "NaN"), which may as well corrupt the database.

This is the second unexpected input bug we found in this function, in addition to the items array bug we showed in the previous section. While both of these bugs were saved by other measures in place (or sheer luck!), both of these bugs must be remediated, as they pose serious risk to the functionality of the web application. In the null safety sections, we will demonstrate how null or NaN values can lead to serious consequences. In any case, there is no immediate action we can perform to take advantage of this bug, so we will continue with the rest of the function.

**Manipulating Amounts**

So, let's further dissect the process we went through as we did the debugging shown above. If we continue and review the subscription switch case, we see that the function first retrieves the subscription name from the DB, and then uses its cost value to calculate its price, and multiplies that by the amount we sent. The same is done for exam. The function keeps aggregating the total sum throughout the for loop to calculate the total price for all items based on the amounts we sent.

After that, the function ensures that the calculated total is within our card's balance with if (total > card.balance). Then, the function moves into processing each item, by looping over each and passing it to its appropriate function for purchase processing (buyCubes, buySubscription, and buyExam). It also repeats this by the amount count, so that purchasing multiple items would be processed multiple times.

Finally, the function adjusts our cards balance by the total prices for all items, which means it would deduct the total price from our payment card.

Code: js

card.balance = card.balance - total;

await PaymentCard.updateOne(

{

userId,

\_id: cardId,

},

card

);

Throughout all of this, we never see it validating (or even considering) that the amount may be a negative amount, all while using the amount to calculate the total price and the number of times an item gets processed. So, we could potentially send a negative amount for a certain item, and when it multiplies this negative amount with the item's price, this should lead it to reduce the total price instead of increasing it, when it does the following:

Code: js

total += (parseInt(name) \* amount) / 10;

We do not care whether it would process this item or not later on, but as long as this does not lead the entire function to fail, we can potentially use this flaw to get a reduced total price, or potentially pay nothing at all. We will verify this finding in the next section.