**PoC and Patching - Unexpected Input**

After all of our testing, we can finally test our findings to see if we can truly buy cubes for free, by exploiting the type bug we identified.

**Proof of Concept**

Let's assume we want to get some free cubes or a reduced fee. We can include an item with a negative amount, and this amount should reduce the total price. For example, we can specify a 100 cubes, which should have a price of $10 (according to the calculations we showed previously or according to the front-end). If we specify the amount to be -1, then the total price should be adjusted by (10 \* -1 = -10). If we add another item with an equal but positive amount, we should get a total of $0 charge, and our card's balance should not be affected.

Let's reuse the previous payload we sent, and adjust it to match the above criteria, as follows:

Code: json

{

"cardId": "64b85d58cabeffbc46ce76c9",

"items": [

{

"name": "100",

"category": "cubes",

"price": 0,

"amount": 1

},

{

"name": "100",

"category": "cubes",

"price": 0,

"amount": -1

}

]

}

When the function processes this request, it will first validate our cardId, then iterate over the items to calculate the total price, which should first be at 10, and on the second iteration should be reduced by -10, netting a total of 0. Then, the function would validate that 0 is within our card's balance, which it should be even if our balance is 0.

After that, the function would process the items by their amounts. So, it would process the 100 cubes once, and it should not process the negative amount at all. Finally, it would adjust our card's balance by the total price of 0, so it should not be affected.

Let's test this by sending the above request and seeing what we get:

A screenshot of a computer

Description automatically generated

As we can see, the response confirms that it Successfully processed payment with a total of $0. We can now login to the front-end with the provided credentials, and we will see that our cube count has indeed increased without having to pay anything:

A screenshot of a computer

Description automatically generated

We can see how a minor mistake in the way cart items were being validated lead to a complete compromise of the payment system, allowing us to practically purchase any item completely free of charge, even if our payment card had a zero balance.

**Exercise:** It is possible to carry the same attack using nothing but the browser and the front-end web application. Try to do that. Hint: How's the browser holding our cart items?

This example demonstrates the importance of thoroughly and adequately examining and validating every input sent by the user, including any sub-items that may be included in that input/object. Without a solid validation mechanism, such an issue and many others may arise, leading to various logic bugs.

**Challenge:** Try to replicate what we have shown here on your local environment. After that, try to find other issues with the processPayment function, as well as the other function we shortlisted in the previous section.

**Patching**

For Unexpected Input logic bugs, we must ensure that we only accept the specific type of input we expect and refuse everything else. This function's main issue was insufficient input validation, as the CartItemSchema only checked if the amount is a number, but did not verify whether it is a positive number with a minimum of 1. So, all we need to do is add these checks to the schema, as shown below:

Code: js

// to process cart items in payment requests

export const CartItemSchema = yup

.object({

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

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

price: yup.number().positive().min(1).required(),

// in usd

amount: yup.number().positive().min(1).required(), // item count

})

.required();

As for the two other 'minor' bugs we identified, we should also address them in our remediation notes. For the items array bug, we can simply throw an error if the items variable is not an array, as follows:

Code: js

try {

if (!Array.isArray(items)) {

throw new Error();

}

for (const item of items) {

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

<SNIP>

}

} catch (err) {

<SNIP>

}

Finally, we should fix the name integer parsing bug. For this bug, we need to confirm that the value of name can be parsed as an integer, but only within the cubes case, as the other cases expect strings. We can use a similar method that was used to confirm the date format that we saw previously in the module, by using isNaN(parseInt(name)) to confirm that the value can be parsed safely, as follows:

Code: js

// add cost to total

switch (item.category) {

case "cubes":

if (isNaN(parseInt(name))) {

throw new Error();

}

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

break;

<SNIP>

}

In general, to patch such vulnerabilities, we need to ensure that validation tests (e.g. schema) accurately match the expected type of input, and do the same for every single input. If the validation tests allow extra room for manipulation (i.e. accept a wide range of input types), then they will likely be vulnerable for such attacks.