**Local Testing - Validation Logic Disparity**

Now that we have identified a potentially interesting function, in this section, we will study the back-end API's it interacts with, and try to see if there are any disparities between the front-end validation logic and the back-end ones.

**Note:** Before we continue, you may restart your Docker container to reset it to its original state.

**getExamAvailability()**

Let's start with the getExamAvailability() function. First, we see that it sets {id, startDate, endDate} from the request body and performs a few checks to validate the date format. It then locates the exam in the database with Exam.findOne using the sent id, as a form of id and exam verification:

Code: js

const { id, startDate, endDate } = req.body;

// validate date format

if (

!startDate ||

!endDate ||

isNaN(Date.parse(startDate)) ||

isNaN(Date.parse(endDate))

) {

return next({

message: "Please provide a valid date range.",

statusCode: 400,

});

}

let exam;

try {

// ensure exam exists

exam = await Exam.findOne({

id,

});

<SNIP>

}

After that, the function uses the following code to find existing exam slots within the provided date range:

Code: js

const bookedExams = UserExam.find({

examId: exam.id,

date: {

$gte: new Date(startDate).setUTCHours(0, 0, 0, 0),

$lte: new Date(endDate).setUTCHours(23, 59, 59, 999),

},

used: false,

});

**Note:** Whenever the code interacts with the mongodb database, it is using pre-defined modules, like Exam and UserExam in this case. Try to read their code to understand how this is working. This should not affect our code review, but it does make it easier for the developers to interact with the database, and for us to understand the code.

Finally, the code simply returns a JSON list of any booked dates, and if it does not find any results, or if it encounters any issues, it returns an empty array:

Code: js

res.json({

unavailableSlots: (await bookedExams).map((exam) => exam.date),

});

We can confirm this by sending a basic request through RapidAPI. If we go to the RapidAPI extension tab, hit + and then set the method to POST and URL to http://localhost:5000/api/exams/availability​. Lastly, we can set the data type to JSON, and use the following request body "from the front-end request":

Code: json

{

"id": 1,

"startDate": "2023-09-01T15:47:14.843Z",

"endDate": "2023-10-01T15:47:14.843Z"

}

Now, if we send the request, we get a list of unavailable slots in this date range:

A screen shot of a computer

Description automatically generated

All of this seems quite normal, as the function simply returns a list of unavailable dates. This approach is quite common, where an endpoint would send a list of unavailable dates instead of the available ones. Either way, this does not affect or cause any logic bugs "yet".

**bookExam()**

Checking the bookExam() function, we see that it also sets {id, date} from the request body, and then sets the user ID from req.user?.id. We also see that it validates the date format just like the getExamAvailability() function, and then it retrieves the exam details to confirm its existence. Finally, it updates the user's purchased exam ticket with the booked exam date:

Code: js

const updateReq = await UserExam.findOneAndUpdate(

{

examId: exam.id,

userId,

used: false,

date: {

// date must be null 'unbooked' -> can't change date once booked

$eq: null,

},

},

{

date: new Date(date),

}

);

We see that it retrieves the exam by its id, which has already been verified to exist, and uses the userId to ensure that the user has already purchased an exam ticket and not used it (used: false). It also ensures that the user has not already booked the exam by searching for a ticket with an empty date (date: { $eq: null }). Finally, it updates this ticket with the requested exam date (date: new Date(date)).

Let's confirm that our understanding of the function is correct. We want to modify the value of userId, but since it is retrieved from the JWT token, which is signed with a secret key, we cannot simply modify it "the key in real target differs from the local one". So, instead, we will set a breakpoint right after the line where userId is set, and then send a request to the endpoint. Once the breakpoint is hit, we can slightly modify the userId to something else, and it should tell us that we have not purchased this exam.

So, first, we'll add the breakpoint by clicking on the line number in VSCode (or using the shortcut SHIFT+F9). We may also right-click on the userId variable and select Add to Watch, to keep an eye on its value:

A screen shot of a computer

Description automatically generated

Lastly, we can send a POST request to /api/exams/book with the same id/date body data we saw in the previous section. As this endpoint requires authentication, we will also need to add our token to the request, which we can copy from the storage tab in the Browser Dev Tools under Local Storage, then we can click on the token and select Copy Row. Then, in the RapidAPI request, we add it in the Auth tab with the Bearer option "make sure you delete the token word when you paste the value". Once the request is set up properly, we can click Send, and we should automatically hit the breakpoint we set:

A screen shot of a computer

Description automatically generated

We can examine the request values on the left panes, including the userId value that we added to watch:

A screen shot of a computer program

Description automatically generated

Now, we can right-click on userId under the WATCH group, select Set Value, and change it to anything else, so it would be any other ID that doesn't have a booked exam. Once set, we can click F5 to continue the execution of the request, and will receive the error "User has not purchased this exam" in the response to the request we just sent:

A screen shot of a computer

Description automatically generated

With this process, we were able to confirm that the function correctly prevents unauthenticated users, properly validates the date format, correctly validates the exam id, ensures that users can only book once they have purchased the exam, and prevents them from booking again once a date has been set. It seems secure, right? Not quite.

While the function does validate the date format, it performs no validations on the availability of the exam date, like checking whether it is already booked or whether it is in the future. Not only can a user book an exam on a date that is not available "which leads to over booking", they can also book a date in the past "which may potentially lead to other logic issues".

The front-end app does all of that, as it disables any dates in the past, and it disables any dates found in the unavailableDates[] array from /exam/availability. However, this disparity between the two ends leads to the flaw we are discussing in this section.

While this is quite a fundamental flaw (i.e. relying on front-end validation), in reality, this is a prevalent logic flaw found even with some of the largest online retailers, as we have mentioned in the intro section.

This flaw can be due to many reasons, like having a gap in communications between front-end and back-end developers, the complexity of the code and thinking this may be validated somewhere else, and many other reasons. That is why we must keep this in mind when reviewing code bases or performing mobile/web application penetration tests, and we must always ensure each function has a solid validation logic on both ends and is in complete parity.