**Code Review - Validation Logic Disparity**

As mentioned in the previous section, the easiest way to identify logic disparity bugs is by comparing validation tests on both ends of the application, and trying to find a disparity on the back-end code. Otherwise, we can manually test functions in the front-end and see which rely on dynamic validation tests. Then, we can start reviewing the back-end code to identify disparities.

Doing this for huge code bases and applications may sound cumbersome, especially ones with many forms and fields. However, once you have the proper understanding of how validation logic disparity bugs look like, then you are likely to identify them when reviewing any function on the back-end.

We are also starting with a mixture of code analysis and dynamic application testing for this specific use case, but as we go through the module, we will utilize more advanced techniques to review larger parts of the code base and filter them for specific criteria.

**Note:** Did you complete the exercise at the end of the previous section? If so, see if one of the functions you noted down/shortlisted is what we will discuss in the section. If you got it, then you are off to a great start. Amazing job!

**Dynamic Validation Tests**

If we start going through the front-end web application, we can easily identify a few forms and fields that are clearly applying client-side data validation. This is easily verifiable by the instant feedback, and the fact that no requests get sent as the validation is applied, indicating that it is in fact happening in the client-side. For example, the very first form we face is the login form, and we can bring up the Network tab from the browser developer tools, and see that, indeed, no requests are being sent to process this validation:

A screenshot of a computer

Description automatically generated

As mentioned in the previous section, we are specifically looking for dynamic front-end validation filters that are relying on data pulled from the back-end. The filters applied by the login form do not appear to rely on any data from the back-end, so they can be considered as static filters that always try to match a specific pattern (e.g. email format). Furthermore, none of the requests sent by our browser when visiting the page seem to have data that may be utilized by it.

We can keep going with the other forms we can find in the web application, while keeping an eye on the network tab for sent requests: A screenshot of a computer

Description automatically generated

**Note:** Keep in mind that not all forms pull data from the back-end in a similar manner. Some may pull the data when the form is clicked, while others may pull the data when the page is visited, and so on.

**Exam Booking**

If we keep going with other forms and fields within the web application, we will notice that the exam booking function does appear to meet all 3 of the criteria we specified in the previous section. So, let's take a look at it in the front-end, to further understand how it functions and whether it is vulnerable to this flaw.

If we go to the /exams page, we see the available exams listed for purchase. If we had already purchased one of them "as is the case for the user with the provided credentials", then the next step would be to Book our exam slot. This is a common practice for many certification bodies, and is also used for many online appointments and reservations.

A blue rectangular object with white text

Description automatically generated

If we click on Book, we are presented with a calendar view for available slots that we may book. Once we select an available slot and click Confirm, the exam gets booked and we get a confirmation message with our exam date.

A screenshot of a computer

Description automatically generated

We can see that:

1. It does accept user input: in the form of our selected date in the calendar
2. It does apply client-side validations: by disabling certain unavailable dates
3. It does rely on back-end data to adjust the validation: which we can verify by monitoring the requests and how the filter changes

Personally, whenever I see a similar case I immediately ask myself:

1. How is it applying the availability filters?
2. Does the back-end revalidate the availability upon booking?

To further understand how to confirm this, we can once again bring up our browser developer tool and go to the Network tab. Once we do, we can click the trash icon on the top-right corner to clear network items. Now, if we click the book button again, we see that the application sends a request to get the details about which slots are unavailable:

A screenshot of a computer

Description automatically generated

Let's investigate this further. If we select Request from the top-right drop-down menu "instead of Response, as shown above", we see that it is sending a request with the following data:

Code: json

{

"id": 1,

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

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

}

To find the exact API endpoint URL, we can click on the headers tab, and see under Request that it is sending a POST request to the /api/exams/availability endpoint:

A screenshot of a computer

Description automatically generated

Finally, let's select any available slot and click confirm, to see which request gets sent. As we can see, this time another POST request was sent to /api/exams/book, with the exam id and the selected date, as follows:

Code: json

{

"id": 1,

"date": "2023-09-14T23:00:00.000Z"

}

A screenshot of a computer

Description automatically generated

From our brief interaction with the endpoint, we can see that the first request to /availability was used to modify how the filter is applied, while the second request to /book was the regular request used for exam booking. To test for validation logic disparity, we need to locate the functions responsible for these endpoints and review their code, and compare their filters to the once we observed on the front-end.

**Note:** You may notice slight differences between the date you selected and the date in the request. This is completely normal, as the request uses an absolute timezone to ensure it is the same on both ends.

**Locating Endpoints**

To locate these endpoint functions, we can open app.js, and we'll see that it is under examRoutes:

Code: js

// set up API routes

<SNIP>

app.use("/api/exams", examRoutes);

We can CMD/CTRL+click it, and it'll open in a new tab in VSCode, as we have seen previously:

Code: js

router.get("/", getAllExams);

router.get("/:id", getExamById);

router.post("/availability", getExamAvailability);

// secure private routes for content (use req.user in private controllers)

router.use(verifyToken);

router.get("/user/exams", getUserExams);

router.post("/book", bookExam);

router.get("/content/:id", getExamContent);

We see that the /availability endpoint falls before verifyToken, while the /book endpoint falls after it. It is important to understand how this affects the endpoints and the application in general. If we review its code, we will see that it is responsible for decoding and verifying the user details from the authentication token. So, any endpoints that fall after it would require an authenticated token.

**Exercise:** Try to read the verifyToken function to see how it is decoding the authentication token and adding the user details to the request, and note what user details are being added to the request. Do you think it would be possible for us to manipulate our user id?

So, /availability endpoint is publicly accessible, while the /book endpoint does require authentication, and will require a valid token. This is expected for the /book endpoint, as the above request didn't provide any details about our user to book the exam for us, so it must be getting these details from our token.

In the next section, we'll go through these functions, and will make tests to validate our understanding of how they work.