**Code Review - Unexpected Input**

In the previous sections, we understood how API routes are mapped and linked to functions in our NodeJS app. So, let's focus in this section on identifying all functions that accept user input through GET and POST requests. We will then filter out any type-safe functions or ones using proper user input validation, and will focus on the remaining ones in our Local Testing phase in the next sections.

**Identifying Functions with User Input**

If we search the code base for functions containing either GET parameters (req.params) or POST parameters (req.body), we will reduce our scope to the following 18 functions (sorted by search results order in VSCode):

| **File** | **Function(s)** |
| --- | --- |
| coupon-controllers.js | applyCoupon() |
| exam-controllers.js | getExamById(), getExamAvailability(), bookExam(), getExamContent() |
| modules-controllers.js | getModuleById(), getModuleUnlockedStatus(), unlockModule() |
| payment-controllers.js | processPayment() |
| sections-controllers.js | getModuleSectionsById(), getSectionsProgress(), getSectionContent(), markSectionAsCompleted() |
| users-controllers.js | createUser(), login(), updateUserDetails(), resetPassword(), requestPasswordResetLink() |

**Tip:** We can use the (req.body)+|(req.params)+ regex pattern in the VSCode search panel, and enable the Use Regular Expression option with the ALT+R or CMD+ALT+R shortcut. Furthermore, we can expand the search details and use src/src/controllers in files to include to only search controller functions.

A screen shot of a computer program

Description automatically generated

**Filtering Out Functions with Input Validation**

To further reduce our testing scope, let's filter out any functions that apply validation on the user input. If we start with the applyCoupon() function, we see that it only has one user input, which is the coupon code, and it applies a validation test on it before processing it. If it fails, it will return an error:

Code: js

// validate coupon

const errors = await validateCouponCode({ coupon });

if (errors) {

return next(errors);

}

Although validation tests sometimes fail to properly validate user input (e.g., using an insecure regular expression), the validation tests used here seem to be secure. We can CTRL/CMD click on validateCouponCode(), we will be taken to the function, and will see that it uses the CouponCodeSchema and the validate function from the yup package, which ensures that the user input meets all of the requirements specified in the schema.

This process makes it very solid in performing its validation tests, assuming that the schema itself is well designed, so we can skip all functions that apply these validation tests (at least for now, and if we don't identify anything, we can go back and investigate them further).

**Filtering Out Functions with ID Verification**

Let's now continue to the second function in the above list getExamById(). This function simply accepts an exam id, and returns its details. For example, if we visit /api/exams/1, we will get the details of exam 1 in JSON format, which will then be properly processed and formatted by the front-end and displayed to the user. Even though this function does not perform any input validation on the id parameter, it still verifies this ID by using it to find an exam in the database, and if the ID does not return any matches, it will simply throw a 404 error to the user:

Code: js

try {

<SNIP>

if (!exam) {

throw new Error();

}

} catch (err) {

return next({

message: "Could not find a matching exam with the provided id.",

statusCode: 404,

});

}

So, this means that all id parameters do not need input validation, as long as they have this step to ensure they match a result, so we can filter out those functions as well. Of course, the code can always be vulnerable to injections or may have weak access control (e.g. this API may display private information publicly), but as we mentioned previously in the module, we are not covering these types of vulnerabilities in this module, and will only be focusing on logic bugs. In a real code review exercise, you may want to double-check how the back-end is limiting its access control and how it is sanitizing user input.

**Exercise:** Go ahead and review all of the remaining 16 functions above, and filter out any ones that do apply validation on all of the user input received through GET/POST requests. You can consider any id parameters to be safe, as long as they return an error when no matches are found.

**Reviewing Schema Models**

If we go through all of the remaining functions, we will see that all of them appear to be secure, as they all apply validation tests on all user input, either through schema.validate or by verifying the passed id. We seem to be dealing with a code that is securely coded, so does this mean that this code is secure from unexpected input logic bugs? Once again, not quite!

While it is an important exercise to filter out seemingly secure functions that apply validations on user input to quickly identify any low hanging fruits that may be easily exploitable, this by no means is the end of the road, as it only targets clearly vulnerable functions. When dealing with sources that have a higher level of security, we can usually (but definitely not always) expect it to have these basic security measures in place, as the developers would likely have coding standards that mandate them.

So, where do we go next? As mentioned earlier, if we hit a road-block, we will return to those schemas used in the validation tests and investigate them further. So, let's filter the previous list down to only those functions that utilize validation tests. We can do so by searching for .validate( within our code, which should return all schemas being used for validation:

* validateCouponCode -> CouponCodeSchema
* validateCartItemDetails -> CartItemSchema
* resetPassword -> passwordResetSchema
* validateUserDetails -> UserSchema

If we cross-search where these are being used, we'll get the following endpoints:

| **File** | **Function/Endpoint** |
| --- | --- |
| coupon-controllers.js | applyCoupon() |
| payment-controllers.js | processPayment() |
| users-controllers.js | createUser(), login(), updateUserDetails(), resetPassword(), requestPasswordResetLink() |

We can once again start with applyCoupon(), CMD/CTRL click on validateCouponCode() to get to the function, and then go to the schema by CMD/CTRL clicking on CouponCodeSchema, and we will see the following schema:

Code: js

export const CouponCodeSchema = yup.object({

// coupon must be an md5 hash of the coupon code

coupon: yup

.string()

.matches(/^[a-f0-9]{32}$/i, "Invalid coupon.")

.required(),

});

It is important to learn the skill of reviewing validation tests and functions and be able to spot any potential weaknesses where the test may be missing certain cases. This is not exclusive to schema validation, but also applies to any custom validation tests in general, like ones using regex or any other condition. Usually, we are only interested in custom validation tests, and will not be testing external packages or core validation functions, as this would be considered out-of-scope (i.e. tested when reviewing the code of those packages).

So, the above schema simply requires the input to be a string and match the specified regex expression. For example, sending a number in the JSON request would fail (e.g. {"coupon": 1}). Furthermore, we can use online regex tools, or the [Regex Previewer](https://marketplace.visualstudio.com/items?itemName=chrmarti.regex) VSCode extension to further test the regex expression, but eventually, we will realize that it safely ensures that the passed string matches an md5 hash, as mentioned in the comment above it.

Continuing with the remaining functions, we see that both exam-controllers.js functions only receive dates as input, and validate their format using isNaN(Date.parse(date)), which after some basic testing and online research appear to be a solid way of validating that the date is in the expected format using a JavaScript built-in function.

Next, we have the processPayment() function, which uses the validateCartItemDetails() function to validate each item in the items list from the POST request. Checking the validation function, we see that it uses CartItemSchema, which has 4 different parameters. We can shortlist this function for now, and investigate it further later on, so we can move on with the rest without much delay.

This leaves the five functions from users-controllers.js, all of which deal with user login and user details. These functions mostly use validateUserDetails(), which relies on the UserSchema. Furthermore, the resetPassword() function uses its own custom passwordResetSchema to validate its user input. A quick look at the UserSchema shows that it is quite basic and would be of low priority in our testing. Another quick look at passwordResetSchema shows that it has some custom stuff, so this may be worth shortlisting as well for further investigation.

So, after searching the entire code base for functions with user input, and then filtering them down based on their input validation tests, we are left with two shortlisted functions: processPayment() and resetPassword(). In the next section, we'll see if we can identify any weaknesses in either of them.