**Avoiding Parameter Logic Bugs**

By now, you should have a very good understanding of different types of logic bugs that are related to user input and parameter manipulation. You should also have had plenty of practice on how to identify such logic bugs. This should give you the necessary understanding of such bugs to avoid introducing them to your code during the development process, as well as being able to spot them when reviewing code.

In this section, we will summarise tips on how to avoid introducing such logic bugs to our code.

**Validation Logic Disparity**

The main thing we are looking for here is Logic Parity between the front-end and the back-end. We cannot rely on validation tests only in the front-end, as they may lead to restriction bypasses, or validation tests only in the back-end, as they may lead to bad user experiences and lost revenue. Any difference or disparity between the two ends may lead to consequences, as we've seen in this module.

This also applies to validation tests and type checks, as will be shown next. So, any tests that are carried in the front-end, must also be replicated on the back-end, and vice-versa. Furthermore, the validation tests should also be identical or very similar, to avoid introducing other disparity bugs.

**Unexpected Input**

To avoid logic bugs arising from unexpected input, we should ideally perform type checks on three different levels:

* The Client-side
* The Server-side
* The Database

All of these should include checks like: input type tests, pattern matching, required/optional fields, parameter size checks, and other types of tests that allow us to only accept exactly what we want, and nothing else. Of course, we do not need to keep manually repeating the code at every level, nor is this recommended, as it may introduce disparities and redundancies due to human error.

So, we can use tools like the Yup schema that was used in this application, which supports the front-end (e.g. React), the back-end (e.g. Node/Express), and even the database with mongoose using models, so we can use the same schemas across our web application. You may refer to the code of this application for examples on how to do so, and on how to derive MongoDB models from the Yup schema using mongoose. As for APIs, this is also possible through tools and plugins like tRPC, GraphQL, and others, which allow building and mirroring APIs on both ends, to ensure there are no conflicts or duplication of efforts.

If this is done accurately, it should adequately prevent unexpected input logic bugs. This is because both the server and the database would only accept values that accurately match the options we've set, and the front-end would also reduce server-side errors by only allowing these types, and not sending requests that may be rejected for these reasons.

Of course, it is also recommended to use strongly typed languages, like TypeScript in this case, as it would remediate most of these issues "but not all" during edit-time and before deploying the application to production. This also helps with null safety issues, as we'll see next.

In addition to the above, it is also recommend to write [unit tests](https://en.wikipedia.org/wiki/Unit_testing) for each function in the application, as they help greatly reduce and uncover potential type/null safety issues. They are also useful for confirming that the function's original purpose works as expected, which can be helpful after applying security patches.

**Null Safety**

To avoid null safety logic bugs, we need to ensure that our code never processes any variables if they were not assigned a value. As we have seen, this is easier said than done with languages that do no support sound null safety, but if using such languages was not an option, then we need to do whatever is possible to ensure null safety issues do not occur.

So, whenever we declare any nullable variables without an initial value, or whenever a function or endpoint accepts optional parameters, we need to either set a default value or perform proper null tests before proceeding an always wrap any code that uses these variables with try/catch blocks. Basically, we need to get the null under control!

For JavaScript, as shown in [this article](https://synergycodes.com/blog/how-to-do-your-null-checks-safely-in-javascript), we must use strict equality (===) and check for both null and undefined. We should also avoid using the not operator (e.g. (!count)), since this can be easily bypassed, as shown before.

As for optional user input parameters, whenever we do not use the .required() option (or similar), then we must do the above tests whenever these parameters are used, and can assume that the schema validation would not be enough. Furthermore, as we have seen before, we can utilize optional validation to ensure that certain parameters would be required or optional depending on the use case, which also reduces potential null safety issues.

In the end, the only real solution is to use languages that support null safety, as mentioned earlier. Even then, we must be careful when bypassing these measures, and only use the null safety bypasses (e.g. !) if we are absolutely sure that this variable at this point of the code would never be null. After all of that, we would still need to wrap such variables with a try/catch block to avoid disrupting the entire application.

**Other Tips**

Finally, there are a few other tips that may help us to avoid parameter logic bugs further. In general, we want to avoid using user-input whenever possible, and instead rely on data already stored on the database. For example, if we need a logged-in user's uid, then we can use their session to retrieve their uid from the database, instead of asking the user for the uid through the request.

Furthermore, it is always recommended to default to preventing access, instead of defaulting to allowing access. This is also important for front-end null safety, like mobile applications or modern web applications. For such applications, a minor null logic bug may allow users access to front-end paid-only material.

Of course, this also applies to back-end applications, like the resetPassword example we saw earlier, where the code only threw an error if the tokens did not match, and it would continue executing otherwise, which can be prevented as discussed in the previous section.

With these tips, and the general understanding of parameter logic bugs, one should be able to write code with solid logic, and avoid introducing such vulnerabilities.