**Validation Logic Disparity**

Parameter Manipulation logic bugs are primarily caused by direct modification of the user input to affect how the application reacts to it. So, throughout the module, we will focus on user-input and how the application logic handles it under different conditions.

We will start with Validation Logic Disparity logic bugs, which are caused by not applying the same validations on both the front-end and the back-end, leading to a disparity. Such a disparity can cause a variety of logic bugs depending on where the disparity occurs "missing on front-end or back-end", as we will see next.

**Client-Side Logic vs Server-Side Logic**

Most modern web and mobile applications nowadays utilize input validation filters to ensure the user passes the expected input and avoid the various vulnerabilities that may arise from lack of input validation. The developers need to implement these filters on both ends, since it is needed on the back-end to prevent malicious requests, and on the front-end/client-side to provide a better user experience and reduce the number of requests the back-end may reject for not meeting the expected format.

We are also well aware that only relying on client-side validation filters is not safe, as attackers can often attack the back-end directly, which would be vulnerable if it did not apply proper input validation. However, if both the client-side and server-side do apply input validation, does this mean the code would always be safe from type manipulation?

The answer is no. Issues may arise when there is a lack of 1-to-1 parity between both sides. This means that the front-end and the back-end do not apply the same validation filters, or if the back-end trusts validations and filters applied by the front-end, and does not re-apply them to confirm. If any validation filters are missing from either side, then it would lead to a disparity.

For example, the Unreleased iPhones case I discussed in the intro section suffered from this exact issue. When we add an item to the shopping cart in the front-end, it sends a request to the back-end requesting it to add a certain productId to the user's shopping cart (likely stored in the database for persistence between devices and user sessions). I am sure that the back-end must have had all types of input filters to avoid issues like SQL injections and other types of injections.

However, the front-end relied on the availability count from the database to either show add to cart or out of stock/coming soon buttons, which was the only validation against ordering out-of-stock items. The back-end did not revalidate the availability of requested item, and trusted that all requests sent from the front-end to be in-stock.

Of course, this does not account for manual requests sent to the back-end (e.g. by malicious actors), so the missing availability revalidation on the back-end caused this logic bug. Furthermore, the front-end did not have any re-validation to re-check the availability of cart items when sending a purchase request, which is a test that should be in place.

**Types of Validation Logic Disparity Bugs**

As we can see, this issue arose because the client-side logic was not in parity with the server-side logic, and each was coded with different assumptions, leading to this loophole.

So, Logic Disparity bugs occur in web and mobile applications when the front-end does not apply the same filters as the back-end, leading to a disparity in the validation logic that causes such bugs. Both sides must apply filters and validations using the same logic, and the back-end should always re-apply the same validations and filters the front-end applied.

If the front-end is missing some filters that the back-end applies, then this may lead to bugs in the app's user experience, like saying an item is out-of-stock when, in fact, it is not. On the other hand, if the back-end is missing some filters or not re-applying the ones used in the front-end, then this may lead to a logic bug we may exploit, which is what we are looking for.

A diagram of a computer

Description automatically generated

**Identifying Logic Disparity Bugs**

If we had access to the source-code of both sides, then we could simply look for validation functions and compare them between the client-side code and the server-side code. If we notice any discrepancies, then we can add them to our list for testing. But let's assume that we only have access to the back-end code, as this is often the case for whitebox pentesting and secure coding exercises, since more emphasis is usually put on the back-end as it contains the sensitive functionality and control of the application. We should not need access to the front-end code as long as we have a working web/mobile application.

To look for Logic Disparity bugs, we need to test functions that have the following aspects:

1. Accept user input
2. Apply client-side validation on this input
3. Relies on data from the back-end to adjust how the validation test works

This means we are interested in dynamic validation tests, rather than static ones. A static validation test always works the same, like applying the same filter to all inputs (e.g. tries to ensure input matches the email format). Dynamic tests, on the other hand, work differently, often based on data retrieved from the back-end, but sometimes they may rely on other information, like the date or time of day. Once we identify these dynamic validation tests, we can compare them to their back-end counterpart, to ensure the back-end also applies the same filters after receiving our input, and potentially identify some disparities in some of these functions.

In summary, a function vulnerable to Logic Disparity would rely on front-end validation (based on data provided by the back-end), and would not re-validate the input coming from the front-end. In other words, not all of the validation tests are carried at both ends, as the back-end would perform some of them and the front-end would perform others, thus creating a disparity between the two ends.

If all of this sounds a bit overwhelming or complicated, it will make much more sense once we start going through our exercises in the next section.