A function's logic is usually developed around a specific input type, so it is always essential to verify the input type and validate that it matches the expected format before processing it. If an application fails to consistently do so, it may receive an unexpected user input, potentially leading to unintended logic bugs.

With such logic bugs, the issues usually arise due to loose typing of variables, where an input variable is not strongly typed and can accept multiple types of variables depending on the input. So, let's start by looking into the differences between strong types and loose types.

**Strongly-typed vs Loosely-typed Languages**

Strongly-typed languages, such as C#, Java, and TypeScript, require variables to have a specific data type upon declaration, and assert that you must match any function's parameter types. As we will discuss in a bit, some of these languages may allow changing the variable's data type during run-time "dynamic typing", but the above rule applies nonetheless to whatever the current type is (i.e. current type must match function's input type).

Loosely-typed/weakly-typed languages, on the other hand, such as JavaScript and PHP, allow declaring variables without a specific data type. This allows variables to receive values in multiple possible types, and we can even run entire scripts without needing to specify any data type for any variables. This is why loosely-typed languages support strict equality (i.e. ===), to ensure both the value and the type are matched. This is not needed in strongly-typed languages, as the type always matches.

For example, in our JavaScript "node" server, if var or let are used for the parameter that will hold the user input, the variable can become a string, an integer, a double, and a number of other formats, depending on the received user input. Some other languages, like PHP, can even handle arithmetic calculations between strings and numbers (e.g. "10" + 10 = 20), which clearly demonstrates loose typing.

We have already seen in the [Whitebox Attacks](https://academy.hackthebox.com/module/details/205) module that this can lead to serious vulnerabilities. For example, Type Juggling can lead to authentication bypasses if loose equality (i.e. ==) is used for comparisons instead of strict equality (i.e. ===).

A diagram of a strong and weak

Description automatically generated with medium confidence

Then we have static and dynamic typing. Static typing simply means that once you assign a type to a variable, this type cannot be changed during run-time, and only values of that type can be assigned to that variable.

As for dynamic typing, variables can change their data type depending on their usage during run-time. While this allows the application to accept variables in any format (e.g. accept "count" as either "10" or 10), which reduces potential type-checking errors, it may also lead to major vulnerabilities and logic bugs.

For this module, we will see how loose and dynamic typing can lead to serious logic bugs. Later on in the module, we will also go through bugs caused by null variables, which are found in languages that support dynamic typing (both strongly/loosely typed).

**Types of Unexpected Input Bugs**

There are numerous types of unexpected input logic bugs, but one of the most common examples is using negative values in functions that are designed to accept positive values only (e.g. shopping carts). This may mean that a user would not need to pay anything, and in some cases, it may even credit the user's account balance with money after placing an order!

For example, if a cart allows adding a negative amount of an item, it would decrease its total price instead of increasing it, potentially leading to a zero or a negative charge. An even simpler example is when an app accepts any value for tipping, so adding a negative tip can reduce the total price, potentially all the way to zero (or less!), as shown in a real-case example below "posted by a Reddit user".

A screenshot of a bill

Description automatically generated

Another case of negative values logic bugs can be found in legacy banking applications, where transferring a negative amount of money to another customer would cause you to take money from their account "basically stealing". For example, user X transferred -$10 to user Y, so Y gets $10 withdrawn from their account and deposited to X. All of this is caused by simply not verifying that the transfer amount is positive.

Though very common, negative values aren't the only possible way to manipulate user input types. Other examples include sending longer than expected strings, manipulating intended string formats, manipulating the way an input is converted or processed, and so on. Basically, any logic bug caused by input type manipulation can be classified as an unexpected input bug.

**Identifying Unexpected Input Bugs**

We will utilize the function search and filter methodology to identify this type of logic bugs to reduce the code scope we need to manually test and review. First, we can identify functions that utilize any form of direct user input. As mentioned previously, some functions may utilize indirect user input and be vulnerable to the same type of bugs. Still, for this module, we will focus on direct input to understand the logic bug, and exploiting both can be done the same way once we have control over indirect input. For more on indirect input and second-order vulnerabilities, check the [Modern Web Exploitation Techniques](https://academy.hackthebox.com/module/details/231) module.

The second step would be to limit our tests to functions that use loose variables (e.g. var or let) and/or loose comparisons if they perform any type checks on the user input (e.g. == instead of ===). Such functions may accept different types of input than intended, especially if they do not perform any input validation.

**Note:** Don't forget to re-download the source files for this section, as they may be slightly modified from the sources of the last exercise. The app itself is, however, the same.