**Local Testing (functions) - Null Safety**

Now that we have identified a couple of potential null safety issues, it is time to try and take advantage of these flaws, which may be easier said than done, as mentioned previously. With injection and type-safety vulnerabilities, we can use a variety of payloads to bypass potential security measures, and can modify them to change our attack vector.

However, with null safety attacks, the only control we have is sending a null variable, and then seeing how the server handles it. So, we can basically use one value, which is null, and we don't have much control other than that.

**Common Effects of Null Safety Issues**

It is important to note that the vast majority of null safety issues lead to run-time errors or crashes the back-end server process, which often causes a slow or temporary denial of service, and in some cases takes the whole server down. This alone, of course, is a major issue and should be taken very seriously, as denial of service attacks are amongst the most used attacks by malicious actors, especially if they can cause them without relying on a vast network of bots (i.e. DDoS).

In addition to that, there are certain cases where a null variable may cause other issues, especially when paired with another type of logic bugs or vulnerabilities, like corrupting part/all of the back-end database, or even bypassing certain security measures. However, this largely depends on the target application and how it's designed.

**Update User Details**

As we have seen in the previous section, the name and username parameters are optional, so if we assign a null value to them, then they may be stored as such in the database, which can have multiple potential effects. For example, we can search for the uses of these parameters throughout the application, and see when the user details are retrieved from the users database collection. It is likely that null checks will not be done before using these values, as values retrieved from the database are usually trusted to have been filtered and validated before being stored. If this is the case, we can potentially crash the server, or do something else depending on the vulnerable logic.

Before doing that, however, we need to first ensure that we are indeed able to store null values in the database. To do so, we will not include the "name" and "username" parameters in the JSON object, since using an empty value would simply be evaluated as an empty string, which is not null and would not cause the same issues we are discussing. So, let's start with the following POST body, and only include the email parameter, as it is a required field:

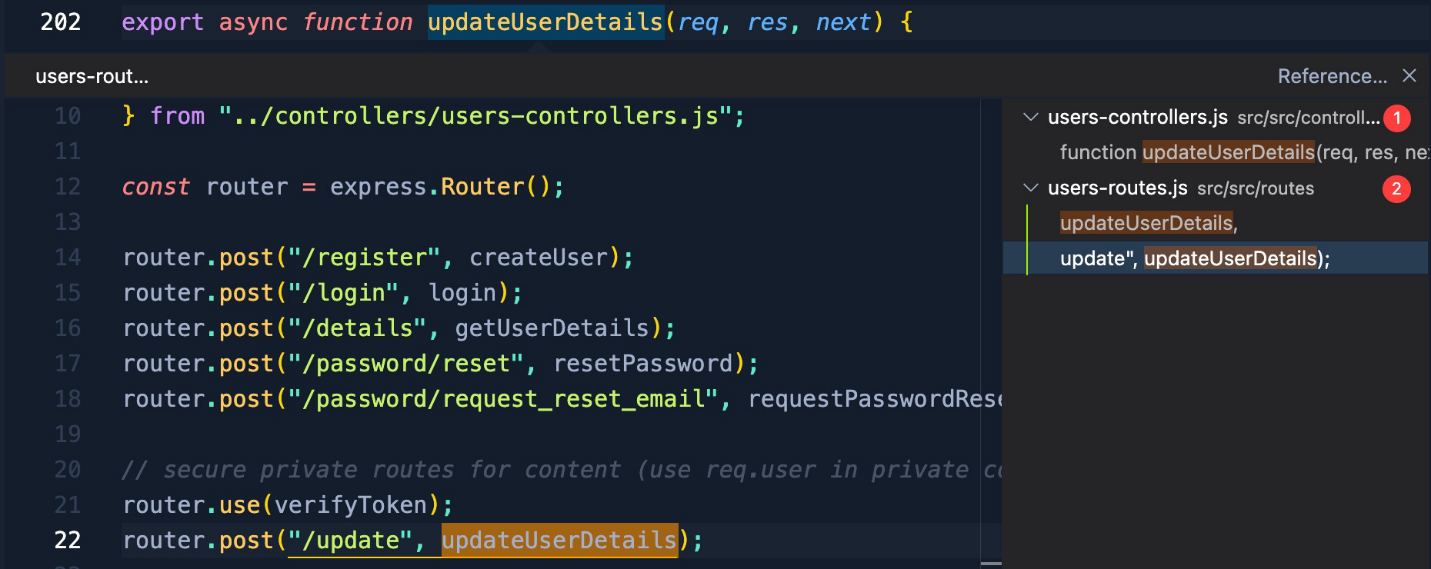
Code: json

{

"email": "htb-student@academy.htb"

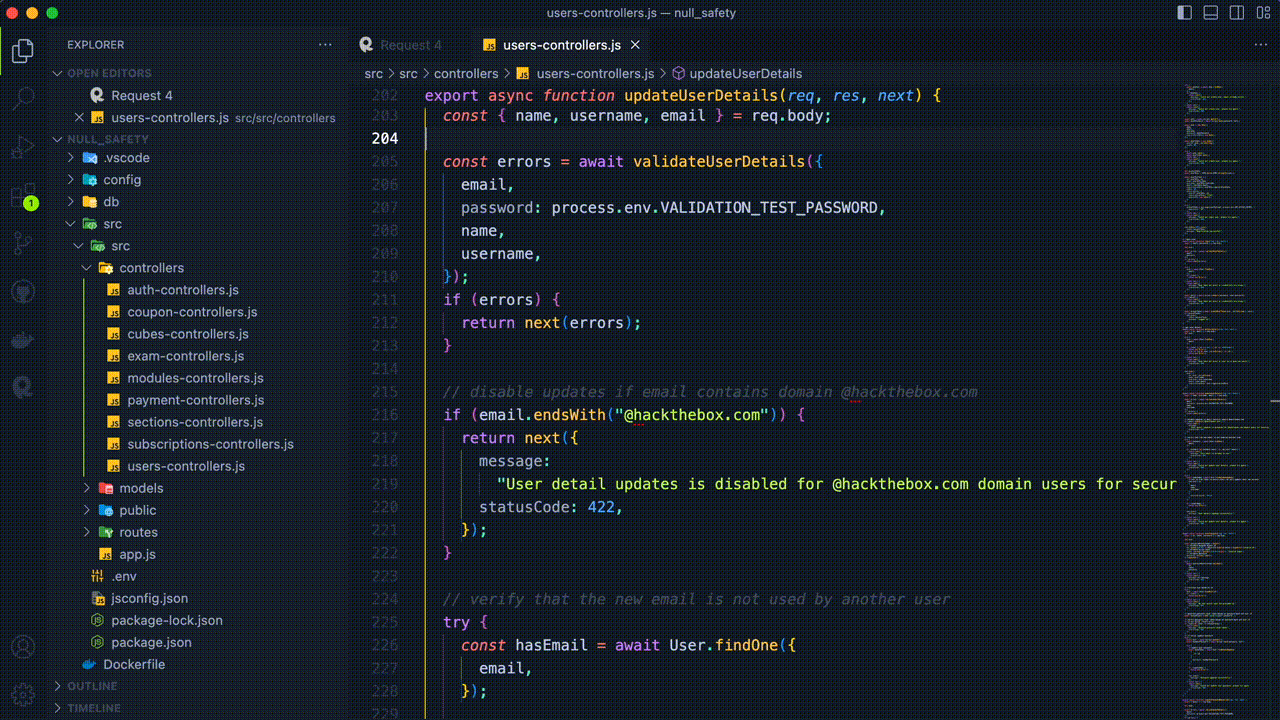
}

To find the route of the API endpoint, we can CMD/CTRL click on the updateUserDetails function name, and we will see that it is being referenced in the user-routes.js file:

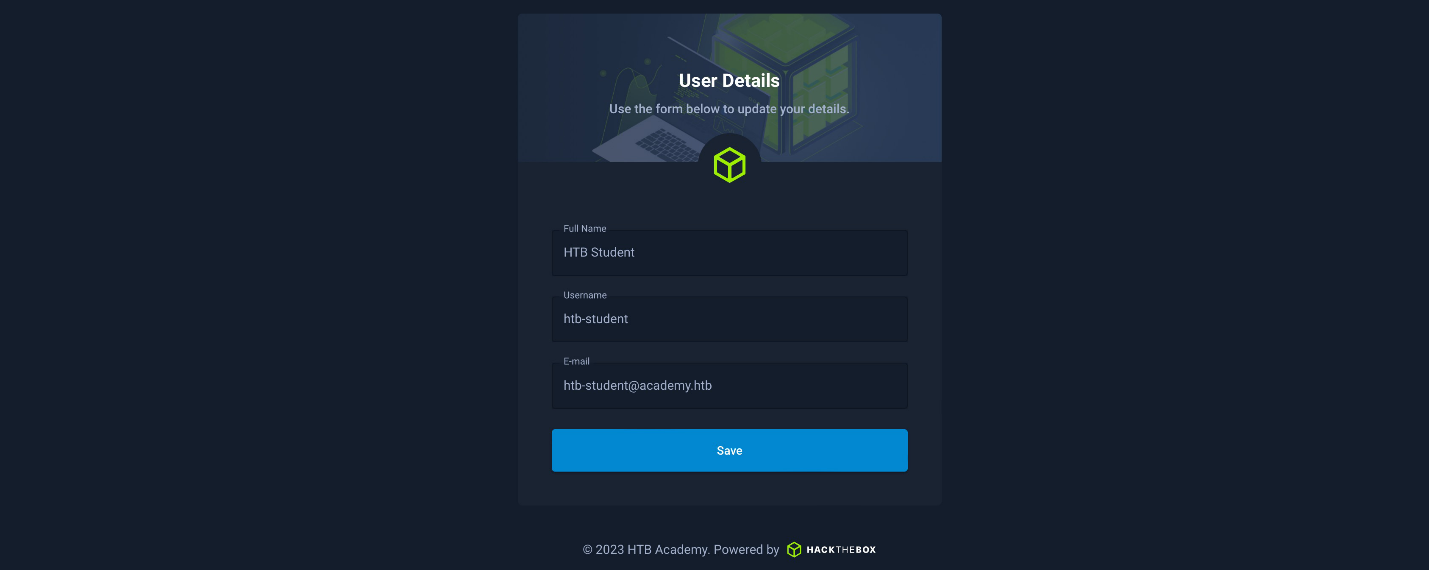


This tells us that we need to send a POST request to /update under the user routes (defined in app.js as /api/users/), so the final request would be to /api/users/update. We also see that this comes after verifyToken, so our request needs to be authenticated with a token. This is also evident by the use of req.user?.id within the function, which is set by decoding the auth token to obtain user details.

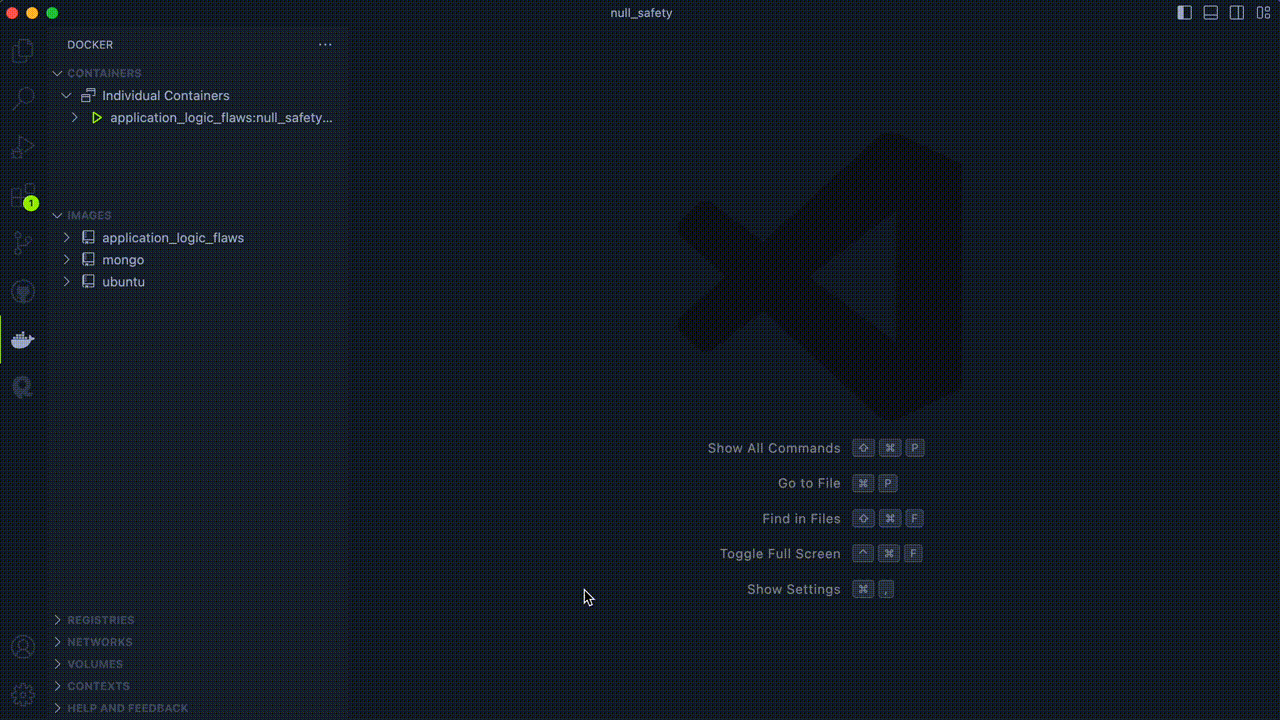
So, we can simply copy the token from our browser session (you may also obtain it by sending a login request, if you're feeling adventurous), add our body payload, and send the request. Before doing that, however, we will set a breakpoint at the beginning of the function, and will add both of the above variables to watch, to ensure they are received as null:



The application did not crash, and simply said "User details updated successfully!". So, did we corrupt these parameters in the database, or did we not cause any harm? Let's login again through our browser, and navigate to /settings to see what our updated user details look like:



Surprisingly, the name and username parameters are neither null nor empty, and they seem to have not been affected at all by our request. We can further verify this by checking our user record within the database inside the Docker container, and we will get the same unchanged values:



It appears that since the code uses a database update call (findByIdAndUpdate), when we send a null value, it will simply not update the parameter, as it will be considered as "no change". This is a stroke of luck! If the code used a different database call, like (set) for example, then we would have been able to set the values to null, and may be able to cause the issues we mentioned previously if null values are stored in the database.

While our attack was unsuccessful, the code definitely has a bug and should be patched, only we could not take advantage of it. Let's hope for better luck with the next issue.

**Reset Password**

Let's now move to the second issue we have identified, which lies within the resetPassword endpoint. We have seen that all parameters are optional, but none of them seems to get properly verified for null safety before being used. Let's go through them one by one, and check if the use of these parameters may cause run-time errors or logic bugs.

Starting with id, as we have established previously, the application will error out when the database does not return any results based on a null id:

Code: js

if (!user) {

throw new Error();

}

Some types of databases may crash or return all results when the same is done, but this is handled correctly given the use of MongoDB, and we can consider it as safe (while still recommending against the use of !).

As for the token parameter, as we've discussed in the previous section, it is mainly being used as a verification check to allow the user to reset their password, since the token is calculated based on secret values. However, the issue that immediately stands out is how the token null check is being used here:

Code: js

if (token && token !== hashedToken) {

return next({

message: "Invalid password reset token.",

statusCode: 403,

});

}

We have already established that developers must check for null values before using any variable. However, the code here is not doing that properly. First, it is not using strict equality (i.e. ===) to compare the variable with null AND undefined, but simply doing if (token), which can be problematic as we've seen before.

Still, this is not the main issue here. The main logic issue lies with the fact that the code will only verify the token if the token exists! So, if we don't provide any token, this entire token verification will be skipped, and the code would proceed to change the password!

For example, if the token is null, then if (token) would evaluate to false, and so the second condition in the if statement would never be tested. This is a very crucial error that many developers make, which is defaulting to success instead of defaulting to fail. It is always recommended not to proceed with any sensitive functions until the user proves they have access, instead of proceeding with execution unless the user was proven not to have access. This particular issue is more related to access control issues, but the logic of defaulting to success is a logic bug, and must be noted here.

Proceeding with the final parameter password:

Code: js

// if valid, update password

else {

const salt = await bcrypt.genSalt();

const newHashPassword = await bcrypt.hash(password, salt);

try {

// update user password

const updateReq = await User.findOneAndUpdate(

{

\_id: id,

},

{

password: newHashPassword,

}

);

<SNIP>

}

}

From a first look, this may appear to be the least problematic parameter, as it is only used to modify the password once we pass all checks, right? Not exactly. We see that this parameter gets used with the bcrypt function to generate a hash for the new password to be stored in the database. However, this is not wrapped within a try/catch block, which is the worst thing you can do with null variables, as this may crash the entire application if it encounters a null, instead of simply returning an error.

Even if the only way to reach this line of code was through having a valid password reset token, any user may request a real password reset link and obtain a real token for their user. Then, if they simply remove the password parameter from their passwordReset request, they may potentially crash the entire application for all users.

In the next section, we will try to test both of these claims, to see if we can indeed exploit these null issues.