15. Parameter Logic Bugs

Setting Up

Throughout the module, we will rely on VSCode and some of its plugins, which should also be helpful in your future code review and whitebox pentesting exercises. The main plugins that we will use are the following:

- RapidAPI
- Docker for VScode

The web application we will be testing throughout the module uses the MERN stack (Mongo, Express, React, Node). We wanted to avoid having you to go through the hustle of setting all of these tools and technologies on your own machine, and then reconfiguring them for each exercise.

So, we created `a separate Docker image for each exercise` that includes both the front-end and back-end codes for the web application, as well as setting up the Mongo database and everything else. This also allows easy debugging of the web application, and easy restart/reset in case anything goes wrong in our testing.

hide01.ir

So, in addition to the above tools, please make sure that you have Docker setup and installed on your machine.

Note: Dockerizing web applications is an important skill to learn, so you can use this opportunity to learn how it's done and use it in your future whitebox pentesting exercises. Another option for larger web applications is to set up a test VM replica of the production server, though this usually takes a lot of effort and isn't easily deployable elsewhere.

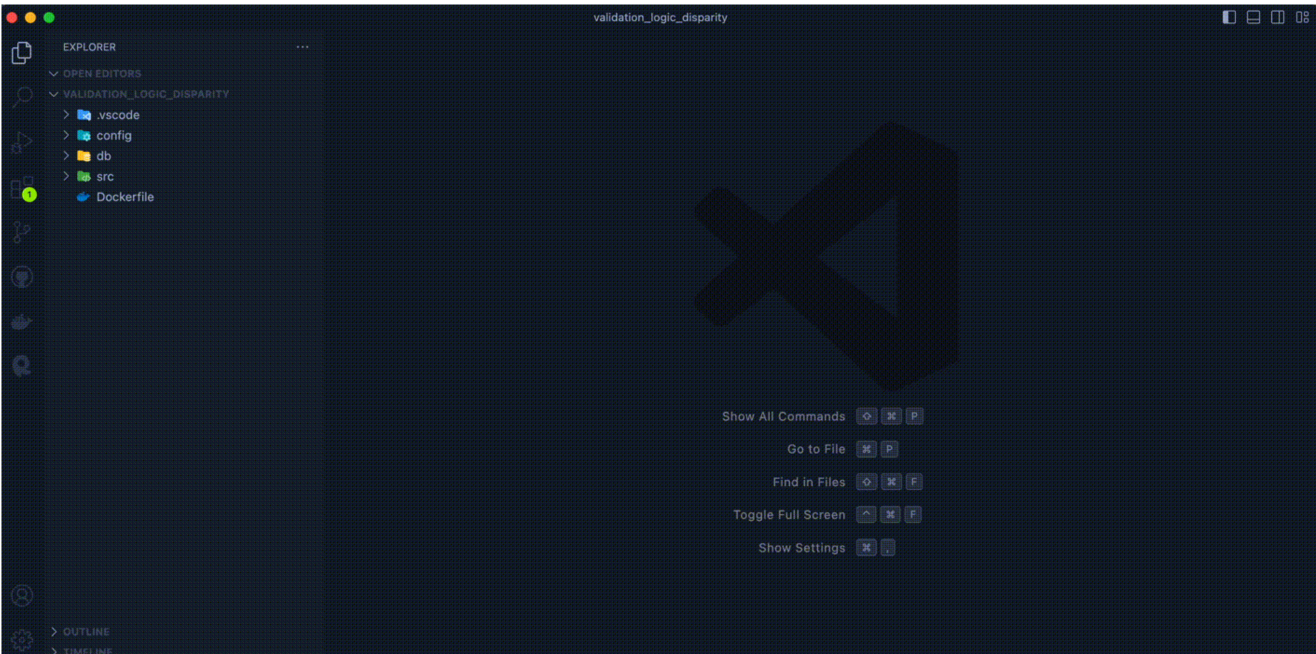
Running Locally

Once you have the above tools set up and running, you can download the provided zip archive at the end of this section, extract its content, and then open the folder using VScode, using `File` > `Open Folder`, or through the terminal with `code ./validation_logic_disparity`.

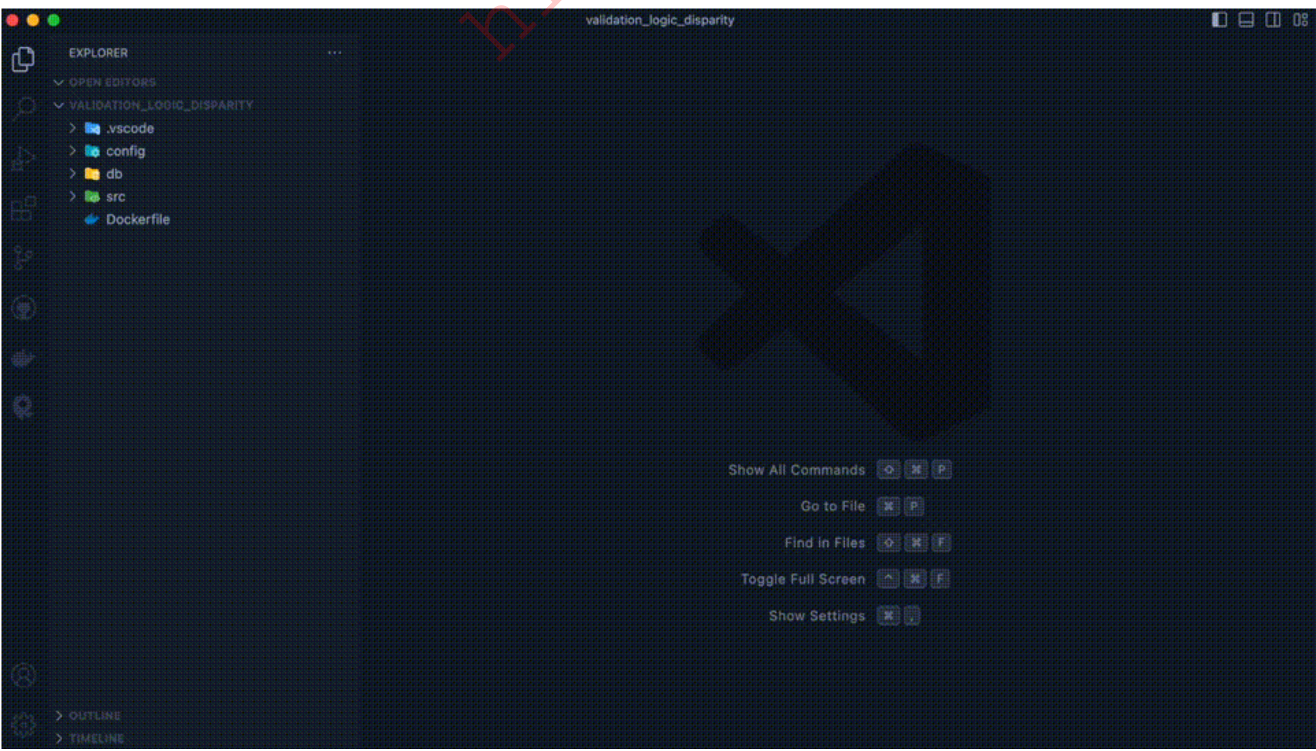
Once that's done, you can open the `Dockerfile` within VSCode, right click on the opened window, and select `Build Image`. This will prompt you to enter a tag for the image, so you can either keep the default value of the folder's name, or use a common host for all exercises under this module (e.g. `application_logic_flaws:validation_logic_disparity`). Once you do so, the docker

https://t.me/CyberFreeCourses

image will start building, which should take 5-15 minutes depending on your machine and internet speeds.



Once the build is done, you can find the new image in the `Docker` icon in the VSCode Sidebar on the right, under `IMAGES`. You can right-click on and select `Run`, and the image should start up. Give it a few seconds to load up everything, and then visit `http://localhost:5000/` with your browser, and login with the credentials provided in the Dockerfile, as that user may have some privileges like cubes or payment cards added under their user. Some tests and exercises may require you to create your own user, so that's another option.

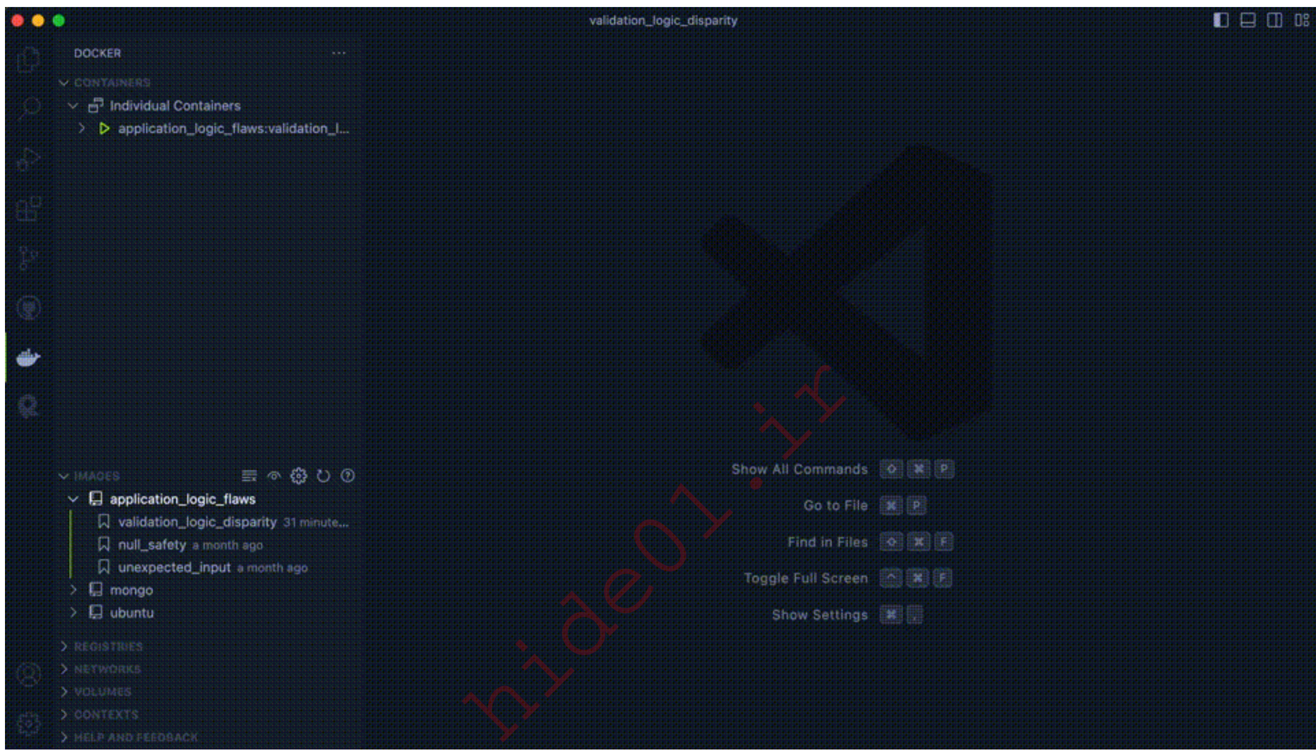


Tip: You may also select `Run Interactively` to keep an eye on the logs, like the back-end server error messages, which will show up in a window within VSCode.

https://t.me/CyberFreeCourses

Local Testing

As part of the whitebox pentesting process, we usually need to perform local testing and debugging, as mentioned in the previous section. Luckily, the Docker images we just setup also enable easy debugging of our code. All we need to do is go to the `Run and Debug` tab in the VSCode sidebar, and click on the `Run` icon next to the text `Docker: Attach to Node`. As the Docker image is already configured for debugging, this will attach to our web application for debugging, and the bottom bar of VSCode should now be red to indicate we are in debug mode. You may also `pause`, `restart`, and `detach` from the debug session using the hovering session buttons.



hide01.ir

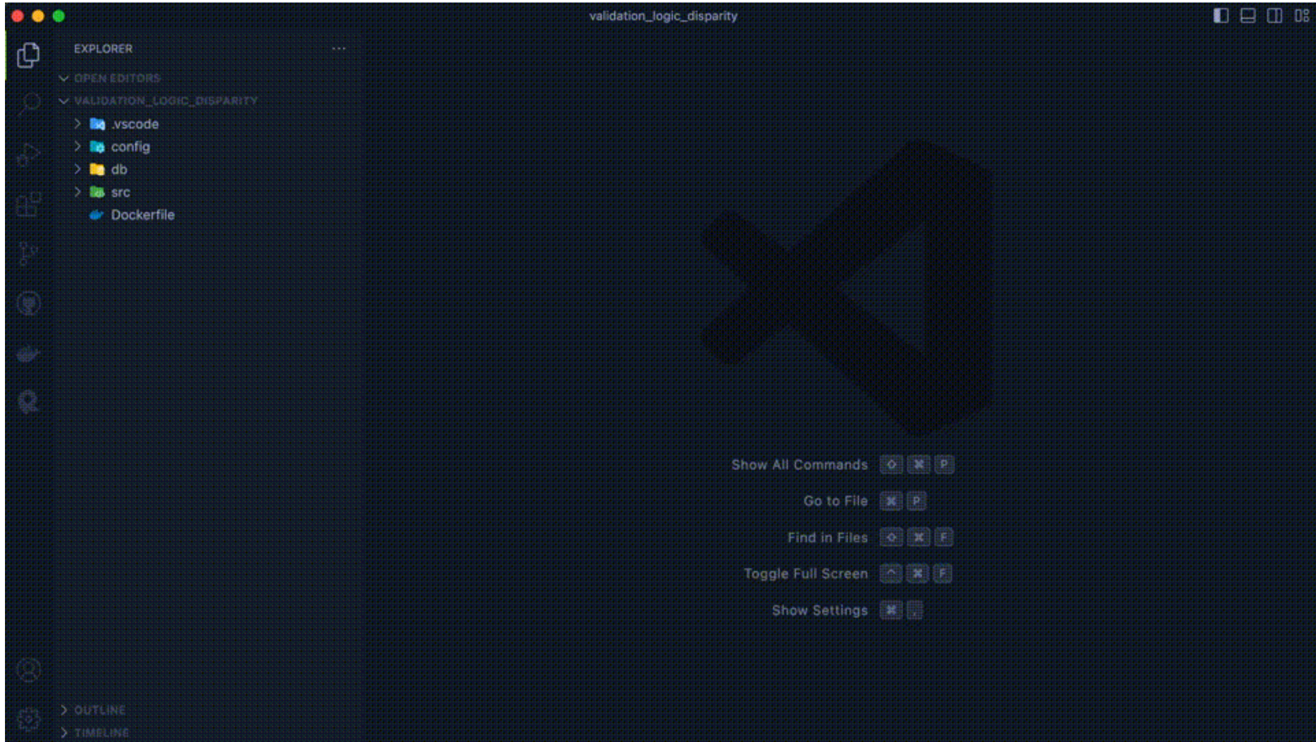
In the coming sections, we will go through more details on how to utilize debugging in our local testing. But as an example, we can add breakpoints by opening the file we want to add a breakpoint in, and go to any line and click on the line number, and this should add a breakpoint. Now, whenever the code reaches this line, the application should break and stop on that point, so we can examine everything and then resume its execution.

Note: It is important to note that even though we can set breakpoints using the code files opened in VSCode, any code changes will not be reflected in the running container. Though this isn't something we will need to do in this module, if you ever need to modify code in a running container, go to the `Docker` tab, find the running image under `CONTAINERS`, then you can click on it to expand it and show its files. You can find the web application files under `/app`, and right-click and select `Open` to open and modify any of them within VSCode.

Another thing we may need to modify and debug throughout our local testing is the `database`. To do so, we can go to the `Docker` tab in VSCode, find our running image under `CONTAINERS`, right-click on it and select `Attach Shell`, and this should open a new window

https://t.me/CyberFreeCourses

terminal in VSCode and drop us into a shell within the running container. Now, we can simply run the `mongosh` command, and should have a MongoDB shell with access to our database.



Note: This module assumes that you are already familiar with MongoDB and how to run basic commands, as well as understand the basic usage of MongoDB with NodeJS. If you are not, you may give it a quick read online to familiarize yourself with it, or check out the Introduction to NoSQL Injection module, which introduces MongoDB and gives a brief about how to use it.

hide01.ir

Application Structure

Before we move on to our first topic, it is worthwhile to take a quick look on the general structure of the application we will be testing. We can start by opening `src/src/app.js`, and going through it.

We will see that the application starts by importing some files and libraries. After that, the application sets up Node/Express along with other configurations, as well as setting up `bodyparser` and `CORS`.

```
const app = express();
const port = parseInt(process.env.PORT || "5000");
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// set up body parser and cors
app.use(bodyParser.json());
app.use(cors());

// serve public assets
```

https://t.me/CyberFreeCourses

```
app.use(express.static(path.resolve(__dirname, "public")));
```

Then, the application sets up the main API middleware with `app.use()` and specifies the routes file for each route. Then, it handles 404 and general app errors.

```
// set up API routes
app.use("/api/auth", authRoutes);
<SNIP>

// forward all other requests to react app, so react router can handle
them 'will handle 404 errors'
app.use((req, res, next) => {
  res.sendFile(path.resolve(__dirname, "public", "index.html"));
});

// handle 404 errors
app.use((req, res, next) => {
  res.status(404).json({
    message: "Could not find this route.",
  });
});

// handle next() errors and general errors
app.use((error, req, res, next) => {
   <SNIP>
});
```

hide01.ir

Finally, it sets up the MongoDB connection and starts listening on the port specified at the beginning. We can read the comments to get a better idea of exactly what each code block does.

```
// start the Express server & db connection
set("strictQuery", false);
connect(process.env.DB_URL ?? "", {
  dbName: process.env.DB_NAME,
  user: process.env.DB_USER,
  pass: process.env.DB_PASS,
})
  .then(() => {
   <SNIP>
});
```

So, the main thing that we need to focus on is the `APIs and their routes middleware`. We can CMD/CTRL click on any of them, like `authRoutes`, and VSCode should open it in its

https://t.me/CyberFreeCourses

corresponding file.

```
const router = express.Router();

// secure private routes for content (use req.user in private controllers)
router.use(verifyToken);
router.get("/update_token", updateUserToken);
```

If we examine the routes middleware, we will see that it is linking each sub-route to a specific function, like `/update_token` to `updateUserToken`, which we can also CMD/CTRL click on to further review, as we will do in upcoming sections.

Take some time to navigate through the code and familiarize yourself of how it's running. In general, the files we will be most interested in are under `/controllers`, as well as `/routes` and `/models`.

With that, you should be able to run this application and debug it, as well as have a general idea of its general structure. In the next section, we will start with our first topic and will go through the code to identify potential issues and logic bugs.

hide01.ir

Introduction to Logic Bugs

In web application testing, we usually follow rules and techniques when testing and identifying vulnerabilities or bugs. However, some bugs fall into the 'logic bugs' category, which can be notoriously difficult to identify because they are primarily caused by logic flaws.

Despite their complexity, a thorough understanding of logic bugs enables us to create guidelines for their identification. Additionally, adhering to a secure coding methodology during web application development can help eliminate or reduce logic bugs, ensuring the creation of robust applications without logic flaws.

Because this may sound confusing and overwhelming, let us start by explaining logic bugs and go through some examples to understand them better.

What are Logic Bugs?

`Logic Bugs` are unintentional flaws in how an application processes user input or interactions. Unlike common bugs that can cause crashes or common vulnerabilities that lead to code execution, logic bugs alter how an application behaves under normal conditions.

For instance, consider an application with paid features. A logic bug in the application's code might inadvertently allow access to these features without payment. Logic bugs in the

https://t.me/CyberFreeCourses

application's flow can often be exploited through normal usage of the application (e.g. front-end only), while parameter-related logic bugs usually require specialized tools for exploitation.

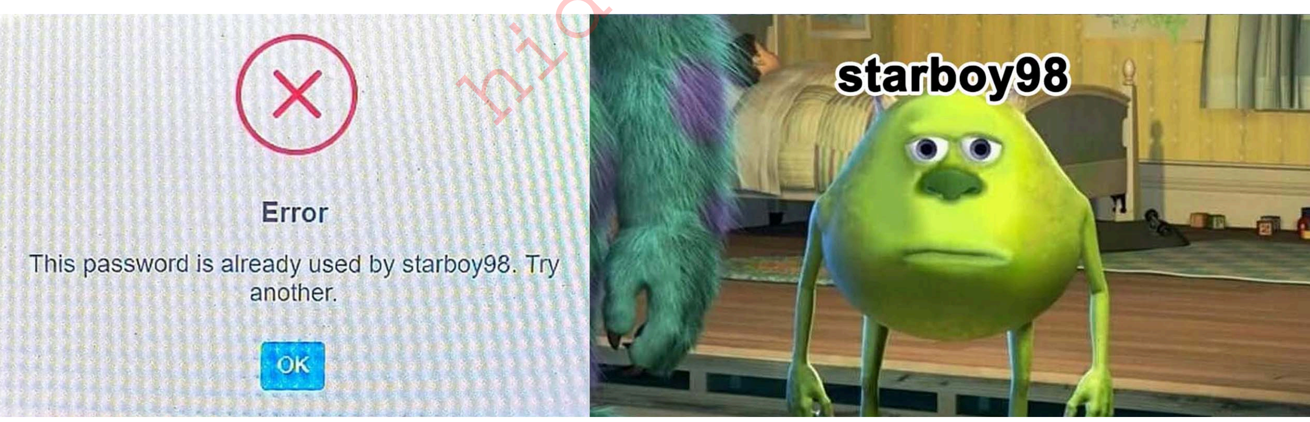
Such flaws would not constitute direct vulnerabilities, nor would they be clear bugs that crash the application or cause a malfunction, but rather cause an `unintended behavior` that we can take advantage of. These bugs can remain hidden during normal application usage, causing no apparent issues. The application may function perfectly, revealing its flaws only under specific conditions that the developer might not have considered during the initial design. This inherent subtlety makes logic bugs particularly challenging to detect. Moreover, it is crucial to understand that logic bugs impact an application's flow and behavior, distinguishing them from issues that cause crashes or malfunctions.

While automated tools have their merits, identifying logic bugs poses a unique challenge, as they often struggle to fully comprehend, parse, and interconnect the intricate logic within an application's codebase and design. However, the landscape may soon change with recent Artificial Intelligence (AI) advancements.

The most reliable approach to identifying logic bugs remains rooted in human expertise and logic. These skills, combined with the guidance and practices we will explore throughout this module, including codebase reviews and static tests, are pivotal in our pursuit of identifying and addressing logic bugs effectively.

hide01.ir

A Trivial Example



The above is a basic example of a Logic Bug, albeit quite comical. It was likely caused by a single line of code written without a proper understanding of its impact, leading to this bug. The developers may have used the `unique()` keyword for the `password` database parameter, which entails that every user's password must be unique from everyone else's! This keyword is usually used for emails and usernames, as such parameters need to be unique. However, in this case, whether it was a mishap or intentional, it caused this logic bug.

Furthermore, the application must have been configured with verbose logging, as it appears to show the direct database error to the front-end users, instead of showing a more generic error (e.g. The password is not unique). This basic example shows us the essence of logic

https://t.me/CyberFreeCourses

bugs, how they occur due to minor mishaps, and how impactful they may be when they are found in sensitive functions "revealing another user's password, in this case".

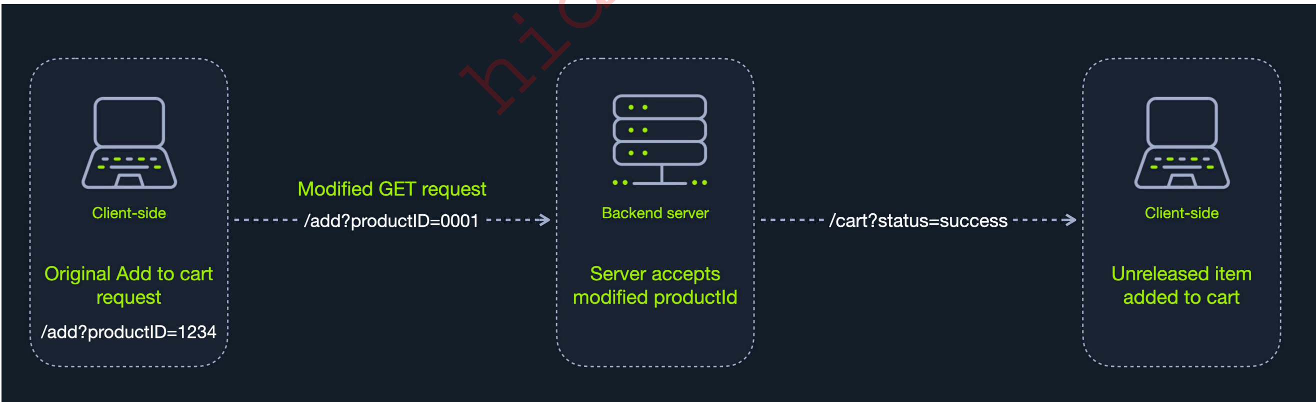
A Real World Example (Unreleased iPhones)

Note: I will modify some facts and vulnerable components to protect the website's identity; however, the idea and concept should be identical.

One of the earliest instances of a web logic bug I faced was in an online e-shop selling electronics (e.g., Amazon, Walmart, or Target). The iPhone 4 was a day away from release in our region (back then, they spread releases over a longer period), and the product page only showed 'coming soon' instead of the 'add to cart' button. Personally, whenever I try to test a web application for logic bugs, I always start with 'I wonder what would happen if...', and so I wondered whether I would be able to complete the purchase if I somehow managed to add this 'unreleased' item to the shopping cart.

Adding the 'unreleased' iPhone to the shopping cart was a straightforward client-side bypass. I modified the `productId` in the GET request when adding any other item and replaced it with the unreleased iPhone's product ID. Once added to the cart, instead of displaying "coming soon," the cart revealed the available quantities for each iPhone option. Consequently, I selected the iPhone configuration I desired and chose the option 'pick up from store' because the "delivery" options appeared incorrect, likely due to the web application's inability to handle an "unreleased item." Below is a diagram showcasing the general flow of the attack:

hide01.ir



When clicking the checkout button, I was confident that the back-end server would handle this error and inform me that the product was 'out of stock' or 'unavailable.' However, to my surprise, I was directed to the payment details page and completed the purchase. I still thought that they would automatically cancel the order. However, to my amazement, after the iPhone was released to the public the next day, I received a pickup appointment and a purchase confirmation. Thus, this bug made the iPhone 4 (in my region) available for purchase before its release date.

I mentioned this incident before picking up the product, but they informed me it was rightfully mine since it had been paid for. As a result, I bypassed the waiting line and placed the order before the release. I promptly reported this bug and suggested a fix. However, as is often the

https://t.me/CyberFreeCourses

case, no action was taken to address it. My intention had always been to identify and report potential logic bugs rather than exploit them for personal gain.

Logic Bugs CVEs

The iPhone 4 example only illustrates one aspect of logic bugs: they can encompass much more and have a more substantial impact. For instance, macOS systems are recognized for their robust Code Execution protection, which only allows code execution with the user's consent. Nevertheless, in 2021, a security researcher identified a basic logic bug that enables attackers to easily circumvent these protections by starting their script with (`#!`) and not defining an interpreter (e.g., `/bin/bash`).

Apple did not program the System Policy to handle scripts without an interpreter and considered them safe. This logic bug allowed for the bypass of Code Execution protection on Apple devices. However, `CVE-2021-30853` was not the only logic bug to achieve such bypasses. Another logic bug, disclosed in this 2014 post, resulted in the same kind of bypass.

Because they all depend on logic, web applications, mobile apps, operating systems, games, or any computer program ever developed can contain limitless examples of logic bugs because their logic can always be flawed.

Impact of Logic Bugs

As we will see throughout the module, the impact of logic bugs can vary from minor inconveniences to `financial losses`, `account takeovers`, `denial of service`, `privilege escalation`, and even `remote code execution`.

https://t.me/CyberFreeCourses

Many logic bugs may not be exploitable, due to a certain 'usually unintended' security measure in place. For such cases, there won't be a significant impact other than user inconvenience. However, as a general rule of thumb, the impact of a logic bug is directly related to the sensitivity and importance of the flawed function and its related data. We are always interested in logic bugs in sensitive functions or sensitive data, like item purchases or code execution.

By the end of this module, we will deduce that flawed logic design often stems from insecure coding practices, which frequently lead to critical logic bugs.

Types of Logic Bugs

By now, we should have a fundamental understanding of logic bugs. This section will discuss how we can categorize different logic bugs, discuss our methodology for identifying them, and set up a local environment to follow along with the module.

It is important to note that logic bugs are not as thoroughly researched and studied as other types of web vulnerabilities, such as injections or authentication flaws. So, when writing this module, we needed to come up with new categorization and identification methodology for logic bugs, which required studying and reviewing a great number of logic bug reports, research papers, as well as our own experiences. We hope that such efforts drive the industry forward into advancing the research in logic bugs, and that the categorization and methodologies we created here will help in doing so.

hide01.ir

Having a clear definition of what a logic bug is and its different types is crucial for this module, as it allows us to clearly define what qualifies as one. So, let's start by discussing the different causes of logic bugs, and then categorize each into its own type of bugs.

Causes of Logic Bugs

From what we have seen so far, we know that logic bugs are not only found in web apps, but in any application that follows any form of logic, which means that they can basically be found in any type of applications.

We can also understand why `automated tools provide little to no value in identifying logic bugs`, as they are caused by a flaw in the logic design, and tools would need to fully comprehend the application's logic and design to identify such an issue. For the same reason, tools like Web Application Firewalls (`WAF`) and `Anti-Virus` software usually `cannot protect against such vulnerabilities`, as malicious behaviour to exploit logic bugs usually looks like normal application usage.

`So, what exactly causes logic bugs?` As you may be able to tell, there isn't a single cause for all logic bugs, but certain common bad coding practices often cause them.

Perhaps the single most important cause for logic bugs is `having a weak application logic design`. However, designing a sound application logic is much easier said than

https://t.me/CyberFreeCourses

done, as it requires a firm understanding of the whole application's flow and process to do so. Throughout the module, we will provide different tips and methods for avoiding logic bugs, which should help build a more robust logic design.

A weak logic design can mean that our application does not know how to handle certain kinds of 'unexpected' inputs or conditions, and may default to the wrong action/function that leads to a logic bug. Many developers make the mistake of only designing the application logic to handle the intended use, and do not consider what users may actually do, or other unlikely outcomes of user actions.

A weak logic design can also be due to lack of proper testing of every possible scenario and type of input, or not programming the application to have a default way to handle any other type of input or any unexpected conditions. For example, a developer may use a `switch` statement for a specific type of input, but they do not default to a specific case. Or a developer may use an `if` and an `else if` statements, but not have a general `else` statement. This can be found in many other areas of coding, like `catching` specific types of errors but not having a general `catch` statement, and so on.

In addition to `weak logic design`, other factors can lead to logic bugs. One example is not having `parity between the front-end and back-end logic` in web applications, which may cause logic bugs even if the logic is well-designed. There are many other causes for logic bugs, and we'll discuss some of them throughout the module.

hide01.ir

Logic Bugs Types

To categorize logic bugs into different types, we need to consider the above-mentioned causes. Many online resources and research papers consider all types of web vulnerabilities (e.g. injections, file uploads, IDORs, LFI, etc) also to be logic bugs. While this is fundamentally true, as even a file upload vulnerability can be considered a logic bug (since the upload logic does not prevent certain file types), we will not be taking this route. Such vulnerabilities are usually caused by `weak filtering`, like injections, or `weak configurations`, like weak user access control. So, we will not consider such vulnerabilities to be logic bugs under our own definition.

Instead, we will be mainly focusing on vulnerabilities caused by a `weak logic design`. Furthermore, we will split this into two types of Logic Bugs: `parameter manipulation` and `flow bypasses`. In this module, we will focus on `Parameter Manipulation` logic bugs only, which has the following types:

Type	Description
Parameter Manipulation	An application's logic cannot properly handle a specific type of parameters
`Validation Logic Disparity`	Where validation logic varies between front-end and back-end.

https://t.me/CyberFreeCourses

Type	Description
`Unexpected Input`	Where bugs are caused by different types of input the application cannot handle.
`Null Safety`	Where bugs are caused by not properly handling null parameters.

As a forward look into `flow logic bugs`, here are the different types of bugs that fall under it:

Type	Description
Flow Bypass	An application's flow can be broken due to certain flaws in its design
`Repeating One-off Actions`	Where we are able to repeatedly use actions that should only be carried once.
`Out of Order Steps`	Where we break a multi-step process by doing steps out of order.
`Unexpected Behavior`	Covers all other flow logic bugs, usually caused by a user behavior the application isn't designed to handle.

hide01.ir

This categorization should cover the majority of logic bugs. We went through various logic bugs reports, and we found that most would always fall under one of the above categories. In the next section, we will go through the methodology and scenario that we will be following throughout this module.

Module Methodology

Now, let's discuss the methodology we will utilize throughout this module to identify and study the different types of logic bugs we just defined, and the scenario we'll follow to explain it.

Methodology

The methodology we will follow in this module is quite similar to the `Whitebox Pentesting Process` as defined in the Intro to Whitebox Pentesting module, which is split into four main steps:

1. `Code Review`
2. `Local Testing`
3. `Proof of Concept`
4. `Patching & Remediation`

https://t.me/CyberFreeCourses

We will start each logic bug type by explaining it in detail, and give a real example of how it may look like. Then, we will follow the above methodology/process, as follows:

Order	Step	Description
1	`Code Review`	We start by trying to `navigate a large code base` and `understand the application design` to `identify potentially vulnerable functions`. We will do so using both `static and dynamic analysis`, and we will combine app usage with code reviews to better understand the application logic.
2	`Local Testing`	With a list of interesting functions, we will begin `testing them for potential logic bugs`, and will learn through each section how to identify each type.
3	`Proof of Concept`	As this is a secure coding exercise and not a whitebox pentesting exercise, we will not be writing exploits. Instead, we will focus on `exploiting the vulnerable functions` based on our understanding from the previous two steps, as a way of 'proof of concept'. Then, we can confirm the vulnerability on the real target, but this must be done in a safe way that doesn't cause any downtime or data loss.
4	`Patching & Remediation`	At the end of each logic bug type, we will discuss how to patch this type of logic bugs and the example we just exploited. We will also test the attacks again to ensure it has indeed been patched.

hide01.ir

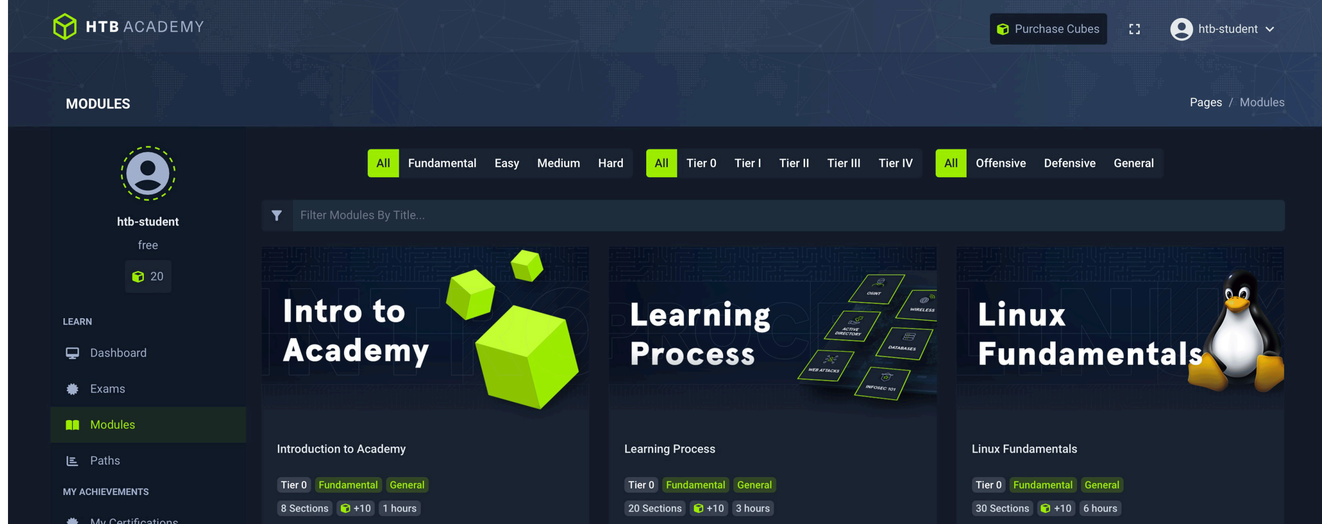
Before we start discussing the different types of logic bugs, let's take a quick look at the scenario we will be following throughout this module.

Note: As a defensive secure coding module, our main target is to identify and patch the logic bugs. We will still show how we can take advantage (attack) some of the vulnerabilities, though this is not our primary objective, so we will not go into a lot of attacking details, other than what's necessary to understand why the logic bug exists.

Scenario

To fully understand real-world `Logic Bugs` and thoroughly comprehend the exercises and module material, we need a fully fletched real-world web application that we are familiar with and can easily navigate to identify its inconsistencies. There is one web application we can be certain that whoever takes this module is familiar with, `Hack The Box Academy`!

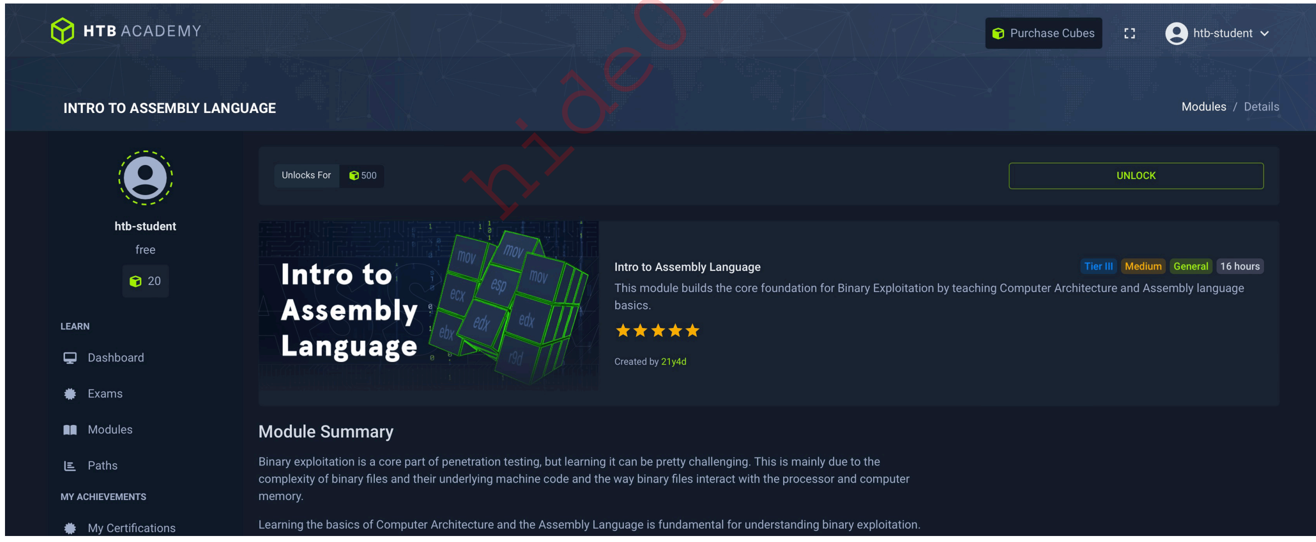
https://t.me/CyberFreeCourses



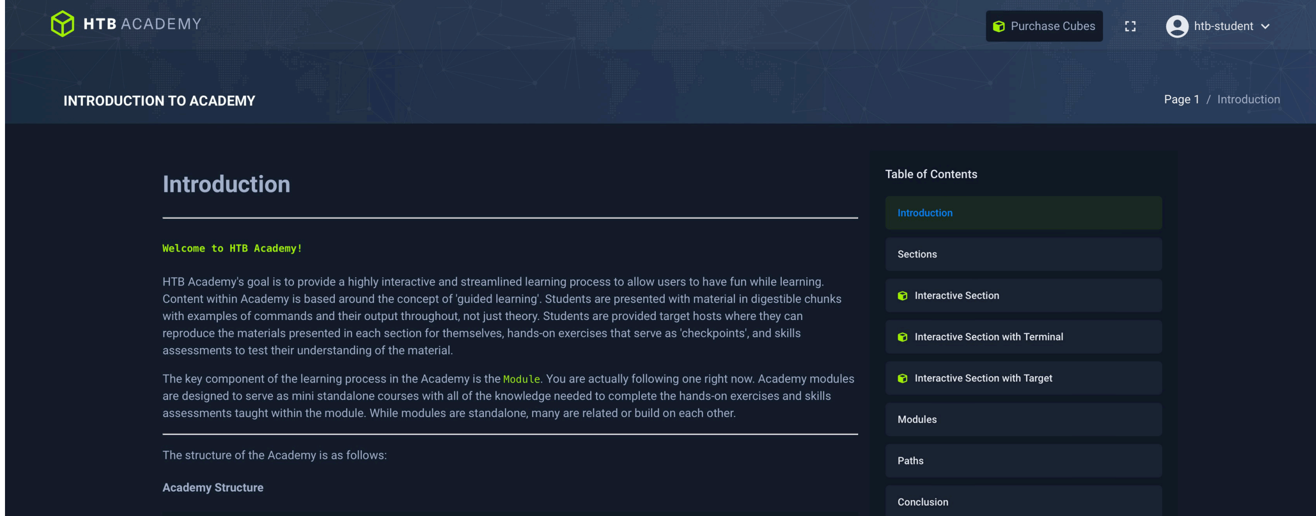
The above is `an Academy replica that we re-built from scratch`, both its `front-end and back-end` (using the MERN stack, for those interested). We put a lot of effort in creating a realistic demo with a sizable codebase, so that we can demonstrate all of the different types of Logic Bugs defined in the previous section with Academy as the vulnerable target.

We all know how the "real" Academy works, so this should give us a very clear contrast of how a logic bug would look, since we can compare the `solid-logic-design (real academy)` vs the `weak-logic-design (replica academy)`.

hide01.ir



https://t.me/CyberFreeCourses



Note: The development frameworks used in this Academy replica are not the same as the real Academy (e.g. Real Academy uses PHP while this uses JS). This, of course, will not have any effects whatsoever on the realism of the examples, as we are mainly interested in the application logic, which is irrelevant to its development framework, since these issues can occur regardless of what framework we use.

As a secure coding module, we will always provide the source code with each exercise, so that you can follow along the methodology we will describe next as you go through the various exercises. Even though we will use the same target web application for all examples of Logic Bugs, each would be configured slightly differently to showcase a different Logic Bug. So, `be sure to download the source code of each new logic bug type`.

hide01.ir

IMPORTANT: Throughout the module, we will suggest various exercises and challenges, and we strongly recommend that you do each and every one of them. Even if you are not able to complete all of them, you will greatly improve your code-reviewing abilities by simply attempting to solve them, before following up with the section instructions, which usually cover the same so you can compare your work and see what you missed and where you can improve your work.

Setting Up

Throughout the module, we will rely on VSCode and some of its plugins, which should also be helpful in your future code review and whitebox pentesting exercises. The main plugins that we will use are the following:

- RapidAPI
- Docker for VScode

The web application we will be testing throughout the module uses the MERN stack (Mongo, Express, React, Node). We wanted to avoid having you to go through the hustle of setting all

https://t.me/CyberFreeCourses

of these tools and technologies on your own machine, and then reconfiguring them for each exercise.

So, we created `a separate Docker image for each exercise` that includes both the front-end and back-end codes for the web application, as well as setting up the Mongo database and everything else. This also allows easy debugging of the web application, and easy restart/reset in case anything goes wrong in our testing.

So, in addition to the above tools, please make sure that you have [Docker] setup and installed on your machine.

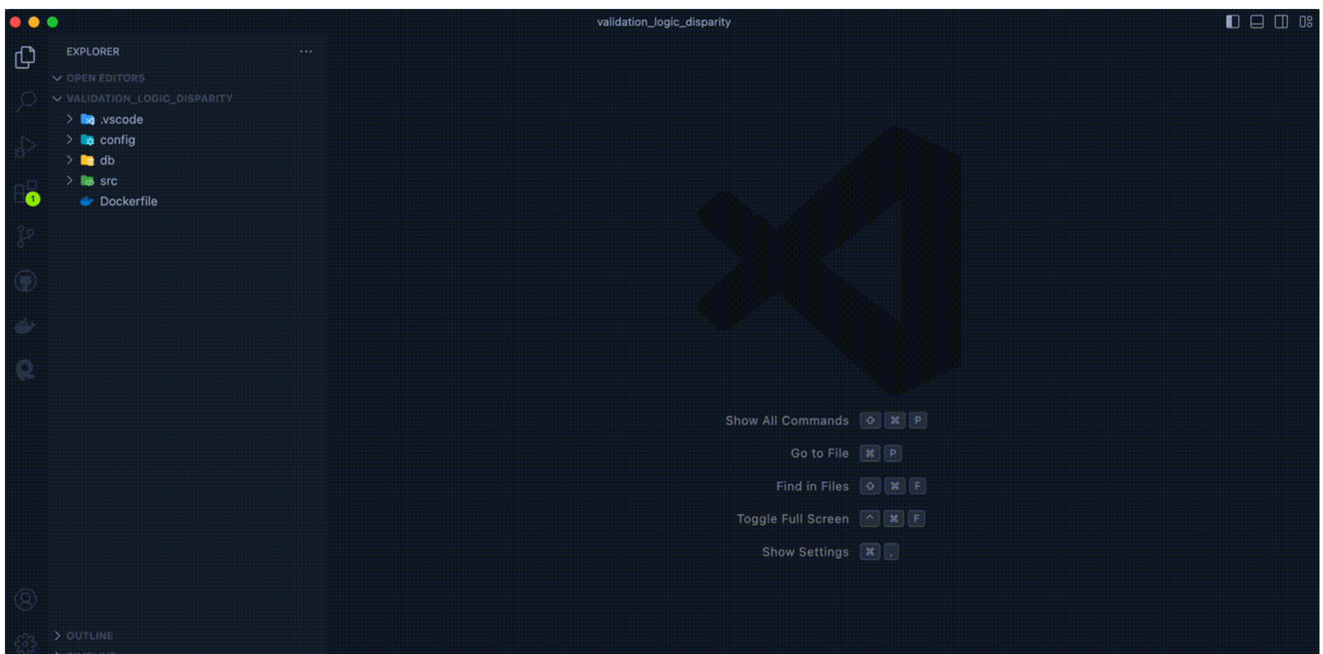
Note: Dockerizing web applications is an important skill to learn, so you can use this opportunity to learn how it's done and use it in your future whitebox pentesting exercises. Another option for larger web applications is to set up a test VM replica of the production server, though this usually takes a lot of effort and isn't easily deployable elsewhere.

Running Locally

Once you have the above tools set up and running, you can download the provided zip archive at the end of this section, extract its content, and then open the folder using VScode, using `File` > `Open Folder`, or through the terminal with `code ./validation_logic_disparity`.

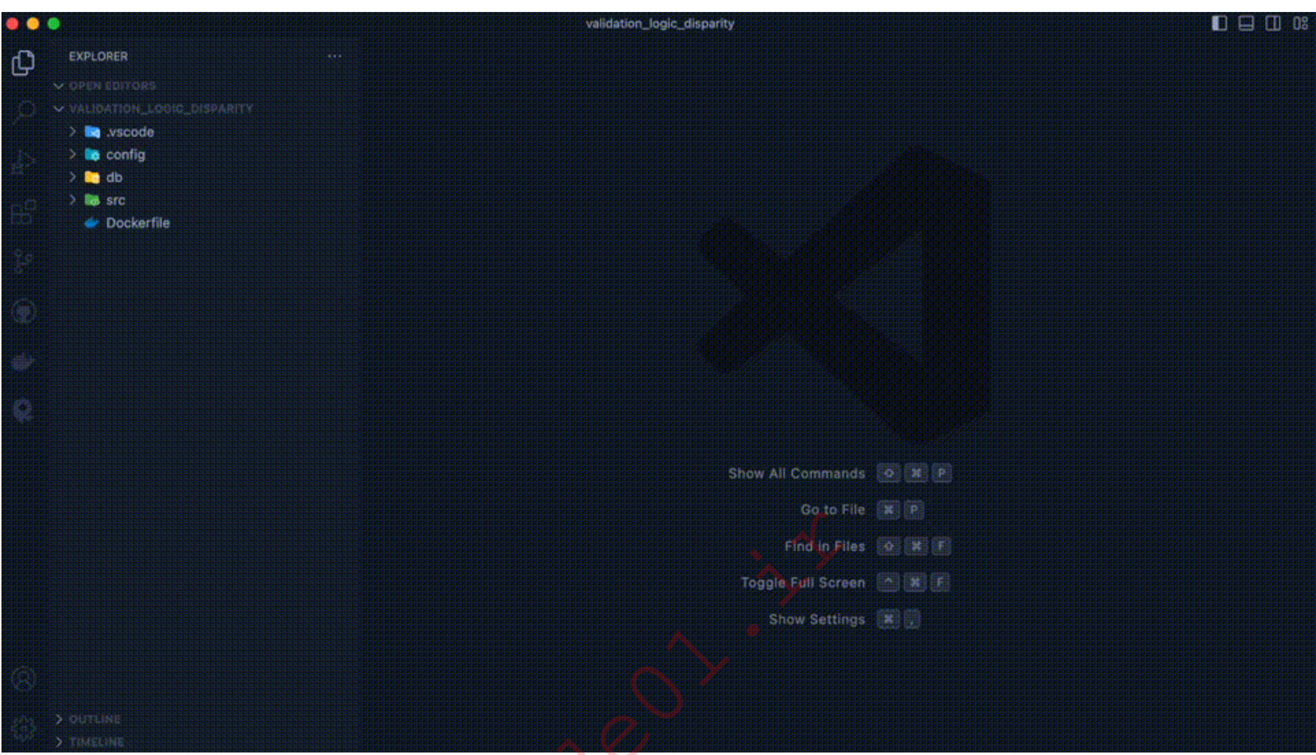
Once that's done, you can open the `Dockerfile` within VSCode, right click on the opened window, and select `Build Image`. This will prompt you to enter a tag for the image, so you can either keep the default value of the folder's name, or use a common host for all exercises under this module (e.g. `application_logic_flaws:validation_logic_disparity`). Once you do so, the docker image will start building, which should take 5-15 minutes depending on your machine and internet speeds.

hide01.ir



https://t.me/CyberFreeCourses

Once the build is done, you can find the new image in the `Docker` icon in the VSCode Sidebar on the right, under `IMAGES`. You can right-click on and select `Run`, and the image should start up. Give it a few seconds to load up everything, and then visit `http://localhost:5000/` with your browser, and login with the credentials provided in the Dockerfile, as that user may have some privileges like cubes or payment cards added under their user. Some tests and exercises may require you to create your own user, so that's another option.



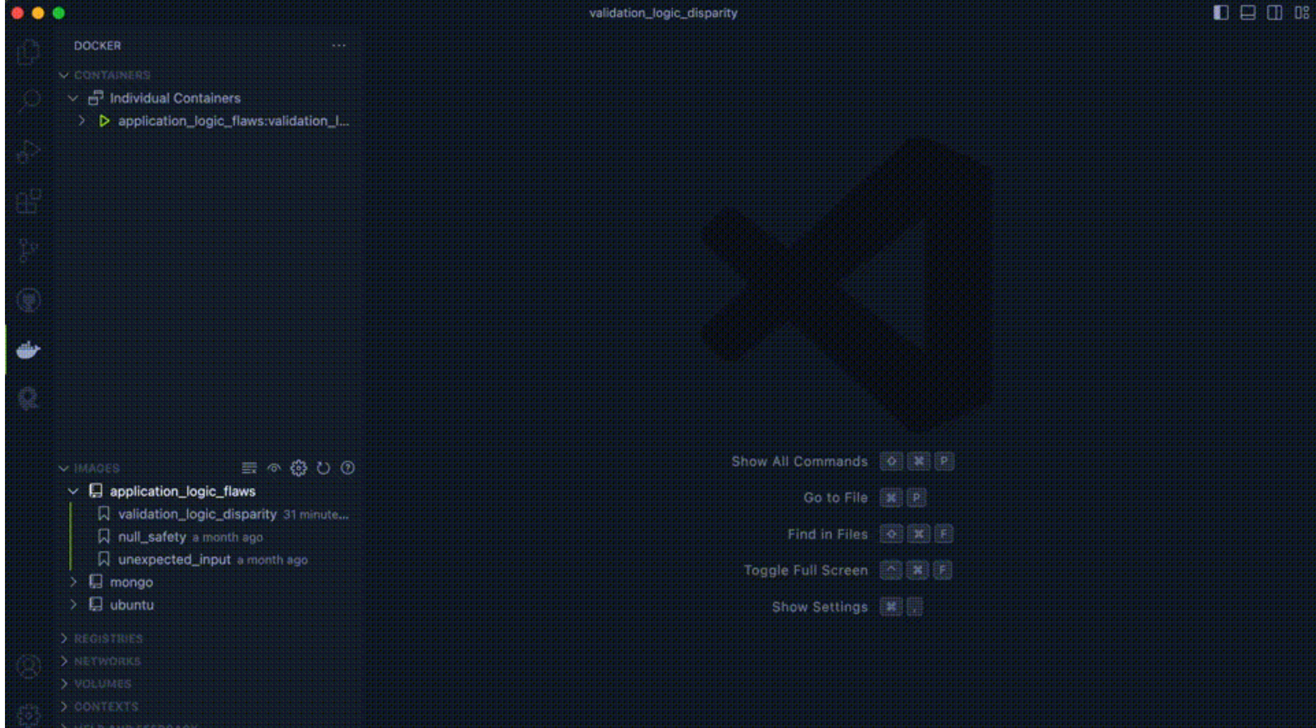
hide01.ir

Tip: You may also select `Run Interactively` to keep an eye on the logs, like the back-end server error messages, which will show up in a window within VSCode.

Local Testing

As part of the whitebox pentesting process, we usually need to perform local testing and debugging, as mentioned in the previous section. Luckily, the Docker images we just setup also enable easy debugging of our code. All we need to do is go to the `Run and Debug` tab in the VSCode sidebar, and click on the `Run` icon next to the text `Docker: Attach to Node`. As the Docker image is already configured for debugging, this will attach to our web application for debugging, and the bottom bar of VSCode should now be red to indicate we are in debug mode. You may also `pause`, `restart`, and `detach` from the debug session using the hovering session buttons.

https://t.me/CyberFreeCourses



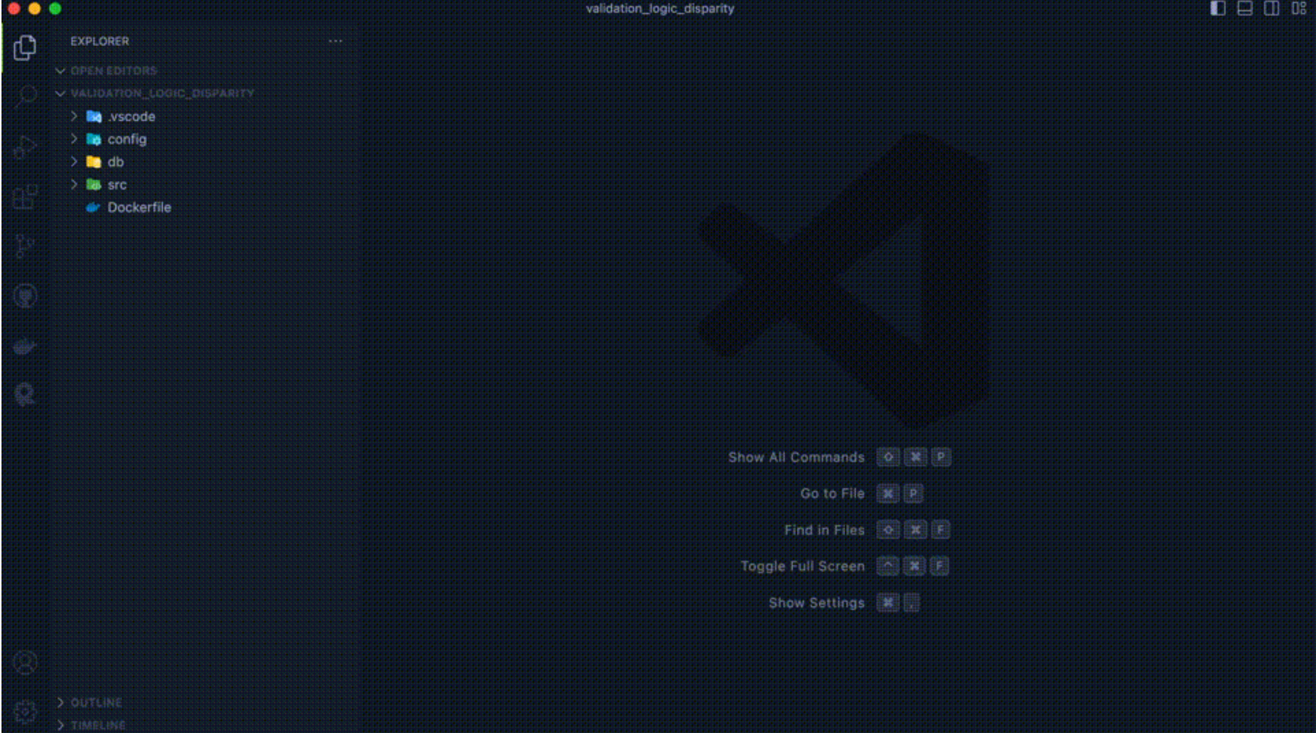
In the coming sections, we will go through more details on how to utilize debugging in our local testing. But as an example, we can add breakpoints by opening the file we want to add a breakpoint in, and go to any line and click on the line number, and this should add a breakpoint. Now, whenever the code reaches this line, the application should break and stop on that point, so we can examine everything and then resume its execution.

Note: It is important to note that even though we can set breakpoints using the code files opened in VSCode, any code changes will not be reflected in the running container. Though this isn't something we will need to do in this module, if you ever need to modify code in a running container, go to the `Docker` tab, find the running image under `CONTAINERS`, then you can click on it to expand it and show its files. You can find the web application files under `/app`, and right-click and select `Open` to open and modify any of them within VSCode.

Another thing we may need to modify and debug throughout our local testing is the `database`. To do so, we can go to the `Docker` tab in VSCode, find our running image under `CONTAINERS`, right-click on it and select `Attach Shell`, and this should open a new window terminal in VSCode and drop us into a shell within the running container. Now, we can simply run the `mongosh` command, and should have a MongoDB shell with access to our database.

hide01.ir

https://t.me/CyberFreeCourses



Note: This module assumes that you are already familiar with MongoDB and how to run basic commands, as well as understand the basic usage of MongoDB with NodeJS. If you are not, you may give it a quick read online to familiarize yourself with it, or check out the Introduction to NoSQL Injection module, which introduces MongoDB and gives a brief about how to use it.

hide01.ir

Application Structure

Before we move on to our first topic, it is worthwhile to take a quick look on the general structure of the application we will be testing. We can start by opening `src/src/app.js`, and going through it.

We will see that the application starts by importing some files and libraries. After that, the application sets up Node/Express along with other configurations, as well as setting up `bodyparser` and `CORS`.

```
const app = express();
const port = parseInt(process.env.PORT || "5000");
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// set up body parser and cors
app.use(bodyParser.json());
app.use(cors());

// serve public assets
app.use(express.static(path.resolve(__dirname, "public")));
```

https://t.me/CyberFreeCourses

Then, the application sets up the main API middleware with `app.use()` and specifies the routes file for each route. Then, it handles 404 and general app errors.

```
// set up API routes
app.use("/api/auth", authRoutes);
<SNIP>

// forward all other requests to react app, so react router can handle
them 'will handle 404 errors'
app.use((req, res, next) => {
  res.sendFile(path.resolve(__dirname, "public", "index.html"));
});

// handle 404 errors
app.use((req, res, next) => {
  res.status(404).json({
    message: "Could not find this route.",
  });
});

// handle next() errors and general errors
app.use((error, req, res, next) => {
   <SNIP>
});
```

Finally, it sets up the MongoDB connection and starts listening on the port specified at the beginning. We can read the comments to get a better idea of exactly what each code block does.

```
// start the Express server & db connection
set("strictQuery", false);
connect(process.env.DB_URL ?? "", {
  dbName: process.env.DB_NAME,
  user: process.env.DB_USER,
  pass: process.env.DB_PASS,
})
  .then(() => {
   <SNIP>
});
```

So, the main thing that we need to focus on is the `APIs and their routes middleware`. We can CMD/CTRL click on any of them, like `authRoutes`, and VSCode should open it in its corresponding file.

hide01.ir

https://t.me/CyberFreeCourses

```
const router = express.Router();

// secure private routes for content (use req.user in private controllers)
router.use(verifyToken);
router.get("/update_token", updateUserToken);
```

If we examine the routes middleware, we will see that it is linking each sub-route to a specific function, like `/update_token` to `updateUserToken`, which we can also CMD/CTRL click on to further review, as we will do in upcoming sections.

Take some time to navigate through the code and familiarize yourself of how it's running. In general, the files we will be most interested in are under `/controllers`, as well as `/routes` and `/models`.

With that, you should be able to run this application and debug it, as well as have a general idea of its general structure. In the next section, we will start with our first topic and will go through the code to identify potential issues and logic bugs.

Validation Logic Disparity

hide01.ir

`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`

https://t.me/CyberFreeCourses

`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

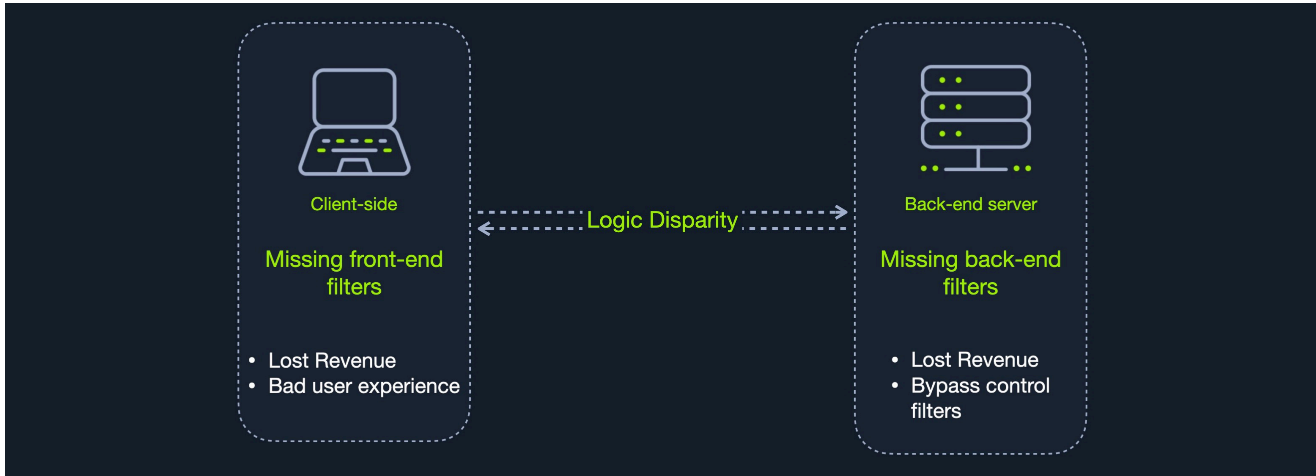
As we can see, this issue arose because the `client-side logic was not in parity with the server-side logic`, and each was coded with different assumptions, leading to this loophole.

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.

hide01.ir

https://t.me/CyberFreeCourses



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.

https://t.me/CyberFreeCourses

Code Review - Validation Logic Disparity

As mentioned in the previous section, the easiest way to identify `logic disparity` bugs is by comparing validation tests on both ends of the application, and trying to find a disparity on the back-end code. Otherwise, we can manually test functions in the front-end and see which rely on dynamic validation tests. Then, we can start reviewing the back-end code to identify disparities.

Doing this for huge code bases and applications may sound cumbersome, especially ones with many forms and fields. However, once you have the proper understanding of how validation logic disparity bugs look like, then you are likely to identify them when reviewing any function on the back-end.

We are also starting with a mixture of code analysis and dynamic application testing for this specific use case, but as we go through the module, we will utilize more advanced techniques to review larger parts of the code base and filter them for specific criteria.

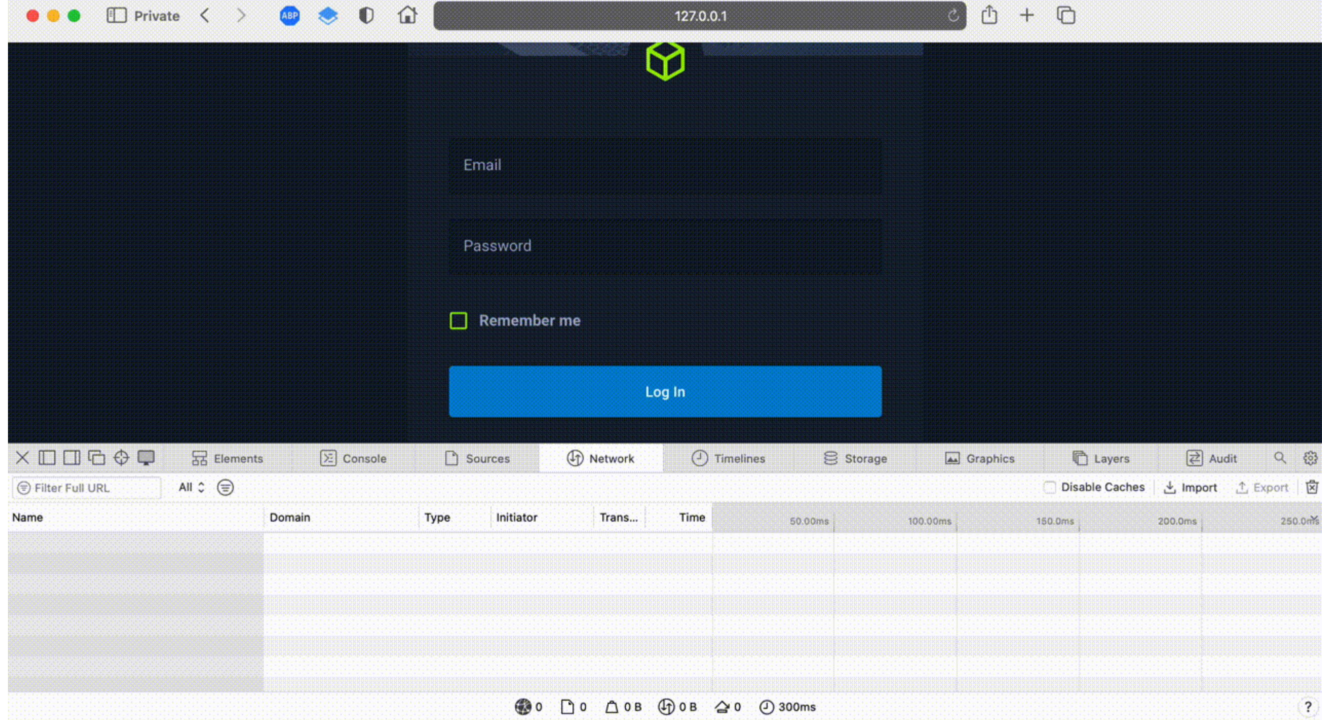
Note: Did you complete the exercise at the end of the previous section? If so, see if one of the functions you noted down/shortlisted is what we will discuss in the section. If you got it, then you are off to a great start. Amazing job!

hide01.ir

Dynamic Validation Tests

If we start going through the front-end web application, we can easily identify a few forms and fields that are clearly applying client-side data validation. This is easily verifiable by the instant feedback, and the fact that no requests get sent as the validation is applied, indicating that it is in fact happening in the client-side. For example, the very first form we face is the `login` form, and we can bring up the `Network` tab from the browser developer tools, and see that, indeed, no requests are being sent to process this validation:

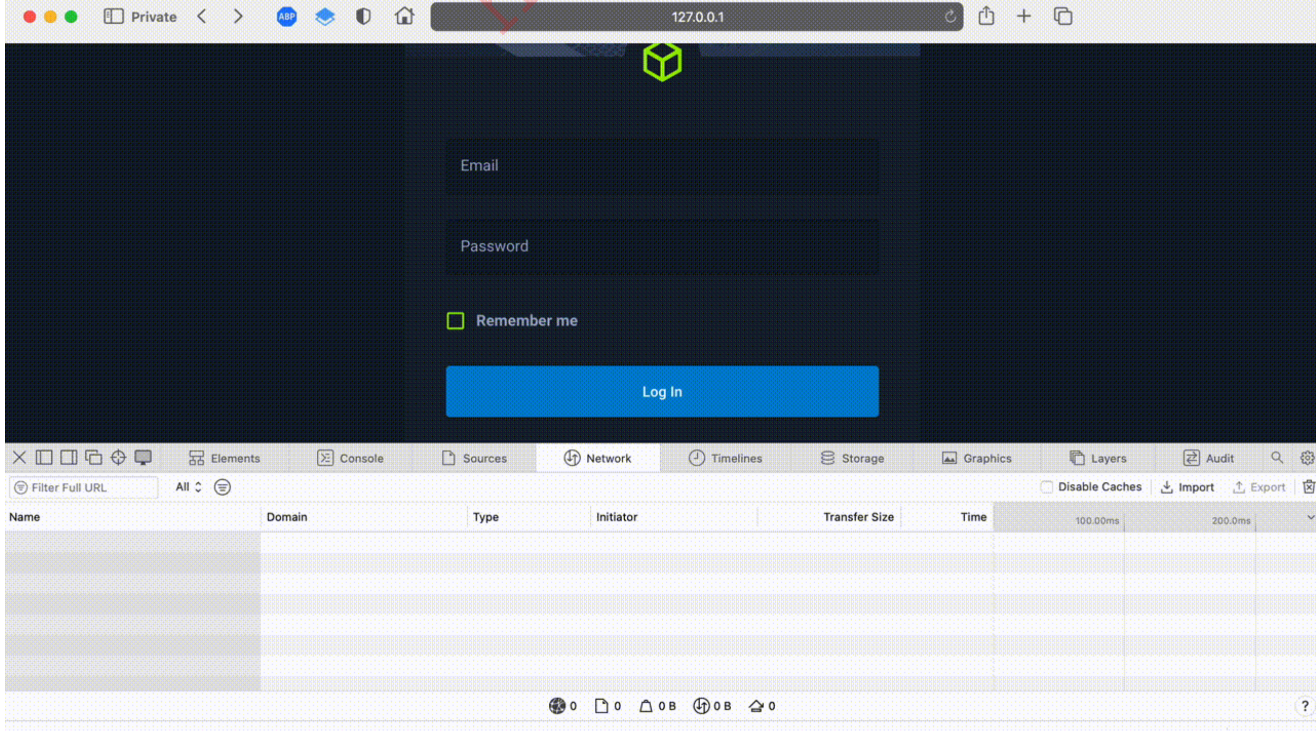
https://t.me/CyberFreeCourses



As mentioned in the previous section, we are specifically looking for `dynamic front-end validation filters` that are `relying on data pulled from the back-end`. The filters applied by the `login` form do not appear to rely on any data from the back-end, so they can be considered as `static filters` that always try to match a specific pattern (e.g. email format). Furthermore, none of the requests sent by our browser when visiting the page seem to have data that may be utilized by it.

hide01.ir

We can keep going with the other forms we can find in the web application, while keeping an eye on the network tab for sent requests:



Note: Keep in mind that not all forms pull data from the back-end in a similar manner. Some may pull the data when the form is clicked, while others may pull the data when the page is

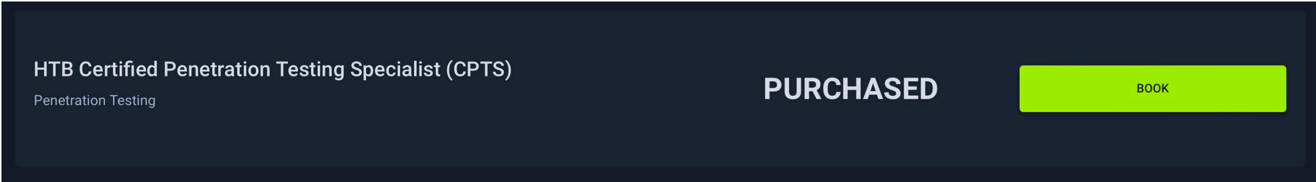
https://t.me/CyberFreeCourses

visited, and so on.

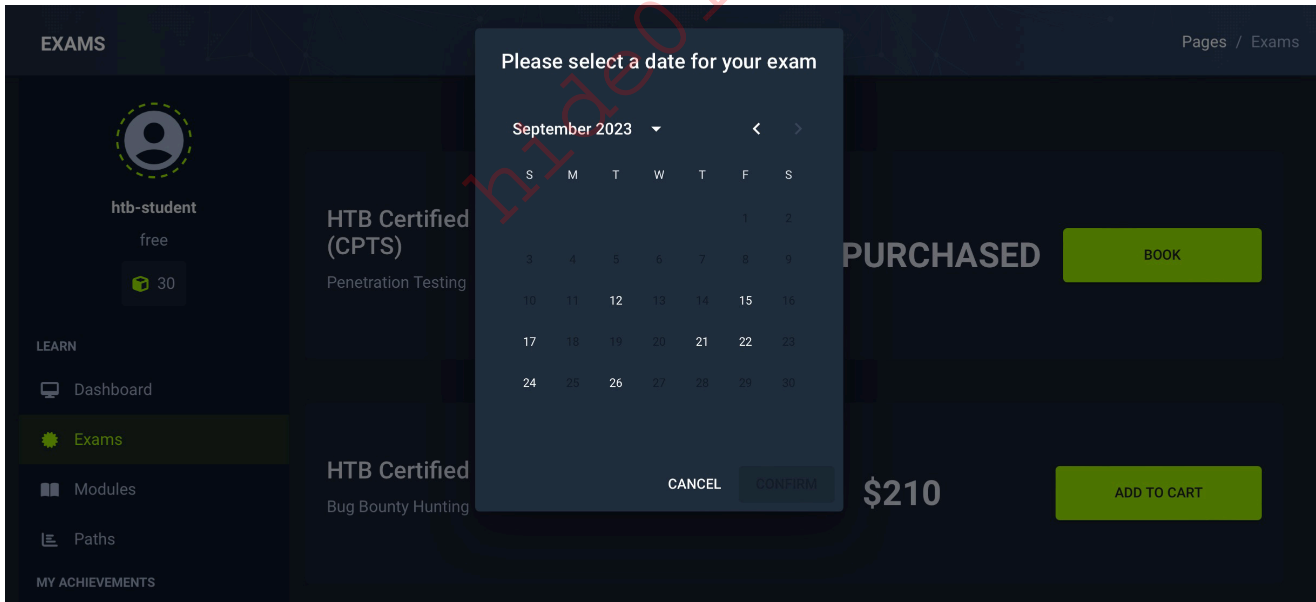
Exam Booking

If we keep going with other forms and fields within the web application, we will notice that the `exam booking` function does appear to meet all 3 of the criteria we specified in the previous section. So, let's take a look at it in the front-end, to further understand how it functions and whether it is vulnerable to this flaw.

If we go to the `/exams` page, we see the available exams listed for purchase. If we had already purchased one of them "as is the case for the user with the provided credentials", then the next step would be to `Book` our exam slot. This is a common practice for many certification bodies, and is also used for many online appointments and reservations.



If we click on `Book`, we are presented with a calendar view for available slots that we may book. Once we select an available slot and click `Confirm`, the exam gets booked and we get a confirmation message with our exam date.



We can see that:

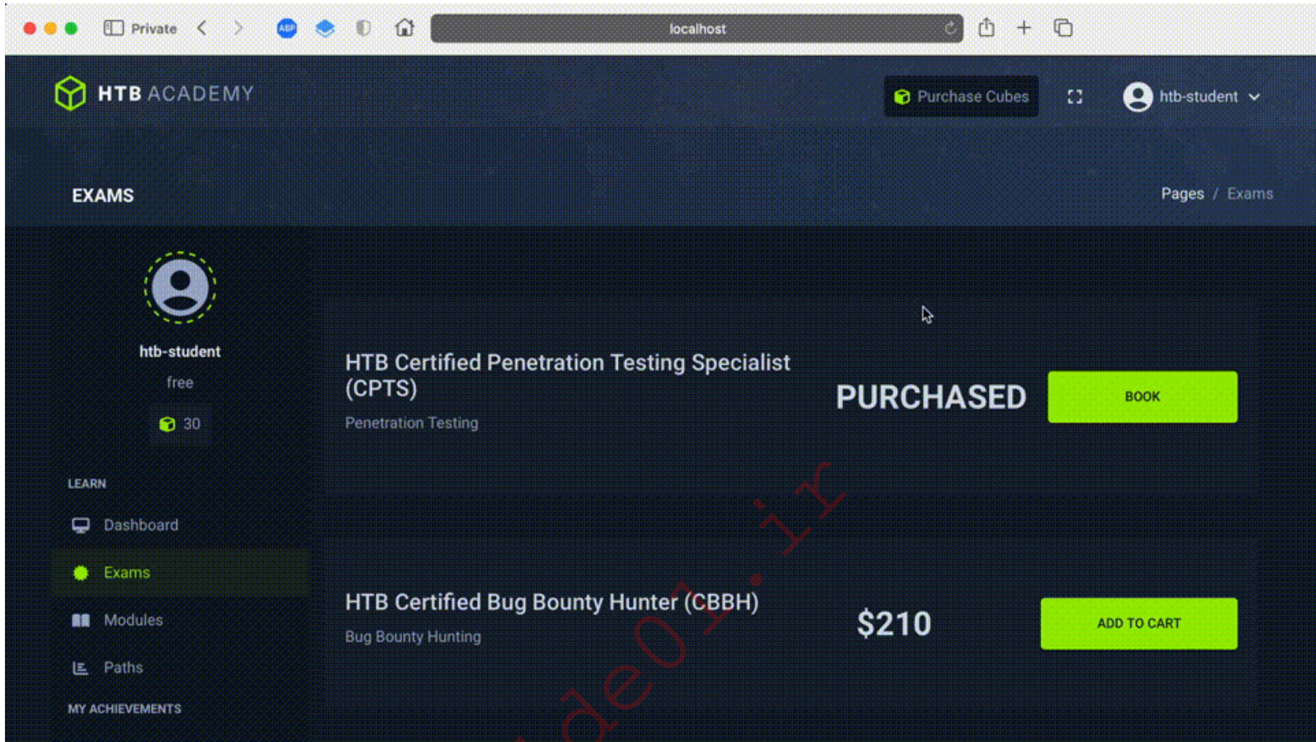
1. It does accept user input: in the form of our selected date in the calendar
2. It does apply client-side validations: by disabling certain unavailable dates
3. It does rely on back-end data to adjust the validation: which we can verify by monitoring the requests and how the filter changes

Personally, whenever I see a similar case I immediately ask myself:

https://t.me/CyberFreeCourses

1. `How is it applying the availability filters?`
2. `Does the back-end revalidate the availability upon booking?`

To further understand how to confirm this, we can once again bring up our `browser developer tool` and go to the `Network` tab. Once we do, we can click the `trash` icon on the top-right corner to `clear network items`. Now, if we click the `book` button again, we see that the application sends a request to get the details about which slots are `unavailable`:



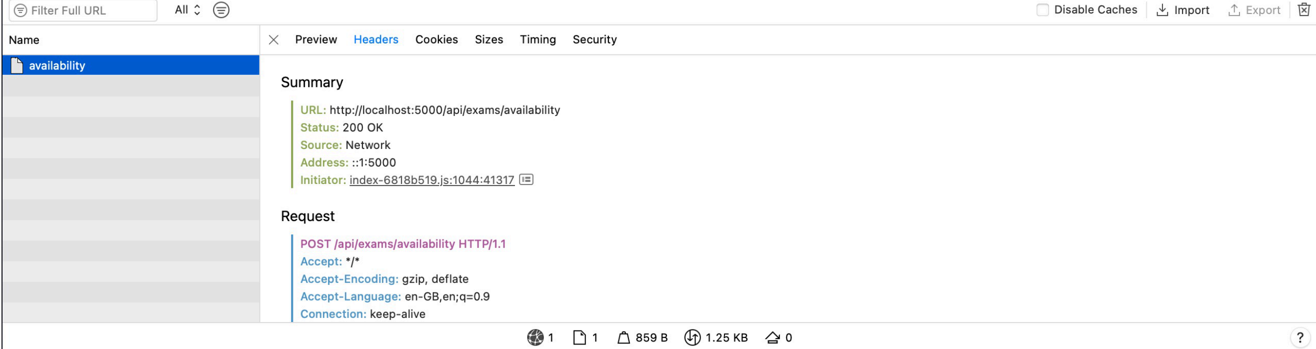
hide01.ir

Let's investigate this further. If we select `Request` from the top-right drop-down menu "instead of `Response`, as shown above", we see that it is sending a request with the following data:

```
{
  "id": 1,
  "startDate": "2023-09-01T15:47:14.843Z",
  "endDate": "2023-10-01T15:47:14.843Z"
}
```

To find the exact API endpoint URL, we can click on the `headers` tab, and see under `Request` that it is sending a `POST` request to the `/api/exams/availability` endpoint:

https://t.me/CyberFreeCourses



Finally, let's select any available slot and click `confirm`, to see which request gets sent. As we can see, this time another `POST` request was sent to `/api/exams/book`, with the exam `id` and the selected `date`, as follows:

```
{
  "id": 1,
  "date": "2023-09-14T23:00:00.000Z"
}
```

From our brief interaction with the endpoint, we can see that the first request to `/availability` was used to modify how the filter is applied, while the second request to `/book` was the regular request used for exam booking. To test for `validation logic disparity`, we need to locate the functions responsible for these endpoints and review their code, and compare their filters to the once we observed on the front-end.

Note: You may notice slight differences between the date you selected and the date in the request. This is completely normal, as the request uses an absolute timezone to ensure it is the same on both ends.

https://t.me/CyberFreeCourses

Locating Endpoints

To locate these endpoint functions, we can open `app.js`, and we'll see that it is under `examRoutes`:

```
// set up API routes
<SNIP>
app.use("/api/exams", examRoutes);
```

We can CMD/CTRL+click it, and it'll open in a new tab in VSCode, as we have seen previously:

```
router.get("/", getAllExams);
router.get("/:id", getExamById);
router.post("/availability", getExamAvailability);

// secure private routes for content (use req.user in private controllers)
router.use(verifyToken);
router.get("/user/exams", getUserExams);
router.post("/book", bookExam);
router.get("/content/:id", getExamContent);
```

hide01.ir

We see that the `/availability` endpoint falls before `verifyToken`, while the `/book` endpoint falls after it. It is important to understand how this affects the endpoints and the application in general. If we review its code, we will see that it is responsible for decoding and verifying the user details from the authentication token. So, any endpoints that fall after it would require an authenticated token.

Exercise: Try to read the `verifyToken` function to see how it is decoding the authentication token and adding the user details to the request, and note what user details are being added to the request. Do you think it would be possible for us to manipulate our user id?

So, `/availability` endpoint is publicly accessible, while the `/book` endpoint does require authentication, and will require a valid token. This is expected for the `/book` endpoint, as the above request didn't provide any details about our user to book the exam for us, so it must be getting these details from our token.

In the next section, we'll go through these functions, and will make tests to validate our understanding of how they work.

Local Testing - Validation Logic Disparity

---

https://t.me/CyberFreeCourses

Now that we have identified a potentially interesting function, in this section, we will study the back-end API's it interacts with, and try to see if there are any disparities between the front-end validation logic and the back-end ones.

Note: Before we continue, you may restart your Docker container to reset it to its original state.

getExamAvailability()

Let's start with the `getExamAvailability()` function. First, we see that it sets { `id`, `startDate`, `endDate` } from the request body and performs a few checks to validate the date format. It then locates the exam in the database with `Exam.findOne` using the sent `id`, as a form of `id` and `exam` verification:

```
const { id, startDate, endDate } = req.body;

// validate date format
if (
  !startDate ||
  !endDate ||
  isNaN(Date.parse(startDate)) ||
  isNaN(Date.parse(endDate))
) {
  return next({
    message: "Please provide a valid date range.",
    statusCode: 400,
  });
}

let exam;
try {
  // ensure exam exists
  exam = await Exam.findOne({
    id,
  });
  <SNIP>
}
```

After that, the function uses the following code to find existing exam slots within the provided date range:

```
const bookedExams = UserExam.find({
  examId: exam.id,
  date: {
    $gte: new Date(startDate).setUTCHours(0, 0, 0, 0),
    $lte: new Date(endDate).setUTCHours(23, 59, 59, 999),
```

hide01.ir
https://t.me/CyberFreeCourses

```
  },
  used: false,
});
```

Note: Whenever the code interacts with the mongodb database, it is using pre-defined modules, like `Exam` and `UserExam` in this case. Try to read their code to understand how this is working. This should not affect our code review, but it does make it easier for the developers to interact with the database, and for us to understand the code.

Finally, the code simply returns a JSON list of any booked dates, and if it does not find any results, or if it encounters any issues, it returns an empty array:

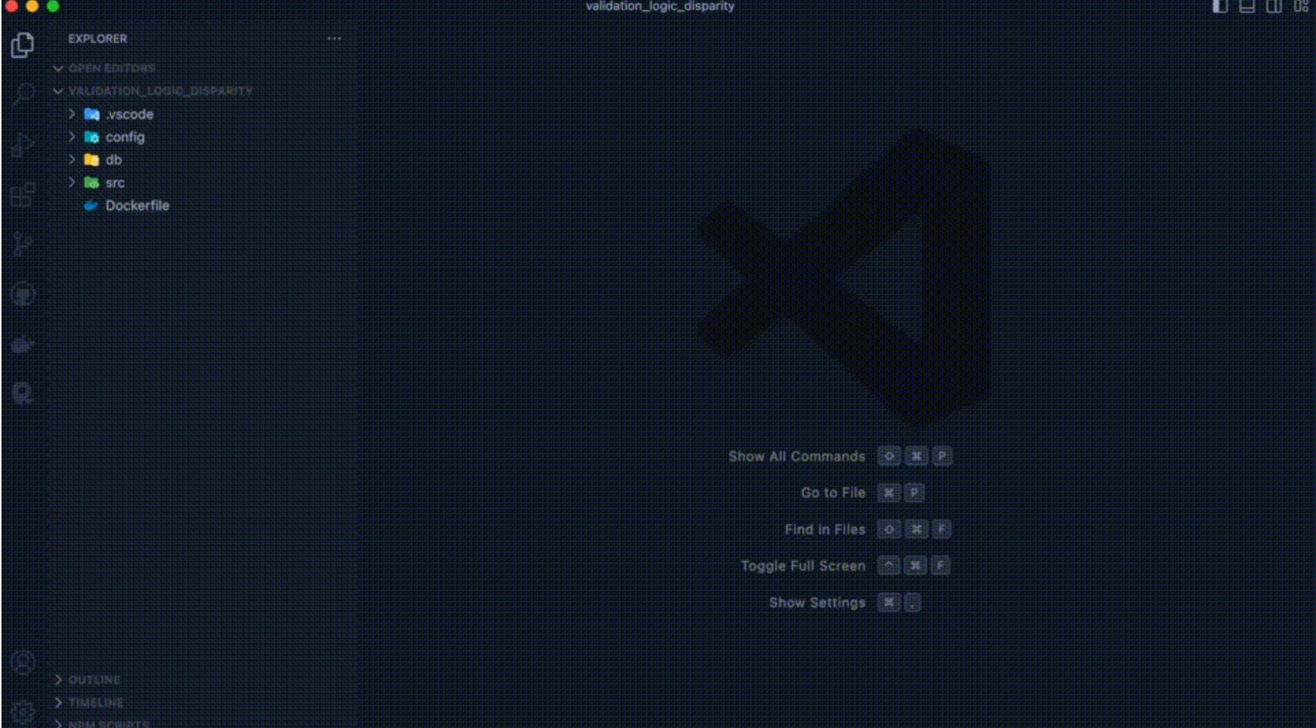
```
res.json({
  unavailableSlots: (await bookedExams).map((exam) => exam.date),
});
```

We can confirm this by sending a basic request through `RapidAPI`. If we go to the `RapidAPI` extension tab, hit `+` and then set the method to `POST` and URL to `http://localhost:5000/api/exams/availability`. Lastly, we can set the data type to `JSON`, and use the following request body "from the front-end request":

```
{
  "id": 1,
  "startDate": "2023-09-01T15:47:14.843Z",
  "endDate": "2023-10-01T15:47:14.843Z"
}
```

Now, if we send the request, we get a list of unavailable slots in this date range:

https://t.me/CyberFreeCourses



All of this seems quite normal, as the function simply returns a list of unavailable dates. This approach is quite common, where an endpoint would send a list of unavailable dates instead of the available ones. Either way, this does not affect or cause any logic bugs "yet".

bookExam()

Checking the `bookExam()` function, we see that it also sets { `id`, `date` } from the request body, and then sets the user ID from `req.user?.id`. We also see that it validates the date format just like the `getExamAvailability()` function, and then it retrieves the exam details to confirm its existence. Finally, it updates the user's purchased exam ticket with the booked exam date:

```
const updateReq = await UserExam.findOneAndUpdate(
  {
    examId: exam.id,
    userId,
    used: false,
    date: {
      // date must be null 'unbooked' -> can't change date once booked
      $eq: null,
    },
  },
  {
    date: new Date(date),
  }
);
```

hide01.ir

https://t.me/CyberFreeCourses

We see that it retrieves the exam by its `id`, which has already been verified to exist, and uses the `userId` to ensure that the user has already purchased an exam ticket and not used it (`used: false`). It also ensures that the user has not already booked the exam by searching for a ticket with an empty date (`date: { $eq: null }`). Finally, it updates this ticket with the requested exam date (`date: new Date(date)`).

Let's confirm that our understanding of the function is correct. We want to modify the value of `userId`, but since it is retrieved from the `JWT` token, which is signed with a secret key, we cannot simply modify it "the key in real target differs from the local one". So, instead, we will set a breakpoint right after the line where `userId` is set, and then send a request to the endpoint. Once the breakpoint is hit, we can slightly modify the `userId` to something else, and it should tell us that we have not purchased this exam.

So, first, we'll add the breakpoint by clicking on the line number in VSCode (or using the shortcut SHIFT+F9). We may also right-click on the `userId` variable and select `Add to Watch`, to keep an eye on its value:

Lastly, we can send a `POST` request to `/api/exams/book` with the same `id` / `date` body data we saw in the previous section. As this endpoint requires authentication, we will also need to add our token to the request, which we can copy from the `storage` tab in the Browser `Dev Tools` under `Local Storage`, then we can click on the token and select `Copy Row`. Then, in the `RapidAPI` request, we add it in the `Auth` tab with the `Bearer` option "make sure you delete the `token` word when you paste the value". Once the request is set up properly, we can click `Send`, and we should automatically hit the breakpoint we set:

https://t.me/CyberFreeCourses

We can examine the request values on the left panes, including the `userId` value that we `added to watch`:

Now, we can right-click on `userId` under the `WATCH` group, select `Set Value`, and change it to anything else, so it would be any other ID that doesn't have a booked exam. Once set, we can click `F5` to continue the execution of the request, and will receive the error " `User has not purchased this exam` " in the response to the request we just sent:

https://t.me/CyberFreeCourses

With this process, we were able to confirm that the function correctly prevents unauthenticated users, properly validates the date format, correctly validates the exam id, ensures that users can only book once they have purchased the exam, and prevents them from booking again once a date has been set. `It seems secure, right? Not quite.`

While the function does validate the date format, `it performs no validations on the availability of the exam date`, like checking whether it is already booked or whether it is in the future. Not only can a user `book an exam on a date that is not available` "which leads to over booking", they can also `book a date in the past` "which may potentially lead to other logic issues".

The front-end app does all of that, as it disables any dates in the past, and it disables any dates found in the `unavailableDates[]` array from `/exam/availability`. However, this `disparity` between the two ends leads to the flaw we are discussing in this section.

While this is quite a fundamental flaw (i.e. relying on front-end validation), in reality, this is a prevalent logic flaw found even with some of the largest online retailers, as we have mentioned in the intro section.

This flaw can be due to many reasons, like having a gap in communications between front-end and back-end developers, the complexity of the code and thinking this may be validated somewhere else, and many other reasons. That is why we must keep this in mind when reviewing code bases or performing mobile/web application penetration tests, and we must always ensure each function has a solid validation logic on both ends and is in complete parity.

PoC and Patching - Validation Logic Disparity

hide01.ir

https://t.me/CyberFreeCourses

Now, we can go ahead and test our theory, and validate that we are able to book an exam slot even if there aren't any available ones.

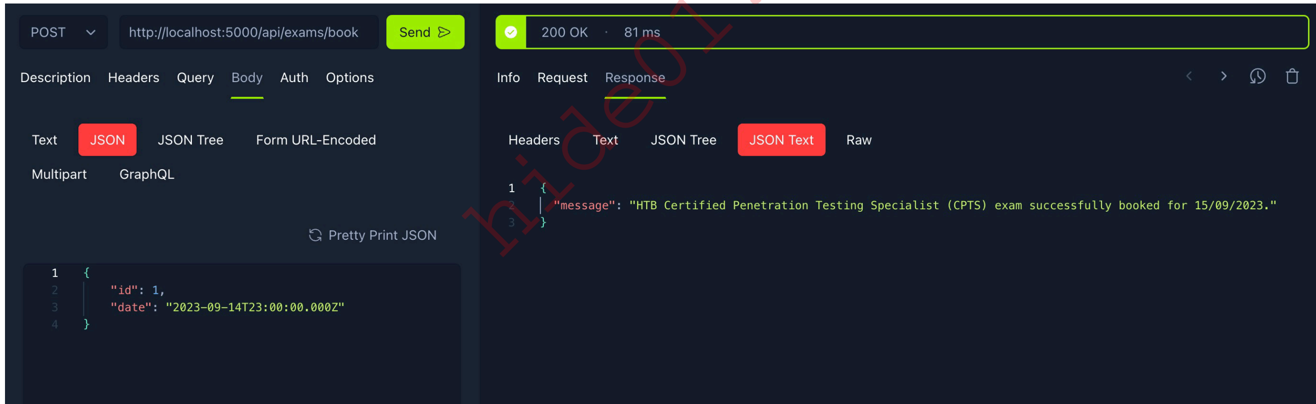
Proof of Concept

First, we will restart our Docker container again to restore the original state of exam booking and remove any breakpoints we may have added to the code. For this demo, we have filled the remaining exam slots in the database to reflect what we will have on the real target (no available slots).

After that, all we need to do is send an exam booking request, like the one we saw when reviewing the front-end application, and only change the `date` value to any unavailable date, as follows:

```
{
  "id": 1,
  "date": "2023-09-14T23:00:00.000Z"
}
```

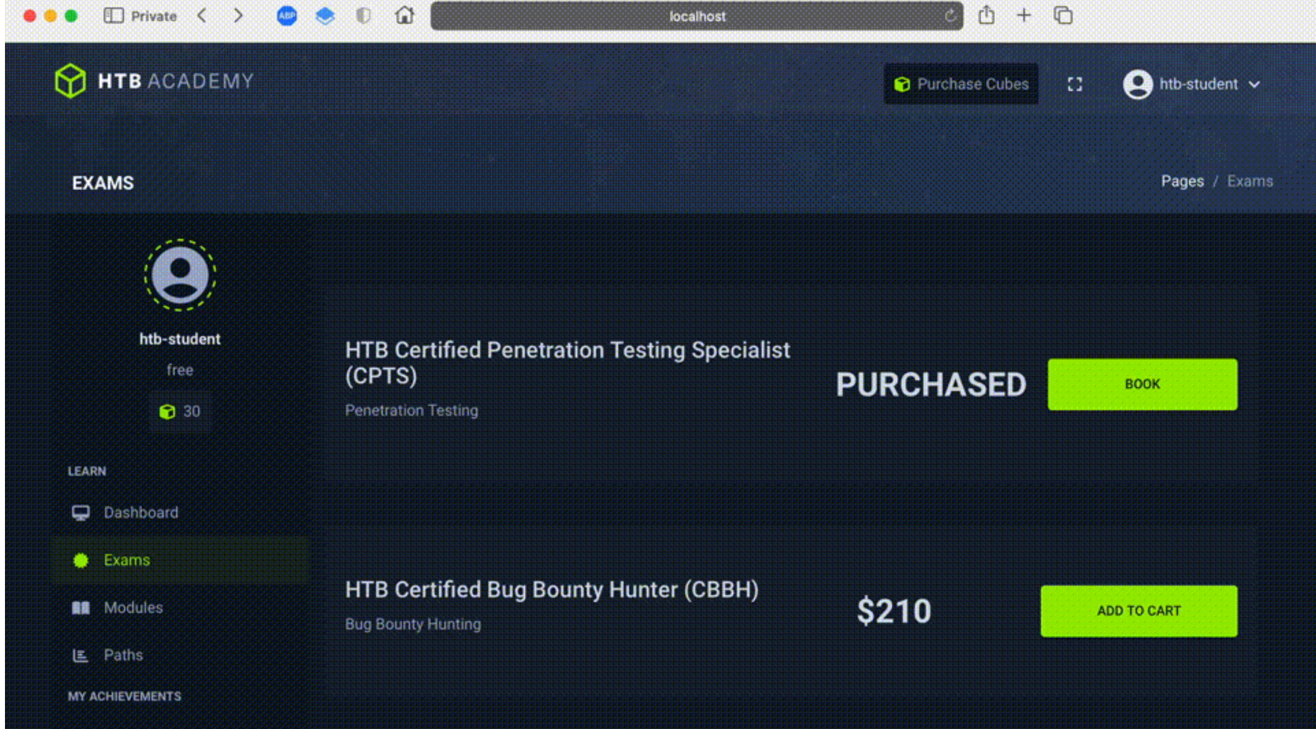
Once we do, we do indeed get a booking confirmation:



Finally, we can refresh the `/exams` page, and we will see that our exam now says `BOOKED`, and we can view the exam:

hide01.ir

https://t.me/CyberFreeCourses



As we can see, this `logic disparity` flaw allowed us to book an exam, even though all exam slots were fully booked. The same flaw applies to other scenarios, like allowing us to purchase an item that is `unreleased` or `out of stock`.

In this case, the missing validation test was on the back-end, and such flaws are always more serious as they allow us to modify data on the database. However, `Logic Disparity` issues may also be caused by missing validations on the front-end, as mentioned before. For example, and item may be `in stock` and available for purchases, and the back-end may be validating items correctly. However, an issue with front-end validation may show the item as `out of stock`, making customers unable to purchase it and leading to lost revenue, which is another common logic bug.

All of this should give us a very clear idea of how `Validation Logic Disparity` issues may arise and how to identify them and exploit them. Next, we will see tips on how to avoid such flaws.

Note: As this is a Hard module, the exercises will not match the same demo shown in the sections, and this specific vulnerability is patched on the real target. We will instead test your understanding of the content with a similar Logic Bug but in a different context. You may still use the same source code to test what is being shown in the section, as well as test and identify another logic bug, as discussed in the exercise.

Patching

We have proven that the vulnerability does exist and can be exploited, so let's see how we would remediate it by patching the code. The main thing that led to this vulnerability is a `missing exam availability test on the back-end`, as the application relied on the test done on the front-end.

https://t.me/CyberFreeCourses

So, we simply need to add this before proceeding with the exam booking. To do so, we can add the below code on `line 191` in the `/controllers/exam-controllers.js` file:

```
// ensure exam slot is available
const bookedExams = UserExam.find({
  examId: exam.id,
  date: new Date(date),
  used: false,
});

if ((await bookedExams).length > 0) {
  return next({
    message: "Exam slot is not available.",
    statusCode: 400,
  });
}
```

This test will simply ensure that the selected exam slot is empty, before proceeding with the booking. Of course, this patch is for this specific case. In general, the remediation of a `logic disparity` is by bringing the logic back to parity on both ends, which means adding any missing tests on either end, like the one above that was missing on the back-end.

Exercise: Try to patch your code and then re-apply the same above PoC, as a way of confirming that fix. If it is still vulnerable, then the patch does not properly remediate the issue.

hide01.ir

Unexpected Input

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

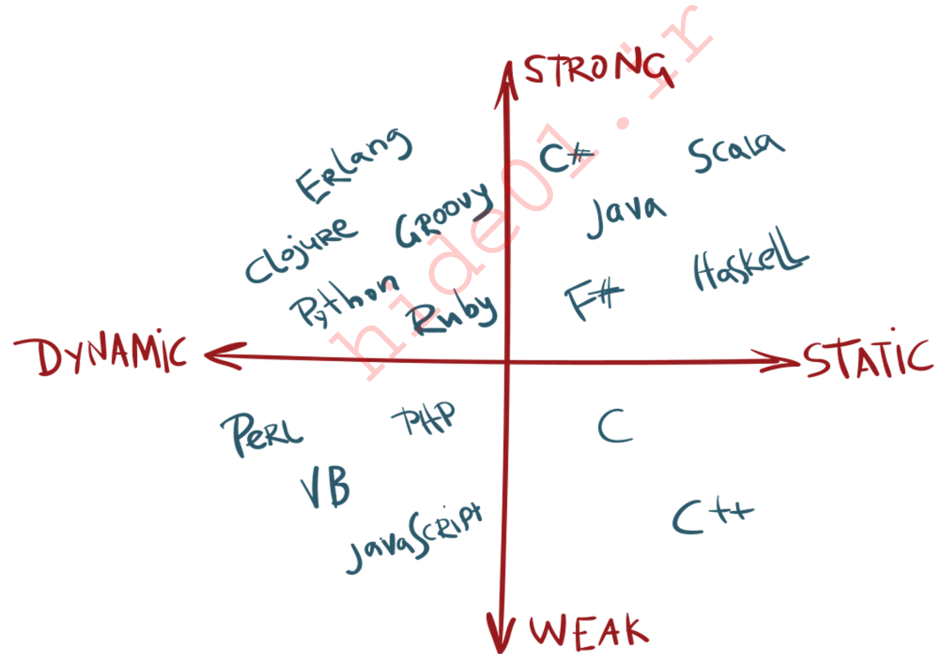
https://t.me/CyberFreeCourses

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] 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. `===`).



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

https://t.me/CyberFreeCourses

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".

https://t.me/CyberFreeCourses

Tab/Check

Jan 24, 2023

1	[illegible]	$12.99
1	[illegible]	$14.19

Show More ▼

Select a Tip Amount $-38.59

Cash Tip	22% $6.79	20% $6.17	18% $5.56	Custom $

TIP AMOUNT

$ -38.59

Subtotal	$30.87
Tax	$2.16
Additional Tip	$-38.59
Srvc Chrg 18%	$5.56
Amount Due	**$0.00**

Pay Now

hide01.ir

https://t.me/CyberFreeCourses

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] 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.

hide01.ir

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.

So, in the next section, we will do just that, and will then start going through any interesting functions we identify.

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

https://t.me/CyberFreeCourses

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()`

hide01.ir

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.

```
import { Coupon, validateCouponCode } from "../models/coupon.js";
import { buyCubes } from "./cubes-controllers.js";
import { buySubscription } from "./subscriptions-controllers.js";
import { buyExam } from "./exam-controllers.js";

export async function applyCoupon(req, res, next) {
  const { coupon } = req.body;
  const userId = req.user?.id;

  // validate coupon
  const errors = await validateCouponCode({ coupon });
  if (errors) {
    return next(errors);
  }
```

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:

https://t.me/CyberFreeCourses

```
// 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:

```
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

hide01.ir
https://t.me/CyberFreeCourses

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.

hide01.ir

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()`

https://t.me/CyberFreeCourses

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:

```
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 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.

hide01.ir

https://t.me/CyberFreeCourses

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.

Local Testing (Validation) - Unexpected Input

With two shortlisted functions, we can start testing them individually, starting with `processPayment()` from **payment-controllers.js**. The main thing we are looking for is weakness in type safety, and potential cases where we may be able to use an unexpected type that passes the validation filter.

processPayment()

From a first look, we can see that this function is longer than average, which means we may have a larger attack surface within the function. As mentioned in the previous section, the function has two direct user inputs we can control; `cardId` and `items`.

We see that `cardId` gets used to retrieve the user's payment card with `PaymentCard.findOne`. However, this is also coupled with the `userId` from the auth token, so it is unlikely that we would be able to trick it to somehow use another user's payment card, even if we knew their card's ID. If a user doesn't have a card, or provides an incorrect `cardId`, we get an error, as we can notice in the front-end `cart`.

hide01.ir

```
try {
  card = await PaymentCard.findOne({
    userId,
    _id: cardId,
  });

  if (!card) {
    throw new Error();
  }
} catch (err) {
  return next({
    message: "Could not find a card with this id for this user.",
    statusCode: 404,
  });
}
```

Exercise: Try to confirm the validity of the above test, by sending a request to the `charge` endpoint with any string as the `cardId`. You can use an empty array as the value of `items`.

https://t.me/CyberFreeCourses

So, this leaves us with `items`. We see that the function loops over each `item` of the `items` we sent. We do not, however, see anywhere that the function `validates that the items parameter is actually an array`. So, what would happen if we do send a number instead of an array?

```
// validate items + get prices
try {
  for (const item of items) {
    const { name, category, price, amount } = item;
    <SNIP>
  }
}
```

Note: We chose to send a number instead of a string, because JavaScript sees a string as an array as well, and can loop over it. If we send an empty string, it would be the same as sending an empty array.

Let's try to send a request with `RapidAPI`, as we did previously. Backtracking `processPayment()` in `/routes`, we see that it needs a `POST` request to `/api/payment/charge`.

Challenge: Try to see how we get the API for this function, by searching for `processPayment` in the `/routes` directory, and linking that with the routes defined in `app.js`.

hide01.ir

Obtaining cardId

As shown earlier, the `/charge` API endpoint requires us to send a valid `cardId`; otherwise, the function would stop executing before we reach the part we are testing. If we log in with the provided user credentials, the pre-added user has a payment card added to their name, so we'll try to grab their ID. If this were another production application, we would probably just add a payment card and grab its ID, but here, the process of adding a card is irrelevant, so we pre-added it to the DB.

To grab the `cardId`, we have two options: Monitoring requests on the front-end, or reviewing code on the back-end. We'll go with the "more fun" route, and will review back-end code. To do so, we can utilize the other `API` endpoint we see in `payment-routes.js`, which is `getUserCards()`:

```
router.use(verifyToken);
router.get("/cards", getUserCards);
router.post("/charge", processPayment);
```

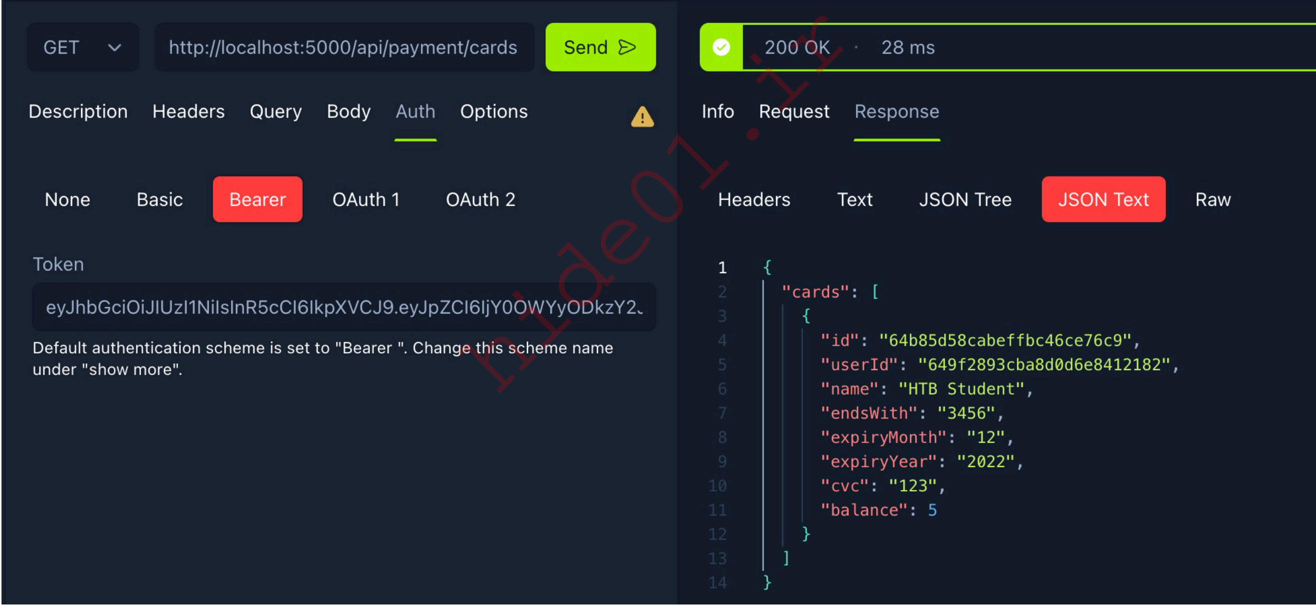
https://t.me/CyberFreeCourses

If we CMD/CTRL click on it, we can read the function, and will see that it does not take any input, and basically uses our `userId` from our auth token to obtain our payment cards, and then return them to us:

```
const userId = req.user?.id;
let userCards = null;

try {
  userCards = await PaymentCard.find({
    userId,
  });
  <SNIP>
}
```

So we only need to send a `GET` request to `/api/payment/cards` along with our auth token, and we should get the `cardId`. Once we send the request, we simply get a list of our cards, which includes each card's `id`:



This is likely used to avoid sending sensitive card details over the network (e.g. full card number), and rather use a reference to obtain it from the back-end. With `cardId` at hand, we can proceed with our attempt at tampering with sending an unexpected type for the `items` parameter.

Note: As mentioned earlier, we could have reached the same conclusion by monitoring the requests sent by the front-end at the checkout page "on the cart". However, unlike the previous demo, we are trying here to stick to back-end analysis only in order to learn a different approach of code review. For the sake of thoroughness, we will also demonstrate the other approach in a bit.

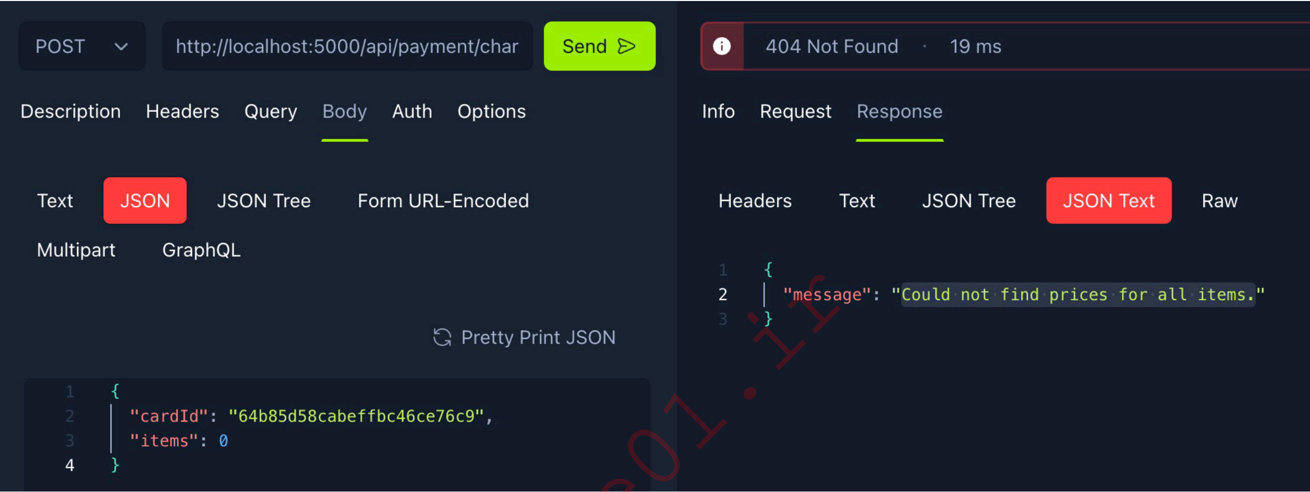
Expected Arrays vs Unexpected Numbers

https://t.me/CyberFreeCourses

We can prepare a `POST` request to `/api/payment/charge` as planned earlier, and include our auth token in `Bearer`, as shown previously. For the JSON body, we will add the `cardId` we just obtained, along with the `items` key with any number for its value, like the following:

```
{
  "cardId": "64b85d58cabeffbc46ce76c9",
  "items": 0
}
```

Once we send the request, we see that it takes quite a while, and then responds with `Could not find prices for all items`:



Checking the code, we see that this is within the `catch` block in the function, as the function must have failed trying to loop over a number "instead of an array". We can confirm this by watching the `items` variable and setting a `breakpoint` within the `try` block, then repeating the above request. After that, we can step into it the block to better understand what happened exactly and where it broke.

https://t.me/CyberFreeCourses

```
export async function processPayment(req, res, next) {
  const { cardId, items } = req.body;
  const userId = req.user?.id;

  // get card from db
  let card = null;
  let total = 0;

  try {
    card = await PaymentCard.findOne({
      userId,
      _id: cardId,
    });

    if (!card) {
      throw new Error();
    }
  } catch (err) {
    return next({
      message: "Could not find a card with this id for this user.",
      statusCode: 404,
    });
  }

  // validate items + get prices
  try {
    for (const item of items) {
```

As we can see, the specific error we got was (`TypeError: items not iterable`). This confirms that this, indeed, is an `unexpected input` bug, albeit harmless as far as we can tell. The `try/catch` block, in this case, saved the function from continuing with a faulty `items` value, though the function should have validated that the `items` value is indeed an array and is not empty; otherwise, it would not land in the for loop and reach the `validateCartItemDetails()` validation within.

We can consider this a `false positive` and move on with reviewing the function code. This is normal in any code review practice, as not everything we test will be useful or vulnerable. Most things we test will be false alerts, especially with securely coded applications. Still, the function filtering we did in the previous section should reduce the count of false alerts, and hopefully get us to test the most promising functions of the application.

validateCartItemDetails()

Now we get to the interesting part that caught our eyes in the previous section, which is the function that validates our user input. The function starts by looping over the `items` array, as we have established already, and then does the following for each item:

```
const { name, category, price, amount } = item;

// validate CartItemType array
const errors = await validateCartItemDetails({
  name,
  category,
  price,
  amount,
});
```

hide01.ir
https://t.me/CyberFreeCourses

```
  if (errors) {
    return next(errors);
  }
```

We see that it obtains four variables from each `item`, and then validates them with `validateCartItemDetails()`. We can once again CMD/CTRL click on it to go through what it is doing, and we see that it is running a `yup` validation test based on the `CartItemSchema`, which we can find right below it:

```
export const CartItemSchema = yup
  .object({
    name: yup.string().required(),
    category: yup.mixed().oneOf(["subscription", "exam",
"cubes"]).required(),
    price: yup.number().required(), // in usd
    amount: yup.number().required(), // item count
  })
  .required();
```

hide01.ir

As we have mentioned in the previous section, any code reviewers need to be able to validate the solidity of any validation test of any form, so let's go through this schema and try to pick it apart.

CartItemSchema

The first thing we notice is that every item in the schema is marked as `required()`, along with the entire `object` being required. So, if the `item` object is entirely missing, or if any of the 4 keys are missing, then it would error out.

Apart from that, it starts by ensuring that the `name` value is a string, which appears to be acceptable, as this is the expected format. For `category`, it ensures that it has to be one of the three allowed values (`subscription`, `exam`, `cubes`), which also appears to be acceptable as it is quite restrictive. As for `price` and `amount`, we see that it ensures that their type is a number, but it provides no further validation on top of that, even though these are the main elements in this JSON object, and are very sensitive to the `processPayment` function.

What is missing from the `price`/ `amount` validation? First of all, it does not ensure that the `number is a positive number`, so we can provide a negative amount or price, which can lead to various issues, as we will see. Furthermore, `it does not set a minimum amount` for either, so we can use `0` for the amount, which does not make any sense from a purchase processing point of view, but the function does not validate that as well. So, `the validation test is clearly flawed`.

https://t.me/CyberFreeCourses

So, in the next section, we will see if we can take advantage of this logic bug, and if we can validate that, we will move to create a proof of concept.

Local Testing (Manipulation) - Unexpected Input

Now that we have solid evidence suggesting that the validation test is flawed in `CartItemSchema`, let's continue our testing to see if we can take advantage of this. Our goal is to prove that we can impact the total price we would pay by sending unexpected input, which negatively affects the total price calculations and alters the intended logic.

Manipulating Prices

Let's start with the easier option, which is to provide a price of `0` for any item we purchase, which is allowed by the schema since the only condition for the parameter is to be a number. This may potentially allow us to buy any item completely free of charge.

We can start with a dynamic test, and send a `charge` request to buy some cubes with the price of `0`. Once again, we can simply copy the request sent by the front-end, but we'll try to stick to the back-end, and will create our own JSON object. We can reuse the previous JSON object that we sent, but instead of a `0` for the `items` value, we will use an object with the 4 elements shown earlier, and set "`price`" to `0`, as follows:

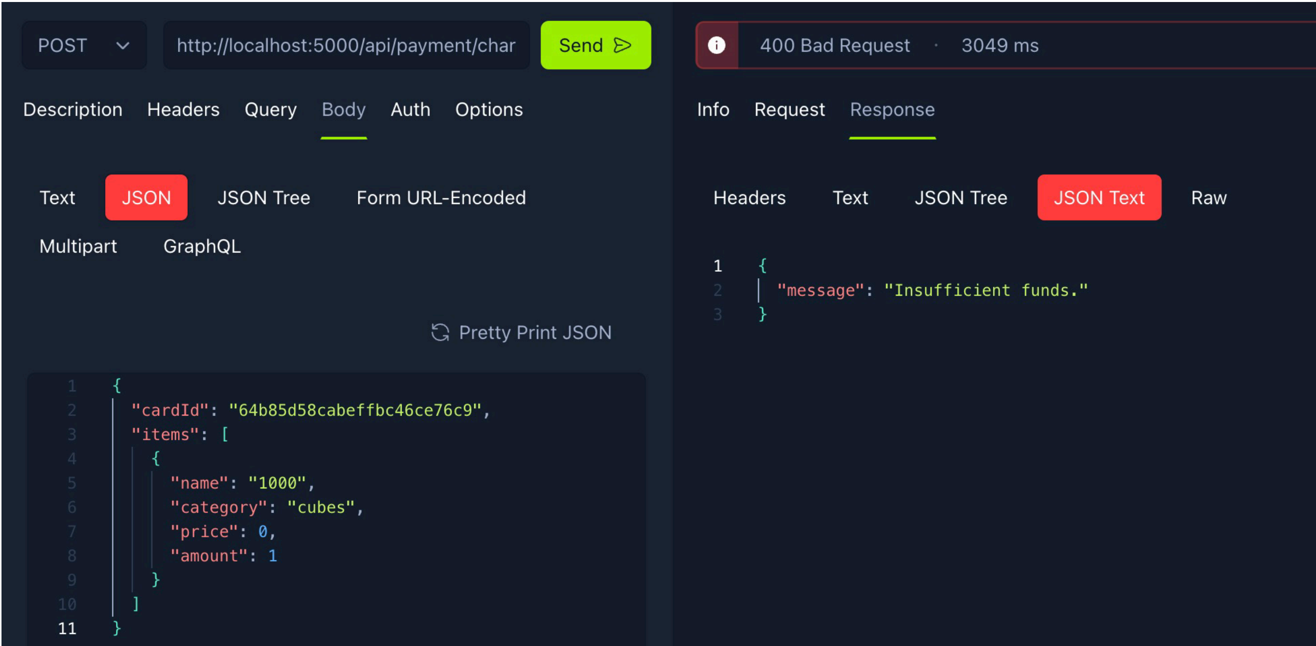
hide01.ir

```
{
  "cardId": "64b85d58cabeffbc46ce76c9",
  "items": [
    {
      "name": "1000",
      "category": "cubes",
      "price": 0,
      "amount": 1
    }
  ]
}
```

Note: By reviewing the code, we understand that the `name` value is meant to match the `subscription` or `exam` names, but what about `cubes`? If we read the function's code, we see that it parses it as a number and uses it as the number of cubes in (`parseInt(name)`), so we will enter the number of cubes we want to get "hopefully for free!". Once again, we could have easily used the front-end to obtain this information, but we chose the code review route.

Unfortunately, if we send the request, we get an error message saying `Insufficient funds`:

https://t.me/CyberFreeCourses



How can we possibly not have enough funds, when the price we set is `0`? If we continue reading the remaining part of the `for` loop, we will see that the function is not using the prices we sent, and instead either manually calculating them or pulling them from the DB:

```
case "cubes":
    total += (parseInt(name) * amount) / 10;
    break;
```

So, with the `price` value not being used, it would appear that it does not cause a logic bug, even though it was not being properly validated.

hide01.ir

Challenge: As the function parses the `name` value as an integer, what would happen if we send a value that cannot be parsed? Does the code validate this information as well? Try and see what happens, and try to find out why it is happening.

Manipulating Items

Even though the `price` value is not being used, the `amount` value is definitely being used and does indeed affect both the total price and the later item processing function:

```
switch (item.category) {
  case "cubes":
    total += (parseInt(name) * amount) / 10;
    break;
  <SNIP>
}

// process items
try {
  for (const item of items) {
```

https://t.me/CyberFreeCourses

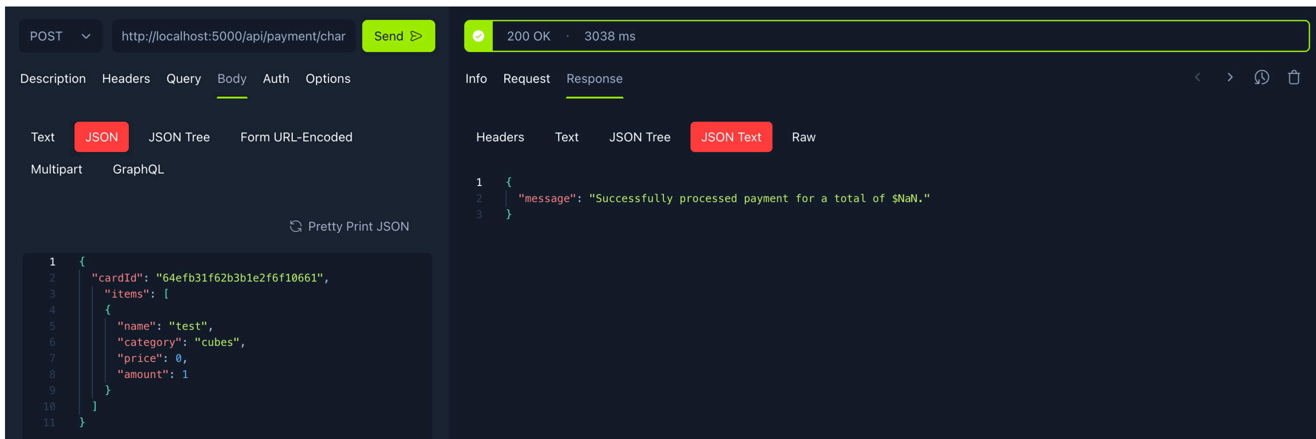
```
      // repeat by the amount
      for (let i = 0; i < item.amount; i++) {
        switch (item.category) {
          // if coupon.type is cubes, then buy cubes for user with item.name
          case "cubes":
            await buyCubes(userId, parseInt(item.name), next);
            break;
          <SNIP>
        }
      }
    }
  }
```

With `amount` not being properly validated, can we manipulate it by sending an unexpected value to take advantage of this flaw? Let's review the rest of the code to see where and how it is being used. Within the above `process items` for loop, we see that for each item, it uses a `switch` statement to execute the proper type of calculation depending on the specified `category`. For `cubes`, it simply parses the `name` into an integer, and then multiplying that by the `amount` to calculate the total price:

```
case "cubes":
    total += (parseInt(name) * amount) / 10;
    break;
```

hide01.ir

As we asked earlier, what would happen if we use a string that cannot be parsed into an integer, like `"test"`? If we do so, e see that the request passes all tests, and even returns a successful purchase, as shown below:



We can set watchers to the `total` and `name` variables, and set a breakpoint at the beginning of the `items` loop, and then follow the function execution, as shown below:

https://t.me/CyberFreeCourses

```
export async function processPayment(req, res, next) {
  const { cardId, items } = req.body;
  const userId = req.user?.id;

  // get card from db
  let card = null;
  let total = 0;

  try {
    card = await PaymentCard.findOne({
      userId,
      _id: cardId,
    });

    if (!card) {
      throw new Error();
    }
  } catch (err) {
    return next({
      message: "Could not find a card with this id for this user.",
      statusCode: 404,
    });
  }

  // validate items + get prices
  try {
    for (const item of items) {
```

We will see that when JavaScript tries to parse the string " `test` " into an integer, it fails and returns `NaN` instead, but it does not throw an error or stop execution! This is an odd behavior in JavaScript, and will be discussed further in the upcoming sections. The key learning point is that JavaScript's `dynamic typing` can proceed with the execution, even if the values are completely off (i.e. null or NaN).

So, in this case, it corrupted the total to `NaN`, which passed the `payment card balance test`. When it tried to `buy cubes`, it did the same thing, and tried to adjust our cube count by `NaN`, which may potentially corrupt our data in the database. Finally, it tried to adjust our card's balance with the `total` (i.e. "NaN"), which may as well corrupt the database.

This is the `second unexpected input` bug we found in this function, in addition to the `items` array bug we showed in the previous section. While both of these bugs were saved by other measures in place (or sheer luck!), both of these bugs must be remediated, as they pose serious risk to the functionality of the web application. In the `null safety` sections, we will demonstrate how `null` or `NaN` values can lead to serious consequences. In any case, there is no immediate action we can perform to take advantage of this bug, so we will continue with the rest of the function.

Manipulating Amounts

So, let's further dissect the process we went through as we did the debugging shown above. If we continue and review the `subscription` switch case, we see that the function first retrieves the subscription name from the DB, and then uses its `cost` value to calculate its price, and multiplies that by the `amount` we sent. The same is done for `exam`. The function keeps aggregating the total sum throughout the for loop to calculate the `total price for all items based on the amounts we sent`.

hide01.ir

https://t.me/CyberFreeCourses

After that, the function ensures that the calculated `total` is within our card's balance with `if (total > card.balance)`. Then, the function moves into processing each item, by looping over each and passing it to its appropriate function for purchase processing (`buyCubes`, `buySubscription`, and `buyExam`). It also repeats this by the `amount` count, so that purchasing multiple items would be processed multiple times.

Finally, the function adjusts our cards balance by the total prices for all items, which means it would deduct the total price from our payment card.

```
card.balance = card.balance - total;
await PaymentCard.updateOne(
  {
    userId,
    _id: cardId,
  },
  card
);
```

Throughout all of this, we never see it validating (or even considering) that the amount may be a `negative amount`, all while using the `amount to calculate the total price` and the number of times an item gets processed. So, we could `potentially send a negative amount` for a certain item, and when it multiplies this negative amount with the item's price, this should lead it to `reduce the total price` instead of increasing it, when it does the following:

hide01.ir

```
total += (parseInt(name) * amount) / 10;
```

We do not care whether it would process this item or not later on, but as long as this does not lead the entire function to fail, we can potentially use this flaw to get a reduced total price, or potentially pay nothing at all. We will verify this finding in the next section.

PoC and Patching - Unexpected Input

After all of our testing, we can finally test our findings to see if we can truly buy cubes for free, by exploiting the type bug we identified.

Proof of Concept

Let's assume we want to get some free cubes or a reduced fee. We can include an item with a negative amount, and this amount should reduce the total price. For example, we can specify a `100` cubes, which should have a price of `$10` (according to the calculations we

https://t.me/CyberFreeCourses

showed previously or according to the front-end). If we specify the amount to be `-1`, then the `total` price should be adjusted by (`10 * -1 = -10`). If we add another item with an equal but positive amount, we should get a total of `$0` charge, and our card's balance should not be affected.

Let's reuse the previous payload we sent, and adjust it to match the above criteria, as follows:

```
{
  "cardId": "64b85d58cabeffbc46ce76c9",
  "items": [
    {
      "name": "100",
      "category": "cubes",
      "price": 0,
      "amount": 1
    },
    {
      "name": "100",
      "category": "cubes",
      "price": 0,
      "amount": -1
    }
  ]
}
```

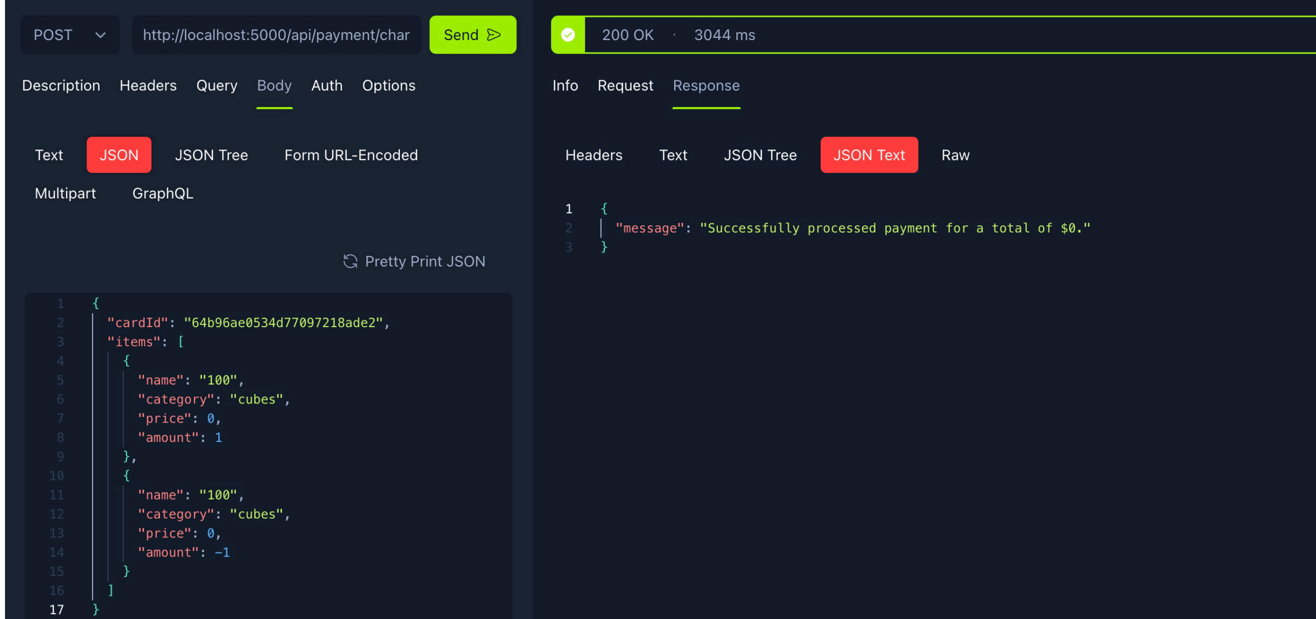
hide01.ir

When the function processes this request, it will first validate our `cardId`, then iterate over the `items` to calculate the `total` price, which should first be at `10`, and on the second iteration should be reduced by `-10`, netting a total of `0`. Then, the function would validate that `0` is within our card's balance, which it should be even if our balance is `0`.

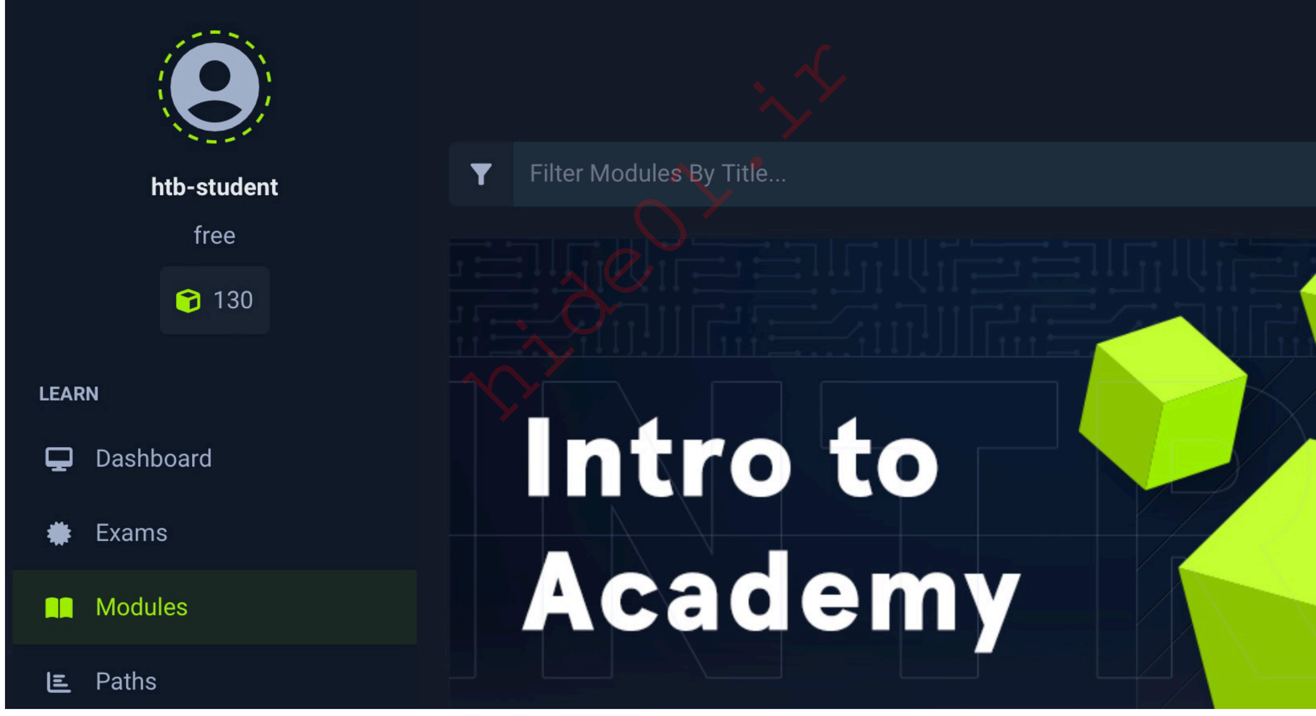
After that, the function would process the items by their amounts. So, it would process the `100` cubes once, and it should not process the negative amount at all. Finally, it would adjust our card's balance by the total price of `0`, so it should not be affected.

Let's test this by sending the above request and seeing what we get:

https://t.me/CyberFreeCourses



As we can see, the response confirms that it `Successfully processed payment` with a total of `$0`. We can now login to the front-end with the provided credentials, and we will see that our cube count has indeed increased without having to pay anything:



We can see how a minor mistake in the way cart items were being validated lead to a complete compromise of the payment system, allowing us to practically purchase any item completely free of charge, even if our payment card had a zero balance.

Exercise: It is possible to carry the same attack using nothing but the browser and the front-end web application. Try to do that. Hint: How's the browser holding our cart items?

This example demonstrates the importance of thoroughly and adequately examining and validating every input sent by the user, including any sub-items that may be included in that input/object. Without a solid validation mechanism, such an issue and many others may arise, leading to various logic bugs.

https://t.me/CyberFreeCourses

Challenge: Try to replicate what we have shown here on your local environment. After that, try to find other issues with the `processPayment` function, as well as the other function we shortlisted in the previous section.

Patching

For `Unexpected Input` logic bugs, we must ensure that we only accept the specific type of input we expect and refuse everything else. This function's main issue was insufficient input validation, as the `CartItemSchema` only checked if the `amount` is a number, but did not verify whether it is a `positive` number with a minimum of `1`. So, all we need to do is add these checks to the schema, as shown below:

```
// to process cart items in payment requests
export const CartItemSchema = yup
  .object({
    name: yup.string().required(),
    category: yup.mixed().oneOf(["subscription", "exam",
"cubes"]).required(),
    price: yup.number().positive().min(1).required(),
    // in usd
    amount: yup.number().positive().min(1).required(), // item count
  })
  .required();
```

hide01.ir

As for the two other 'minor' bugs we identified, we should also address them in our remediation notes. For the `items array bug`, we can simply throw an error if the `items` variable is not an array, as follows:

```
try {
  if (!Array.isArray(items)) {
    throw new Error();
  }

  for (const item of items) {
    const { name, category, price, amount } = item;
    <SNIP>
  }
} catch (err) {
  <SNIP>
}
```

Finally, we should fix the `name` integer parsing bug. For this bug, we need to confirm that the value of `name` can be parsed as an integer, but only within the `cubes` case, as the other cases expect strings. We can use a similar method that was used to confirm the `date`

https://t.me/CyberFreeCourses

format that we saw previously in the module, by using `isNaN(parseInt(name))` to confirm that the value can be parsed safely, as follows:

```
// add cost to total
switch (item.category) {
  case "cubes":
    if (isNaN(parseInt(name))) {
      throw new Error();
    }

    total += (parseInt(name) * amount) / 10;
    break;
  <SNIP>
}
```

In general, to patch such vulnerabilities, we need to ensure that validation tests (e.g. schema) `accurately match the expected type of input`, and do the same for every single input. If the validation tests allow extra room for manipulation (i.e. accept a wide range of input types), then they will likely be vulnerable for such attacks.

hide01.ir

Exercise: Try to patch your code and then re-apply the same above PoC, as well as what we tried in the previous sections for the other two bugs.

Null Safety

The concept of null variables dates back to 1965 in computer programming. The inventor of the `NULL` reference famously referred to it as " The Billion Dollar Mistake", because of the numerous issues it introduced to computer programs throughout the years. Null pointers in C, for example, led to various software vulnerabilities and bugs.

This should give us a basic idea as to why allowing null variables in code is generally considered a bad idea, as they may introduce run-time errors that can break the user experience. Null variables may also introduce flaws that allow users to bypass certain restrictions and access data they should not have access to, not through access control issues, but through logic bugs, as we will see in the coming sections.

Such kinds of vulnerabilities are so common that I personally encountered one while writing this very section, in the Facebook app no less. I noticed that when I accessed a comment by clicking on a notification (which triggers the app to navigate to the URI set by the notification), then whenever I would add a comment, it would fail and ask me to retry. Then, I noticed that it said `reply as (null)`, so it was probably still loading my user details, which made it fail to add the comment, likely to missing identifiers (e.g. userId).

https://t.me/CyberFreeCourses

While this is a very minor user-inconvenience logic bug, it shows that such bugs occur even with organizations that have very strict coding and security standards, which is why we should learn about them and take measures to prevent them.

Null Variables

`A null variable is a variable with no value assigned to it`. It is most often introduced when a variable is declared without an initial value (e.g. `var count;` instead of `var count=0;`). This may be useful under specific conditions, like while awaiting the value of a certain variable to be pulled from a remote resource.

This leads developers to declare the variable without an initial value, and then assign it a value once it is retrieved. If the variable is `never accessed or used before it is assigned a new value`, then the code is considered `null-safe`. However, if the variable is accessed or passed to a function/method (e.g. `count.toString()`), then it would lead to a run-time error, known as a `null reference error`, and crash part/all of the application.

There are numerous situations where the variable may accessed before it is assigned a value. For example, a front-end web application may fail to obtain the value of a variable from a remote resource (e.g. user device not connected to the internet), so it may proceed to render the page without that variable being assigned a value, which would cause a run-time error and crashes the page.

This is why it is important to ensure we know how to write code that is `null-safe` to prevent null-related run-time errors, some of which may potentially lead to logic bugs that we can take advantage of.

Null Safety

hide01.ir
https://t.me/CyberFreeCourses

Modern languages that support null safety ensure that null variables are never accessed without having a value, by enforcing certain checks/rules during edit-time or before the code is compiled. Some of the languages that have implemented null safety solutions are:

- TypeScript
- Rust
- C#
- Swift
- Kotlin
- Dart

However, many other languages, like JavaScript, do not yet fully support null safety, so many developers tend to `use certain checks to ensure a null variable is not accessed before being assigned a value`. For example, a developer may test if the variable is null (e.g. `if (count === null)` or `if !(count)`), or may perform type validation tests to ensure that the current value matches the expected type and is not null.

Even with such checks, we cannot fully avoid null reference errors, as there could be unpredicted cases that lead to accessing variables before assigning them a value.

While each language has a slightly different approach that fits its own environment, most languages utilize similar basic concepts to allow `null-safe code`. Some of these are:

hide01.ir

Concept	Description	Example
Non-Nullable variables	Variables that must be initialized with a value, and cannot hold null when being used, or the compiler/IDE will throw an error.	`let count: number;`
Nullable variables	Variables that may still hold null as their value "must be used with caution"	`let count: number
`?.` : Null aware operator (or optional chaining operator)	Used with nullable variables, and allows to only access the variable objects (e.g. `.length`) if the nullable variable is not null	`total = list?.length`
`??` / `??=` : Null-coalescing operator/assignment	Used with nullable variable, and allows to set a default value if the nullable variable is null	`total = count ?? 0` / `count ?? = 0`
`!` : Non-Null assertion operator	Used with nullable variables, allows us to bypass the null safety checks, if we are sure that at this point the variable would not be null (i.e. override the IDE's null safety check)	`total = list!.length`

https://t.me/CyberFreeCourses

Null-safe languages support both `nullable` and `non-nullable` variables, while other languages usually support only `nullable` variables. For this reason, some non-null-safe languages, like JavaScript, started to support some of the above null operators, like `?.` or `??`, to reduce potential null errors.

This also shows that even though an application may be coded in a null-safe language, it can still produce code that is `not` null-safe, as it `may be overriding some of its null safety checks` (e.g. by using `!`), which may lead to the null safety logic bugs. This is why we should try to avoid overriding these checks as much as possible, and only use null safety overrides when we are 100% sure that the variable will not be null (e.g. with `if (count !== null)`).

Note: You can refer to [this article], which rates each language's null safety, and provides a list of the best/worst languages in terms of null safety.

Identifying Null Bugs

Unlike most logic bugs, null safety issues can be reliably identified through tools, as most languages mentioned above do during edit-time. However, this only identifies potential cases where a variable may be accessed while being null, and does not perform any logic analysis of what would be affected if it does. Also, as mentioned earlier, even null-safe languages that override safety checks (e.g. through `!` or turning off null safety altogether), can still have null safety logic bugs. So, for null-safe languages, we would focus on null safety overrides.

So, the process we can follow is to `determine potential null variables that are user controllable`, and then look into the potential `logic bug that can be caused by that`. For the first step, we can do the following, depending on the type of null safety:

1. **For languages without null safety**: Look for `uninitialized` variables or ones that get assigned null afterward.
2. **For null-safe languages**: review uses of `non-null assertion operator (!)`, and see if it could lead to a null-reference error.

After we identify those, we need to review the null safety checks being done for these variables, like type validation or condition tests `if (count !== null)`. If we can `determine a case that may bypass these checks`, then we can `shortlist this variable` for further local testing and code reviews to determine whether it can lead to a potential logic bug.

With this in mind, in the next section we can start by creating a list of all potential null variables, and then proceed with the next steps.

Exercise: By now, you should have a pretty good idea of how to navigate a codebase and understand how it works. So, it is time to test your skills before continuing with the rest of the module. Try to use the above process to identify potential null safety logic bugs in the codebase. You can then compare your findings, as well as the process you followed, with

bideU.ir

https://t.me/CyberFreeCourses

what the upcoming sections will cover, which is an essential step to know what you may have done better.

Code Review (Null Variables) - Null Safety

Let's start by trying to identify variables that can have a `null` value at some stage. These generally occur in two ways:

1. `Null variables`: Declaring a nullable variable (e.g. `let const;`) without an initial value
2. `Optional parameters`: When a function has a parameter that is optional with no default value (i.e. can be set to null)

The key point for us is to identify ones that may have null safety issues, and then filter those depending on whether we have control over them. At this stage of the module, we are mainly interested in input that we have a direct/indirect control over.

JavaScript Common Null Pitfalls

Before we start identifying potential null issues with our JavaScript application, we need to learn a bit more about common pitfalls of this language. Unfortunately, JavaScript is considered to be one of the worst languages when it comes to null safety. In addition to allowing nullable variables, JavaScript also supports `three different types that do not hold a value`, which may lead to run-time errors. They are:

1. `null`
2. `undefined`
3. `NaN`

This makes checking for null variables quite tricky in JavaScript, and can often lead to unexpected outcomes. As showcased in this article, if we check the type of `null` in JavaScript, we would get an empty object, while `undefined` would simply return `undefined`. Furthermore, running `typeof` on `NaN` returns a number, even though NaN literally means Not-a-Number!

On top of all of that, JavaScript mostly doesn't even produce a run-time error when dealing with a null value, and simply continues its execution as if the variable is assigned a value, due to the flawed implementation of typeof in JavaScript. We saw an example of this in the `unexpected input` section, when the application continue execution with the `total` value being `NaN`, and returned it to us in a message.

Finally, JavaScript sometimes even parses null values to a type (e.g. `int`), which can completely change the intended execution logic. For example, if an `int` variable is being used while having a null value, then JavaScript would simply turn it to `0`. This can lead to many types of logic bugs, like comparing the null variable (now `0`) with `>=`, which would

hide01.ir

https://t.me/CyberFreeCourses

result in `true`, as we will see later on. Another example is if it attempts to obtain the cost of an item from the database and fails to do so, then it may consider the cost of the item to be 0, which could allow malicious users to trick it to `buy items for free`.

As we are dealing with JavaScript in this module, it will be very interesting to evaluate our code against all of these potential null issues, so let's see how we can go about identifying any of them.

Flawed Null Checks

This article also explains some cases where checking for null variables may not always work as expected. For example, if we only checked if a value is `undefined`, and the variable was actually `null`, then the condition (`user === undefined`) would be false, as `null` and `undefined` have different characteristics, as mentioned above. So, the application must check against both `null` and `undefined` using strict equality `===` for both; otherwise, the test may fail. The same thing applies to `NaN`.

Let's take the below code as an example:

```
const { count } = req.body;
if (!count) {
  throw new Error();
}
```

The code tries to obtain the value of `count` from the POST request body, and if this value was not sent (i.e. `!count`), then it would throw an error. However, since JavaScript considers `0` to be equal to `false` and `1` to be equal to `true`, if the sent value of `count` was `0` or `'0'` "as a string", then the code would still throw an error, even though the value was not null! This is because JavaScript would evaluate `(!0)` to be `if not false` (i.e. if true), which would pass the if condition.

This shows that even the use of the `not` operator (`!`) is not safe, and we should instead use `!==` or `===` and check against both `null` and `undefined`.

Identifying Null Variables

Let's start with the simplest case, and get a list of variables that are declared without an initial value. In JavaScript this is usually done with the use of `let` or `var`, as using `const` requires setting a value (since it's constant and can't be changed). So, we can use the VSCode `regex` search with the pattern `(let|var) [A-Za-z]*;` and also limit it to the

hide01.ir

https://t.me/CyberFreeCourses

`src/src/controllers/**` directory, and it should locate all such variables:

As we can see above, the search yielded 42 results found in 9 different files. The first result, as we can see in the screenshot above, attempts to set the value of `user` to the output of `User.findById(userId)`, and if no output is returned (i.e. it is still null), it throws an error and stops execution:

```
if (!user) {
  throw new Error();
}
```

All errors in this web application are passed to the `next()` function, which is whatever function passed to the current function as the `next` parameter, as seen below:

```
export async function createUserToken(userId, next)
```

If we backtrack this `next` function, we will see that it was passed down from the `login()` function, which eventually leads to the Express `app.use` function on line `68` in `app.js`. This basically returns an HTTP error to the user with the error string provided by the `next` / `catch` function. Once it detects a null, it'll throw an error, which will stop execution and return whatever error was thrown to the user.

Exercise: Try to review the other results from above using the same process we just did, and see if any of them skips the null check "e.g. `if (!user)`". If you find any, then it can be shortlisted as a potential null safety issue.

Going through the rest of the results, we will notice that the application is generally consistent in checking for null variables before proceeding with execution. However, it is always validating that a variable is not null through the use of `if (!variable)`, which, as has been discussed earlier, can lead to unexpected outcomes in some cases, which may end up causing a logic bug.

hide01.ir

https://t.me/CyberFreeCourses

We will not focus on these, as their values are usually derived from database functions or other functions, which means that it is unlikely to return a null value, as then the database would return an error and cause the `catch` clause to throw an error and stop execution. Still, it may be worth keeping those as a last resort, as there may be some cases where they would cause a logic bug.

Note: In this module, we are trying to cover different approaches and potential ways to identify the logic bugs we are discussing. While our exercises may not suffer from all of them, it is still important to learn these approaches/techniques, as well as learn the way of thinking when identifying such bugs, as they may be found in other applications you test.

For the time being, let's continue with the second case we are looking for, which are `optional parameters`, as we will see in the next section.

Code Review (Optional Parameters) - Null Safety

In the previous section, we named 2 ways where null bugs may occur: `null variables` and `optional parameters`. We went through all cases of nullable variables, and found that they were mostly being handled safely.

The second common way that leads to null errors is where null values/variables are assigned to nullable/optional parameters, which occurs with `functions` and `objects`. Both `functions` and `objects` may allow optional parameters/keys, and if these parameters do not have a default value, we need to see how the function would handle them if they are not passed (i.e. when they are null).

hide01.ir

In this section, we will mainly try to identify API endpoints that utilize user input, and then see if any of the endpoint parameters is optional (i.e. can be null). If they do allow optional parameters, and do not perform proper tests to ensure that null values are not used, then logic bugs may occur, which is what we will try to identify.

Identifying Null Parameters

JavaScript supports named parameters (e.g. `({param1, param2})`) and positional parameters (e.g. `(param1, param2)`). The main benefit of using named parameters is that we don't need to place them in order correctly and can simply call each argument with the parameter's name. This also allows optional parameters, which accept null as their value, as mentioned above. However, suppose the function accepts an optional parameter, and it `did not perform proper null checks` and eventually did `use the parameter while being null`. In that case, it may lead to run-time errors and logic bugs.

The same thing applies to `objects`, which contain named parameters that we can then be assigned to a function through `direct assignment` (e.g. `const param1 = obj.param1;`) or `object destructuring` (e.g. `const {param1, param2} = obj;`). This is very commonly used, especially with `JSON` objects in POST requests (e.g. `req.body;`), which is much simpler and shorter than assigning each JSON parameter to a variable.

https://t.me/CyberFreeCourses

The question is, `what if an endpoint expects us to send a certain parameter, and we don't?` This is quite normal and is usually handled by verifying the JSON object through schemas (as we've seen before) or any other means. So, if we send an empty body, for example, it should be handled gracefully by throwing an error specifying what exactly was missing. Having any null safety bugs with optional parameters in end-points `may lead to serious logic bugs`, especially since they will be directly controllable by the end-user.

Checking for missing API/JSON parameters is quite similar to the null checks we mentioned in the previous section, and can suffer from the same pitfalls we discussed previously. So, when it comes to null safety logic bugs, this is the area that is most likely to cause issues.

Required vs Optional Parameters

Since we are mainly looking for issues with user input, we can go back to the endpoints that utilize user input, which we listed previously in the `Unexpected Input - Code Review` section. However, this time, we will be checking whether there are sufficient tests to validate that the input is not null.

We have also previously found out that all endpoints either validate the user input through a schema, unless it is an ID, in which case it would be validated by searching for it in the database and throwing an error if no matches are found. It is always useful to go through all user-controllable input and check whether any of them are missing any checks, but as we have already verified that in the `unexpected input` sections, we can safely assume that all input will be validated through the database or with a schema.

hide01.ir

So, `does this mean that the code is safe from null safety issues`? Of course not, as schemas may still allow null parameters if not configured properly.

While it is different from one validation tool to another, almost all of them allow `required` and `optional` parameters, just like `named functions`, as we discussed earlier. In this case, the `yup` package allows us to specify mandatory fields by using the `.required()` option. If the endpoint allows certain optional parameters, then the `.required()` option will not be used, and the endpoint should act accordingly, depending on whether the parameter is used or not. This means that we need to `identify all instances of optional parameters used in different schemas`, and then study those for null safety issues, as the `endpoint may not be coded with solid logic`, which may lead to a logic bug.

Luckily, we have already identified endpoints that utilize schemas with user input in a previous section. Here they are again, along with the names of their schemas:

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

https://t.me/CyberFreeCourses

Exercise: Check the above 4 schemas for optional parameters (missing the `.required` option), and then look for those optional parameters in the endpoints that utilize their schemas (also found in the `Unexpected Input - Code Review` section) to see if they properly check for null values being used those optional parameters.

Identifying Optional Parameters

We can now start identifying optional parameters that may hold null values, and then review their functions to see how they handle optional parameters. Let's start with the above schemas, and go through them one by one to see if they support any optional parameters.

The first one, `CouponCodeSchema`, is quite basic as it only supports one parameter `coupon`, which is denoted with `.required()`, meaning it is not optional, and the schema validation test will throw an error if we don't provide it:

```
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(),
});
```

Next, we have `CartItemSchema` (that we patched from unexpected input bugs), which has four parameters, but all of them have `.required()`, meaning none of them is optional. On top of that, the entire object is itself `.required()`, meaning that if we send an empty cart, it will also error out:

```
export const CartItemSchema = yup
  .object({
    name: yup.string().required(),
    category: yup.mixed().oneOf(["subscription", "exam",
"cubes"]).required(),
    price: yup.number().positive().min(1).required(),
    // in usd
    amount: yup.number().positive().min(1).required(), // item count
  })
  .required();
```

The `passwordResetSchema` comes after that, as follows:

```
const passwordResetSchema = object({
  // validate mongodb object id
  id: mixed((value) => ObjectId.isValid(value)).typeError("Invalid id"),
```

hide01.ir
https://t.me/CyberFreeCourses

```
  // validate bcrypt hash
  token: string().matches(/[0-9a-f]{32}/i, "Invalid token"),
  // validate password
  password: string().min(5),
}).required();
```

We see that while the entire object is `.required()`, none of the parameters has this option, which is a bit of an odd case. So, we have to provide at least one of them, but it does not matter which one we send, since `all of them are considered optional`. This is definitely interesting, and we should take a look at it later on.

Finally, we have one of the most used schemas in the whole application, which is `UserSchema`. Unlike the previous three schemas, all of which were used once each, this schema is used for most user endpoints, like `createUser`, `login`, `updateUserDetails`, and `requestPasswordResetLink`. All of these functions are sensitive and interesting, which makes this schema high on the list of priorities. So, let's look at its parameters:

```
export const UserSchema = yup
  .object({
    id: yup.string(),
    name: yup.string(),
    username: yup.string(),
    email: yup.string().email().required(),
    password: yup.string().min(5).required(),
    registrationDate: yup.date(),
  })
  .required();
```

hide01.ir

As we can see, four out of the six parameters are optional, namely: `id`, `name`, `username`, and `registrationDate`. This is quite an odd choice, as we expect parameters like `id` or `username` to be required, but perhaps this is due to it being used with multiple endpoints, and not all endpoints require all of these fields. In any case, it is worth shortlisting for our local testing.

Local Testing (Schemas) - Null Safety

With a list of endpoints with optional parameters, we can start reviewing the use of these optional parameters, and see how they are handled by the endpoint, as they may have null safety issues. If we do identify any null safety issues, we can move to the more difficult part, which is trying to see how we may take advantage of such flaws.

UserSchema

https://t.me/CyberFreeCourses

In the previous section, we identified optional parameters in two different schemas: `passwordResetSchema` and `UserSchema`. All of the uses of both schemas are found in the `users-controllers.js` file under `/controllers/`. We will start with the more interesting/sensitive of the two schemas, which is `UserSchema`.

As mentioned in the previous section, this schema has four optional parameters: `id`, `name`, `username`, `registrationDate`, and is used to validate user input in 4 different endpoints: `createUser`, `login`, `updateUserDetails`, and `requestPasswordResetLink`. So, let's go through each of these, and see how it is handling potential null variables.

We will start with `createUser`. The first thing we notice is that it performs null checks for the `name` and `username` variables:

```
// name/username are not set as required in the schema, so we need to
check for them here
if (!name || !username) {
  if (!name) {
    <SNIP>
  } else {
    <SNIP>
  }
}
```

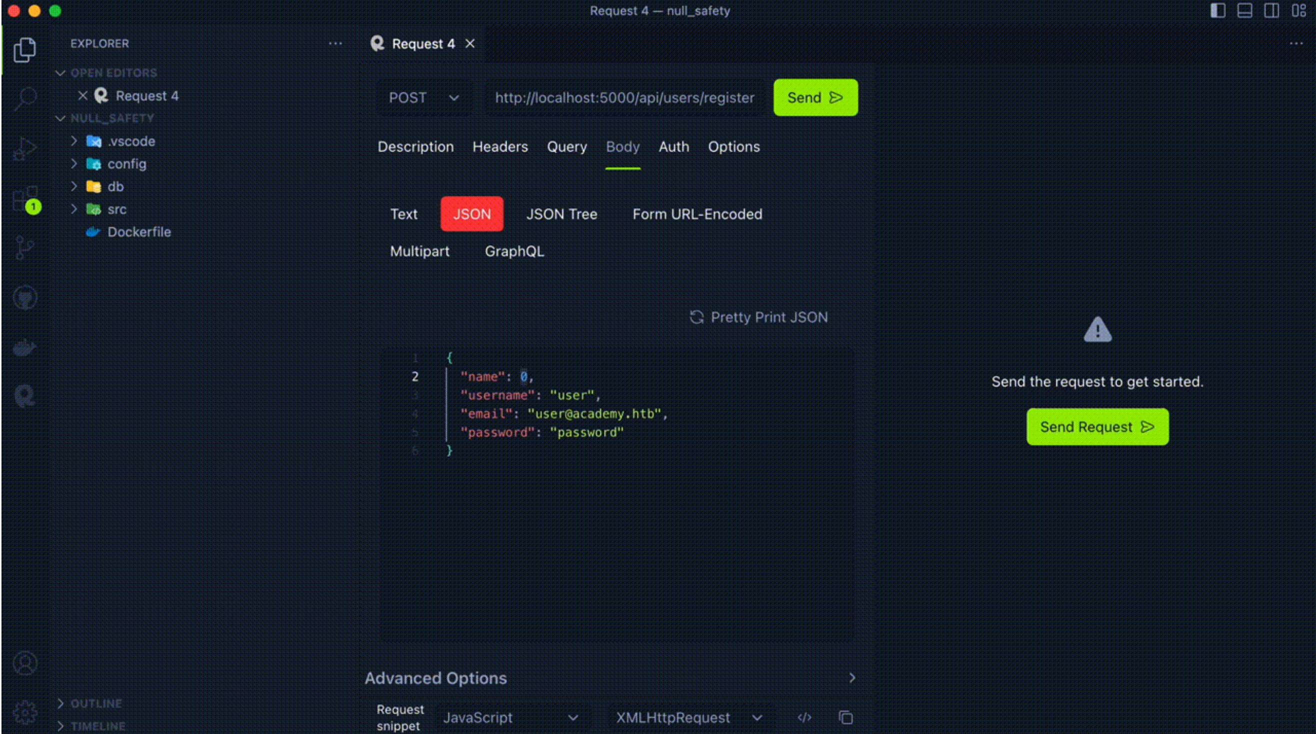
hide01.ir

This raises the question: Why weren't these parameters set as `.required()` in the schema instead of null-checking them here? The most likely answer is that these fields may not be required in all endpoints that use the `UserSchema` for input validation; hence, they are set as optional.

However, this is a bad implementation, as it may lead to null safety issues, and the better implementation would be to use `conditional requirements` for the schema. This means that we set a certain condition under which each parameter would be considered optional or required, like having the `useUsername` toggle, for example, which may set some of the parameters as optional. Another option would be to use a different schema for each use case, although this is not always ideal or efficient, given the amount of duplication it will cause.

In addition to the above, we have already established that the use of the not operator (`!`) has some pitfalls in JavaScript. For example, if we set the name to be `0`, then the above condition would be triggered, even though we did not set it to null, and setting it to `1` or `"0"` or any other string would work, as demonstrated below:

https://t.me/CyberFreeCourses



While this is obvious in this case, as a name should never be `"0"`, some other parameters that expect numeric values may face the same issue, like `amount` in the example we saw in the `unexpected input` sections. So, it is also advisable to use a specific pattern when validating each parameter, wherever possible, to avoid such edge cases. Furthermore, in case we need to perform local null checks, we should avoid using the not operator (`!`), and stick to strict equality (e.g. `if (name === null || name === undefined)`).

Other than `name` and `username`, the `id` and `registrationDate` are automatically generated and not passed as user input, so their usage is safe in this case. If we move to the `login` endpoint, we see that it accepts only two user inputs: `email` and `password`, both of which are required by the schema, so they are safe as well.

Next, we have `updateUserDetails`, which accepts 3 user input parameters: `name`, `username`, `email`, two of which are set as optional by the schema (`name` and `username`). The endpoint uses the schema validation to test the user input:

```
const errors = await validateUserDetails({
  email,
  password: process.env.VALIDATION_TEST_PASSWORD,
  name,
  username,
});
if (errors) {
  return next(errors);
}
```

hide01.ir

https://t.me/CyberFreeCourses

We see that the code uses a pre-set random password for the required `password` field, since it is a required field, but is not used in this endpoint. This is another example of a `bad schema design`, as it is using a workaround to pass the required parameters, even though it does not need to. Once again, this is another case where `conditional requirement` must be used, which should set the `password` parameter as `optional` if the `usePassword` toggle is set to false.

Other than that, we see that both `name` and `username` are being used later on to update user details in the database, even though they are set as optional parameters by the schema, and we don't even see the endpoint checking whether they are null, like the `createUser` endpoint did above:

```
const updateReq = await User.findByIdAndUpdate(
  // use id from token to ensure users can only update their own account
  req.user?.id,
  {
    email,
    name,
    username,
  },
  {
    returnOriginal: false,
  }
);
```

This is `definitely an issue`, as we can store null values in the database, and if anywhere in the code it retrieves these values while not expecting null values, `it may lead to a run-time error or a process crash`. We will further test this later on.

Finally, this schema is used in `requestPasswordResetLink`, which only accepts one parameter (`email`), and this parameter is marked as `.required()` in the schema, and may be considered safe as well.

PasswordResetSchema

Moving on to the second schema. Unlike the previous schema, this one is only used once within the same endpoint it is defined in. We have previously noticed that all of its parameters are set as optional, even though the entire schema is set as `required`. This means that we need to `provide at least one of the parameters` in order to pass the schema validation test, but it does not matter which parameter we choose to use.

Let's try to see where each of the parameters is being used in the endpoint. We see that the `id` parameter is being used to identify the user, and then to calculate the secret password-reset token:

hide01.ir

https://t.me/CyberFreeCourses

```
// retrieve user based on id
try {
  user = await User.findById(id);
  <SNIP>
}
<SNIP>

// generate password reset token based on password hash and user id
const hashedToken = md5(`${id}:${user?.password}`);
```

As for the `token`, it is mainly being used to validate against the secret token, as calculated above:

```
// verify password reset token based on password hash and user id
// if not valid, return error
if (token && token !== hashedToken) {
  return next({
    message: "Invalid password reset token.",
    statusCode: 403,
  });
}
```

The `password` is simply used to modify the password once everything has been verified. So, the application basically accepts the user `id` and the secret token, then dynamically compares the sent token with the one it calculates based on secret user details stored in the databases (mainly the old password hash).

hide01.ir

Most modern applications generate a random and long secret token upon a password reset request, and then store it in the database with an expiration date, to ensure the token cannot be guessed and that it expires on time. Still, some smaller applications use this dynamic approach to avoid storing tokens on the back-end, and simply calculate it on run-time using secret information that users cannot guess. Even though the first approach is recommended, this approach is not necessarily insecure, so we will not consider it as a security risk.

However, `is this the only issue with the code?` The main issue that got us here is that all of these parameters are considered optional. So, `what would happen if any of them is set to null?` This is what we will test in the next section.

Local Testing (functions) - Null Safety

---

https://t.me/CyberFreeCourses

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.

hide01.ir

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:

```
{
  "email": "[email protected]"
```

https://t.me/CyberFreeCourses

```
}
```

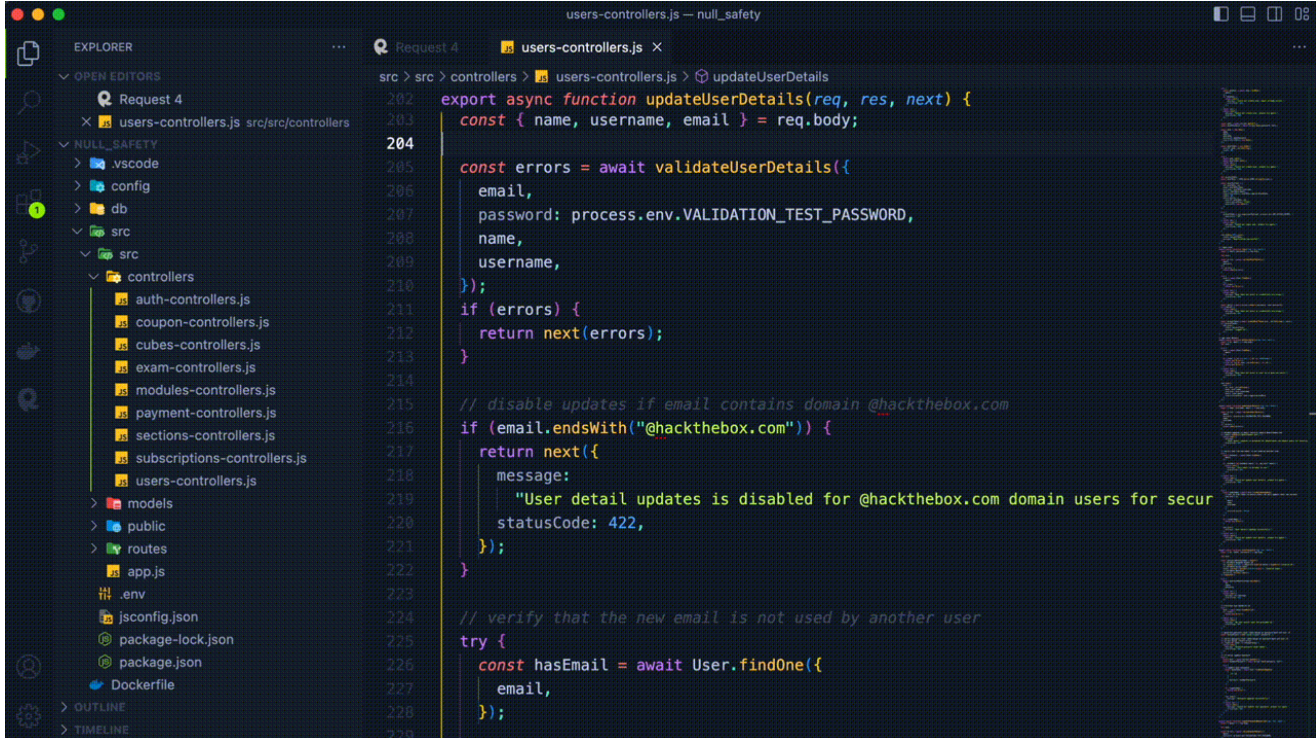
To find the route of the API endpoint, we can CMD/CTRL click on the `updateUserDetails` function name, and we will see that it is being referenced in the `user-routes.js` file:

```
202   export async function updateUserDetails(req, res, next) {
users-rout...                                                        Reference... ×
10    } from "../controllers/users-controllers.js";                 users-controllers.js src/src/controll... 1
11                                                                    function updateUserDetails(req, res, ne
12    const router = express.Router();                               users-routes.js src/src/routes 2
13                                                                    updateUserDetails,
14    router.post("/register", createUser);                           update", updateUserDetails);
15    router.post("/login", login);
16    router.post("/details", getUserDetails);
17    router.post("/password/reset", resetPassword);
18    router.post("/password/request_reset_email", requestPasswordRese
19
20    // secure private routes for content (use req.user in private co
21    router.use(verifyToken);
22    router.post("/update", updateUserDetails);
```

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.

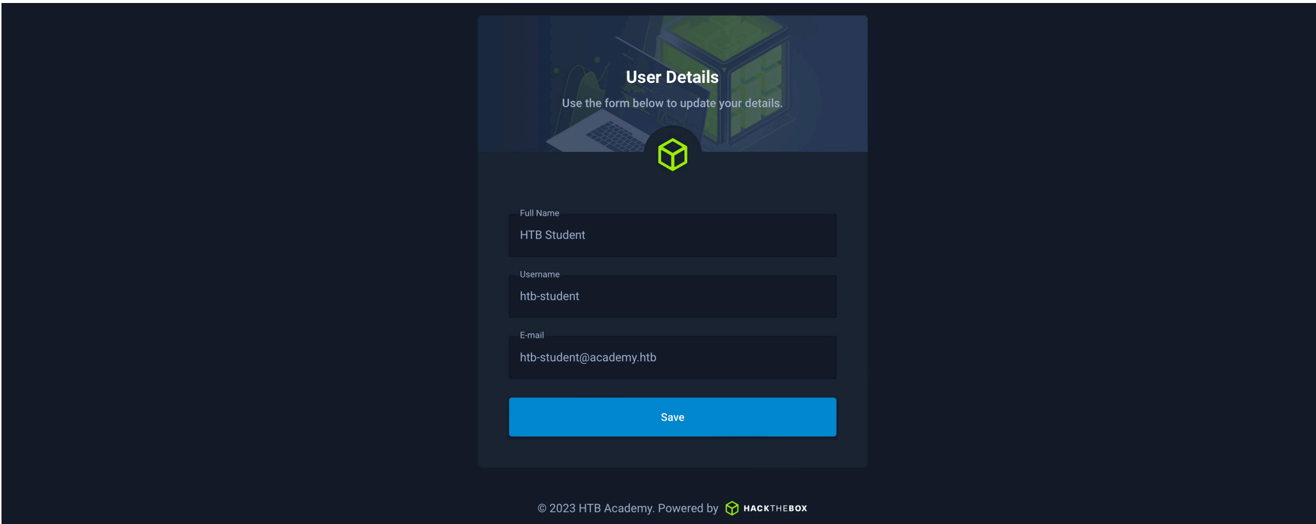
hide01.ir

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:

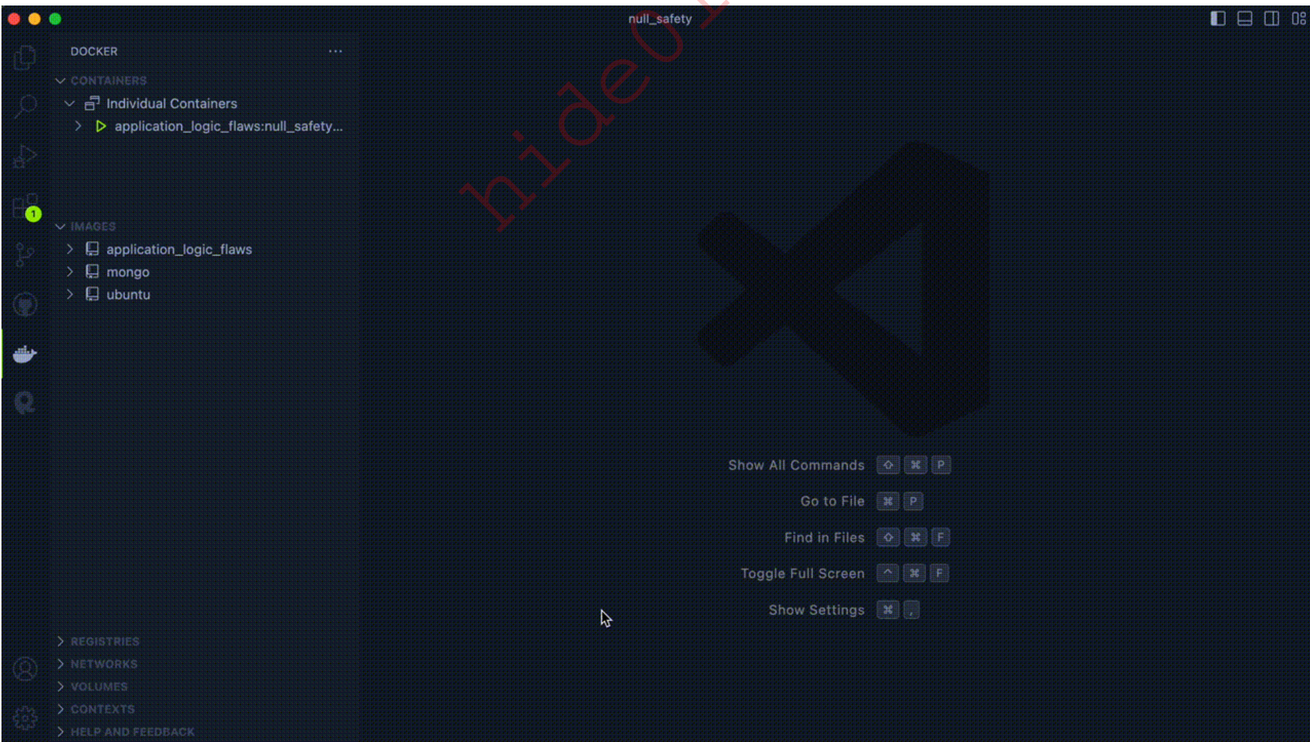


https://t.me/CyberFreeCourses

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.

hide01.ir

https://t.me/CyberFreeCourses

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`:

```
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:

hide01.ir

```
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`!

https://t.me/CyberFreeCourses

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`:

```
// 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>
  }
}
```

hide01.ir

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.

https://t.me/CyberFreeCourses

PoC and Patching - Null Safety

We have strong evidence of three different null issues, two of which should be exploitable by us. Of course, there can always be unforeseen protections in place that prevent us from exploiting these bugs, so in this section, we will try to confirm their exploitability with a proof of concept.

Proof of Concept

Let us start with the first logic bug we identified with the `token` parameter. This bug should enable us to reset the password of any user by simply knowing their `id`, which is not necessarily considered to be a secret value and may be revealed by API calls or even in their profile URL.

Account Takeover

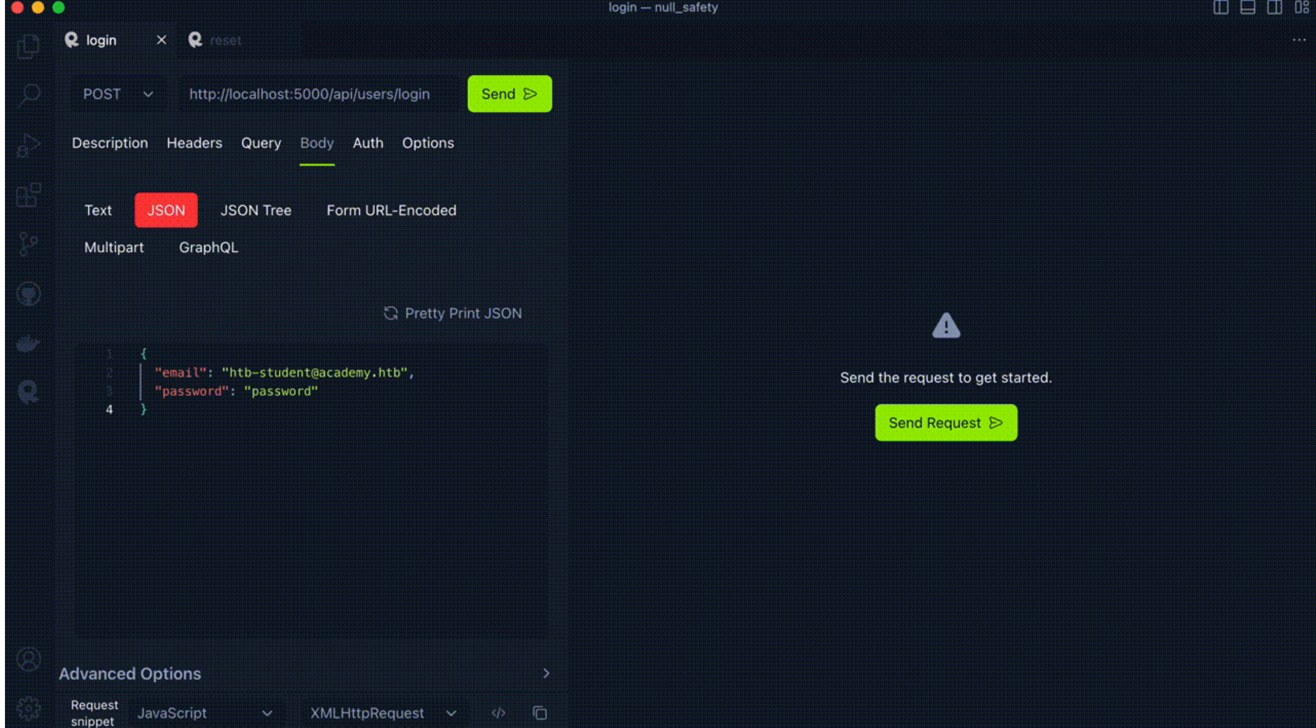
Let's target the local user we have, which has the id of `649f2893cba8d0d6e8412182`. To perform this attack, all we need to do is to send a `POST` request to the `/api/users/password/reset` endpoint, and skip adding the `token` parameter in our request's body:

```json
{
  "id": "649f2893cba8d0d6e8412182",
  "password": "123456"
}
```

hide01.ir

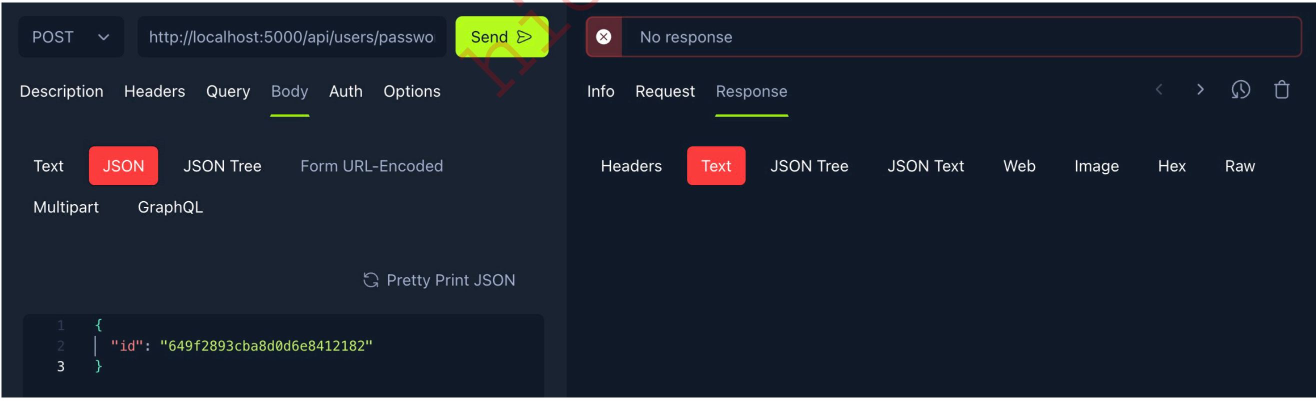
If we send request, we get `Password updated successfully!`. If we attempt to login, we can successfully get in, which means that our first attack is successful:

https://t.me/CyberFreeCourses



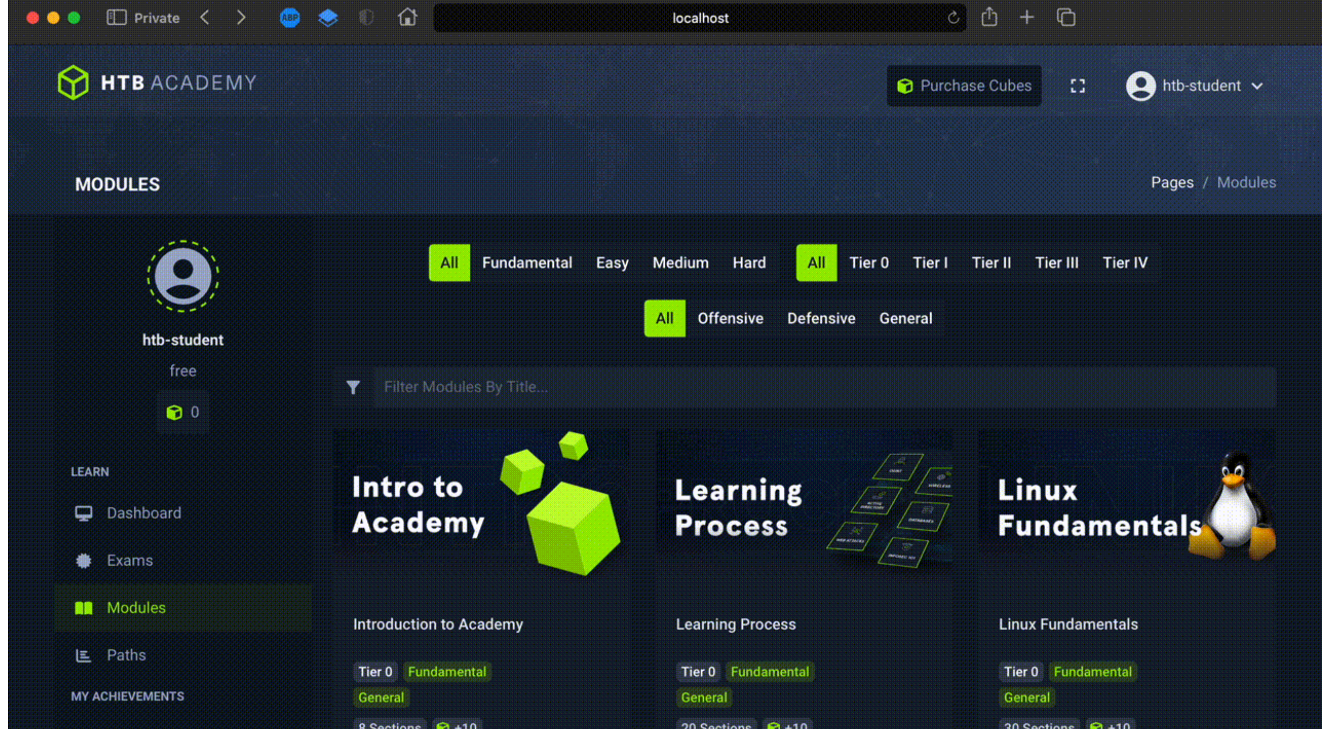
Denial of Service

Excellent! Moving on with the second attack, and this time, we will also remove the `password` parameter from the above payload, and we don't have to include the `token` parameter to skip the check as we did above (or you may calculate the real token and include it, if you're feeling adventurous). Now, we will send the request to see how the application handles it:



We simply get no response back from the server. To check if this has caused any errors, we can go to the `Docker` tab in VSCode, right-click on the running container, and select `View Logs`, and will see that the application appears to have completely crashed. Indeed, if we go back to the browser and refresh, we see that the web application is no longer running:

https://t.me/CyberFreeCourses



With a simple request, we have completely taken the server down, and now no user would be able to access it, until it is manually restarted. Both of these examples show how serious null safety issues can be, and why we should take them seriously.

Note: We have to be very careful when testing our proof of concept when it comes to denial of service attacks, as we should always avoid disrupting the production services or permanently modifying any production data.

Patching

In this code, we have identified multiple issues, and even though we are not able to take advantage of all of them, we still need to apply patches for them all.

updateUserDetails

The first issue was in the `updateUserDetails` endpoint, and the root cause was in the `UserSchema`. We cannot simply require all parameters, as some endpoints need the schema validation, and do not need all parameters (e.g. login only needs `email` / `password`). At the same time, we cannot leave these parameters as optional for all cases.

So, the best solution would be to use `conditional validation`, and specify the cases in which we need these parameters to be `required`. For example, if we need the `username` parameter in `UserSchema` to only be `required` with the `createUser` and the `updateUserDetails` endpoints, and keep it as optional with the rest (as it is not being used there), then we can modify it as follows:

```
export const UserSchema = yup
  .object({
    useUsername: yup.boolean().default(false),
```

hide01.ir
https://t.me/CyberFreeCourses

```
    id: yup.string(),
    name: yup.string(),
    username: yup
      .string()
      .when("useUsername", ([useUsername], schema) =>
        useUsername ? yup.string().required() : yup.string()
      ),
    email: yup.string().email().required(),
    password: yup.string().min(5).required(),
    registrationDate: yup.date(),
  })
  .required();
```

As we can see, we added a new boolean parameter `useUsername` with a default value of `false`, and if this parameter is set to true, it will trigger the `when` option to make the `user` parameter `.required()`. When we use the schema for validation and need to require the `username` parameter, we can do so as follows:

```
await UserSchema.validate({
  useUsername: true,
  username,
  email,
  password,
});
```

hide01.ir

Since `useUsername` is set to true, then this would mean that the `username` parameter is required, and if a null value is passed into it, it will return an error. Doing optional parameter checks this way is much more secure and much more maintainable than scattering them throughout the endpoints, which may introduce issues in any of the endpoints.

If we were using the same schema solution with TypeScript, then all of that would be detected automatically. In that case, if we misuse it, like setting `useUsername: true` but there's a case where `username` would be `null`, then it would notify us during edit-time and prevent the compilation of the code.

resetPassword

As for the `resetPassword` endpoint, we had multiple issues that we need to rectify. First, we need to ensure that all of the parameters are required, as they are only used once (in this endpoint), and all of them are needed, so there's no point in keeping any of them as optional. Keep in mind that requiring the entire object doesn't mean all of its parameters are also required, as we have shown in the previous section. So, we can fix the `passwordResetSchema` schema as follows:

https://t.me/CyberFreeCourses

```
const passwordResetSchema = object({
  // validate mongodb object id
  id: mixed((value) => ObjectId.isValid(value))
    .typeError("Invalid id")
    .required(),
  // validate bcrypt hash
  token: string()
    .matches(/[0-9a-f]{32}/i, "Invalid token")
    .required(),
  // validate password
  password: string().min(5).required(),
}).required();
```

Once this is done, we would know that the `token` will not be null during the execution of the code, so we will not have to test for that when comparing it to `hashedToken`. So, we can simply do:

```
if (token !== hashedToken) {
  return next({
    message: "Invalid password reset token.",
    statusCode: 403,
  });
}
```

Hide01.ir

Furthermore, as the function already uses `strict` inequality (`!==`), then this would also check for null and undefined, as it would not be equal to the type of `hashedToken`, which is a `string`. We need to rectify the `default to success` logic bug that we noticed, even if we know that the `token` variable would never be null at that point, we need to have it patched as a matter of principle. Ideally, we should `only continue if the tokens do match`, and fail otherwise. This may look something like this:

```
if (token === hashedToken) {
  try {
    const salt = await bcrypt.genSalt();
    const newHashPassword = await bcrypt.hash(password, salt);
    // update user password
    <SNIP>
  }
  <SNIP>
} else {
  return next({
    message: "Invalid password reset token.",
    statusCode: 403,
  });
```

https://t.me/CyberFreeCourses

```
}
```

Finally, we must always wrap any lines that use optional/nullable parameters with a `try/catch` block, like the lines that calculate the new `password` hash, which caused a DoS, as we have seen in the previous section:

```
else {
  try {
    const salt = await bcrypt.genSalt();
    const newHashPassword = await bcrypt.hash(password, salt);
    // update user password
    <SNIP>
  }
}
```

Of course, with the updated schema, we know that the `password` parameter would never be null, but this is done as a demonstration of how we should do it in that case. With that, we should have `patched all of the parameter logic bugs` found within our web application. Or did we? `Try to see if you can find any others`, in addition to whatever the exercises are testing.

Exercise: It is very important to test our patches locally before deploying them to production. So, try to apply the above patches to their respective functions/files, then run the application again to ensure it will function correctly. You should test every function we patched, to ensure that it both functions as expected under normal conditions, and is also no longer vulnerable to the above bugs.

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

hide01.ir

https://t.me/CyberFreeCourses

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.

hide01.ir

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 for each function in the application, as they help greatly reduce and uncover potential type/null safety issues. They

https://t.me/CyberFreeCourses

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](), 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.

hide01.ir

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.

https://t.me/CyberFreeCourses

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.

Skill Assessment - Parameter Logic Bugs

Scenario

Your team has been contracted by Hack The Box to review the back-end code of Academy. Your team leader has assigned you the task of identifying various logic bugs that are directly caused by user input and other parameters.

Download the code below, and start applying what you learned throughout the module to identify as many logic bugs as possible. Make sure to test everything locally before testing it on the target. Your main goal is to be able to unlock as many modules and exams as possible, and try to obtain their content.

Tip: Once you can unlock any module you choose, we recommend writing a script to obtain all of its sections' contents (instead of doing so manually). Creating the script will also be helpful as a final Proof of Concept for your team leader.

Extra Challenge: After obtaining the flag, try to patch all vulnerabilities you identified on your local environment. Once that's done, try to run the PoC again to ensure it no longer works, while ensuring that everything else functions normally.

hide01.ir

https://t.me/CyberFreeCourses