# Local Testing

With a prioritized shortlist of potentially vulnerable functions at hand, our next step is to confirm or deny the existence of these vulnerabilities through testing and local debugging. The testing approach we will follow is very similar to what a developer would do to debug an issue found within their application, as vulnerabilities are essentially 'issues' in the code.

We would use multiple techniques to understand better how input and output flow in each function we test, which should be enough to confirm whether the vulnerability exists. Finally, we can explore exploiting this vulnerability to showcase its impact and danger. In many cases, we may identify a serious issue, but it may not be exploitable due to the application's design, which reduces its probability of damage and thus reduces its overall risk.

## Backend Replication

So far, all of our steps have consisted of static analysis, that is to say, without actually running any application functions, and we only relied on analyzing the code. But as we start the dynamic analysis here, we first need to ensure we have a test server that closely resembles the production backend server, as mentioned previously.

If the team provides us with a replica test server, we can skip this step as we've already completed it. However, in most Academy modules, we often try to set up our test servers, which is a critical skill to master.

The difficulty of this step mostly depends on the availability of information about the target. If the team provided us with all the assets and tools they use in their production server, then it is only a matter of setting everything up. But in some cases, some teams may not have a document that details all of this knowledge, as they may have gradually set up their production server without documenting everything. If this were the case, we would need to install all requirements manually so the application is fully running without issues, which is often easier said than done, especially without a complete list of requirements.

For example, if we were testing a NodeJS application, all of its requirements would likely be in a packages.json file, which saves us a lot of time. Then, we only need to focus on setting up a database and any external connections the application may utilize (e.g. to the front-end). In some other development platforms, we may not have this info easily relayed to us. We will likely need to examine the entire code base to identify dependencies, install all of them, and then test whether the application runs correctly. This can be very time-consuming, which is another reason a team with poor documentation would be charged more for a whitebox exercise, as we previously mentioned.

**Note:** The actual process of replicating an unknown production server is outside the scope of this module, as we will be assuming that the client's team will provide us with those. However, this is covered in other whitebox modules on HackTheBox Academy and can generally be closely replicated through basic server/web applications foot-printing steps.

## Testing

We can start our testing process once we have our test server and application up and running. The type of tests we perform depends on the vulnerability we target. In general, we are mainly looking to achieve two things:

1. Trigger the target function.
2. Control how our input reaches/affects the target function.

We will usually need to understand how we can land in the target function, as it may not always be directly accessible and only be triggered under certain circumstances. If this were the case, we would need to do various tests and local debugging until we know how and why our input would trigger the target function.

We will also likely need to trace our input throughout the application, from the front-end to the vulnerable function, and then monitor how our input changes at every stage until it reaches the target function.

Once we have complete control over how to trigger the function and the input reaching it, we should be able to decide decisively whether the function is vulnerable. If it is, we can move to exploitation.

## Exploitation

Our final step of the testing phase is to achieve a basic exploitation of the target function. This does not need to be a fully operational exploit (as we'll get to that in PoC), but we need to confirm what our testing has shown and achieve exploitation. For example, if we were targeting a command injection vulnerability, we would need to execute a command that can safely be assumed to work under most conditions (e.g. touch or ping). If we were targeting an SQL injection vulnerability, we would try to execute a specific query in the database and check the logs to ensure we did execute the intended query.

We do not need to overcome security filters (like WAF) or bypass limitations (like blind exploitation), as we only need a fundamental confirmation that the function is indeed vulnerable and can be exploited. Once we have this confirmation, we will attempt to bypass all difficulties during the PoC step and write a proper exploit. But if we do not confirm exploitation at this stage, we may waste many hours trying to bypass or overcome something only to realize later that the function is not vulnerable. This is why this step is vital.