# Code review

The first and most crucial step of a whitebox pentest is reviewing the application's source code to understand its design and functionality. As mentioned in the previous section, this step is quite advanced as it requires programming knowledge to understand the application functionality. At the same time, this step requires the ability to read multiple programming languages and have knowledge about their potential flaws. Furthermore, the large code base makes the size of the attack surface huge, making it even more challenging to identify and prioritize interesting functions.

Of course, in addition to all of this, there is the primary skill of understanding and identifying advanced web vulnerabilities in a code base. Doing so can give pentesters and bounty hunters the edge, even if their technical knowledge is not as advanced as others.

Let's start seeing the multiple phases we go through to review code and identify potential vulnerabilities correctly.

**Note:** In a whitebox pentesting exercise, the code review step will consist of identifying interesting functions and then testing them one by one without covering the entire code base. On the other hand, in a secure coding exercise, we usually need to cover the entire code base on a rolling basis to ensure it is securely coded. Both exercises share common steps, and only the depth of the test may be different.

## Planning and Data Gathering

Before we start going through the code, we need to have a few meetings with the development team and other stakeholders to get a thorough understanding of the application design, as well as collect all necessary assets for our exercise, like the source code, test server, and other assets mentioned in the previous section.

It is important to remember that all of our tests will be carried out on a test environment first, either a test server provided to us or a test server that we set up to resemble the production environment. In this module and other whitebox Academy modules, we will always have access to the source code as if it was provided to us by the development team.

Furthermore, we will mostly be setting up our test server, which is a critical skill to learn, though some modules will provide a pre-built test server to save us some time with more complicated applications. This will be further discussed in the next section.

## Scope Selection

If we were provided with application design documentation, we could study it to get a general idea of how the application works and each function's role. With huge code bases with millions of lines of code, we will not need to test the entire application, but likely on some of its functions, branches, or tools.

However, in many cases, the team may not have a design document, and we would need to personally reverse engineer the code base and try to understand what each function does. This is often more difficult if the team does not provide good documentation/comments within the code base (e.g. comments/steps/processes).

This is why the cost of performing a whitebox pentest depends not only on the size and scale of the project but also on how well it is designed and documented, as both of these things can make the exercises much longer and much more challenging to carry.

## Prioritization & Scope Reduction

Whether the developers provide design documents or not, we should be able to get a general idea of how the application works and what each function is generally responsible for. Once we do, we can start prioritizing functions by their sensitivity.

For example, functions that handle authentication are always a priority since they may lead to an authentication bypass. Likewise, functions that interact directly with the operating system, by executing commands or writing files, are always interesting to look at. The nature of our priorities may also depend on the type of application or even the type of vulnerabilities we are most skilled at. For example, if we were very good with SQL injection vulnerabilities, then we would prioritize functions that interact with the database.

There are multiple techniques we can utilize to identify interesting functions, and the following are some of them:

1. Select functions based on the application design. In this technique, we select functions solely based on our understanding of the application design, potentially even before looking at the source code. For example, we can understand that files under the /purchases directory deal with online payment handling, so we can prioritize these files in our test, and so on.
2. Select functions and files through search. This technique is much faster and quicker, though it is not as comprehensive and may lead to many missed opportunities. We can search for certain sensitive functions through the code base (e.g. with find/grep or text-based search). For example, if we were dealing with a PHP web application, we may consider searching for functions that execute system code, like exec, system, passthru, and others, as we will see later in the module. Then, we can study the use of each and look for any that do not correctly filter the user input. We often mention interesting functions for various programming languages in our web modules, as they can be utilized for such purposes.
3. Select functions through the use of the application. Unlike the first two techniques that only relied on static analysis, this technique utilizes a mix of dynamic analysis and static analysis by testing the application and examining its different pages and requests, and then prioritizing them based on what seems to be most sensitive or what may look the weakest. For example, if we notice a specific page that throws a lot of errors during the application usage, we may want to focus on this page, and so on.

These are only a few techniques that we suggest, though each of us can come up with different techniques that suit our style, which may utilize one or many of the above techniques as well. Furthermore, other whitebox pentesting modules in HackTheBox Academy may use some of these techniques or others, which would be another opportunity to learn different methods for code reviews.

**Exercise:** Throughout different modules on HackTheBox Academy, whenever you read through a whitebox attack scenario, try to see which of the above techniques were used to reduce the scope, or if a totally different technique was used.

## Reverse Engineering

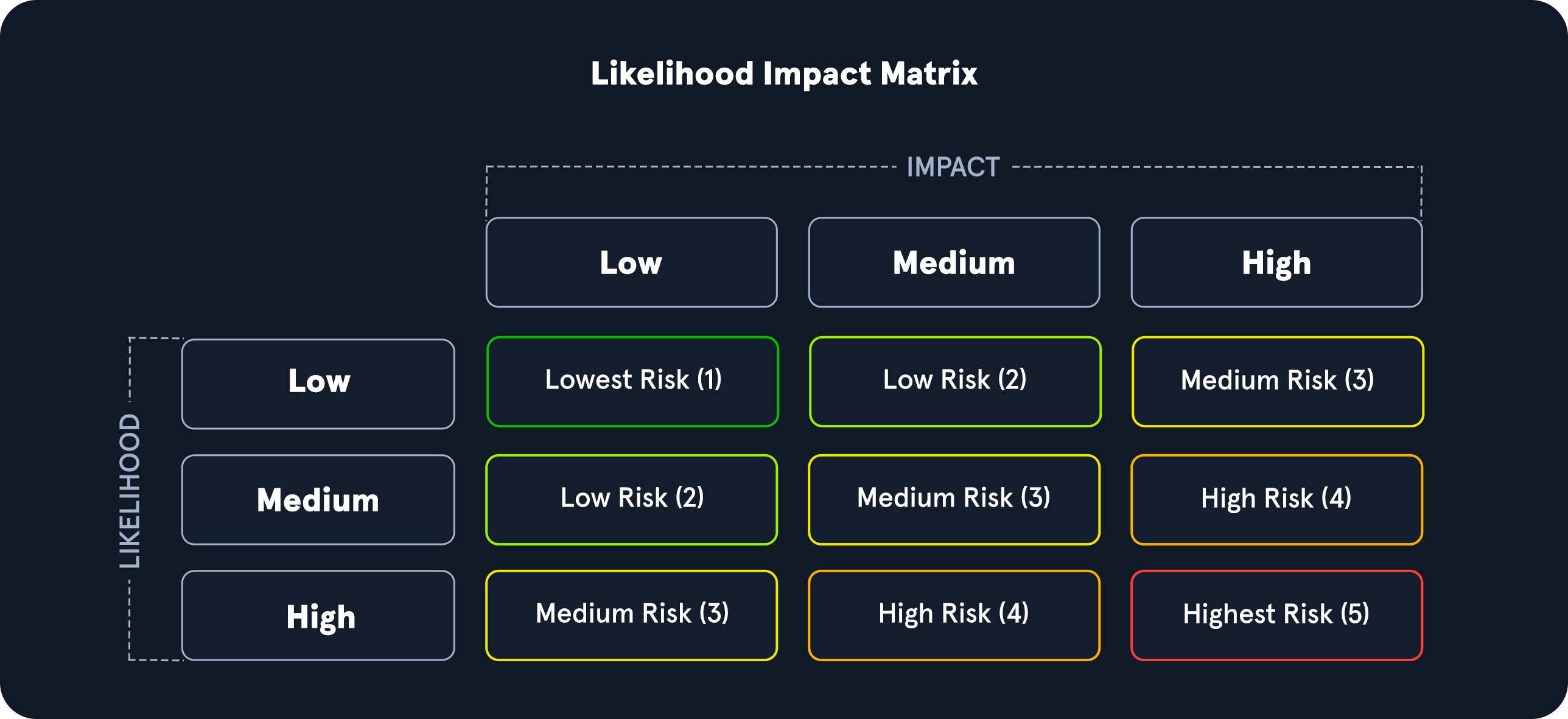
Once we have a prioritized list of functions and application files we want to test, we come to the code reading step of the Code Review. To determine whether a function is securely coded, we first need to fully understand how it works and how each input and output is traced and processed.

We can read the function line by line, adding comments as we go for what each line and parameter does. In most cases, we would not stick to a single file, but also need to trace each variable and external function to their sources in other files and review them. As we can see, this can be a very strenuous exercise without good documentation. Still, this is vital for whitebox pentesting, without which we may be unable to identify any advanced vulnerabilities.

This step is unique because it may reveal vulnerabilities that are impossible to identify through a blackbox pentest only. For example, suppose we identify a dangerous function that interacts with the system directly in an unsafe manner. If we stick to a single file, we may find it is not vulnerable as the input is very well filtered. However, reversing the different parts of this function may reveal that under some conditions, the input may not be filtered, or the function may have an additional source of input that does not get filtered properly (e.g. stored on the database). Without a good understanding of the function, we may overlook such issues, but through thorough reverse engineering and a good study of the source, we should increase our chances of catching such flaws.

## Prioritize Targets

After the above step, we may have several instances where we think a function is worth shortlisting. So, the last phase of Code Review is to prioritize our findings so that we can start testing them one by one. As for how to prioritize them, we can utilize the Impact X Probability risk matrix, where risks with high probability and high impact would have the highest priority, and so on.



For example, through reverse engineering, we may find functions that we think are very likely to be vulnerable based on our understanding of the code and our knowledge of application vulnerabilities. However, these vulnerabilities may be on a local-only page or low-impact vulnerabilities, so they would have a medium priority.

Likewise, we may find some functions that we think may be vulnerable under some conditions, but they may reside in a critical function within the application, so they would have a high priority.

Similarly, we can prioritize all of our findings based on their impact and probability and then start testing them individually in the next step of whitebox pentesting.