HABOOB

Java Deserialization in ViewState

By **Haboob Team**

Java Deserialization in ViewState

Table of Contents

1. Introduction

In this paper we are going to talk about a technique used to exploit Java web applications. This technique relies on some vulnerable Java libraries (such as apache "CommonCollections"), which deserializes any received object without performing any checks.

2. What is Serialization?

Serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (such as, across a network connection link) and reconstructed later (possibly in a different computer environment). It converts an in-memory data structure to a value that can be stored or transferred. This process of serializing an object is also called marshalling an object, and it is typically used when data must be moved between different parts of a computer program or from one program to another.

3. What is Deserialization?

Deserialization takes a series of bytes and converts it to an in-memory data structure that can be consumed programmatically. A serialized object which was used for communication cannot be processed by a computer program; therefore, deserialization transforms the object that was used for storage or transmission to a representation of the object that is executable.

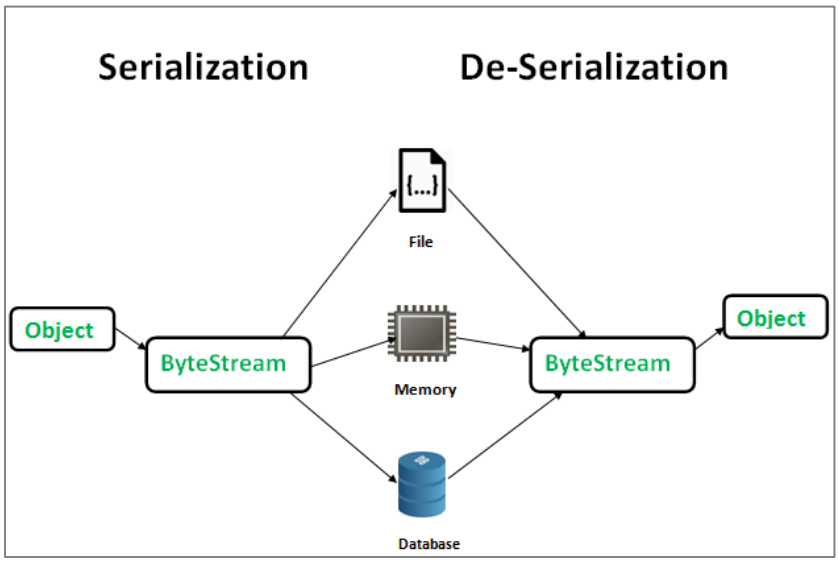

Figure 1: Serialization vs Deserialization

4. Vulnerability Analysis

Before digging any deeper, we need to understand what "ViewState" is. Without state management in JSF, typically all information associated with the page and the controls on the page would be lost with each round trip. The purpose of "ViewState" is to memorize the state of the user, even after numerous HTTP queries (stateless protocol). The objective is to store and restore results of users actions that impacted the user interface of a web page (choice within drop-down list, check-box selection, etc.). The "ViewState" of a page is, by default, stored in a hidden form field in the web page named "javax.faces.ViewState". It works by either saving the state in memory of the server and binds it to the session, or serialize/deserialize the state in the request/response each time. There are some Java libraries that accepts any serialized object without performing any checks. This allows leveraging these libraries to execute an arbitrary code.

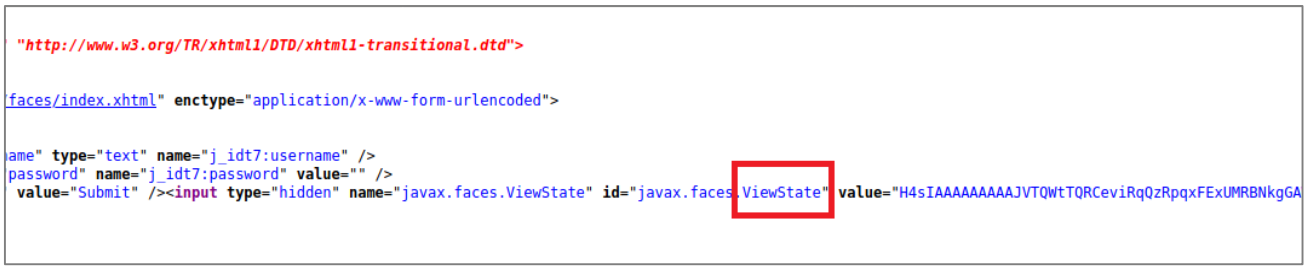

Figure 2: "ViewState" field in the page source of a web application

5. ViewState Encodings

By default, "ViewState" data is stored in the page in a hidden field and is encoded using base64 encoding. "ViewState" can also be encoded as "base64 and gzip" (Base64Gzip), which starts with "H4sIAAA". Below is a demonstration of a "ViewState" decoding process in Burp Suite.

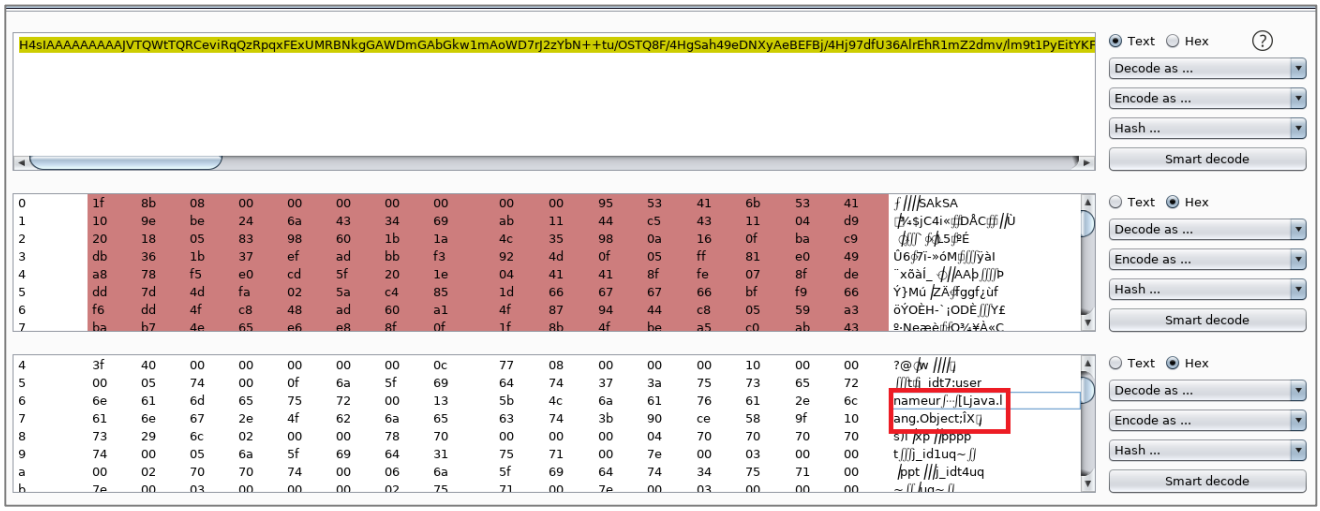

Figure 3: ViewState decoding

The “ViewState” string in the first box was first decoded as Base64 (second box), and then decoded as Gzip as shown in the last box. The decoding process showed the readable text “java.lang.object” and other Java related information.

6. Attack Requirements

There are three requirements for this attack: Burp Suite, “Ysoserial”, and a vulnerable JSF web application. Other extensions were used such as Burp collaborator and Java Deserialization Scanner, which are both available on the professional version of Burp Suite. The Java Deserialization Scanner extension can also be downloaded from GitHub.

7. Attack Demonstration

In this section, we are going to demonstrate the attack and how to identify a blind RCE to get a fully interactive shell. We used a tool called “Ysoserial”, which is a collection of utilities and property-oriented programming "gadget chains" discovered in common java libraries that can, under the right conditions, exploit Java applications performing unsafe deserialization of objects.

In our demonstration, we first installed an extension in Burp Suite called “Java Deserialization Scanner”. Java Deserialization Scanner is a Burp Suite plugin, which generates customs payloads aimed at detecting and exploiting Java deserialization vulnerabilities.

Then, we intercepted the request and sent it to the extension's dashboard by clicking "Send to DS - Manual Testing".

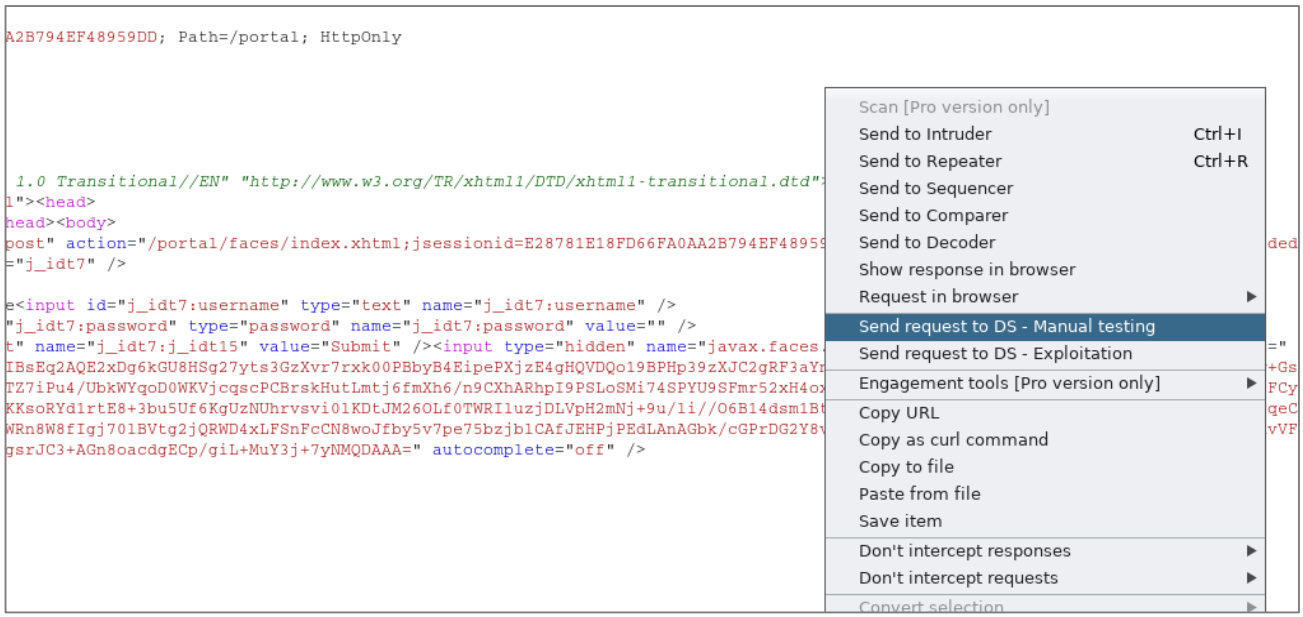

Figure 4: Request sent to extension for testing

As we can see in figure 5, the "ViewState" starts with "H4sIAAAA", which represent the Base64Gzip encoding format. Then, we selected the ViewState value to test it for deserialization by choose "Base64Gzip" attack.

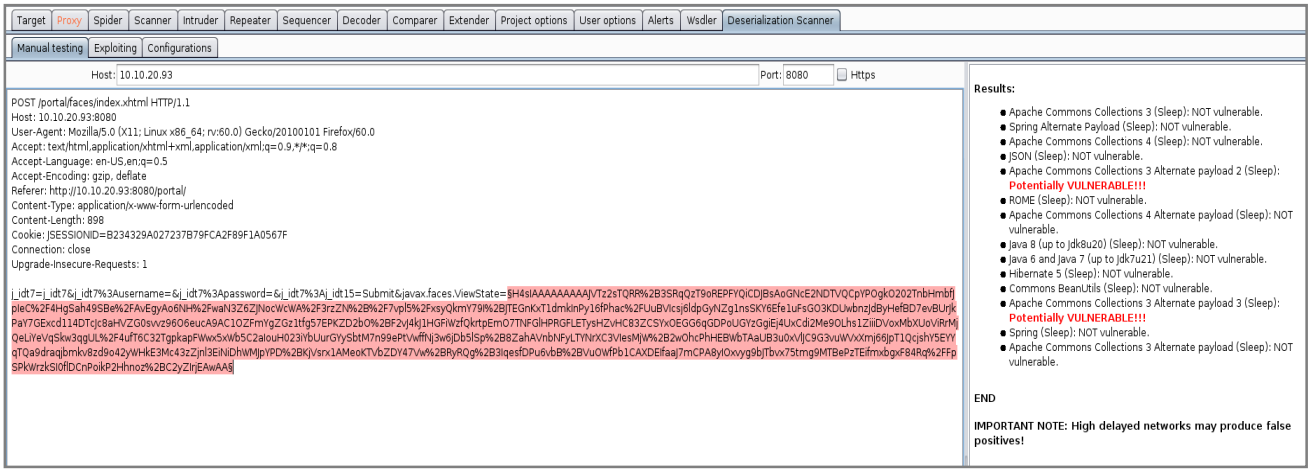

Figure 5: Scanning result

The scan results show that the ViewState parameter is vulnerable to deserialization via "CommonsCollection" library. Next step in our exploitation process is sending the request to the exploitation tap as shown in figure 6

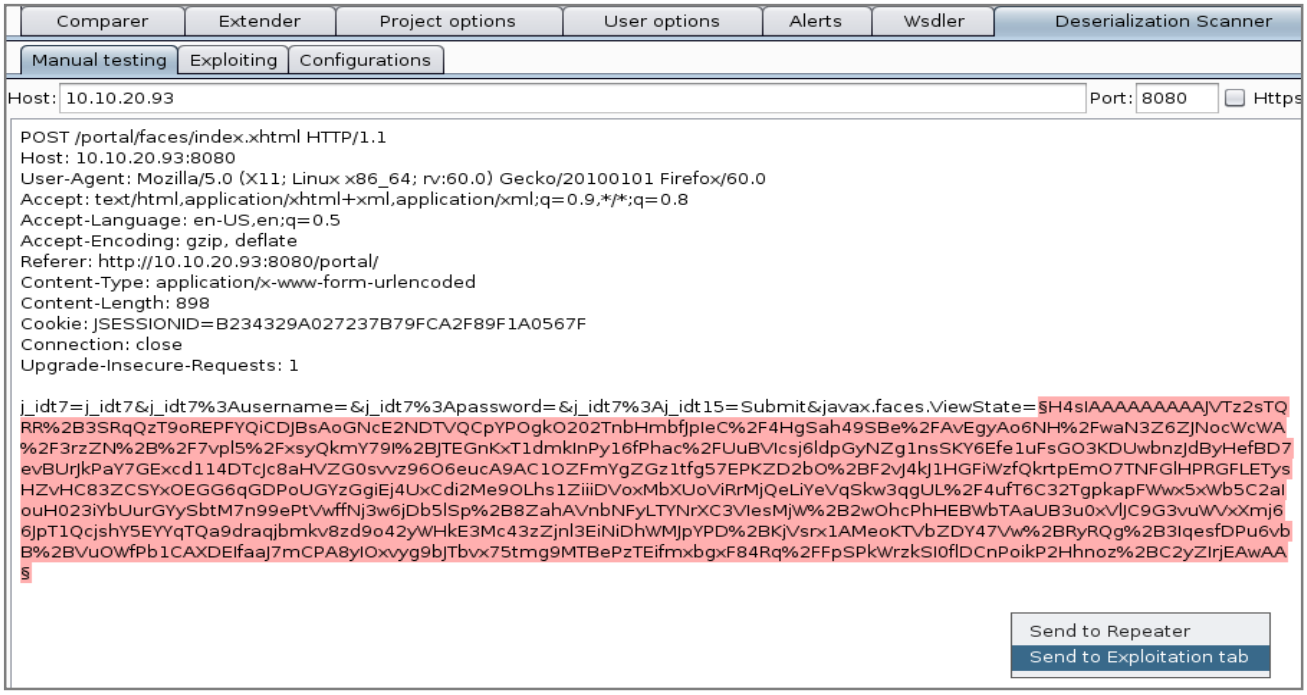

Figure 6: Exploitation tap

Since this is a blind RCE, we need a way to determine that our exploitation is working. For this reason, we used Burp Collaborator to generate DNS traffic, shown below in figure 7.

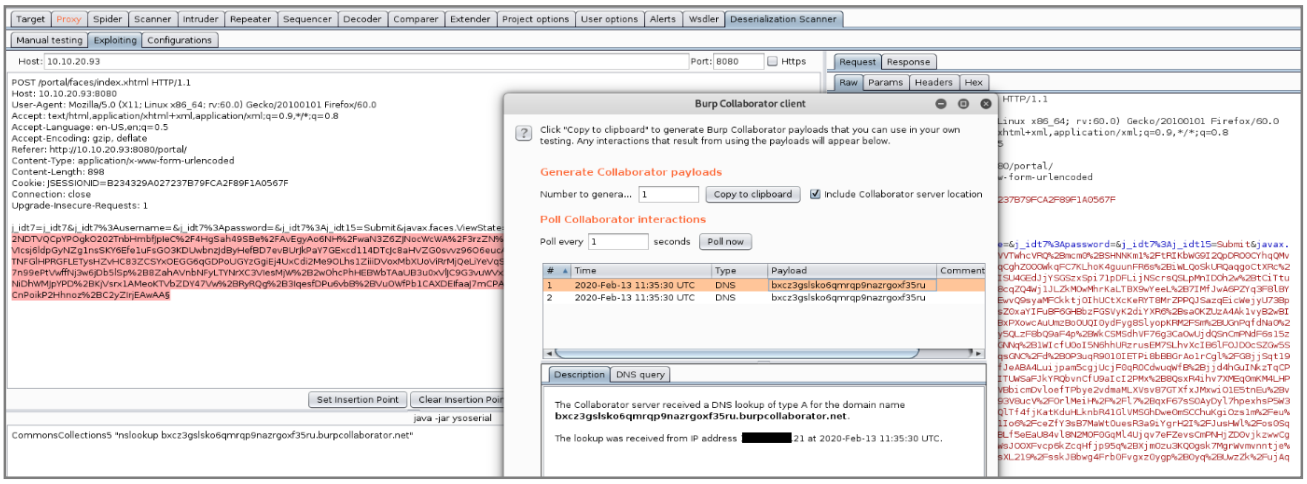

Figure 7: DNS lookup

Another way of identifying a blind RCE is by pinging our host and monitoring Wireshark for ICMP packets as demonstrated in figure 8.

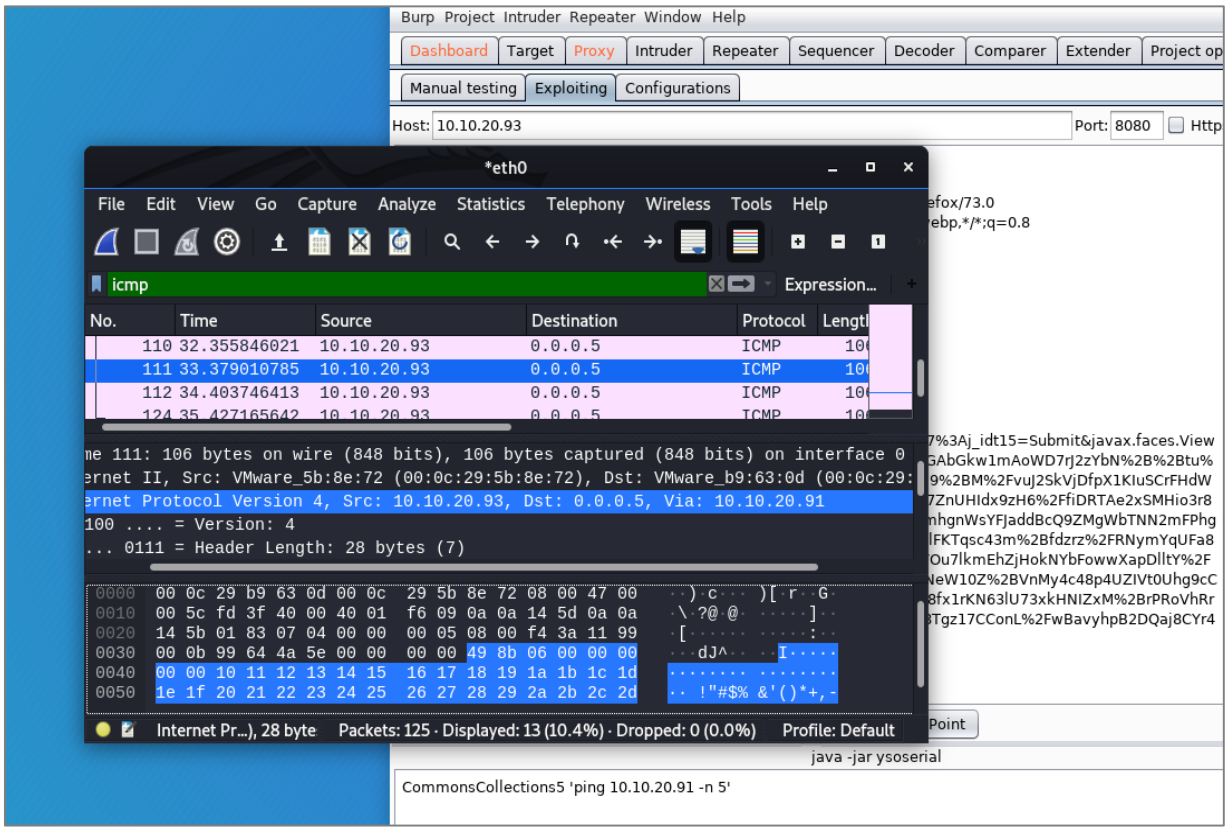

Figure 8: ICMP requests

Finally, we prepared a listener on our machine to get a fully interactive reverse shell on the vulnerable web application, shown in figure 9.

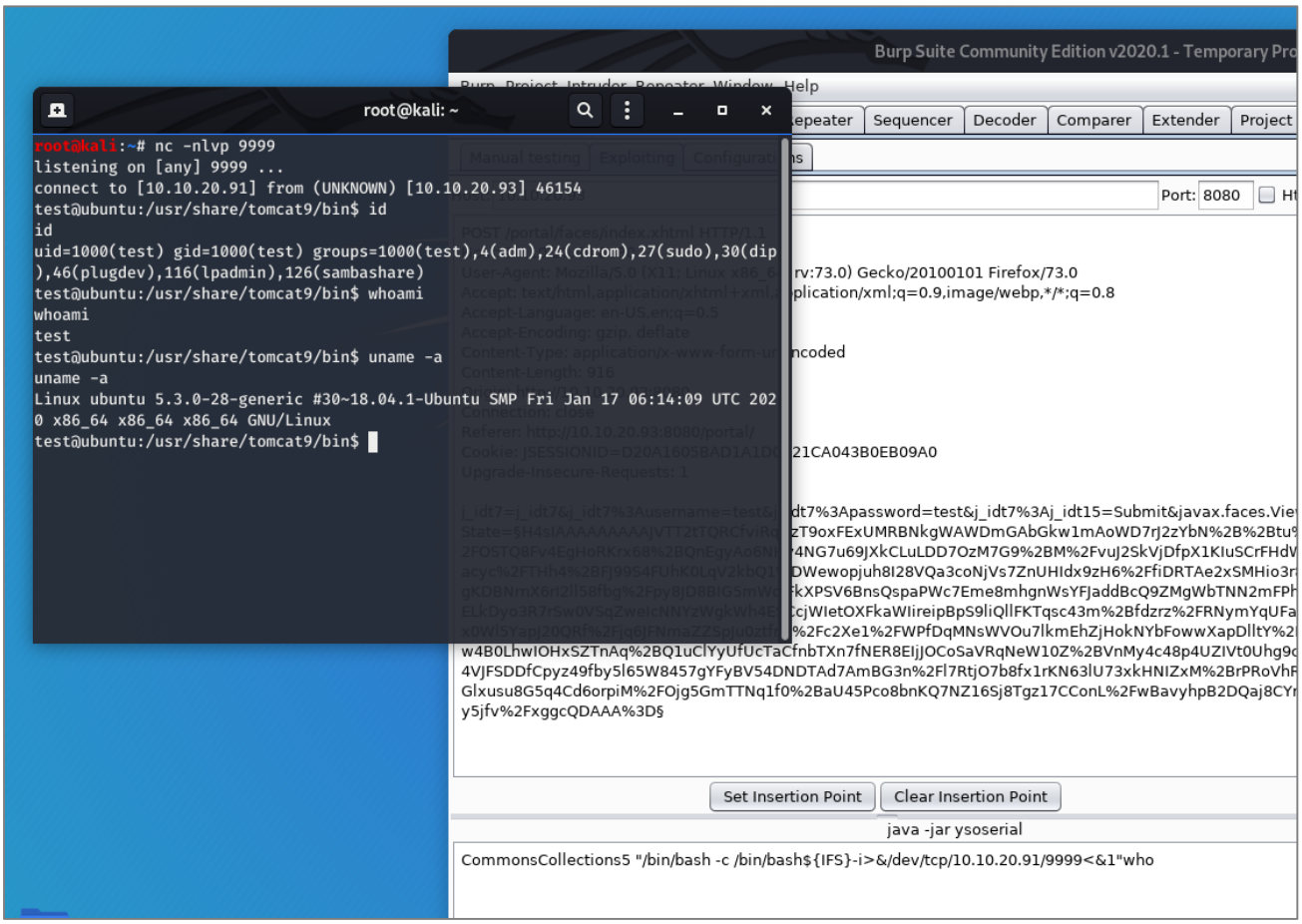

Figure 9: Interactive shell

8. References

[1] https://owasp.org/www-project-cheat-sheets/cheatsheets/Deserialization_Cheat_Sheet

[2] https://github.com/frohoff/ysoserial

[3] https://medium.com/better-programming/serialization-and-deserialization-ba12fc3fbe23

[4] https://en.wikipedia.org/wiki/Unmarshalling

[5] https://en.wikipedia.org/wiki/Serialization

[6] https://medium.com/@mirzafarrukh13/state-management-785f375521a6

[7] https://github.com/federicodotta/Java-Deserialization-Scanner

[8] https://www.synacktiv.com/ressources/JSF_ViewState_InYourFace.pdf

[9] https://www.geeksforgeeks.org/serialization-in-java/