[Advertiser Disclosure](#)

developer.com

Custom Search

- Java
- Microsoft & .NET
- Mobile
- Android
- Open Source
- Cloud
- Database
- Architecture
- Other
  - Project Management
  - PHP
  - Perl
  - Ruby
  - Services
  - Other Languages
  - White papers
  - Research Center
- NEW: Slideshows
- Sponsored

January 4, 2020
Hot Topics:
prev

- Android
- Java
- PHP
- Microsoft & .NET
- Cloud
- Open Source
- Database

next

Advertisement

Developer.com
Java
Data & Java
Read More in Data & Java »

How to Bypass Accessibility Checks Through Reflection in Java

- June 1, 2015
- By Manoj Debnath
- Send Email »
- More Articles »

Tweet

Programming in reflection is sometimes like playing with the privacy of Java internals and getting into the diversion of exploiting secure Java code. We do not get into the details of whether to keep the loopholes for exploitation alive (not likely) or an intentional convulsion of security breaches (nope, rather, an open kitchen, where you can extend your ideas to work) within JVM. The main idea is that this feature provides some value and really appreciates those peepholes so that we can create our own recipe rather than eulogize a bullet proof black box of language paradigm. Let's dive into another perspective of complexity in reflection.

A Little Diversion to What We Know

In a Java class, private fields are inaccessible except to members; only public fields are visible through its instances. And protected fields? Your class must be a part of the cult or, more technically, extend the class to access them. In all sobrieties, we are taught this accessibility mechanism in Java. Now, this reflection dropped from nowhere and says that we can access even non-accessible fields, methods, and constructors of a class (psst… only if the security manager permits you to do so). Intriguing, isn't it! What we need is to get hold of the reference of the field, method, or a constructor using the APIs provided by the *Class* object and defile it to any un-holistic end. Great! Now, we can put our evil minds to work to desecrate and ransack the chastity of Java classes. Need proof? Here it is:

```
package org.reflection.demo;

import java.io.Serializable;
import java.util.Date;

public class Employee implements
      Serializable, Cloneable{
   private static final long serialVersionUID = 1L;
   private String name=null;
   private Date joinDate=null;

   public Employee() {super();}
   public Employee(String name, Date joinDate) {
      super();
      this.name = name;
      this.joinDate = joinDate;
   }
   public String getName() { return name;}
   public void setName(String name)
      { this.name = name; }
   public Date getJoinDate() { return joinDate; }
   public void setJoinDate(Date joinDate)
      {this.joinDate = joinDate; }
}

package org.reflection.demo;

import java.lang.reflect.Field;
import java.util.Date;

public class AccessibilityCheck {
public static void main(String[] args) {
   Class<Employee> c = Employee.class;
   try {
      Employee e = c.newInstance();
      Field name = c.getDeclaredField("name");
      Field doj = c.getDeclaredField("joinDate");
      name.setAccessible(true);
      doj.setAccessible(true);
      System.out.println("Value of name: "+ name.get(e));
      System.out.println("Value of joinDate: "
         + name.get(e));
      name.set(e, "Harry");
      doj.set(e, new Date());
      System.out.println("Changed value of name: "
         + name.get(e));
      System.out.println("Changed value of joinDate: "
         + doj.get(e));

   } catch (InstantiationException | NoSuchFieldException
      | SecurityException | IllegalAccessException ex) {
         System.out.println(ex.getMessage());
      }
   }
}
```

Output

```
Value of name: null
Value of joinDate: null
Changed value of name: Harry
Changed value of joinDate:
   Fri May 01 15:23:06 IST 2015
```

- Email Article
- Print Article

Observe that we are using the *get()* method to get a reference of the fields typified with private, protected, and public accessibility in the *Employee* class. Similarly, we can retrieve methods and constructor references. But, the catch here is the *Field*, *Method*, and *Constructor* classes are derived from the *AccessibleObject* class. This class stands guard as a default access control and checks when it is used. We can use the *setAccessible(boolean flag)* method and call it explicitly on a field, method, and constructor reference with a *true* argument to make them accessible to our program. In case, we do not call the *setAccessible(boolean flag)* method or provide a *false* value to the flag while accessing a private member of a class; our program will throw an exception. Public and protected members do not care what *true*/*false* value we post in the flag; they are simply accessible.

Security Manager

Security manager keeps our evil minds at bay. The invasion we have been planning to sabotage the puny outpost of the *setAccessible(boolean flag)* method with the militia of our rudimentary intelligence ebbed out because the path to fluid access to otherwise inaccessible class members goes through the barricade of Java Security Manager. The hull is further protected by Java Security policy norms. But, how come we never mentioned anything about security manager in the preceding code and still are able to access private members of *Employee*? The reason is that security manager is not installed in an application by default when we run our program. It is due to its absence we could access private members of the *Employee* class.

If, however, the security manager is installed, the accessibility is restricted and depends on the mercy of the manager. If permission is granted, our application can access; otherwise, it cannot access. We can easily check if security manager is installed or not, as follows:

```
SecurityManager securityManager=System.getSecurityManager();
   if(securityManager==null)
      System.out.println("security manager is not installed");
```

We can, however, install the default security manager while running the application by issuing the following command line argument:

```
C:/>  java -Djava.security.manager <java_file>
```

The application then uses the default security policy. In case we want to provide our own policy, we can do it in a policy file written as:

```
grant {
   permission java.lang.reflect.ReflectPermission
      "suppressAccessChecks";
};
```

Ref: The following table is the ReflectPermission summary extracted from the Java 8u40 docs.

Permission Target Name	What the Permission Allows	Risks of Allowing this Permission
suppressAccessChecks	Ability to suppress the standard Java language access checks on fields and methods in a class; allows access not only to public members but also allows access to default (package) access, protected, and private members.	This is dangerous in that information (possibly confidential) and methods normally unavailable would be accessible to malicious code.
newProxyInPackage.{package name}	Ability to create a proxy instance in the specified package of which the non-public interfaces that the proxy class implements.	This gives code access to classes in packages to which it normally does not have access and the dynamic proxy class is in the system protection domain. Malicious code may use these classes to help in its attempt to compromise security in the system.

And save it in a file named, say, `permission_grant.policy`.

Now, to use our custom policy, we can run the application with the following command line option:

```
java -Djava.security.manager -Djava.security.policy=
   c:/ permission_grant.policy <java_file>
```

Once we have provided the policy file we can check it in Java code as follows:

```
try {
   ReflectPermission permission =
      new ReflectPermission("suppressAccessChecks");
   permission.checkGuard(null);
   System.out.println("Permission granted");
} catch (SecurityException e) {
   System.out.println("Permission denied");
}
```

Conclusion

Without denigrating its need, in my humble opinion security is a farce; it is always the cat and mouse game that matters. Perhaps the most secure code is the code that you have never written and the most secure language is the language never invented. In Java, there are security manager and security policies to shut the loud mouth of security fanatics who, in many ways, want to peek in for those who dwell among the un-trodden. Perhaps the perspective of reflection is openness, a type of nothing to hide gesture. Exploitation depends on the conviction of the programmer.