Software Assurance Tips

A product of the Software Assurance Tips Team[3]

Chris Ramsey

Monday 30th August, 2021

1 A CWE-499 Breakdown: Serializing Sensitive Data

Updated Wednesday 11th February, 2026

Recently, we detected a potential CWE-499 on some of the source being scanned in the lab. It raised some questions and debate among us, and left me scratching my head. Being a curious fellow, I took it upon myself to play around with it to try to figure it out. I'm not a Java developer by any stretch of the imagination, but I am a certified security professional with a background in software development. So, with a fresh install of Eclipse, I went to work.

1.1 The CWE

Below, you can see the summary and description of this CWE:

> **CWE-499: Serializable Class Containing Sensitive Data**
>
> **Description**
> The code contains a class with sensitive data, but the class does not explicitly deny serialization. The data can be accessed by serializing the class through another class.
>
> **Extended Description**
> Serializable classes are effectively open classes since data cannot be hidden in them. Classes that do not explicitly deny serialization can be serialized by any other class, which can then in turn use the data stored inside it.
>
> **Applicable Languages**
> Java (Undetermined Prevalence)
>
> **Demonstrative Example**

```
class PatientRecord {
        private String name;
        private String socialSecurityNum;
        public Patient(String name,String ssn) {
                this.SetName(name);
                this.SetSocialSecurityNumber(ssn);
        }
}
```

Listing 1: Example 1

> [2]

Did I read that right? Sensitive data contained within a class that doesn't explicitly deny serialization can be accessed by serializing the class through another class? This, I've got to see.

1.2 ClassA—Sensitive Data Container

So, let's start with a simple class that contains some sensitive data. Duplicating the example from the CWE should suffice.

```
package swa;
public class ClassA {
        private String name;
        private String socialSecurityNum;
        public ClassA(String name,String ssn) {
                this.SetName(name);
                this.SetSocialSecurityNumber(ssn);
        }
}
```

Listing 2: ClassA

Given this is meant to be a container of sensitive data, I didn't make the class in Listing 2 serializable. Let's create a serializable wrapper around it and see if we can get at the data.

1.3 ClassB—Serializable Wrapper

```
package swa;
import java.io.Serializable;
public class ClassB implements Serializable {
        private static final long serialVersionUID = 1L;
        ClassA obj;
        public ClassB(ClassA objVal) {
                this.obj = objVal;
        }
}
```

Listing 3: ClassB

Listing 3 demonstrates a serializable wrapper class around the class with sensitive information.

1.4 The Driver

With our two classes written, it's time for the show. Let's create a program that creates an instance of the sensitive data container, set it on our wrapper, and then serialize the wrapper to file.

```
package swa;
import java.io.*;
public class SerializeExample {
       public static void main(String[] args) {
               ClassA objA = new ClassA("swa","123456789");
               ClassB objB = new ClassB(objA);
               try {
                       FileOutputStream fos = new FileOutputStream("/home/swa/data.ser");
                       ObjectOutputStream oos = new ObjectOutputStream(fos);
                       oos.writeObject(objB);
                       oos.close();
                       fos.close();
               } catch (IOException i) {
                       i.printStackTrace();
               }
       }
}
```

Listing 4: Serialization Driver

Time for the main event! What happens when I run this?

```
java.io.NotSerializableException: swa.ClassA
          at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1192)
          at java.base/java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1577)
          at java.base/java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1534)
          at java.base/java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1443)
          at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1186)
          at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:352)
```

```
        at swa/swa.SerializeExample.main(SerializeExample.java:14)
```

Listing 5: Driver Output

Alas, no joy. Even though `ClassA` contains sensitive data and does not explicitly deny serialization, I couldn't get at the sensitive data by serializing a wrapper. That's a bit of a bummer–I was looking forward to being able to trumpet this one from the rooftops.

1.5 What Gives?

So I took my disappointment and went to Google, trying to make sense of this mysterious CWE. The first thing I found was this tidbit:

> Notice that for a class to be serialized successfully, two conditions must be met:
>
> - The class must implement the java.io.Serializable interface.
> - All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.
>
> [4]

What a refreshingly reasonable approach to have! For me to access `ClassA`'s sensitive data, it would have to be made serializable.

1.6 Caveats

Not all serialization libraries are created equally. For example the Kryo library will work with the above example and serialize the private data by changing the driver to the one described in Listing 6.[1]

```
package swa;
import java.io.FileOutputStream;
import java.io.IOException;
import com.esotericsoftware.kryo.Kryo;
import com.esotericsoftware.kryo.io.Output;
public class SerializeExample {
        public static void main(String[] args) {
                Kryo kryo = new Kryo();
                kryo.register(ClassA.class);
                kryo.register(ClassB.class);
                ClassA objA = new ClassA("swa", "123456789");
                ClassB objB = new ClassB(objA);
                try {
                        Output fos = new Output(new FileOutputStream("/home/swa/data.ser"));
                        kryo.writeObject(fos, objB);
                        fos.close();
                } catch (IOException i) {
                        i.printStackTrace();
                }
        }
}
```

Listing 6: Kryo Driver

To prevent Kryo from serializing the private data, the fields storing the data must be marked `transient`.

But is that really enough to protect a private class in Java? No! Kryo works by using reflection, and Java reflection permits the developer to change the security context of even private variables. Suppose that `ssn` is marked as `transient`. This variable may still be accessed and serialized with reflection using the driver in Listing 7.

```
package swa;
import java.lang.reflect.Field;
public class SerializeExample {
        public static void main(String[] args) {
                ClassA objA = new ClassA("swa","123456789");
                try {
```

```
                Field f = objA.getClass().getDeclaredField("ssn");
                f.setAccessible(true);
                System.out.println(f.get(objA));
            } catch (NoSuchFieldException e) {
                e.printStackTrace();
            } catch (SecurityException e) {
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }
}
```

Listing 7: Reflection Driver

1.7 Conclusions

So the next time I see a CWE-499, I know to check the following factors:

1. Does the class contain sensitive data?
2. Is the class (or its parent) serializable?
3. Is the field that contains the sensitive data marked transient?
4. Does the project disable reflection?

If the class doesn't truly contain sensitive data, then it would be a False Positive. If it does, then the other three factors help guide us to an appropriate conclusion. The CWE would most likely be confirmed if the class is serializable and the field is not marked transient. But, if either of those conditions are false, then it may be a merely informational finding. An issue could, at least theoretically, arise over the next development cycle as folks make changes.

And, that's not all. I'll go ahead and add one last thing to check to our list:

5. Is the sensitive data stored unencrypted?

While this one falls under the umbrella of a different CWE (311), it's worth checking while you're evaluating a instance of 499.

References

[1] Kryo Contributors. Kryo. Java binary serialization and cloning. Aug. 26, 2021. URL: https://github.com/EsotericSoftware/kryo (visited on 08/26/2021).

[2] CWE Content Team. "CWE-499: Serializable Class Containing Sensitive Data". In: (2021). URL: https://cwe.mitre.org/data/definitions/499.html.

[3] Jon Hood, ed. SwATips. https://www.SwATips.com/.

[4] Tutorials Point. Java - Serialization. Mar. 26, 2018. URL: https://www.tutorialspoint.com/java/java_serialization.htm (visited on 08/26/2021).