UDC 004.415.24

ENSURING INFORMATION SECURITY WHEN SERIALIZING JAVA OBJECTS

M. MAKARYCHEV, V. MAKARYCHEVA, A. OSKIN
Polotsk State University, Belarus

This paper describes how to serialize Java objects with support for encryption and electronic signature. Based on existing tools in Java, its own implementation with fixed security settings is proposed.

Today, Java is the most popular programming language according to the TIOBE Index. Java supports the object-oriented programming (OOP) paradigm. This means that any data is represented as objects of some class. One of the characteristics of the object is the state – information characterizing the object.

Often the state of an object needs to be saved to a file, database, or transferred over the network. Serialization is used for this. Serialization is the process of converting the state of an object into a data stream. Deserialization is the reverse process. Serialization of objects of some class is possible only when this class implements the Serializable interface.

In special cases, the serialization mechanism can be customized. Firstly, using the transient modifier, you can reverse the serialization of a certain field of the class. Secondly, you can change serialization behavior by defining methods:

– private void writeObject(ObjectOutputStream out) throws IOException;

– private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException.

Thirdly, in the place of implementation of the Serializable interface, you can implement the Externalizable interface and define its methods:

– public void writeExternal(ObjectOutput out) throws IOException;

– public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException.

In this case, the serialization protocol is determined by the developer.

Sometimes a serialized object is required to be cryptographically secure. Cryptographic protection refers to ensuring confidentiality, monitoring the integrity and authenticity of information. The serialization control methods listed above have a significant drawback: you need to change the source code of the class. For example, a class must contain encryption and decryption algorithms for its fields. The developers of Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) solved this problem and introduced two classes: SealedObject and SignedObject.

A SealedObject instance acts like a wrapper around another object. It contains an encrypted version of the serialized representation of the wrapped object. The following conditions are required to create an instance of SealedObject:

– the wrapped class must be serializable;

– the constructor, in addition to the wrapped object, requires an instance of the Cipher class, which is a cryptographic cipher [1].

Encrypted content can be decrypted and deserialized to obtain the original object. To do this, you must call one of the overloaded getObject() methods on the SealedObject instance [1].

A SignedObject instance also acts as a wrapper around another object. It contains the wrapped object in serialized form, as well as information about the signature required to verify the authenticity and integrity of the wrapped object. To create an instance of SignedObject, the following conditions must be met.

– The wrapped class must be serializable.

– The constructor needs the private key of the signer, represented by an instance of the PrivateKey class.

– The constructor needs a signature generation mechanism, represented by an instance of the Signature class [1].

Once the SignedObject instance has been created or obtained as a result of deserialization, you can verify the integrity of the wrapped object and the alleged authenticity of the signer. To do this, you must call the verify() method, passing it the public key and an instance of the Signature class, with which SignedObject was originally created [1].

Using the getObject() method, you can get an encapsulated object. The encapsulated object is deserialized before returning. It is worth noting that SignedObject does not encrypt the content, so you can get a wrapped object without first checking the sender's authenticity or integrity [1].

Having examined these classes and their dependencies in detail, you will notice that they also require quite a lot of settings before using them. We solved this problem and wrote classes for working by default with SealedObject and SignedObject.

For convenient use of SealedObject, the class SealedObjectCreator was written. It contains instances of the Key and Cipher classes. These instances are initialized in the SealedObjectCreator constructor. To initialize the Key, a KeyGenerator with the AES algorithm is used, which generates a 256-bit key using a SecureRandom instance with the SHA1PRNG algorithm. To initialize Cipher, the AES algorithm is used in GCM mode without a padding scheme (AES/GCM/NoPadding), the constant Cipher.ENCRYPT_MODE and the key obtained earlier are passed to the init() method of the Cipher instance [2]. SealedObjectCreator provides getter methods and an instance method for creating a SealedObject:

public SealedObject newSealedObject(Serializable wrappedObject) [3].

For convenient use of SignedObject, the SignedObjectCreator class was written. It contains instances of the Signature and KeyPair classes. These instances are initialized in the SignedObjectCreator constructor. Signature is initialized using the SHA512withECDSA algorithm. To initialize KeyPair, a KeyPairGenerator with an EC algorithm is used, which generates a 521-bit key using a SecureRandom instance with the SHA1PRNG algorithm [3]. SignedObjectCreator provides getter methods and an instance method for creating a SignedObject:

public SignedObject newSignedObject(Serializable wrappedObject).

For testing, the User class was created with the string fields name, phone, email and with the integer field id. The object of this class was constructed as follows:

final User user = new User(1L, "John", "8-047-429-28-05", "ragain.j@pol.com").

Using SealedObjectCreator, an instance of SealedObject was created, which was written to the file. An example of the contents of the file is shown in Figure 1.

Figure 1 shows that the information about the user object is encrypted. After deserializing the resulting SealedObject instance, you can call the getObject() method with the key from the SealedObjectCreator instance. In this case, the decrypted user object will be returned.

Using SignedObjectCreator, an instance of SignedObject was created, which was written to the file. An example of the contents of the file is shown in Figure 2.

Figure 2 shows that the information about the user object is not encrypted. For example, the phone number is highlighted in gray. Data types and other technical information are also visible here. It is worth noting that the file is written in binary form, and in a simple text editor you can read only string values. This file also carries digital signature information. If you change the contents of the file or pass the Signature instance different from the one provided by SignedObjectCreator to the verify() method of the deserialized instance of SignedObject, the verify() method will return false. This means that the integrity or authenticity of the data is compromised. Although the user object can be successfully obtained using the getObject() method of the SignedObject class.

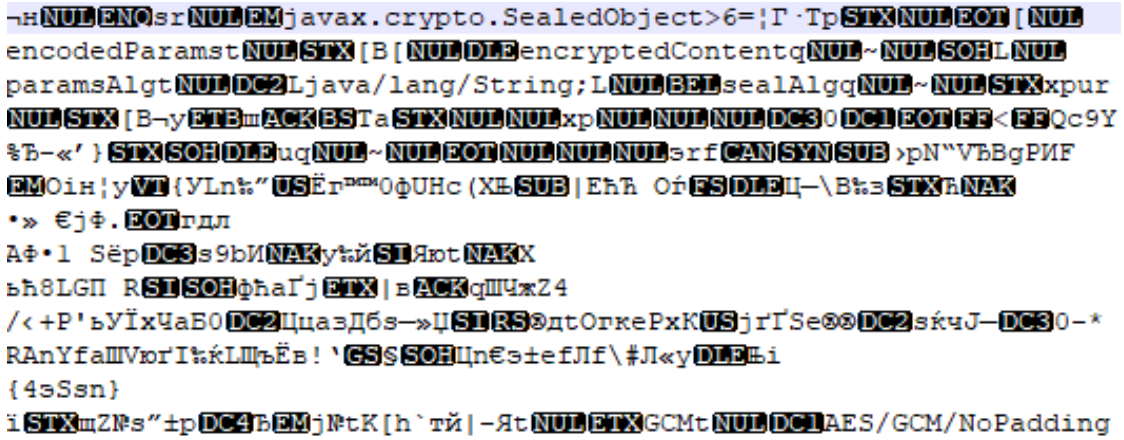

Figure 1. – Serialized SealedObject content

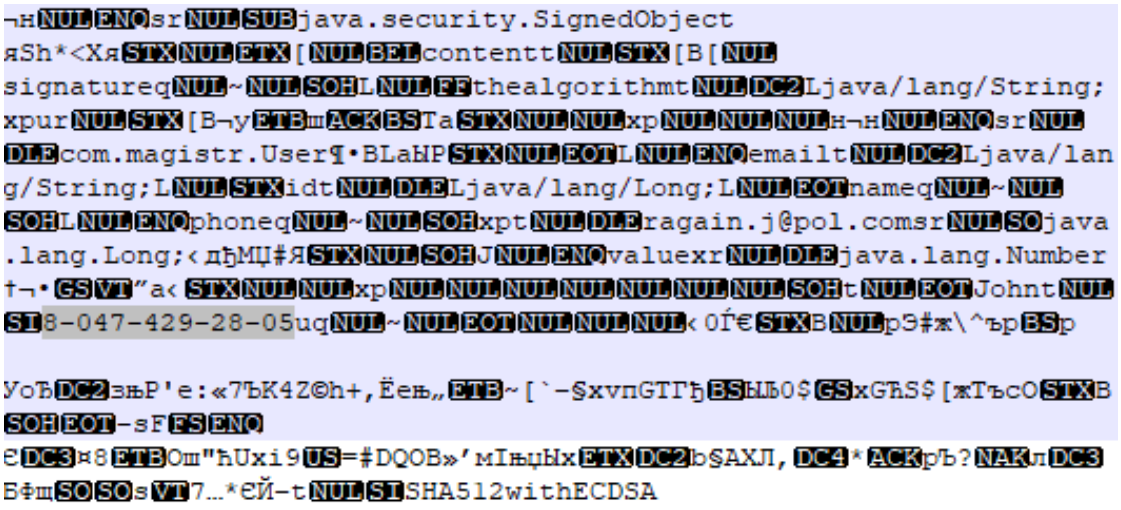

Figure 2. – Serialized SignedObject content

So, the classes SealedObjectCreator and SignedObjectCreator are a useful addition to SealedObject and SignedObject. They allow you to configure secure serializable objects by default and manage them.

REFERENCES

1. Protect information with the SignedObject and SealedObject classes [Electronic resource] / – Mode of access: https://www.javaworld.com/article/2076237/signed-and-sealed-objects-deliver-secure-serialized-content.html. – Date of access: 03.02.2020.
2. Security Best Practices: Symmetric Encryption with AES in Java and Android [Electronic resource] / – Mode of access: https://proandroiddev.com/security-best-practices-symmetric-encryption-with-aes-in-java-7616beaaade9. – Date of access: 05.02.2020.
3. Encryption and Decryption in Java Cryptography [Electronic resource] / – Mode of access: https://www.veracode.com/blog/research/encryption-and-decryption-java-cryptography. – Date of access: 05.02.2020.