SER02-J. Sign then seal objects before sending them outside a trust boundary

Sensitive data must be protected from eavesdropping. All data that crosses a trust boundary must be protected from malicious tampering. An obfuscated transfer object [Steel 2005] that is strongly encrypted can protect data. This approach is known as *sealing* the object. To guarantee object integrity, apply a digital signature to the sealed object.

Sealing and signing objects is the preferred mechanism to secure data when

- Transporting sensitive data or serializing any data.
- A secure communication channel such as Secure Sockets Layer (SSL) is absent or is too costly for limited transactions.
- Sensitive data must persist over an extended period of time (for example, on a hard drive).

Avoid using home-brewed cryptographic algorithms; such algorithms will almost certainly introduce unnecessary vulnerabilities. Applications that apply home-brewed "cryptography" in the `readObject()` and `writeObject()` methods are prime examples of *anti-patterns*. However, using existing cryptography libraries inside `readObject()` and `writeObject()` is perfrectly warranted.

This rule applies to the intentional serialization of sensitive information. SER03-J. Do not serialize unencrypted sensitive data is meant to prevent the unintentional serialization of sensitive information.

Noncompliant Code Example

The code examples for this rule are all based on the following code example:

```
class SerializableMap<K,V> implements Serializable {
  final static long serialVersionUID = -2648720192864531932L;
  private Map<K,V> map;

  public SerializableMap() {
    map = new HashMap<K,V>();
  }

  public Object getData(K key)  {
    return map.get(key);
  }

  public void setData(K key, V data)  {
    map.put(key, data);
  }
}

public class MapSerializer {
  public static SerializableMap<String, Integer> buildMap() {
    SerializableMap<String, Integer> map =
        new SerializableMap<String, Integer>();
    map.setData("John Doe", new Integer(123456789));
    map.setData("Richard Roe", new Integer(246813579));
    return map;
  }

  public static void InspectMap(SerializableMap<String, Integer> map) {
    System.out.println("John Doe's number is " + map.getData("John Doe"));
    System.out.println("Richard Roe's number is " +
                       map.getData("Richard Roe"));
  }

  public static void main(String[] args) {
    // ...
  }
}
```

This code sample defines a serializable map, a method to populate the map with values, and a method to check the map for those values.

This noncompliant code example simply serializes then deserializes the map. Consequently, the map can be serialized and transferred across different business tiers. Unfortunately, the example lacks any safeguards against byte stream manipulation attacks while the binary data is in transit. Likewise, anyone can reverse-engineer the serialized stream data to recover the data in the `HashMap`. Anyone would also be able to tamper with the map and produce an object that made the deserializer crash or hang.

```
public static void main(String[] args)
                        throws IOException, ClassNotFoundException {
  // Build map
  SerializableMap<String, Integer> map = buildMap();

  // Serialize map
  ObjectOutputStream out =
      new ObjectOutputStream(new FileOutputStream("data"));
  out.writeObject(map);
  out.close();

  // Deserialize map
  ObjectInputStream in =
      new ObjectInputStream(new FileInputStream("data"));
  map = (SerializableMap<String, Integer>) in.readObject();
  in.close();

  // Inspect map
  InspectMap(map);
}
```

If the data in the map were sensitive, this example would also violate SER03-J. Do not serialize unencrypted sensitive data.

Noncompliant Code Example (Seal)

This noncompliant code example uses the `javax.crypto.SealedObject` class to provide message confidentiality. This class encapsulates a serialized object and encrypts (or seals) it. A strong cryptographic algorithm that uses a secure cryptographic key and padding scheme must be employed to initialize the `Cipher` object parameter. The `seal()` and `unseal()` utility methods provide the encryption and decryption facilities respectively.

This noncompliant code example encrypts the map into a `SealedObject`, rendering the data inaccessible to prying eyes. However, the program fails to sign the data, rendering it impossible to authenticate.

```
public static void main(String[] args)
                        throws IOException, GeneralSecurityException,
                               ClassNotFoundException {
  // Build map
  SerializableMap<String, Integer> map = buildMap();

  // Generate sealing key & seal map
  KeyGenerator generator;
  generator = KeyGenerator.getInstance("AES");
  generator.init(new SecureRandom());
  Key key = generator.generateKey();
  Cipher cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.ENCRYPT_MODE, key);
  SealedObject sealedMap = new SealedObject(map, cipher);

  // Serialize map
  ObjectOutputStream out =
      new ObjectOutputStream(new FileOutputStream("data"));
  out.writeObject(sealedMap);
  out.close();

  // Deserialize map
  ObjectInputStream in =
      new ObjectInputStream(new FileInputStream("data"));
  sealedMap = (SealedObject) in.readObject();
  in.close();

  // Unseal map
  cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.DECRYPT_MODE, key);
  map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);

  // Inspect map
  InspectMap(map);
}
```

Noncompliant Code Example (Seal Then Sign)

This noncompliant code example uses the `java.security.SignedObject` class to sign an object when the integrity of the object must be ensured. The two new arguments passed in to the `SignedObject()` method to sign the object are `Signature` and a private key derived from a `KeyPair` object. To verify the signature, a `PublicKey` as well as a `Signature` argument is passed to the `SignedObject.verify()` method.

This noncompliant code example signs the object as well as seals it. According to Abadi and Needham [Abadi 1996],

> *When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of the message.*

Any malicious party can intercept the originally signed encrypted message from the originator, strip the signature, and add its own signature to the encrypted message. Both the malicious party and the receiver have no information about the contents of the original message because it is encrypted and then signed (it can be decrypted only after verifying the signature). The receiver has no way of confirming the sender's identity unless the legitimate sender's public key is obtained over a secure channel. One of the three Internal Telegraph and Telephone Consultative Committee (CCITT) X.509 standard protocols was susceptible to such an attack [CCITT 1988].

Because the signing occurs after the sealing, it cannot be assumed that the signer is the true originator of the object.

```java
public static void main(String[] args)
                        throws IOException, GeneralSecurityException,
                               ClassNotFoundException {
  // Build map
  SerializableMap<String, Integer> map = buildMap();

  // Generate sealing key & seal map
  KeyGenerator generator;
  generator = KeyGenerator.getInstance("AES");
  generator.init(new SecureRandom());
  Key key = generator.generateKey();
  Cipher cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.ENCRYPT_MODE, key);
  SealedObject sealedMap = new SealedObject(map, cipher);

  // Generate signing public/private key pair & sign map
  KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
  KeyPair kp = kpg.generateKeyPair();
  Signature sig = Signature.getInstance("SHA1withDSA");
  SignedObject signedMap =
      new SignedObject(sealedMap, kp.getPrivate(), sig);

  // Serialize map
  ObjectOutputStream out =
      new ObjectOutputStream(new FileOutputStream("data"));
  out.writeObject(signedMap);
  out.close();

  // Deserialize map
  ObjectInputStream in =
      new ObjectInputStream(new FileInputStream("data"));
  signedMap = (SignedObject) in.readObject();
  in.close();

  // Verify signature and retrieve map
  if (!signedMap.verify(kp.getPublic(), sig)) {
    throw new GeneralSecurityException("Map failed verification");
  }
  sealedMap = (SealedObject) signedMap.getObject();

  // Unseal map
  cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.DECRYPT_MODE, key);
  map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);

  // Inspect map
  InspectMap(map);
}
```

Compliant Solution (Sign Then Seal)

This compliant solution correctly signs the object before sealing it. This approach provides a guarantee of authenticity to the object in addition to protection from man-in-the-middle attacks.

```
public static void main(String[] args)
                        throws
  IOException, GeneralSecurityException,
                               ClassNotFoundException {
  // Build map
  SerializableMap<String, Integer> map = buildMap();

  // Generate signing public/private key pair & sign map
  KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
  KeyPair kp = kpg.generateKeyPair();
  Signature sig = Signature.getInstance("SHA1withDSA");
  SignedObject signedMap = new SignedObject(map, kp.getPrivate(), sig);

  // Generate sealing key & seal map
  KeyGenerator generator;
  generator = KeyGenerator.getInstance("AES");
  generator.init(new SecureRandom());
  Key key = generator.generateKey();
  Cipher cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.ENCRYPT_MODE, key);
  SealedObject sealedMap = new SealedObject(signedMap, cipher);

  // Serialize map
  ObjectOutputStream out =
      new ObjectOutputStream(new FileOutputStream("data"));
  out.writeObject(sealedMap);
  out.close();

  // Deserialize map
  ObjectInputStream in =
      new ObjectInputStream(new FileInputStream("data"));
  sealedMap = (SealedObject) in.readObject();
  in.close();

  // Unseal map
  cipher = Cipher.getInstance("AES");
  cipher.init(Cipher.DECRYPT_MODE, key);
  signedMap = (SignedObject) sealedMap.getObject(cipher);

  // Verify signature and retrieve map
  if (!signedMap.verify(kp.getPublic(), sig)) {
    throw new GeneralSecurityException("Map failed verification");
  }
  map = (SerializableMap<String, Integer>) signedMap.getObject();

  // Inspect map
  InspectMap(map);
}
```

Exceptions

SER02-J-EX0: A reasonable use for signing a sealed object is to certify the authenticity of a sealed object passed from elsewhere. This use represents a commitment *about the sealed object itself* rather than about its content [Abadi 1996].

SER02-J-EX1: Signing and sealing is required only for objects that must cross a trust boundary. Objects that never leave the trust boundary need not be signed or sealed. For example, when an entire network is contained within a trust boundary, objects that never leave that network need not be signed or sealed. Another example is objects that are only sent down a signed binary stream.

Risk Assessment

Failure to sign and then seal objects during transit can lead to loss of object integrity or confidentiality.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SER02-J	Medium	Probable	High	**P4**	**L3**

Automated Detection

This rule is not amenable to static analysis in the general case.

Related Guidelines

MITRE CWE	CWE-319, Cleartext Transmission of Sensitive Information

Bibliography

[API 2014]	
[Gong 2003]	Section 9.10, "Sealing Objects"
[Harold 1999]	Chapter 11, "Object Serialization"
[Neward 2004]	Item 64, "Use `SignedObject` to Provide Integrity of Serialized Objects" Item 65, "Use `SealedObject` to Provide Confidentiality of Serializable Objects"
[Steel 2005]	Chapter 10, "Securing the Business Tier"