Security for XML Data Binding

Nils Gruschka[1] and Luigi Lo Iacono[2,*]

[1] NEC Laboratories Europe, Heidelberg (Germany)
nils.gruschka@neclab.eu
[2] Europäische Fachhochschule (EUFH), Brühl (Germany)
l.lo_iacono@eufh.de

Abstract. This paper introduces a complementary extension to XML data binding enabling the (selective) protection of structured objects and members. By this contribution, an object can be transformed into a *secured object* which contains encrypted and/or signed parts according to an assigned security policy. The serialization of secured objects results in XML data which is protected by standard XML security means. Thus, this approach introduces a data-oriented security mechanism which seamlessly integrates into XML data binding and therefore enables cross-platform (de)serialization of secured objects without the need of programming against a specific XML security API. Distinct entities in a distributed processing environment then operate transparently either on plain or secured instances of a class.

Keywords: XML, Data Binding, Data Protection, Security, Secured Objects

1 Introduction

The extended markup language (XML) [3] provides a basic syntax and a set of encoding rules that can be used to share data between different kinds of computers, different applications, and different organizations. Its platform and language independent declarative description of data makes XML indispensable especially in system integration scenarios in which heterogeneous components and systems need to be coupled.

Different mechanisms exist to process XML data. The simple API for XML (SAX) [16] provides a mechanism to read data from an XML document. The parser operates in a streaming manner generating events which invoke user-defined and registered callback methods for processing these events. SAX has the advantage to be efficient in terms of resource-consumption and parsing time, but has a complex programming model which allows the read-only access to XML data. The document object model (DOM) [8] offers a tree-oriented API for random access to the XML data. Although DOM provides a much more intuitive programming model in comparison to SAX and allows a read as well as

* This work was performed while Luigi Lo Iacono was with NEC Laboratories Europe, NEC Europe Ltd.

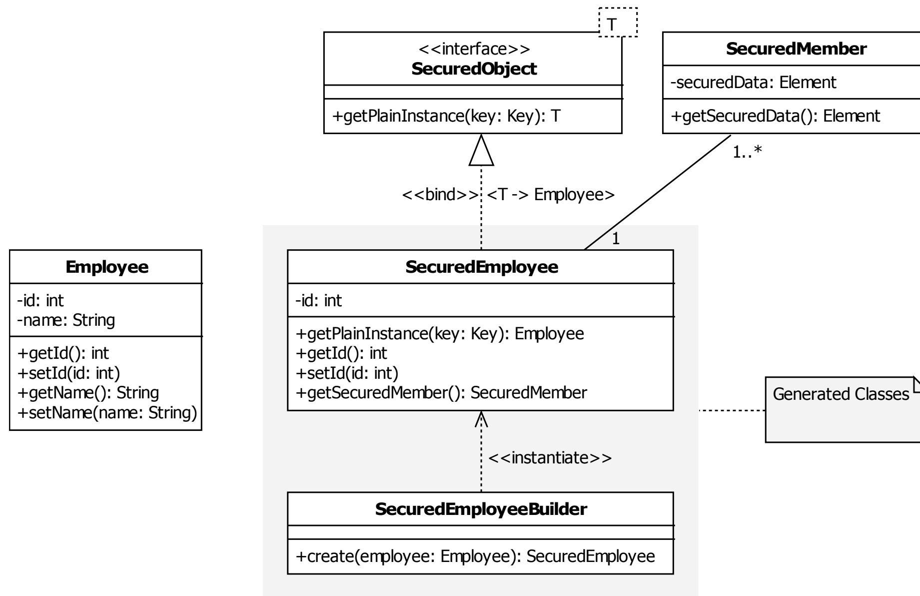


Fig. 1: Pattern of the secured object approach (including an example)

a write access to XML data, the trade-off comes with the required resources to build an in-memory tree representation of the XML data. Additionally, further approaches exist trying to combine the stream- and tree-based schemes in order to exploit the benefits of both [10, 19].

All the above mentioned approaches have in common, that they define some kind of API with an associated programming model to retrieve the data from the direct representation of the XML document itself. With XML data binding [13] instead the information inside the XML document is mapped to an object of a specific programming language, enabling the conversion between an XML representation and an in-memory representation. This allows applications to access the data in the XML document from the object rather than using a specific API operating on the raw XML document. An XML data binding engine accomplishes this task by automatically creating a mapping between XML schema [17] types of the XML document and the programming language classes to be represented in memory.

XML data binding has become an essential building block to implement communications in distributed systems. Contemporary Web Service frameworks e.g. rely internally on XML data binding mechanisms to bridge between the service (or client) implementation and the Web Service messaging framework which are commonly based upon XML-based SOAP messages [2].

Of course, security issues are crucial relevant in distributed sytems using open networks. The adoption of transport-based security such as SSL/TLS [4] is a straight-forward solution, but does not fulfill requirements for more fine-grained

security mechanisms applied to individual elements contained in the structured objects. Even the message-oriented security mechanisms defined by WS-Security [14] for protecting SOAP messages only partly fulfill this requirement. While WS-Security allows securing selected parts of the message and the protection mechanisms "survive" multiple SOAP intermediaries, even here the transferred data is completely unprotected inside the ultimate receiver.

For protecting data beyond that point security mechanisms at the data level are needed. Integrating data-oriented security into XML data binding provides a solution for this issue giving developers the full convenience of working with data structures in their prefered programming language instead of dealing with according API and requiring the according knowledge to operate on the raw XML and applying XML security to it.

As a usage scenario consider an online shop offering a Web Service interface to the user. The order request messages contain in addition to the user's address and the ordered goods also the user's credit card number. All of this data should be kept confidential during transport from the user to the service. This task can e.g. be performed by WS-Security means which are even able to encrypt selective parts of the message. This can be useful if intermediaries between the client and the service must have access to parts of the message. But in any case, the message is completely decrypted inside the service. However, it is not necessary for the service to learn the credit card number. Thus, it is desirable that (just) the credit card number remains encrypted inside the order document. For billing purposes the shop can send the encrypted number to the credit card company, which is the only entity able to decrypt it.

An important property of XML data binding is the ability to (de)serialize objects across programs, languages, and platforms without the need of programming XML generation or processing code. Thus, requirements for a security solution for XML data binding include the capability to protect selective parts of structured objects while keeping the property of being platform and language independent. Additionally, from a programming perspective, a security system for XML data binding should not require programming of code to protect the objects or the serialized secured XML documents.

2 Related Work

To the knowledge of the authors, the existing approaches to secure software objects while being serialized to a a certain data sink (such as a file system, database or communication channel) are characterized by an "all-or-nothing" approach. Commonly, cipher stream classes are provided by XML data binding frameworks which can be plugged into the processing chain. The encipherment is performed before the serialized XML document is passed to its target (file system or communication channel). The decipherment is performed exactly in the opposite direction. First, the data is read in, then decrypted to an XML fragment and finally deserialized to the corresponding object hierarchy. JBoss Remoting [11] is an example for such a system. It provides a single API for

network-based invocations that uses pluggable transports and data marshallers, including an `EncryptingMarshaller` class for encrypting.

Other available approaches are even more general focusing either on the objects themselves or raw I/O streams. Examples again taken from the Java domain include the `SealedObject` [15] and `CipherOutputStream` [18] respectively. These approaches are not specific to XML data binding and can therefore not fulfill the granularity requirements at the object or XML document level.

Using XML security offers of course the required flexibility and selectiveness in applying security means to—also parts of—the XML document. Making use of XML security directly, however, breaks the processing model of XML data binding, since developers need to cope again with raw XML processing and XML security APIs which the data binding is aiming to hide. Moreover, the direct use of XML security requires a profund knowledge of the underlying concepts and specifications in order to ba able to implement effective security systems. An XML data binding integrated approach can henceforth lower the burden for developers and generate more reliable code which e.g. is not vulnerable to XML Signature Wrapping attacks [12].

Summarizing, no technology currently exists, which allows protecting sensitive data contained in software objects while being serialized to XML and at the same time preserves all benefits provided by XML in general and XML data binding in particular.

3 Security for XML Data Binding

To overcome these limitations of current approaches and to seamlessly integrate a (selective) protection of structured objects in XML data binding without requiring the programming of code and without the loss of platform and language neutrality, a new approach is introduced in this paper.

3.1 General Approach

The basic idea is illustrated in Figure 2. First, the field (or fields) to be secured are serialized to an XML tree fragment. On this tree fragment the intended securing means are applied using standard XML security means. The result is stored as XML element in the corresponding secured object. This secured object can be serialized using XML data binding methods and send to other peers. Depending on the scenario those peers operate either on the secured object or the plain object.

This approach leads to a number of interesting characteristics. First of all, such a secured object can be serialized and deserialized using standard data binding methods. This enables a non-security enabled peer to transparently operate on the secured object. Further, the transport message is an interoperable XML document conforming to the XML security standards. Additionally, the serialized message as well as the secured object contain all the necessary meta-data (algorithms, certificates etc.) for processing the secured parts.

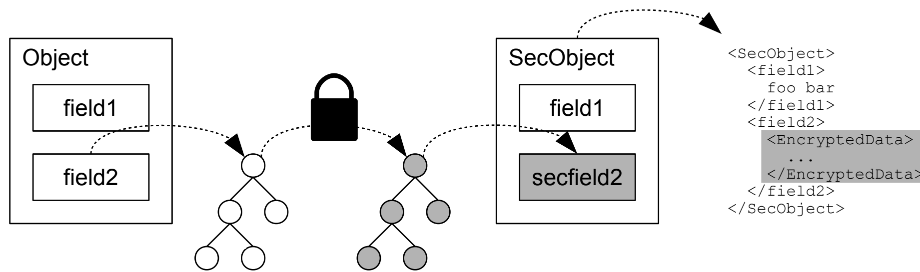


Fig. 2: Illustration of general approach

To define the parts to be protected an enhancement of the mapping specification between the object hierarchy and the XML document by security expressions is introduced. Developers can henceforth define object members to be protected by declaring them in the mapping specifications. A corresponding tool chain generates the secured counterpart objects according to this defition. The newly developed approach uses the XML security techniques XML Signature [1] and XML Encryption [9] to protect the sensitive object parts. This leads to a generic approach to manage the protected parts as secured XML elements inside specific secured mirror classes. Thus, together with the developed code generation mechanisms, the effort for a developer to introduce code for protecting the data in XML serialization in their own code is reduced to zero.

The two main artifacts introduced with this approach are the `SecuredObject` interface and the `SecuredMember` class (see Figure 1). The `SecuredObject` interface simply insures that all secured classes—which are generated by the framework—implement an operation to revert the security measures. The `SecuredMember` class contains the secured parts serialized as DOM element protected by XML security means. Depending on whether the parts to be protected are all within the same subtree or in distinct and disconnected subtrees, the secured class contains one or multiple associations with the `SecuredMember` class.

The proposed approach starts from the class, which defines the XML root element. For this class, a corresponding protected class is generated in case the mapping specification includes security declarations. As naming convention, the class name of the generated class will be constructed by prepending the term “`Secured`” in front of the original class name. This is also illustrated in Figure 1. The class which serves as XML root element is the `Employee` class. For this class and the contained members, the developer can specify what needs to be protected and how. In the same way as the developer specifies what members should become part of the serialized XML and whether as element or attribute, the developer is able to declare whether the members should be encrypted, signed, encrypted-before-signed or signed-before-encrypted. In this example, we assume the `name` member is to be encrypted. One can see in the

example, that the `SecuredEmployee` does not contain the name, but instead a `securedMember` hiding it (in encrypted form).

As already stated, most of the code required to handle protected instances of classes is generated. A specific command line tool takes the name of the class representing the XML root element as starting point. It then generates two classes: the mirror class which contains the specified security mechanisms along with the enforcement and a builder class to create secured instances out of unprotected ones. Again, a naming convention is used to construct the class name of the builder. The word "`Builder`" is appended to the end of the corresponding `SecuredObject` class. In Figure 1 the class to build instances of `SecuredEmployee` is the `SecuredEmployeeBuilder`.

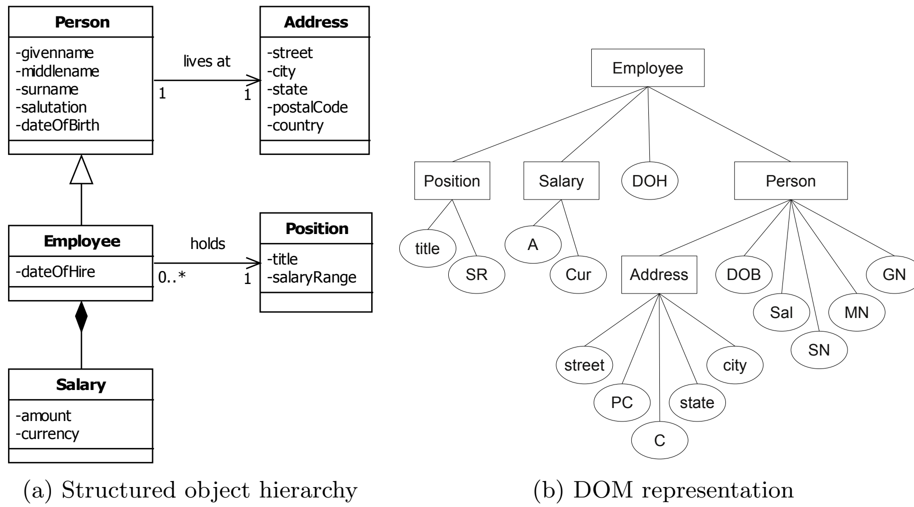
The reverse operation, i.e. generating an unprotected instance out of a secured one, is performed by calling the `getPlainInstance()` operation. This method is defined in the `SecuredObject` interface and needs to be implemented by all secured classes. Since the code of such classes is generated and due to the usage of XML security standards this is done automatically without the interaction or coding by a developer.

3.2 Inheritance and Composition

In cases in which a secured member is not a simple data type but a complex data structure or object hierarchy, all included data must be secured and is treated as defined for the class or its members respectively. Assume—as an extension of the previous example—that the whole `Employee` class presented before obtains some data items by inheritance and others by some form of association (see Figure 3a). By specifying that the `Employee` class must be encrypted, all members contained in it and especially the `Person`, `Position` and `Salary` class (either simple or complex) will be encrypted. Thus, security protection rules diffuse inside complex types. If this is too coarse grain (e.g. if the postal code must stay accessible), more selective policies can be defined for the contained complex object structures, by labeling all members but the `postalCode` member of the `Address` class to be encrypted.

This flexibility is realized by serializing the data structure to an in-memory DOM representation. Figure 3b shows the graphical DOM representation of an instance of the data model given in Figure 3a. On this tree XML Encryption and XML Signature means are applied and thus their capabilities regarding granularity can be fully exploited, meaning for example that security mechanisms can be applied to single nodes up to (distinct) sub-trees or even the complete DOM tree.

Finally, the developed design of the presented approach allows security expressions to be nested, resulting in an inside-out (i.e. starting with the adoption at the lowest level of the object hierarchy) application of the cryptographic primitives.



(a) Structured object hierarchy (b) DOM representation

Fig. 3: Example data structure

4 Prototype Implementation

The proposed approach has been prototypically implementated based on the Java Architecture for XML Binding (JAXB) [13] framework. Figure 4 gives an overview of the JAXB concept. On the left-hand one can see, that classes are mapped to XML Schema elements. Depending on the scenario, the developer either creates a Java class and the JAXB framework generates the appropriate XML Schema or vice versa. This part is performed during compile-time. At runtime the framework converts the appropriate Java objects to XML documents and vice versa.

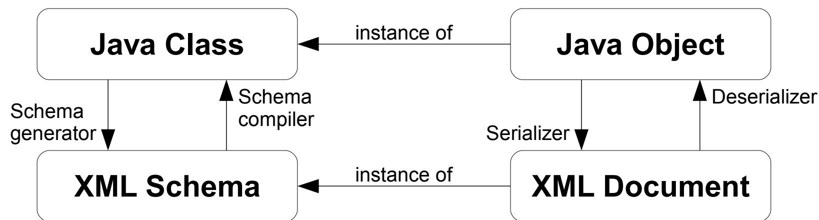


Fig. 4: JAXB concept

```
@XmlRootElement
public class Employee {
   private int id;

   @Encrypt
   private String name;
...
```

Listing 1.1: Security-related annotations

Since JAXB uses annotations to specify the mapping between the Java class and its XML representation, the proposed approach extends this scheme by security-related annotations for the expression of security policies. The prototype framework defines the `@Encrypt`, `@Sign`, `@EncryptBeforeSign` and `@SignBeforeEncrypt` annotations for this purpose. Note, that the annotations do not require to be configured with cryptographic parameters. Such are specified while creating instances of the secured class and can be defined dynamically during runtime as will be explained later in this section.

Listing 1.1 shows how the `Employee` class (used as an example in Figure 1) is enriched with these annotations, declaring to encrypt the employee's name. In order to generate the secured class, the `sogen` command line tool scans the Java class which defines the XML root element for these annotations in post-order tree traversal. For the example given in section 3, the `SecuredEmployee` class is generated for securing `Employee` instances. Listing 1.2 shows how to serialize a protected object using the generated classes.

```
// Create an instance of SecuredEmployee
Employee employee = new Employee(123456, "Luigi");
SecuredEmployeeBuilder seb = new
   SecuredEmployeeBuilder(cryptoSuite, key);
SecuredEmployee securedEmployee =
   seb.create(employee);

// Serialize the SecuredEmployee instance
// to the console
JAXBContext context =
   JAXBContext.newInstance(SecuredEmployee.class);
Marshaller marshaller =
   context.createMarshaller();
marshaller.setProperty(
   Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(securedEmployee, System.out);
```

Listing 1.2: Securing object instances before serialization

`SecuredEmployee` instances are created out of `Employee` instances by the `SecuredEmployeeBuilder` using a specified cryptographic suite and keying

material. The code example given in Listing 1.2 also shows, that the usage of the JAXB framework is not changed by our security extension. The only difference is, that instead of providing a clear text instance of an object, its secured counterpart instance is passed to the `marshal()` operation. Since the protected object instance contains secured members as DOM elements, these elements are serialized as is. Listing 1.3 shows an example of a serialized `SecuredEmployee` resulting from the security policy contained in Listing 1.1. One can see that it contains an employee identifier in plaintext and one `SecuredMember` containing ciphertext using standard and interoperable XML security mechanisms.

```
<securedEmployee>
  <id>123456</id>
  <securedMember>
    <xenc:EncryptedData xmlns:xenc=".../xmlenc#"
      Type=".../xmlenc#Element">
      <xenc:EncryptionMethod
        Algorithm="...#aes128-cbc"/>
      <xenc:CipherData>
        <xenc:CipherValue>
          ClZU7W/aj4ElpRDeofETz8FsPI8T2T2dl9vedc0+
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </securedMember>
</securedEmployee>
```

Listing 1.3: XML serialization of a secured object

The simple procedure for retrieving an unprotected object instance from its protected counterpart is shown in Listing 1.4. Since the generated `SecuredEmployee` class implements the `SecuredObject` interface, it contains the `getPlainInstance()` method. This operation perform the necessary transformations and checks (e.g. signature verification), inverts the cryptographic operations and returns an instance of the `Employee` class.

```
// Create an instance of Employee from a
// SecureEmployee instance
Employee employee =
   securedEmployee.getPlainInstance(key);
```

Listing 1.4: Reversal of a secured object

Note, that due to the explicit design and the usage of XML security to store the protected members inside the `SecuredEmployee` class, the meta data included inside the XML element of the `SecuredMember` class contains all required information to perform the cryptographic transformations and checks. Thus, just the keying material for secret key operations has to be provided .

5 Discussions and Conclusion

Despite all the criticisms related its verbosity, XML has established itself in a broad variety of applications and systems. XML data binding helps in eliminating XML processing drawbacks by offering a convenient method of handling XML data from within applications. Together with its ability to (de)serialize objects across programs, languages, and platforms it is a powerful tool for implementing heterogeneous distributed systems, e.g. as the core of SOAP-based Web Service frameworks.

For objects containing sensitive information security issues become relevant. The presented approach for securing objects introduces a mechanism that integrates seamlessly with XML data binding and allows for the (selective) protection of an object's structure and members without the need of using additional APIs and knowledge to work on the raw XML document. The concept of managing the protected members as secured XML elements using the XML security standards enables flexible data-oriented protection and furthermore keeps the properties of being interoperable and program, language and platform independent.

Other XML based systems can benefit from the data-oriented security approach introduced by this paper. Consider SOAP frameworks or XML databases as an example. When coupling the approach of secured XML data binding with HTTP [7] and the REST [6] concepts, light-weighted secured Web Services can even be realized. Further research is required to investigate this idea as a simple yet powerful replacement of SOAP as an alternative technical foundation for service-oriented architectures (SOA) [5]. In particular, the data-oriented vs. message-oriented security capabilities are an interesting study target, especially regarding the need for increased security within the endpoints themselves and not only during the transfer.

A proof of concept implementation in Java based on JAXB has been discussed, showing the ease-of-use and potential of this approach. During the evaluation of the presented prototype, it became clear, that additional policy expressions can further improve the usability. In cases, in which the number of members which are to remain unprotected is much larger than the number of members which must be protected, expressions to declare the exclusion from the protection inherited from the upper object hierarchy layer decreases the overall number of expressions required. Currently missing features are the creation of Java classes out of an XML schema definition and related aspects such as security policy expressions for XML schema.

Finally, it is planned to release these developments as an open source project in order to provide a platform to support future research and development work as well as to raise the awareness for the presented approach in standardization forums.

References

1. M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML-Signature Syntax and Processing. *W3C Recommendation*, 2002.

2. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. *W3C Note*, 2000.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Male, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). *W3C Recommendation*, 2006.
4. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. *IETF request for comments*, RFC 4346, 2006.
5. T. Erl. *Service-Oriented Architecture – Concepts, Technology, and Design*. Prentice Hall, 2005.
6. R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, 2000.
7. R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF request for comments*, RFC 2616, 1999.
8. A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *W3C Recommendation*, 2004.
9. T. Imamura, B. Dillaway, and E. Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.
10. Java Web Services Performance Team. Streaming APIs for XML Parsers. Technical report, Sun Microsystems, 2005.
11. JBoss Community. JBoss Remoting. *http://jboss.org/jbossremoting/*.
12. M. McIntosh and P. Austel. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 20–27, New York, NY, USA, 2005. ACM Press.
13. B. McLaughlin. *Java and XML Data Binding*. O Reilly, 2002.
14. A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). 2006.
15. T. Sundsted. Signed and sealed objects deliver secure serialized content. *JavaWorld*, 2000.
16. The SAX Project. Simple API for XML – SAX 2.0.1. *http://www.saxproject.org/*, 2002.
17. E. van der Vlist. *XML Schema*. O'Reilly, 2002.
18. A. Ward. Encrypting the Java serialized object. *Journal of Object Technologies*, 2006.
19. J. Zhang. Simplify XML processing with VTD-XML. *JavaWorld*, 2006.