ADITYA K SOOD

insecurities in designing XML signatures

Aditya K Sood, a.k.a. oknock, is an independent security researcher and founder of SecNiche Security, a security research arena. He is a regular speaker at conferences such as XCON, OWASP, and CERT-IN. His other projects include Mlabs, CERA, and TrioSec.

adi.zerok@gmail.com

XML DIGITAL SIGNATURE TECHNOLOGY is on high heels nowadays, but potential insecurities have been encountered because of insecure programming practices. This article discusses the weak spots in the coding of XML signatures and related operations. The procedural approach involves inline cryptography to combat application vulnerabilities. Stress is placed on secure coding practices.

This article encompasses the practical problems in designing XML signatures through the use of APIs. XML signatures are used to provide security to data of any kind, whether XML or binary. The confidentiality, integrity, and authenticity of the message have to be preserved when designing a SOAP request for communication. XML API functionality is very versatile but at the same time protection measures have to be included to prevent loss of data. Verification of data on the client end becomes a formidable task, owing to the persistence of errors, leading to failure of the post-verification signing process. The prerequisites will be listed and discussed from the application point of view to thwart Web-based errors in XML signing of messages.

XML Digital Signatures

The digital signing of messages has become an efficient security measure. XML signatures are being used extensively to initiate a realm of security. The implementation is done through Apache structural libraries or XML digital signature APIs. However, using digital signatures is not devoid of implementation problems. This article serves to dissect the process of implementation of XML API.

XML security is considered an intermediate process in designing Java security components. As such, the element of security is implemented before interaction with an application. The major component is XWSS, which stands for XML Web Service Security. This component functions directly with the Apache XML security provider.

To understand XWSS, let's look first at the XML security stack.

The APIs are standardized under JSR 105. The two pluggable components present are the Apache XML Security Provider and the SUN Java Cryptography Architecture (JCA) Provider. Both compo-

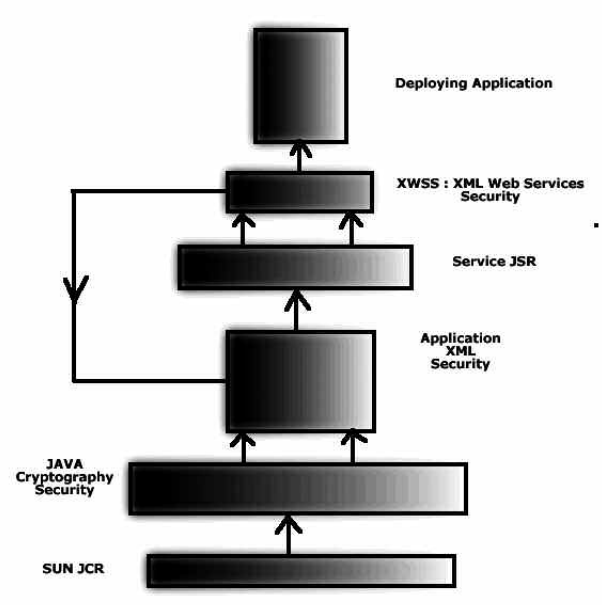

FIGURE 1: THE COMPONENTS USED IN XML SIGNING

nents interface directly with the JCA component. The JSR 105 standard is implemented through the Apache XML Security Provider, which ensures proper interaction with the XWSS component. So, overall a message is signed with an XML signature before an application receives it. The use of XML APIs is an effective method for protecting data integrity. The cryptography architecture followed is elucidated in Figure 1. The practical implementation is made possible through the design of cryptographic libraries imported during the time of execution.

Let's have a look at the XML signature packages:

- The java.xml.crypto package contains classes that are used directly in the implementation of design and generation and encryption of messages through the digital XML signature.
- The java.xml.crypto.dsig package comprises interfacial components that describe the cryptographic-related W3C specifications. It is used in signing and validation of digital signatures.
- The javax.xml.crypto.dsig.keyinfo package constitutes interfaces to various key structures that are defined in the W3C XML digital signature recommendation.
- The javax.xml.crypto.dsig.spec package contains classes for input parameters such as digests and keys.
- The javax.xml.crypto.dom and javax.xml.crypto.dsig.dom packages contain DOM-related classes.

The presentation of these various packages is initiated to trigger the JAVA Cryptography Architecture. The XML signature structures are implemented by the various interfaces provided by these crypto packages. For example, the Key-related interfaces Keyinfo, KeyName, KeyValue, and PGPData are defined on the basis of the W3C recommendations. Developers basically generate abstract factories such as XMLSignatureFactory or KeyInfoFactory based on the interfaces provided by these crypto packages. Developers also create their own URI dereferencing implementation based on the URI dereference class.

Secure coding is a precursor to secure implementation. Improper handling and implementation can marginalize the entire structure of Web applications. These implementation problems are described next.

Let's have a look at the XML signature layout (see Fig. 2).

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="urn:envelope">
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
<SignedInfo>
<CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComme
nts"/>
<SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>

<Reference URI="">
<Transforms>
<Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
</Transforms>

<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<DigestValue>uooqbWYa5VCqcJCbuymBKqm17vY=</DigestValue>
</Reference>
</SignedInfo>

<SignatureValue>KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRI26S2tmXjqNND7MRG
toew==
</SignatureValue>
<KeyInfo>
<KeyValue>
<DSAKeyValue>

<P>/KaCzo4Syrom78z3EQ5SbbB4sF7ey80etKII864WF64B81uRpH5t9jQTxeEu0Im
bzRMqzVDZkVG9xD7nN1kuFw==</P>
<Q>li7dzDacuo67Jg7mtqEm2TRuOMU=</Q>

<G>Z4Rxsnqc9E7pGknFFH2xqaryRPBaQ01khpMdLRQnG541Awtx/XPaF5Bpsy4pN
WMOHCBiNU0NogpsQW5QvnlMpA==</G>
<Y>qV38IqrWJG0V/mZQvRVi1OHw9Zj84nDC4jO8P0axi1gb6d+475yhMjSc/BrIVC
58W3ydbkK+Ri4OKbaRZlYeRA==
</Y>

</DSAKeyValue>
</KeyValue>
</KeyInfo>
</Signature>
</Envelope>
```

FIGURE 2: EXAMPLE OF AN XML SIGNATURE

The example in Figure 2 clearly depicts the implementation structure of an XML signature. It comprises a Keyinfo structure, which further incorporates the KeyValue. The full structure is placed into an envelope for transmission across the entities for secure communication. The XML signature specifications are based on the W3C recommendations and are applied directly on defined benchmarks. The benchmarks here refer to the standard specification provided by the W3C for effective structural design of XML documents and related applications. It actually provides a hierarchical implementation of XML objects. Also present is the signed info structure, which holds the desired information bearing the signature. It is implemented in a canonical form, in which a reference element is called by a URI. The value of the URI is always undertaken as a string. If the string is empty or NULL, then the root of the document is defined by that URI.

With this introduction to XML digital signatures, we can now dissect the implementation problems that cause discrepancies in communication.

Parsing Anatomy in Instantiating a Signature

The very first problem occurs in developing the instantiation of an XML digital signature. Parsing is actually undertaken by a JAXP builder library. Usually the builder library is present in a default state to be used independently. The developer can make a mistake in parsing an XML signature instance object through the predefined builder library. The proper implementation hierarchy is shown in Figure 3.The benchmark of standard XML implementation seen in Figure 3 is a basic procedure for designing an XML signature. First a signature instance object is created, then the Name space is set. Once this is done the builder library is called to parse the created object. Let's have a look at the code:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(new FileInputStream(argv[0]));
```

The code is stated in the hierarchy provided in Figure 3. The flaw occurs mainly in setting NameSpaceAware true and in argument-passing in parsing through file-streaming. File-streaming is a process in which file-handling functions are dynamically used based on the variance of input. Because argv[0], the value of the input parameter changes when different numbers of arguments are passed. If the proper argument is not passed, the instantiation of the XML signature goes awry, because it affects the signature stats. So parsing must be taken into account to hinder any further passing of errors in the signature. If the instantiation is not done properly, it can cause stringent errors in the application of the signature. Thus developers should be careful in accomplishing this task.

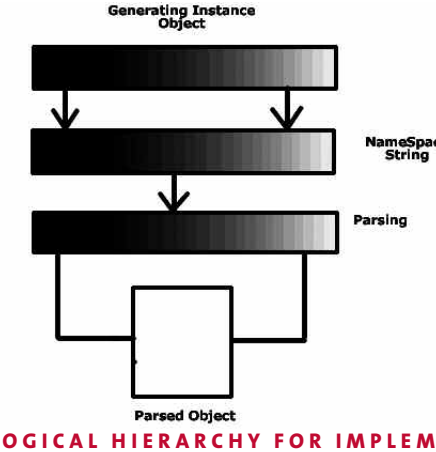

FIGURE 3: LOGICAL HIERARCHY FOR IMPLEMENTING XML SIGNATURES

Signature Specification Error Checks

Once the object is instantiated, the next step is to specify a signature, which then has to be validated. The major problem occurs when the error checks are not implemented properly, and consequently wrong elements are not filtered and get passed as such. For example, in the absence of a string check, an application error occurs whenever a null string or large string is passed. During signature specification the error checks have to be executed by the developers to ensure that security is not constrained. Overlooked errors have the potential to throw the entire application into disarray. Let's look at the code for a better view:

```
NodeList nl = doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");
if (nl.getLength() == 0) { throw new Exception("Cannot find Signature element");}
```

As you see, handling errors during object implementation mutes their impact.

KeySelector Problem in the Validation Context

The validation context states that the context in which an XML signature instance is validated by passing input parameters. For instance, if a developer is using a DOM (Document Object Model), the developer has to instantiate a DOM validation context instance. The problem occurs mainly in passing the reference parameters to the generated validation context. In this a Key-

Selector KeyValue and a reference to a signature element are passed. The coding flaw occurs in passing the KeySelector pair and elements. This hampers the process of validation and leads to false references.

The following code represents the implementation of KeyValue and KeySelector structures:

```
private static class KeyValueKeySelector extends KeySelector {
    public KeySelectorResult select(KeyInfo keyInfo, KeySelector.Purpose purpose, AlgorithmMethod
    method,XMLCryptoContext context) throws KeySelectorException {

if (keyInfo == null) { throw new KeySelectorException("Null KeyInfo object!"); }
    SignatureMethod sm = (SignatureMethod) method;
    List list = keyInfo.getContent();
    for (int i = 0; i < list.size(); i++) {
    XMLStructure xmlStructure = (XMLStructure) list.get(i); if (xmlStructure instanceof KeyValue) {
    PublicKey pk = null;  try { pk = ((KeyValue)xmlStructure).getPublicKey(); } catch (KeyException ke) {

throw new KeySelectorException(ke); }
// make sure algorithm is compatible with method
    if (algEquals(sm.getAlgorithm(), pk.getAlgorithm())) { return new SimpleKeySelectorResult(pk); } } }

throw new KeySelectorException("No KeyValue element found!"); }
    static boolean algEquals(String algURI, String algName) {
    if (algName.equalsIgnoreCase("DSA") && algURI.equalsIgnoreCase(SignatureMethod.DSA_SHA1)) {
    return true; } else if (algName.equalsIgnoreCase("RSA") &&
    algURI.equalsIgnoreCase(SignatureMethod.RSA_SHA1)) { return true; } else { return false;} } }
```

Actually, KeySelector tries to find a suitable key for the validation of data. The Key is stored in KeyValue. So a wrapper class is designed for applying this. Remember, to subdue the impact of KeySelector problems, KeySelector exceptions should be implemented with desired checks. The context is implemented as follows:

```
DOMValidateContext valContext = new DOMValidateContext (new KeyValueKeySelector(), nl.item(0));
```

Developers should take this into account in developing robust signatures.

Mismanagement in Assembling XML Signature Components

Once different components are designed and articulated with code, they must be assembled into a singular object. This assembly is required because the application of a signature is possible only after the completion of the centralized object (i.e., XML signature object). As stated earlier, the application calls DOM to get the handle of the required XML signature, which is possible through the XMLSignatureFactory object. Three steps must be completed prior to the implementation:

- Signing the URI of an object
- Specifying the digest method
- Transforming the enveloped layout

Whenever an application calls a specific code related to XML signatures from the server, a URI is required to complete the action. The URI describes the root of the element. If a mismatch occurs in passing arguments, information can be leveraged, because the infection vector is randomized and it can direct the execution vector in any sphere of application.

The envelope transformation causes the signature to be removed prior to the calculation of the signature value. Insecurity occurs in passing arguments. In this example, no specific object is supplied but a null string is subjected as an argument and the transformation object is set directly. This is bad programming practice in the context of signature designing. Look at this code snippet:

```
Reference ref = fac.newReference
    ("", fac.newDigestMethod(DigestMethod.SHA1, null),
    Collections.singletonList (fac.newTransform(Transform.ENVELOPED,
    (TransformParameterSpec) null)), null, null);
```

References should be applied with caution; the wrong reference points to the wrong application entity, thereby creating considerable inefficiency. The SignedInfo object should be created carefully with argument fusing. The problem here is that the first parameter in the reference is supplied as [" "], but it should be supplied with some proper argument or NULL (pointing to no memory). Reference parameters should be supplied in a correct manner with standard objects, as follows:

```
    Reference ref = fac.newReference("#object",
      fac.newDigestMethod(DigestMethod.SHA1, null));
SignedInfo si = fac.newSignedInfo (fac.newCanonicalizationMethod (CanonicalizationMethod.
INCLUSIVE_WITH_COMMENTS, (C14NMethodParameterSpec) null),
fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null), Collections.singletonList(ref));
```

Once the SignedInfo object is created, key generation comes next. The keys should be handled and generated in a standard manner in the context of the specific application:

```
KeyInfoFactory kif = fac.getKeyInfoFactory(); KeyInfo object:
KeyValue kv = kif.newKeyValue(kp.getPublic());
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
XMLSignature signature = fac.newXMLSignature(si, ki);
```

Strict vigilance is required for assembling XML signature components. The issues presented here cover some of the major problems related to XML signature designing.

Conclusion

Application security requires a well-planned and security-oriented coding layout to work efficiently. Dethroning insecure vectors requires secure coding practices. Application functionality can be jeopardized by the absence of even one of these factors. Protection should be applied through effective mechanisms or by adopting a security development life cycle when designing applications. Secure coding is considered to be a good proactive defense in combating application flaws.

FURTHER READING

[1] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon, "XML-Signature Syntax and Processing," in W3C Recommendation, World Wide Web Consortium, 12 February 2002, D. Eastlake, J. Reagle, and D. Solo, editors: http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[2] T. Imamura, B. Dillaway, and E. Simon, "XML Encryption Syntax and Processing," in W3C Recommendation, World Wide Web Consortium, 10 December 2002, D. Eastlake and J. Reagle, editors: http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[3] D. Eastlake and K. Niles, *Secure XML: The New Syntax for Signatures and Encryption* (Upper Saddle River, NJ: Pearson Education, 2002).

[4] J. Rosenberg and D. Remy, *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature and XML Encryption* (Indianapolis: Sams, 2004).

[5] http://java.sun.com/security/javaone/2002/javaone02.3189-jsr105-bof.pdf.