Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption

Juraj Somorovsky, Jörg Schwenk
Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
{Juraj.Somorovsky, Joerg.Schwenk}@rub.de

***Abstract*—At CCS'11 a new chosen-ciphertext attack on XML Encryption [13] has been presented. This attack is of high relevance, since it allows one to decrypt arbitrary encrypted XML payload by issuing 14 server requests per byte on average.**

In this paper we discuss several countermeasures against this attack, which have been considered by different framework developers for different scenarios. We analyze the scenarios and show why these countermeasures do not work. Thereby, we motivate for the application of authenticated encryption in the XML Encryption specification.

I. Introduction

The W3C XML Encryption specification [8] is a part of the XML Security standards. It is responsible for ensuring confidentiality in XML-based messages. It is adopted in various standardized scenarios (e.g. Web Services or Single Sign-On) and customized applications. It is applied in major web-based applications, ranging from business and e-commerce, financial services and healthcare, to governmental and military infrastructures. It is, for example, implemented in frameworks of several major organizations like Apache, Redhat, IBM, Microsoft, or SAP.

However, as previously shown [13], the XML Encryption specification is vulnerable to a specific chosen-ciphertext attack. This attack can be applied in all scenarios, where the attacker is able to send messages including modified ciphertext to an oracle that decrypts the message and responds with 1 or 0 according to the message validity. According to the server responses, the attacker can decrypt the whole ciphertext. The attack is based on a fact that the server parses the decrypted XML data. If the parsing fails, the server interrupts the message processing. This new side-channel has led to a generalization of padding oracle attacks applied by Vaudenay [23] and considered in further works [1], [20], [7], [24], [5], [6], and to a development of a more performant practical attack. The attacker has to issue only 14 requests to decrypt one byte on average.

In this paper we outline the attack and introduce two exemplary scenarios: Web Services and Single Sign-On. We have to note that this attack is however applicable to all the custom applications using XML Encryption and revealing a decryption oracle. We summarize discussions with various developers and introduce the proposed countermeasures. We explain why standard-conformant and customized countermeasures in most of the scenarios do not work. We e.g. show why unifying error messages is not a valid countermeasure or why XML Signatures [9] in Web Services cannot mitigate these attacks if the Web Services are secured using WS-Security Policy [16].

To this end, we propose the standardization of mode of operation including authentication and integrity check. We hope that the discussed difficulties by application of different countermeasures in different scenarios will force the standardization groups to include authenticated modes of operations in their specifications and reconsider deprecation and removing unauthenticated modes of operations. This includes the described XML Encryption specification [8][1] as well as the newly developed JSON Web Encryption standard [15].

II. Basics

In this section we give an overview of the Cipher block chaining mode of operation and security standards used in XML and Web Services.

A. Cipher Block Chaining

AES and 3DES are well-used symmetric-key encryption algorithms. They allow to encrypt and decrypt data, whose length is 16 or 8 bytes, respectively. In order to apply these algorithms to data of arbitrary length, the data has to be padded and split into blocks, which are then chained using a mode of operation.

Cipher block chaining (CBC) [18] is the most popular blockcipher mode of operation in practice. Its functionality in the XML Encryption specification including the padding scheme is depicted in Figure 1.

For its description, suppose a byte string $data'$ of arbitrary length. The $data'$ string is first padded in order to achieve a length l, which is an integer multiple of the block-size bs. XML Encryption specifies the following padding scheme:

1) Compute the smallest integer $p > 0$ such that $|data'|+p$ is an integer multiple of bs of the block cipher.
2) Append $(p-1)$ random bytes to $data'$.
3) Append one more byte to $data'$, whose integer value equals p.

Using this padding scheme we get $data$, whose length is multiple of bs. Now, we can split $data$ into blocks: $data = (data^{(1)}, \ldots, data^{(d)})$. These blocks are processed as follows:

[1]The newest version of the standard already includes Galois Counter Mode (along with CBC) as a mandatory mode of operation: http://www.w3.org/TR/xmlenc-core1/

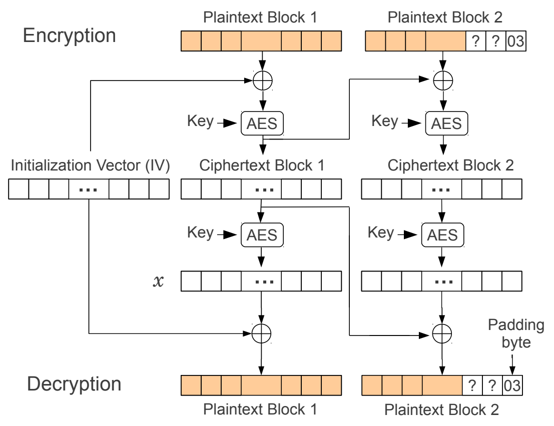


Fig. 1. CBC mode of operation with the XML Encryption padding scheme

- An *initialization vector* $iv \in \{0,1\}^{8 \cdot bs}$ is chosen at random. The first ciphertext block is computed as

$$x^{(1)} := data^{(1)} \oplus iv, \qquad C^{(1)} := \mathsf{Enc}(k, x^{(1)}). \quad (1)$$

- The subsequent ciphertext blocks $C^{(2)}, \ldots, C^{(d)}$ are computed as

$$x^{(i)} := data^{(i)} \oplus C^{(i-1)}, \qquad C^{(i)} := \mathsf{Enc}(k, x^{(i)}) \quad (2)$$

for $i = 2, \ldots, d$.
- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(d)})$.

The decryption procedure reverts this process in the obvious way.

B. XML and Web Services

The Extensible Markup Language [2] defines a structure for flexible storage and transmission of tree-based data. It is currently used in Single-Sign On scenarios, custom applications, or in Web Services [12].

Web Services is a W3C standard used to achieve inter-process interactions over networks between different software applications. The communicating applications use SOAP messages [11]. SOAP messages are XML-based messages generally consisting of *header* and *body*. The header element includes message-specific data (e.g. timestamp, user information, or security tokens). The body element contains function invocation and response data, which are mainly addressed to the business logic processors.

C. XML Security

XML documents often contain data whose confidentiality, authenticity, and integrity must be protected. Therefore, the W3C consortium developed standards describing the XML syntax for applying cryptographic primitives to XML data: XML Encryption [8] and XML Signature [9].

1) XML Encryption: XML Encryption specifies a method for achieving confidentiality of the stored XML elements. In order to encrypt XML data, in most scenarios *hybrid encryption* is used: First, the encryptor chooses a *session key* k. This key is encrypted using a public-key encryption scheme (please note that the public-key encryption scheme is out of scope of this paper since we describe an attack on the part encrypted with a symmetric key). Afterwards, the payload data is encrypted with a symmetric cipher. For this purpose, until March 2012, the standard allowed to choose between two symmetric ciphers, namely AES-CBC and 3DES-CBC.

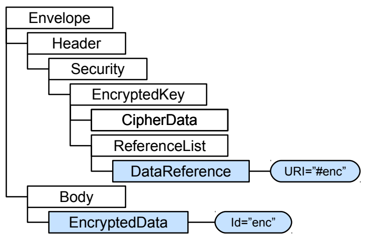


Fig. 2. Example of a SOAP message with encrypted data

Figure 2 gives an example of a SOAP message containing such a hybrid ciphertext. This message consists of the following parts related to our next attack description:

1) The `EncryptedKey` part. This part consists of two components. The `CipherData` element contains the encrypted session key k. `ReferenceList` contains references to all `EncryptedData` elements that can be decrypted with the session key k.
2) The `EncryptedData` part. This part contains the initialization vector iv in a clear followed by the payload data, encrypted using the session key k.

Let us mention that there may be *multiple* `EncryptedData` elements sharing the same session key. These elements are referenced using the `ReferenceList` element.

A SOAP Web Service receiving such an XML document processes it as follows. It first decrypts the session key k. Then, it uses k to decrypt all the parts containing encrypted payload. Finally, the payload data is parsed with an XML parser and the whole document is forwarded to the next module or business process invocation.

It is important to mention that the XML Security processing module typically does not know, which parts of the decrypted payload is later processed by the business logic. Thus, it could also happen that the encrypted payload is decrypted, parsed, and *not processed further*.

2) XML Signature: In order to ensure integrity and authenticity of exchanged XML documents, the XML Security working group defined the XML Signature specification [8]. A simplified structure of an XML Signature applied to a SOAP message according to WS-Security [19] gives Figure 3.

The depicted SOAP message includes a function invocation `"MonitorInstances"` defined in the SOAP body. The authenticity and integrity of the SOAP body is ensured by the XML Signature defined in the SOAP header. The XML Signature element consists of two mandatory elements: `SignedInfo` and `SignatureValue`. The `SignedInfo` element includes an Id-reference pointing to the SOAP body and a digest value computed over the referenced element. In order to secure the `SignedInfo` element, a signa-

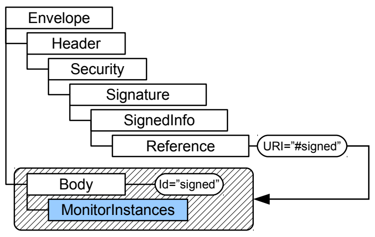


Fig. 3. Example of XML Signature applied on the SOAP body

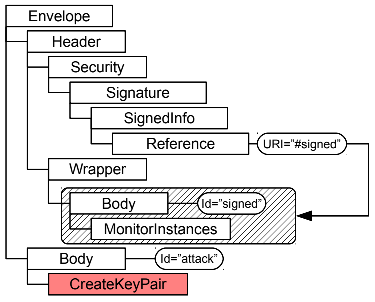


Fig. 4. Signature Wrapping attack

ture value over this element is computed and put to the `SignatureValue` element. This is achieved by a public-key algorithm such as RSA or DSA.

Processing of the given SOAP message would usually look as follows: The recipient first *searches for the referenced element* given in `SignedInfo`. He computes the digest value over this element and compares it to the value given in the `DigestValue` element. Afterwards, he verifies the signature value over `SignedInfo`. At the end, he can execute the function *defined in the SOAP body*.

D. XML Signature Wrapping

The *XML Signature Wrapping* attack [17] compromises integrity of the signed XML documents. An example of this kind of attack applied on the message given in Figure 3 shows Figure 4. In this example, the attacker moves the original SOAP body to a `Wrapper` element in the SOAP header. Afterwards, he creates a SOAP body with a new `Id="attack"` and defines an arbitrary function, e.g. `CreateKeyPair`. As the Id of the original SOAP body stays same and the concerning parts are not altered, the security logic can verify its integrity and authenticity. The business logic however invokes the newly inserted function `CreateKeyPair`.

Let us mention that there exist more sophisticated XML Signature Wrapping attack techniques. They misuse different properties of the validation logic [10], [21] or completely different referencing mechanisms [14].

E. WS-Security Policy

In order to define security properties of the exchanged SOAP messages, on both server and client side, WS-Security Policy [16] is used. This specification allows to indicate policy assertions that apply to Web Services. Policy assertions are grouped into policy alternatives. A set of policy alternatives

```
<Policy>
  <ExactlyOne>
    <All>
      <EncryptedParts><Body/></EncryptedParts>
    </All>
  </ExactlyOne>
</Policy>
```

Fig. 5. WS-Security Policy defining that the SOAP body in the incoming message has to be encrypted. However, it does not restrict encryption of other elements.

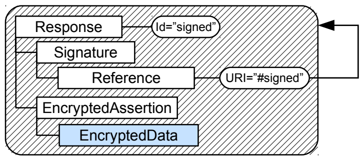


Fig. 6. Encrypted claims about identities are placed into the `EncryptedAssertion` element. This element is defined in the `Response` element and secured using an XML Signature.

gives a WS-Policy definition. For grouping of policy assertions two XML tags are used: `All` and `ExactlyOne`. `All` indicates that all child node assertions have to be fulfilled. `ExactlyOne` indicates a logical `XOR` and it contains assertions, from which *exactly one* has to be fulfilled. However, most of the Web Services implementations handle `ExactlyOne` as a logical `OR` and thus also accept messages with *more* assertions fulfilled.

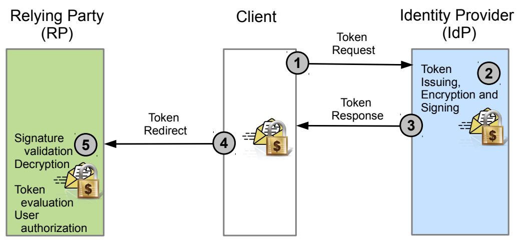
An example of WS-Security Policy document gives Figure 5. This Policy definition would enforce the server to process only the SOAP messages that contain encrypted SOAP body. It is however important to mention that using this policy all the analyzed frameworks also accepted SOAP messages including *additional* encrypted document parts, e.g. elements in the SOAP header.

F. Single Sign-On and SAML

The growing number of user identities in Internet has led to development of different Single Sign-On solutions. These solutions are based on one login-procedure. The users authenticate only by a trustworthy Identity Provider (IdP). After successful login, the Identity Provider can issue different tokens for the user giving him access to Relying Parties (RP). Single Sign-On can be realized by different standards including OAuth, OpenID, or Security Assertion Markup Language (SAML) [4].

In the following, we will consider the XML-based SAML standard. This standard allows to define arbitrary claims about identities and secure them by applying XML Signature and XML Encryption. An example of an encrypted SAML assertion is given in Figure 6. In this figure the `EncryptedAssertion` element is stored in a binding element – in this case in the `Response` element. Authenticity and integrity of the whole message are secured using an XML Signature.

A simplified SAML-based Single Sign-On scenario is depicted in Figure 7. In this scenario, a logged-in user first sends a token request including the desired RP to the IdP (1). The



Relying Party (RP)
Client
Identity Provider (IdP)
Token Request
Token Issuing, Encryption and Signing
Token Response
Token Redirect
Signature validation Decryption
Token evaluation User authorization

Fig. 7. A typical SAML-based Single Sign-On scenario: The user sends a token request to the IdP. The IdP issues a token for the user, which is afterwards encrypted and signed. The issued token is sent to the user and forwarded to the RP.

IdP issues a token for the user including e.g. his rights, roles, or expiration time. The token is afterwards encrypted and signed (2). The IdP sends the secured token to the user (3), who forwards it to the desired RP (4). The RP first validates the signature over the token and decrypts it. Afterwards, it evaluates the token and gives the user rights to access his domain (5).

III. High-Level Attack Description

In this section we give a brief attack description. Afterwards, we show two real-world attack scenarios. Please note that these scenarios are only exemplary, the attack can also be applied to other custom applications.

A. Attack Description

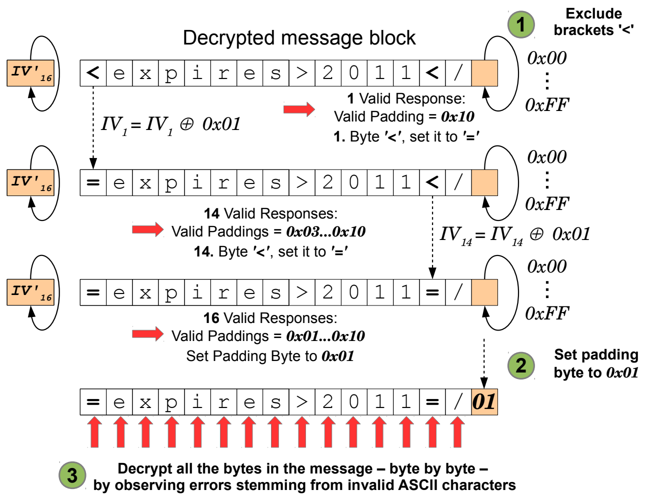
The attack on XML Encryption is based on the malleability of the CBC mode of operation: It allows an attacker, who is in possession of the ciphertext $C = (iv, C^{(1)})$, to flip arbitrary bits in the plaintext $m = Dec(C, k)$ without knowing the session key k. He can simply achieve this by flipping appropriate bits in the initialization vector iv.

Changing different bits in the encrypted XML data can lead to errors in the server processing, which forces the server to respond with a fault message. These fault messages can have the following reasons:

- **Decryption errors.** This error stems from an incorrect padding. Recall that the last byte of a padded plaintext must include a valid padding number p (in case of AES $p \in \{0x01 \ldots 0x10\}$).
- **Parsing errors.** This error may have two reasons. Either the plaintext contains an "unprintable" ASCII character (XML parser can parse only valid characters, otherwise it stops processing). The other reason is that the syntax of the decrypted XML part is not valid. The latter means that the special escape character $0x38$ (&) is not followed by a valid entity reference, or an element starting with the bracket $0x60$ (<) is not properly closed.

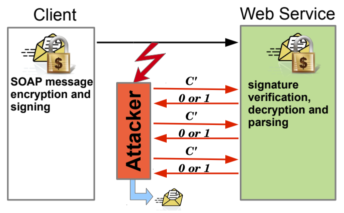
Sending differently adapted ciphertexts to the server and observing the server responses gives the attacker the possibility to efficiently decrypt the eavesdropped ciphertext.

Given a ciphertext $C = (C^{(0)}, C^{(1)}, \ldots, C^{(d)})$, which contains valid XML data including no escape characters (&), the algorithm looks as follows. The attacker performs the attack in d rounds (each ciphertext block is decrypted in each



Exclude brackets '<'
Decrypted message block
IV'16
< e x p i r e s > 2 0 1 1 < /
0x00
0xFF
IV1 = IV1 ⊕ 0x01
1 Valid Response: Valid Padding = 0x10
1. Byte '<', set it to '='
= e x p i r e s > 2 0 1 1 < /
14 Valid Responses: Valid Paddings = 0x03...0x10
14. Byte '<', set it to '='
IV14 = IV14 ⊕ 0x01
= e x p i r e s > 2 0 1 1 = /
16 Valid Responses: Valid Paddings = 0x01...0x10
Set Padding Byte to 0x01
Set padding byte to 0x01
= e x p i r e s > 2 0 1 1 = / 01
Decrypt all the bytes in the message – byte by byte – by observing errors stemming from invalid ASCII characters

Fig. 8. An example of the attack on the ciphertext block containing the incomplete plaintext `<expires>2011</`



Client
Web Service
SOAP message encryption and signing
Attacker
C'
0 or 1
signature verification, decryption and parsing

Fig. 9. Attack on Web Services: Adversary sends eavesdropped SOAP message with modified ciphertexts c' to the Web Service server and observes the server responses. With each response he learns a few plaintext bits.

different round). In each round $i = 1 \ldots d$, he takes two ciphertext blocks $c = (C^{(i-1)}, C^{(i)}) = (iv, C^{(i)})$ and with these two blocks he proceeds as follows (we show an example in Figure 8):

1) He excludes all the left brackets from the plaintext in order to overcome the parsing errors originating from improperly closed elements. To this end, he iterates over the last iv byte and sends in each iteration the two blocks $c = (iv', C^{(i)})$ to the server. The number of valid responses gives him the position of the first left bracket. Afterwards, he can flip the bit in the byte containing $'<'$ to convert it to a different valid character. He repeats this step until he excludes all the brackets.
2) He sets the last iv byte so that the last plaintext byte contains $0x01$. This gives him a possibility to access all the proceeding bytes in the block.
3) Now, he is able to decrypt all the bytes in the block byte-by-byte. Thereby, he uses server error responses returned by invalid parsing processing.

After execution of these three steps, the attacker has knowledge of vector $x^{(i)} = data^{(i)} \oplus iv$ (see Section II-A). Therefore, he would also be able to encrypt arbitrary XML elements with a slightly adapted algorithm [7].

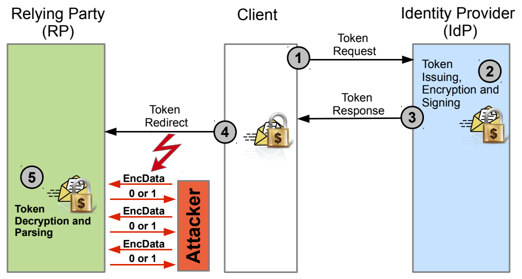


Fig. 10. Attack in a SAML-based scenario: the adversary gains the encrypted SAML message. He applies the attack by sending the modified SAML assertions to RP.

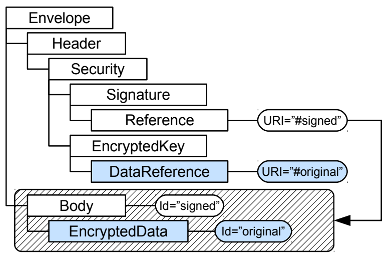


Fig. 11. XML Signature applied on the encrypted payload in a SOAP message

B. Attack Scenario: Web Services

The first scenario describes an attack on a Web Service server and is depicted in Figure 9. In this scenario the adversary first eavesdrops a SOAP message transmitted between the client and the Web Service server. He modifies the ciphertext c included in the SOAP message according to the algorithm described above. He sends the message to the server and observes its response. He needs only to distinguish between two response types: fault (0) and valid (1). According to the response type he is able to learn a few bits of the plaintext and construct new ciphertexts c'. He repeats this step until he decrypts the whole encrypted content.

C. Attack Scenario: SAML-based Single Sign-On

The second scenario presents an attack applied in SAML-based Single Sign-On and is depicted in Figure 10. It can be executed by an adversary who issues modified ciphertexts, includes them into the SAML assertions, and sends them to the RP. If the RP responds with 1 or 0 according to the message validity, the adversary can distinguish between valid and invalid ciphertexts. This gives him the possibility to decrypt the whole assertion or create new ciphertexts with arbitrary content.

In this scenario we have to consider also an adversary who is registered by the IdP and is a valid user of the RP. By applying the attack, this adversary can modify the encrypted SAML assertion sent from the IdP in order to execute privilege escalation and e.g. get administrator rights. Please note that in this case SSL/TLS would be of no help since the adversary is able to see the forwarded SAML messages.

IV. Countermeasures

In this section we give an overview of some countermeasures against the attack on XML Encryption and we analyze the scenarios, in which they work.

A. XML Signature

Application of XML Signatures on ciphertexts can ensure their authenticity and integrity. This standard describes two types of signatures, namely *public-key* XML Signatures (which use classical digital signature schemes) and *secret-key* XML Signatures (which use message authentication codes).

Generally, XML Signatures can thwart these attacks *if and only if*:

1) the attacker is not able to create validly signed messages.
2) the encrypted part cannot be moved to any unsigned part of the document.

If the application could ensure these two points, the attack could not be applied. However, in the following we illustrate that this is not that trivial. For this purpose, please consider the SOAP message depicted in Figure 11. In this message the SOAP body contains an encrypted payload, which is signed using XML Signature and Id-based referencing.

1) Attacker Able to Create Validly Signed Messages: First problem by application of public-key XML Signatures comes with a scenario where more parties are allowed to communicate with a Web Service server. Consider for instance there are two clients E_1 and E_2 of a Web Service provider, where both clients can send encrypted and digitally signed messages to the server. Assume that E_1 creates a SOAP message with encrypted and signed content, and sends it to the server. Now if E_2 wants to learn the contents of this message, then it could record this message, simply remove the signature, compute its own signature over the ciphertext, and mount the attack. The crucial point here is that the server cannot distinguish whether E_2 has encrypted the payload itself, or copied it from a ciphertext that E_1 has created. Still, in the digital signature setting the server can at least identify the attacker uniquely.

2) XML Signature Wrapping: Another problem with application of XML Signatures on ciphertexts is XML Signature wrapping attack. This attack affects public-key as well as secret-key XML Signatures.

Figure 12 gives an example of signature wrapping attack application on the message presented in the previous figure. In order to execute this attack, the attacker first copies the authenticated SOAP body into the security header. As the Id of the SOAP body stays the same, the signature component is able to validate this element. Afterwards, the attacker needs only to apply the attack on XML Encryption on the content of the newly defined SOAP body: he must force the server to decrypt the content of this element. Thus, he simply changes the `DataReference` element in `EncryptedKey` and makes it point to the content of the newly defined SOAP body.

The server would process the depicted message as follows. It would first validate the XML Signature. Afterwards, it would decrypt and parse the content of the newly defined SOAP body. After successful data decryption and parsing,

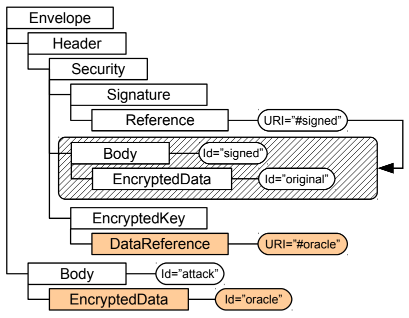


Fig. 12. XML Signature wrapping moves the signature validation to the SOAP header and thereby offers a new possibility for mounting of the XML Enryption attack

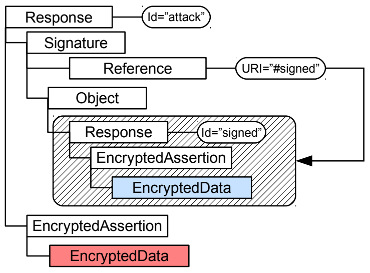


Fig. 13. An example of XML Signature wrapping attack application on Guanxi: the attacker places a copy of the original message into the `Object` element. Afterwards, he is able to send arbitrary ciphertexts to the RP and apply the XML Encryption attack.

the payload would be processed by the business logic. This step would most probably fail, since the randomly generated payload could not be processed by the server's business logic. Thus, by applying this attack the attacker must rely on differences between fault messages coming from the *decryption processing* and the *business logic*.

We evaluated Apache Rampart[2] and JBossWS[3] regarding their resistance against XML Signature wrapping attacks. Our experiments showed that all basic examples are vulnerable to simple XML Signature wrapping attacks. Thus, the attacker could mount the the attack on XML Encryption.

Similar attacks can be executed on SAML. An application of the XML Signature wrapping attack on the Guanxi implementation[4] is depicted in Figure 13. Using this attack, the adversary copies the whole document and extracts its `Signature` element. He inserts it into the `Object` element of the original document. Afterwards, he is able to construct arbitrary ciphertexts and apply the attack on XML Encryption against the RP. This way, he would be able to break integrity and confidentiality in the federated identity scenarios. More details on the topic of XML Signature wrapping in SAML are given in [22].

3) XML Encryption Wrapping: One could say signature wrapping attacks could be thwart using different countermea-

[2]Apache Axis2 security module: http://axis.apache.org/axis2/java/rampart
[3]http://www.jboss.org/jbossws
[4]http://sourceforge.net/projects/guanxi

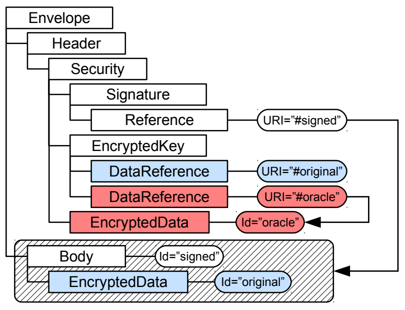


Fig. 14. XML Encryption wrapping copies the encrypted payload to an unsigned document part and thereby offers the attacker construction of a server oracle

sures. Apache Rampart e.g. uses in its advanced configuration WS-Security Policy verification, which explicitly checks, if the SOAP body is signed (this is a generally applicable countermeasure). This countermeasure makes it impossible to move the signed SOAP body to another parts of the document. However, as described in Section II-E, WS-Security Policy cannot define the maximal number of secured elements. This gives the attacker an opportunity to force the server to decrypt *additional* elements and apply the so called XML Encryption wrapping attack.

The basic idea of this attack is depicted in Figure 14. By its application the attacker simply copies the encrypted payload to any unsigned document part, which does not violate XML Schema. Afterwards, the attacker creates a new `DataReference` inside of the `EncryptedKey` element and makes it point the newly created `EncryptedData` element (please note that the attacker can also create a completely new `EncryptedKey` element). This forces the server to decrypt both `EncryptedData` elements. The original encrypted payload stays in the signed part. Thus, the XML Signature stays valid, the original payload is decrypted and parsed, and the server's business logic processes it *without* failures. On the other hand, the newly created `EncryptedData` element in the SOAP header is decrypted and parsed. The business logic does not process it, as it only needs the content of the SOAP body.

By application of this simple attack the attacker creates a server oracle, which responds with 0 or 1 depending on the validity of the ciphertext. Therefore, even encrypted server responses could simply be recognized and the attack applied.

We tested *all* Apache Rampart and JBossWS configuration examples, showing their vulnerability against XML Encryption wrapping. Moreover, we proved that these attacks are also applicable to security configurations in SAP gateways.

B. Unifying Error Messages

The possibly most obvious countermeasure to our attack consists in unifying the SOAP fault messages sent in response to invalid SOAP request messages so that an attacker can not distinguish between a decryption error and an application level error. However, this approach has some serious drawbacks.

At first, meaningful error messages are generally considered as "good programming practice". In fact, they are necessary for developers that have to implement client-side applications for encryption-enabled Web Services.

Secondly, even with unified SOAP fault messages, there are additional side-channels that can be exploited for determining what type of error a certain request message triggered. For instance, measuring the time consumed until a (unified) SOAP fault message arrives may already indicate the level in the application stack at which the error occurred.

Finally, we stress that this countermeasure is not effective when XML Encryption wrapping attacks as described above are applicable, since copying the encrypted data to a deeper level in the SOAP header would exclude them from XML Schema validation and business logic processing. Thus, the server would respond with a SOAP fault *if and only if* the encrypted data in the SOAP header are incorrect.

C. *Other Countermeasures*

In this section we describe other countermeasures, which were proposed by different software vendors.

1) Revocation of Session Keys: By application of the attack on one `EncryptedData` element the attacker uses for each server request the same symmetric key. Revocation of symmetric keys could be considered as a valid countermeasure. However, this countermeasure could cause the following problems:

- It needs to apply serious changes to applications or libraries.
- It requires a shared state across servers that are working in a cluster. Even if this state would be achieved, it would potentially be possible to get some responses back before all the servers know about the revoked key.
- Sometimes, even a few bits of information are enough to decrypt an important part of a message. You can e.g. think of messages including boolean values ("yes"/"no") or credit card numbers.

2) Blacklisting Clients' Public Keys: This countermeasure would bring similar problems as described above. However, it would partially solve the public-key signature problems described in Section IV-A as the not honest clients having server access would be blocked after sending invalid messages.

3) Inclusion of Signed Nonces: Inclusion of signed nonces could be seen as another valid countermeasure. However, it causes similar problems as the above mentioned revocation of session keys. Moreover, its application includes signature problems as described in Section IV-A.

4) Blacklisting Clients' IP addresses: A countermeasure proposed by some developers is blocking the clients that already sent a few number of invalid messages causing security faults. Please note that there are the same drawbacks as in the countermeasures described above. Moreover, this countermeasure does not work if the attacker is able to execute his attack from machines with different IP addresses. This is a valid assumption when considering allocation of virtual instances in cloud scenarios.

Another drawback would be a simple application of Denial-of-Service attacks. The attacker having the same IP as the honest client could send an amount of invalid messages causing security errors. This would again be a valid assumption when considering that attacker and honest client stay in the same subnet and are accessing a remote server.

5) Decryption only of Signed Elements: As described in the previous sections, the WS-Policy unfortunately does not allow to restrict element decryption. This makes it possible to apply XML Encryption wrapping attacks. Moreover, it is possible to execute Denial-of-Service attacks by inclusion of a large number of `EncryptedKey` elements, which force the server to execute expensive public-key decryption. Therefore, we propose inclusion of new policy tags allowing to decrypt the element only if it is signed.

Please note that application of this countermeasure still would not solve signature problems as described in Section IV-A.

D. *Changing Mode of Operation*

Finally we would like to highlight some cryptographic countermeasures. One option is to use a *symmetric* cryptographic primitive that does not only provide confidentiality, but also integrity. One option may be to add a *message authentication code* (MAC) like HMAC (see [18]) over the ciphertext to the encrypted message. In contrast to a digital signature, which can simply be replaced by a different signature, the security properties of a MAC ensure that it is not possible for an attacker to modify a ciphertext while keeping the MAC valid. In this case, our attack becomes impossible. Another option, which provides the same improvement in security, would be to replace the CBC mode of operation with a mode of operation that provides message integrity, like the *Galois counter mode* (GCM) [18], for instance.

The XML Security Working Group included already this mode of operation into the new standard version. The first implementation can be found in the Apache CXF framework[5].

1) Streaming-based XML Encryption processing: Even when applying message authentication codes, the developers should pay attention to other side-channel attacks. These can appear when applying streaming-based XML processing.

Consider an XML security module that accepts an encrypted byte stream. It decrypts this stream and sends it block-by-block to a streaming-based parser. The parser processes the incoming elements and sends them to another processing module. At the end of the stream the security module checks the MAC over all encrypted data. Consider that if a decryption or parsing error occurs, the parser interrupts the message processing and immediately sends a fault message. Moreover, consider an attacker who is in possession of a valid encrypted text and is able to flip the first bits in the plaintext (either by flipping bits in the initialization vector when applying CBC or by flipping bits directly in the ciphertext when applying GCM). If the encrypted text is long enough, the attacker could observe the

[5]http://cxf.apache.org

server response time differences. Longer response time would indicate correct payload and failure by the MAC verification. Shorter response time would indicate an incorrect payload and a failure by its parsing.

This is a valid assumption also when applying standard DOM-based (tree-based) parsers [3]. Namely, some DOM-parsers include an underlying streaming-parser, which is used for preprocessing of the incoming elements. An example gives `org.apache.xerces.parsers.DOMParser`, which is included as a default parser in JDK. Therefore, developers should pay attention when implementing modes of operations including MAC-checking: The encrypted part must always be completely processed and parsed and the MAC must be validated afterwards.

V. Conclusion

In this paper we analyzed different countermeasures against the attack on XML Encryption and described several side-channels, which allow to mount the attack in the Web Services and Single Sign-On scenarios. For the vendors we proposed two practical countermeasures.

By applying XML Signatures, the server must *not* decrypt any unsigned content. This trivially looking countermeasure cannot be defined within WS-Security Policy as this standard misses an assertion describing this processing. We propose the developers to implement this processing property by default.

Application of public-key XML Signatures alone does however not work in specific scenarios, where the adversaries could construct validly signed messages. In these scenarios we suggest usage of authenticated modes of operations. XML Encryption specification already included Galois Counter Mode of operation, which would fend the described attacks.

Because of the compatibility reasons, the CBC mode of operation is not going to be extracted from the XML Encryption specification. This motivates us for further research in this direction. We ask especially what happens if the adversary intercepts a message containing a GCM-ciphertext, but is able to enforce the server to decrypt it as CBC-ciphertext and thus omit integrity validation? Other research steps can lead to the analysis of the asymmetric algorithms like PKCS#1.5.

Acknowledgments

We would like to thank Thomas Alt, Martijn de Boer, Scott Cantor, Florian Kohlar, Tibor Jager, Jan Lieskovsky, Martin Rex, and Alessio Soldano for their contributions.

This work was partially funded by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

References

[1] John Black and Hector Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In Dan Boneh, editor, *USENIX Security Symposium*, pages 327–338. USENIX, 2002.

[2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 2008.

[3] Steve Byrne, Arnaud Le Hors, Philippe Le Hégaret, Mike Champion, Gavin Nicol, Jonathan Robie, and Lauren Wood. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[4] Scott Cantor, John Kemp, Rob Philpott, and Eve Maler. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

[5] Jean Paul Degabriele and Kenneth G. Paterson. Attacking the IPsec standards in encryption-only configurations. In *IEEE Symposium on Security and Privacy*, pages 335–349. IEEE Computer Society, 2007.

[6] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 493–504. ACM, 2010.

[7] Thai Duong and Juliano Rizzo. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *IEEE Symposium on Security and Privacy*, 2011.

[8] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.

[9] Donald Eastlake, Joseph Reagle, David Solo, Frederick Hirsch, and Thomas Roessler. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation*, 2008.

[10] Nils Gruschka and Luigi Lo Iacono. Vulnerable Cloud: SOAP Message Security Validation Revisited. In *ICWS '09: Proceedings of the IEEE International Conference on Web Services*, Los Angeles, USA, 2009. IEEE.

[11] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.

[12] Hugo Haas, David Booth, Eric Newcomer, Mike Champion, David Orchard, Christopher Ferris, and Francis McCabe. Web services architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[13] Tibor Jager and Juraj Somorovsky. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[14] Meiko Jensen, Lijun Liao, and Jörg Schwenk. The curse of namespaces in the domain of XML signature. In *ACM Workshop on Secure Web Services (SWS)*, pages 29–36, 2009.

[15] M. Jones, E. Rescorla, and J. Hildebrand. Json web encryption (jwe) – draft-jones-json-web-encryption-01, oct 2011. http://tools.ietf.org/html/draft-jones-json-web-encryption-01.

[16] Chris Kaler and Anthony Nadalin. Web Services Security Policy Language (WS-SecurityPolicy) 1.1. 2005.

[17] Michael McIntosh and Paula 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.

[18] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.

[19] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard*, 2006.

[20] Kenneth G. Paterson and Arnold Yau. Padding oracle attacks on the ISO CBC mode encryption standard. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 305–323. Springer, February 2004.

[21] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW)*, October 2011.

[22] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On Breaking SAML: Be Whoever You Want to Be. In *Proceedings of the 21st USENIX Security Symposium, Bellevue, WA, USA*, August 2012.

[23] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, April / May 2002.

[24] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 299–319. Springer, February 2005.