How to Break XML Encryption*

Tibor Jager
Horst Görtz Institute for IT Security
Chair for Network- and Data Security
Ruhr-University Bochum
tibor.jager@rub.de

Juraj Somorovsky
Horst Görtz Institute for IT Security
Chair for Network- and Data Security
Ruhr-University Bochum
juraj.somorovsky@rub.de

ABSTRACT

XML Encryption was standardized by W3C in 2002, and is implemented in XML frameworks of major commercial and open-source organizations like Apache, redhat, IBM, and Microsoft. It is employed in a large number of major web-based applications, ranging from business communications, e-commerce, and financial services over healthcare applications to governmental and military infrastructures.

In this work we describe a practical attack on XML Encryption, which allows to decrypt a ciphertext by sending related ciphertexts to a Web Service and evaluating the server response. We show that an adversary can decrypt a ciphertext by performing only 14 requests per plaintext byte on average. This poses a serious and truly practical security threat on all currently used implementations of XML Encryption.

In a sense the attack can be seen as a generalization of padding oracle attacks (Vaudenay, Eurocrypt 2002). It exploits a subtle correlation between the block cipher mode of operation, the character encoding of encrypted text, and the response behaviour of a Web Service if an XML message cannot be parsed correctly.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Code breaking

General Terms

Security

1. INTRODUCTION

The W3C XML Encryption specification [6] today marks the de-facto standard for data encryption in complex distributed applications. The use of XML as core data syntax, e.g. for major business, e-commerce, financial, healthcare, governmental and military applications, has resulted in broad adoption of XML Encryption to protect confidential data—especially, but not exclusively, in the context of Web Services. On the technical level, the XML Encryption specification precisely describes the process and syntax to be used when applying a cryptographic algorithm for data encryption to arbitrary XML-structured data. Moreover, it also describes how to process this syntax in order to decrypt the encrypted contents at the data recipient's side. XML Encryption does not describe a new cryptographic algorithm itself, but merely allows a set of standard block ciphers, namely AES and Triple-DES (3DES), to be used at will. In order to be able to encrypt messages which are longer than the input size of the block cipher, the *cipher-block chaining* (CBC) mode of operation is used.

In this paper, we present an attack technique that enables an adversary to decrypt arbitrary data that have been encrypted according to the XML Encryption specification. Based on a cryptographic weakness of the CBC mode, we are able to perform a chosen-ciphertext attack which recovers the entire plaintext from a given ciphertext. The only prerequisite for this attack consists in availability of an "oracle" telling us whether a given ciphertext contains a "correctly formed" plaintext. "Correctly formed" means here that the plaintext contains a valid encoding (e.g. in UTF-8 or ASCII) of a message. In practice, this oracle may be provided for instance by a Web Service that returns suitable error messages, or that provides some other side-channel allowing us to distinguish correct from invalid ciphertexts, like a different timing of data processing, for instance.

To prove the practical relevance of our attack, we apply it examplarily to a realistic Web Service based on the Apache Axis2 [18] XML framework. Axis2 is one of the most popular frameworks to support the building of Web Services client and server applications. We show that a moderately optimized implementation of the attack is able to decrypt 160 bytes of encrypted data within 10 seconds by issuing 2,137 queries to the Web Service. The complexity of the attack grows only linearly with the ciphertext size, thus recovering a larger plaintext of 1,600 bytes takes about 100 seconds and 23,000 queries.

Despite the fact that the details of the attack, and thus our results in context of the Axis2 framework, are of course rather application-specific, we want to stress that the attack itself is generic, and can be adapted to other scenarios like alternate XML frameworks and possibly even other systems

*This work has been supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II and the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

CCS'11 October 17–21, 2011, Chicago, Illinois, USA.

beyond XML Encryption as well. For instance, we have verified that the attack works against redhat JBoss [9] as well without any modifications.

In general chosen-ciphertext attacks can be avoided by ensuring the integrity of the ciphertext. One would therefore expect our attack can easily be thwarted by using XML Signature [7] to ensure integrity (note that XML Signature specifies not only classical public-key signatures, but also "secret-key signatures", i.e., message authentication codes). However, for several reasons this is not true, since we can show how to perform our attack *even if* either public-key or secret-key XML Signatures over the ciphertext are used. We achieve this either by applying classical Signature Wrapping [10] techniques, or by using a new attack technique that we call *Encryption Wrapping*. Thus, fixing current implementations or developing secure new implementations without changing the XML Encryption standard seems non-trivial. We illustrate this in more detail in the countermeasures section below.

Responsible disclosure. The attack described in this paper was announced to the W3C XML Encryption Working Group and to several providers and users of implementations of XML Encryption in February 2011. This includes The Apache Software Foundation (Apache Axis2), redhat Linux (JBoss), IBM, Microsoft, and a governmental CERT. All have acknowledged the validity of our attack. The CERT has disclosed our attack to governmental CERTs of other countries. Furthermore, redhat has immediately forwarded a short announcement to the `vendor-sec` mailing list.

We are providing advice to developers that are working on fixes. As already mentioned, this is difficult without changing the standard itself (see Section 6 for details).

Related Work and Our Contribution. It is well-known that the CBC encryption mode is malleable unless additional methods for ensuring integrity are applied. This was exploited by Vaudenay [19], who showed that it is possible to decrypt a ciphertext which is encrypted in CBC mode by issuing a small number of queries to a so-called *padding oracle*. Subsequent work refines the idea of Vaudenay [19], for instance to other padding schemes and modes of operations [1, 14], random or secret initialization vectors [20], attacks on real-world systems like IPSec [3, 4] and ASP.NET, JSF CAPTCHA, the Ruby on Rails framework, and an OWASP security system [15, 5]. Duong and Rizzo [15, 5] also make the observation that a padding oracle does not only allow to decrypt ciphertexts, but also to obtain valid *encryptions* of arbitrary plaintexts. It is even possible to describe padding schemes which are secure against padding oracle attacks [1, 13], a corresponding formal security model was given by Paterson and Watson [13].

By their nature, padding oracle attacks work only for certain padding schemes. In particular, the above attacks are *not* applicable to XML Encryption, since the standard specifies a different padding scheme.

In contrast, we use the *encoding* of data as a side-channel that allows to attack encryption schemes using a weak mode of operation, which allows to exploit the in most cases inevitable fact that an adversary is able to observe whether a decrypted plaintext is processed by an application after decryption, or discarded since the encoding could not be recognized. Note that this works independent of the padding scheme, and thus potentially also in scenarios where padding oracle attacks are not applicable. Mitchell [11] has already outlined such a generalization of padding oracle attacks, but without giving any specific example.

Our attack on XML Encryption is highly efficient, as it needs only 14 queries per byte on average to break XML Encryptions. For comparison, the related attack of Rizzo and Duong [15] issues 128 oracle queries per byte on average in order to break CAPTCHA.

2. PRELIMINARIES

Throughout the paper we write $\{0,1\}^\ell$ to denote the set of all bit strings of length ℓ. For $a, b \in \{0,1\}^\ell$, we write $|a|$ to denote the length of a (i.e., $|a| = \ell$), $a \oplus b$ to denote the bitwise XOR of a and b, and $a||b$ to denote concatenation of a and b. We write $\{0,1\}^*$ shorthand for $\{0,1\}^* = \bigcup_{i=0}^{\infty}\{0,1\}^i$.

2.1 Block Ciphers

The XML Encryption standard specifies AES and Triple-DES (3DES) as block ciphers. Since our attack does not exploit specific properties of these algorithms, but works with any cipher in a similar way, we will consider an abstract block cipher in the following.

To this end, we define a *block cipher* as a pair of algorithms $(\mathsf{Enc}, \mathsf{Dec})$. The encryption algorithm $c = \mathsf{Enc}(k, m)$ takes as input a key $k \in \{0,1\}^\ell$ and an ν-byte plaintext $m \in \{0,1\}^n$, where $n = 8 \cdot \nu$,[1] and returns a ciphertext $c \in \{0,1\}^n$. The decryption algorithm $m = \mathsf{Dec}(k, c)$ takes a key k and a ciphertext c, and returns $m \in \{0,1\}^n$. For instance, if AES is used then we have $n = 128$ and thus $\nu = 16$.

2.2 Padding Scheme

Suppose we want to encrypt an XML message $m \in \{0,1\}^*$ of arbitrary bit-length $|m|$. Since XML data is represented by UTF-8 characters, we may always assume that $|m|$ is an integer multiple of 8. However, $|m|$ does not need necessarily be an integer multiple of the block size n of the block. Thus a padding algorithm π with inversion algorithm π^{-1} must be applied to the message, to obtain a padded message $m' = \pi(m)$ whose bit-length $|m'|$ is an integer multiple of n.

The XML Encryption standard specifies the usage of the following padding scheme π:

1. Calculate the smallest non-zero number *plen* of bytes that must be suffixed to the plain text to bring it up to a multiple of the block size n.
2. Append $plen - 1$ arbitrary pad bytes to m.
3. Set the last byte equal to *plen*.

For instance, the example given in [6] considers a three-byte message $m = \texttt{0x616263}$ and a block cipher with $n = 64$, so that $\nu = 8$. In this case, we have

$$\pi(m) = m' = \texttt{0x616263}????????\texttt{05},$$

where ?? is an arbitrary byte value.

To remove the padding, one simply reads the last byte of m' and removes the required number of bytes to obtain m.

[1] Throughout the paper we assume that the block size n is always an integer multiple of 8.

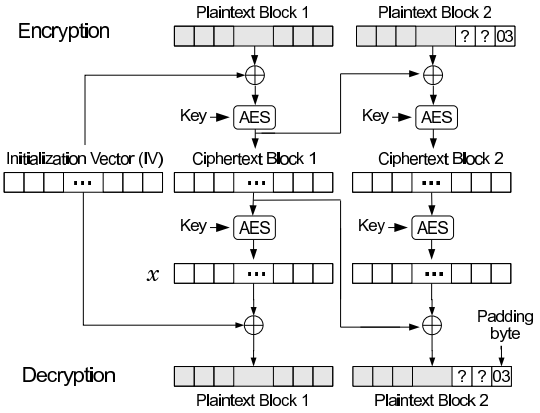

Figure 1: Illustration of the CBC mode of operation with the padding scheme from Section 2.2.

2.3 Cipher-Block Chaining

Cipher-block chaining (CBC) is the most popular block-cipher mode of operation in practice, and the only mode specified in the XML Encryption standard. It processes a message m', whose length is an integer multiple dn of the block size of $(\mathsf{Enc}, \mathsf{Dec})$, as follows.

- An *initialization vector* $iv \in \{0,1\}^n$ is chosen at random. The first ciphertext block is computed as
$$x := m'_1 \oplus iv, \qquad C^{(1)} := \mathsf{Enc}(k, x). \tag{1}$$
- The subsequent ciphertext blocks $C^{(2)}, \ldots, C^{(d)}$ are computed as
$$x := m'_i \oplus C^{(i-1)}, \qquad C^{(i)} := \mathsf{Enc}(k, x) \tag{2}$$
for $i = 2, \ldots, d$.
- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(d)})$.

See Figure 1 for an illustration of this scheme. The decryption procedure reverses this process in the obvious way.

In the sequel we will write $C = \mathsf{Enc}_{\mathsf{cbc}}(k, \pi(m))$ to denote encryption and $\pi(m) = \mathsf{Dec}_{\mathsf{cbc}}(k, C)$ to denote decryption in CBC mode.

3. BASIC IDEA OF THE ATTACK

In this section, we describe the basic idea behind our attack, and give a toy example which illustrates how to apply this idea in a simplified scenario. In Section 4 we will show how to adapt this idea to the "real world".

3.1 CBC-Mode is Malleable

One ingredient to our attack described below is that ciphertexts encrypted in CBC mode can be modified by an adversary such that the resulting ciphertext is related to the original ciphertext in a certain way. This works for *any* block cipher.

Suppose a ciphertext $C = (IV, C^{(1)}, \ldots, C^{(d)})$ encrypting a message $m = (m^{(1)}, \ldots, m^{(d)})$ in CBC mode is given. Then a related ciphertext can be constructed as follows. Let $IV' := IV \oplus msk$ for some $msk \in \{0,1\}^n$. Then the ciphertext
$$(IV', C^{(1)})$$
is a valid encryption of the message $m^{(1)} \oplus msk$. This can be seen by inspecting Equations (1). Similarly, Equations (2) show that the ciphertext
$$(C^{(i-1)} \oplus msk, C^{(i)})$$
is a valid encryption of the message $m^{(i)} \oplus msk$ for all $i \in \{2, \ldots, d\}$. Here we use that the decryption algorithm interprets $C^{i-1} \oplus msk$ as an initialization vector, if the ciphertext starts with this value.

In our attack described below, we choose different values for msk and send the resulting ciphertext to the XML server. The response (i.e., the type of error message or application response) of the server allows us to learn the plaintext contained in a given ciphertext.

3.2 A Toy Example

In this section we describe a simple attack on ciphertexts encrypted in CBC mode, which allows to recover the plaintext message, if a certain *oracle* (to be described below) is given. The actual attack on XML Encryption from Section 4 is based on the same idea, but in addition handles some technical obstacles that arise when the theoretical concept is adopted to the "real world".

In the following let us assume that a plaintext consists only of 8-bit characters (e.g. ASCII), and that no padding scheme is used (i.e., the length of the encrypted data is always an integer multiple of the block-length of the cipher). Let us partition the set of all characters into two sets $\mathcal{A}$ and $\mathcal{B}$. We say that $\mathcal{A}$ contains "Type-A" characters, and $\mathcal{B}$ contains "Type-B" characters. In this toy example we assume that $\mathcal{A} = \{w\}$ contains only a *single* character w. For instance, $w =$ `0x00` may be the `NULL` character.

Definition 1. *We say that a ciphertext C is* well-formed *w.r.t. key k, if the plaintext $m = \mathsf{Dec}_{\mathsf{cbc}}(k, C)$ contains* only *Type-B characters.*

Let us assume that we are given a (not necessarily well-formed) ciphertext
$$C = (IV, C^{(1)}) = \mathsf{Enc}_{\mathsf{cbc}}(k, m)$$
consisting of an initialization vector IV and a single encrypted block $C^{(1)}$, which encrypts a message m. Furthermore, suppose that we may query an oracle $\mathcal{O}$. The oracle takes as input CBC-encrypted ciphertexts $C = (IV, C^{(1)})$. It computes the decryption $\mathsf{Dec}_{\mathsf{cbc}}(k, C)$ and replies as follows.

- $\mathcal{O}(C) = 1$, if the plaintext $m = \mathsf{Dec}_{\mathsf{cbc}}(k, C)$ contains *only* Type-B characters.
- $\mathcal{O}(C) = 0$, otherwise.

We will show how to use this oracle to recover the message m contained in $C = (IV, C^{(1)})$ byte-by-byte. To this end, we proceed in three steps.

1. Use the oracle to compute an initialization vector IV' such that $C' = (IV', C^{(1)})$ is well-formed.
2. Use the oracle to recover the CBC decryption intermediate value $x = \mathsf{Dec}(k, C^{(1)})$.
3. Recover the message m by computing $m = IV \oplus x$.

It is easy to compute an initialization vector IV' such that the ciphertext $C' = (IV', C^{(1)})$ is well-formed. To this end, we can first query the oracle whether $\mathcal{O}((IV, C^{(1)})) = 1$. In this case we can set $IV' := IV$. Otherwise we set IV' to a random bit string. The probability Π that this yields a well-formed ciphertext depends on the number ν of bytes per block. For the current definition of $\mathcal{A}$, the probability is equal to

$$\Pi(\nu) = (1 - 1/256)^{\nu}.$$

Thus, when using AES we have $\Pi(16) = (1 - 1/256)^{16} \approx 0.94$, while using 3DES we have $\Pi(8) \approx 0.97$. Thus, we can expect that we find a suitable IV' after a few trials. We query the oracle to test whether we are successful.

Now we have a well-formed ciphertext $(IV', C^{(1)})$. Next we show how to recover an arbitrary byte x_j of the CBC decryption intermediate value $x = \mathsf{Dec}(k, C^{(1)})$. To this end, let us write $IV' = (IV'_1, \ldots, IV'_\nu)$ and $IV'' = (IV''_1, \ldots, IV''_\nu)$ to denote the individual bytes of IV' and IV''. Next we modify the initialization vector IV' by XOR-ing a byte-mask msk to the j-th byte of IV', until a mask msk is found such that

$$\mathsf{Dec}_{\mathsf{cbc}}(k, (IV'', C^{(1)})) = IV'' \oplus \mathsf{Dec}(k, C^{(1)}) = IV'' \oplus x$$

contains a character from $\mathcal{A} = \{w\}$. Since we have only modified the j-th byte of IV', we can conclude that

$$w = IV''_j \oplus x_j.$$

Thus we can recover x_j by computing $x_j = w \oplus IV''_j$. Since this procedure works for all j, we can thus determine x byte-wise.

Algorithm 1 Recovering x_j.

Input: A single-block ciphertext $C' = (IV', c^{(1)})$ and an index $j \in \{1, \ldots, \nu\}$.
Output: The j-th byte x_j of $x = \mathsf{Dec}(k, C^{(1)})$.

1: $msk := \texttt{0x00}$
2: **repeat**
3: $\quad msk := msk + 1$
4: $\quad IV'' := IV' \oplus 0^{8(j-1)} || msk || 0^{n-8j}$
5: **until** $\mathcal{O}((IV'', C^{(1)})) = 1$
6: **return** $x_j := w \oplus IV''_j$

Finally, if we are given $x = \mathsf{Dec}(k, C^{(1)})$, then we can recover the message m contained in the original ciphertext $(IV, C^{(1)})$ by computing $m = IV \oplus x$. The process of recovering x_1 is illustrated in Figure 2.

4. ATTACKING XML ENCRYPTION

In this section we show how to apply the attack described above on a real world Web Service framework. We have chosen Apache Axis2 [18], since it is one of the major frameworks. We start with some basics on how encryption in XML documents works, and how Axis2 handles incoming XML-based messages. These messages are formatted according to the SOAP standard [8], the details of this format are not relevant here and thus omitted. Then we describe how we can use such an Axis2-based Web Service endpoint as an oracle $\mathcal{O}_{\mathsf{Axis}}$ for our attack. Finally, we describe algorithms that execute the attack using this oracle by first preparing a given multi-block ciphertext (by e.g. adjusting the padding) and then recovering the plaintext byte-wise from such a prepared ciphertext.

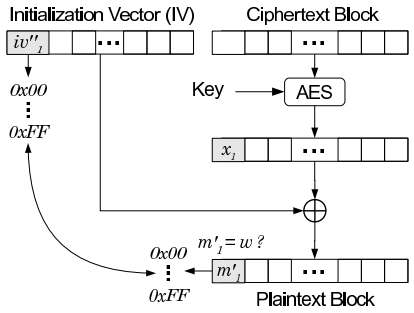

Figure 2: Using the malleability of CBC and the oracle $\mathcal{O}$ to recover x_1.

```
<EncryptedData
       Type='http://www.w3.org/2001/04/xmlenc#Element'
       xmlns='http://www.w3.org/2001/04/xmlenc#'>
   <EncryptionMethod
         Algorithm='http://www.w3.org...#aes256-cbc'>
   </EncryptionMethod>
   <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <KeyName>John Smith</KeyName>
   </KeyInfo>
   <CipherData>
      <CipherValue>A23B45C56...</CipherValue>
   </CipherData>
</EncryptedData>
```

Figure 3: Example of an XML Encryption metadata block

4.1 XML and XML Encryption

XML. The *eXtensible Markup Language* [2], or XML for short, is a syntax for serializing tree-shaped data structures. An essential characteristic of the XML syntax is that the use of the `'<'` and `'>'` characters is allowed only for the declaration of XML element nodes. Hence, if a text node happens to contain any of these characters prior to XML serialization, it must be escaped to a different set of characters (here: `<` and `>`). In the same way, the ampersand (`&`) is escaped to `&`. These properties of XML are relevant to the attack description.

XML Encryption. Along with specifying XML, the W3C fostered the standardization of means to apply cryptographic primitives like digital signatures and encryption to arbitrary XML data. The resulting specifications were the *W3C XML Signature* recommendation [7] and the *W3C XML Encryption* recommendation [6].

We focus on the XML Encryption specification, of which a syntax example is given in Figure 3. As can be seen, the XML Encryption specification describes the encryption metadata, such as encryption key identifiers, algorithms, encryption schemes, etc. The most important part, however, is the `<CipherValue>` element, which contains the ciphertext of the original data.

When receiving an encrypted message, it is processed as follows. At first, it is necessary to search the whole XML document for `<EncryptedData>` elements. For each of these, the metadata elements containing key information must be parsed, processed, and used to gain the proper cryptographic

key required for decryption. The mechanism for this task is out of scope for this paper, it can be assumed that the processing application has means to determine and gather the correct key based on the information from the `<KeyInfo>` element. Next, the content of the `<CipherValue>` is extracted and decoded in order to gather the ciphertext. Then, the decryption itself can be performed, using the algorithm specified in `<EncryptionMethod>`. Finally—and this is a crucial step for the attack described below—the decrypted contents are parsed and put back to the XML document tree. If an error occurs during the decryption or subsequent parsing process, this error is propagated to the XML processor, which typically raises an exception for the application to handle.

Since XML Encryption can be applied to arbitrary subtrees of an XML document tree, there are different types of encryption modes depending on the nature of the encrypted contents. *Encrypted Element* implies that a single XML element including all of its descendant nodes has been encrypted. *Encrypted Content* states that the ciphertext may contain arbitrary node types, e.g. multiple subsequent XML elements, comments, processing instructions etc. *Encrypted Text Content* is a special case of *encrypted content*, and implies that the ciphertext is completely made of text contents.

Though the `Type` attribute of the `<EncryptedData>` element gives a hint on which of these three cases applies, the common XML decryption frameworks in general do not treat them differently. Decryption, parsing, and processing works identical for all these cases.

4.2 Character Encondings

The XML Encryption standard prescribes that characters and symbols are encoded according to the UTF-8 code, which specifies a bit-representation of characters from various alphabets (latin, greek, cyrillic, hebrew, arabic, and countless more) plus many special symbols (e.g. from mathematics or music). The most important subset of UTF-8 characters consists of latin characters, arabic numerals, and some special symbols like `line feed` and `carriage return`. It is important to know that for these characters UTF-8 is identical to ASCII.

The ASCII code represents characters as single bytes, and allows to encode $2^7 = 128$ different characters, as depicted in Figure 4. Note that the encoding uses only 7 out of 8 bits, the most-significant bit is always equal to 0 for all ASCII characters.

Figure 4 lists also a classification of ASCII characters into "Type-A" and "Type-B" characters, which is not relevant yet, but will be useful to explain our attack.

4.3 The Axis2 Web Services Framework

In recent years many Web Services frameworks emerged [17, 16, 9]. One of the most popular open source Web Services Framework is Apache Axis2 [18]. The Rampart module of Axis2 contains an implementation of the WS-Security [12] standard, which allows to apply XML Encryption and XML Signature in SOAP messages.

To use a module in the Axis2 framework, the module must be engaged to the Axis2's message *flow*. A flow is a collection of modules, where each module takes the incoming SOAP message context, processes it, and passes it to the next module. When the SOAP message comes to the end of the flow, it is forwarded to a Message Receiver. The Message Receiver invokes the function implemented in the Service class and passes the result to the output flow.

The Axis2 flow consists typically of three modules, namely Transport, Security, and Dispatch. The Security module processes the security elements. In particular, encrypted elements are first *decrypted* and then *parsed* by an XML parser in order to update the SOAP message context. The decrypted and validated content is then passed on to the Dispatch module. Each module in the flow and the Message Receiver can stop the SOAP message processing if an error occurs. In this case the processing is terminated and an appropriate SOAP fault is returned.

We distinguish between two types of server responses. We say that a `security fault` is returned, if the server replies with a `WSDoAllReceiver: security processing failed` message. If an application-specific error or no error message is returned, then we say that the server replies with an `application response`.

4.4 Axis2 Security Faults

To construct our Axis2-based oracle $\mathcal{O}_{\mathsf{Axis}}$, we evaluate the SOAP faults returned by the security handler of an Axis2 server. This handler returns a `security fault` fault whenever a problem during the processing of security elements in a message occurs. This fault can have several reasons, which can be divided into two categories:

Decryption error. This results from incorrect padding.

> Recall that the last byte of a padded plaintext must include a valid padding number in the range from `0x01` (indicating only the last byte is padded) to `0x10` (indicating the whole last block is a padding block).

Parsing error. This error may have two reasons.

> Either the plaintext contains an "invalid" character. Invalid charactes are all ASCII characters from `0x00` to `0x1F`, except for `0x09` (`horizontal tab`), `0x0A` (`line feed`), and `0x0D` (`carriage return`).
>
> The other reason is that the syntax of the decrypted XML part is not valid. The latter means that the special escape character `0x26` (`&`) is not followed by a valid entity reference, or the bracket `0x3C` (`<`) is not properly closed.

Since in both cases the *same* error message is returned, we cannot distinguish between them.

4.5 An Axis-based Oracle

We classify the set of ASCII characters in two categories, which we call "Type-A" and "Type-B" characters. This classification is depicted in Figure 4. We denote with $\mathcal{A}$ the set of all Type-A characters, and with $\mathcal{B}$ the set of all Type-B characters. Observe that $\mathcal{A}$ contains primarily "invalid" characters, plus the reserved XML characters "`&`" and "`<`".

Based on this classification, we can construct an oracle $\mathcal{O}_{\mathsf{Axis}}$, which is similar to the oracle $\mathcal{O}$ from Section 3.2, as follows. $\mathcal{O}_{\mathsf{Axis}}$ takes as input a CBC-encrypted ciphertext $C = (IV, C^{(1)})$, which consists of an initialization vector IV and a single ciphertext block $C^{(1)} \in \{0,1\}^n$. It embeds the ciphertext C into a SOAP message, sends this document to the Axis2 server, and replies as follows.

Dec.	Hex	Char.	Type	Dec.	Hex	Char.	Type	Dec.	Hex	Char.	Type	Dec.	Hex	Char.	Type
Block 0				Block 2				Block 4				Block 6			
0	00	NUL	A	32	20	SPC	B	64	40	@	B	96	60	‘	B
1	01	SOH	A	33	21	!	B	65	41	A	B	97	61	a	B
2	02	STX	A	34	22	"	B	66	42	B	B	98	62	b	B
3	03	ETX	A	35	23	#	B	67	43	C	B	99	63	c	B
4	04	EOT	A	36	24	$	B	68	44	D	B	100	64	d	B
5	05	ENQ	A	37	25	%	B	69	45	E	B	101	65	e	B
6	06	ACK	A	38	26	&	**A**	70	46	F	B	102	66	f	B
7	07	BEL	A	39	27	’	B	71	47	G	B	103	67	g	B
8	08	BS	A	40	28	(	B	72	48	H	B	104	68	h	B
9	09	HT	**B**	41	29	)	B	73	49	I	B	105	69	i	B
10	0A	LF	**B**	42	2A	*	B	74	4A	J	B	106	6A	j	B
11	0B	VT	A	43	2B	+	B	75	4B	K	B	107	6B	k	B
12	0C	FF	A	44	2C	,	B	76	4C	L	B	108	6C	l	B
13	0D	CR	**B**	45	2D	-	B	77	4D	M	B	109	6D	m	B
14	0E	SO	A	46	2E	.	B	78	4E	N	B	110	6E	n	B
15	0F	SI	A	47	2F	/	B	79	4F	O	B	111	6F	o	B
Block 1				Block 3				Block 5				Block 7			
16	10	DLE	A	48	30	0	B	80	50	P	B	112	70	p	B
17	11	DC1	A	49	31	1	B	81	51	Q	B	113	71	q	B
18	12	DC2	A	50	32	2	B	82	52	R	B	114	72	r	B
19	13	DC3	A	51	33	3	B	83	53	S	B	115	73	s	B
20	14	DC4	A	52	34	4	B	84	54	T	B	116	74	t	B
21	15	NAK	A	53	35	5	B	85	55	U	B	117	75	u	B
22	16	SYN	A	54	36	6	B	86	56	V	B	118	76	v	B
23	17	ETB	A	55	37	7	B	87	57	W	B	119	77	w	B
24	18	CAN	A	56	38	8	B	88	58	X	B	120	78	x	B
25	19	EM	A	57	39	9	B	89	59	Y	B	121	79	y	B
26	1A	SUB	A	58	3A	:	B	90	5A	Z	B	122	7A	z	B
27	1B	ESC	A	59	3B	;	B	91	5B	[	B	123	7B	{	B
28	1C	FS	A	60	3C	<	**A**	92	5C	\	B	124	7C	|	B
29	1D	GS	A	61	3D	=	B	93	5D	]	B	125	7D	}	B
30	1E	RS	A	62	3E	>	B	94	5E	^	B	126	7E	~	B
31	1F	US	A	63	3F	?	B	95	5F	_	B	127	7F	DEL	B

Figure 4: ASCII Character Encoding Table and Classification of Characters.

- $\mathcal{O}_{\mathsf{Axis}}(C) = 1$, if the Axis2 server returns a `security fault`.
- $\mathcal{O}_{\mathsf{Axis}}(C) = 0$, otherwise.

As described in Section 4.3, the Axis2 server will return no `security fault`, if

- the decryption $\pi(m) = IV \oplus \mathsf{Dec}(k, C^{(1)})$ yields a message with *valid padding*, and
- the plaintext $m = \pi^{-1}(\mathsf{Dec}_{\mathsf{cbc}}(k, C))$ has a valid XML structure. That is,
  - if m contains an XML tag `<a>` for some string `a`, then it must also contain the corresponding closing tag `</a>`,
  - if m contains the `&` ampersand character, then it must be a valid entity reference, like `&gt;` for instance,
  - m does not contain any characters from `0x00` to `0x1F`, except for `0x09` or `0x0A` or `0x0D`.

Otherwise a `security fault` is returned. This allows us to use $\mathcal{O}_{\mathsf{Axis}}$ in a way similar to the oracle $\mathcal{O}$ from Section 3.2.

4.6 Using $\mathcal{O}_{\mathsf{Axis}}$ to recover plaintexts

In this section we describe an algorithm that uses the $\mathcal{O}_{\mathsf{Axis}}$ oracle to decrypt a given ciphertext $C = (IV, C^{(1)}, \ldots, C^{(d)})$. Note that C may consist of multiple blocks (i.e., $d \geq 1$). Due to the rather complex structure of the set $\mathcal{A}$ and some optimizations to reduce the number of oracle queries, this procedure is rather complex. For better readability, we present only simplified algorithms, which illustrate the basic attack idea better. However, we have implemented the optimized algorithms. We will furthermore make the (in practice reasonable) assumption that the plaintexts contains only ASCII characters, but no characters from the extended character set of UTF-8. The attack can however be extended to arbitrary UTF-8 characters.

First we need a new definition of well-formedness.

Definition 2. *We say that a single-block ciphertext $C = (IV, C^{(1)})$ is* well-formed *w.r.t. key k, if*

$$m = (m_1, \ldots, m_\nu) = \mathsf{Dec}_{\mathsf{cbc}}(k, C)$$

has a single byte padding (i.e. $m_\nu = \texttt{0x01}$) and consists only of Type-B characters (i.e. $m_j \in \mathcal{B}$ for all $j \in \{1, \ldots, \nu-1\}$).

The algorithm is a composition of two sub-procedures, which we call FindIV and FindXbyte.

- The FindIV procedure prepares the ciphertext for our attack. It takes as input a multi-block ciphertext $C = (IV, C^{(1)}, \ldots, C^{(d)})$ and an index $i \in \{1, \ldots, d\}$, and returns an initialization vector iv such that the ciphertext $c = (iv, C^{(i)})$ is well-formed.
- The FindXbyte procedure takes as input an index $j \in \{1, \ldots, \nu\}$ and a well-formed (w.r.t. the target key) single-block ciphertext $c = (iv, c^{(1)})$ such that $c^{(1)} = C^{(i)}$ (as provided by the FindIV procedure). It returns the j-th byte x_j of the CBC decryption intermediate value

$$x = (x_1, \ldots, x_\nu) = \mathsf{Dec}(k, C^{(i)}).$$

Using these procedures, Algorithm 2 recovers the plaintext m contained in C. The algorithm loops through all d ciphertext blocks of C, each time performing essentially three steps.

1. First, it calls the FindIV procedure, which computes an initialization vector iv such that $c = (iv, C^{(i)})$ is a well-formed ciphertext (Line 2).
2. Then it runs the FindXbyte procedure ν times to recover all ν decryption intermediate values
$$x^{(i)} = (x_1^{(i)}, \ldots, x_\nu^{(i)}) = \mathsf{Dec}(k, C^{(i)})$$
(Lines 3 to 5).
3. Knowledge of $x^{(i)} = \mathsf{Dec}(k, C^{(i)})$ allows us to recover the i-th plaintext block as
$$\begin{aligned} m^{(i)} &= \mathsf{Dec}(k, C^{(i)}) \oplus C^{(i-1)} \\ &= x^{(i)} \oplus C^{(i-1)} \end{aligned}$$
(Lines 6 and 7).

Algorithm 2 Using $\mathcal{O}_{\mathsf{Axis}}$ to recover plaintexts.

Input: $C = (C^{(0)} = IV, C^{(1)}, \ldots, C^{(d)})$
Output: $m = (m^{(1)}, \ldots, m^{(d)})$
1: **for** $i = 1$ **to** d **do**
2: $iv := \mathsf{FindIV}(C, i)$
3: **for** $j = 1$ **to** ν **do**
4: $x_j^{(i)} := \mathsf{FindXbyte}(C^{(i)}, iv, j)$
5: **end for**
6: $x^{(i)} := (x_1^{(i)}, \ldots, x_\nu^{(i)})$
7: $m^{(i)} := x^{(i)} \oplus C^{(i-1)}$
8: **end for**
9: **return** $(m^{(1)}, \ldots, m^{(d)})$

Note that the above algorithm makes exactly d calls to the FindIV procedure and $d \cdot \nu$ calls to the FindXbyte procedure.

4.6.1 Procedure FindIV

In this section we describe the FindIV procedure. This procedure takes as input a ciphertext $C = (C^{(0)}, C^{(1)}, \ldots, C^{(d)})$ and an index i, and returns an initialization vector iv such that $(iv, C^{(i)})$ is a well-formed ciphertext.

For simplicity we explain the algorithm for the case where a block cipher with block size $\nu = 16$ bytes is used. This corresponds to the case where AES is used in the XML Encryption standard. With a few minor changes the procedure can be adapted to ciphertexts of arbitrary block length. Moreover, we suppose the input ciphertext has the following properties:

- The plaintext of $C' = (C^{(i-1)}, C^{(i)})$ does not contain any "Type-A" character, except for (possibly) the "<" character.
- Each encrypted block contains only incomplete XML elements (i.e. there exists no start tag followed by an element content and an end tag).

If the input ciphertext meets these ciphertext properties, then there are two more issues our FindIV procedure must solve. First, it has to remove all occurrences of the "<" character. Thus we obtain an *encrypted text content* ciphertext consisting only of characters from $\mathcal{B}$. Second, it sets the last byte of the newly created iv so that the padding byte becomes equal to `0x01`. Thus, we obtain a valid padded ciphertext with a single padding byte. These are exactly the prerequisites for our FindXbyte procedure.

We start the description of our FindIV procedure with an observation on the padding byte. The padding byte can be modified by changing the last byte of $C^{(i-1)}$. When we iterate over all the 256 possible values for the last byte $C_\nu^{(i-1)}$ of $C^{(i-1)}$, then we implicitly modify the last byte of the (padded) plaintext contained in $(C^{(i-1)}, C^{(i)})$. Note that in the case $\nu = 16$ there are at most 16 valid padding bytes, namely all values from `0x01` (one padding byte) to `0x10` (all 16 bytes are padding).

First observe that, since we know that the first bit of any ASCII character and any valid padding byte is always equal to 0, we can copy the first bit from the last byte of the original initialization vector $C^{(i-1)}$. Our algorithm iterates only over the remaining at most 128 possible values of the last byte of $C^{(i-1)}$.

Observe now that, if the plaintext of $(IV, C^{(i)})$ contains only characters from $\mathcal{B}$ (no "<" character), then $\mathcal{O}_{\mathsf{Axis}}$ returns exactly $\nu = 16$ responses such that $\mathcal{O}_{\mathsf{Axis}}(C) = 0$. Otherwise, the number of $\mathcal{O}_{\mathsf{Axis}}(C) = 0$ responses *depends on the position* of the first "<" character in the block. For example, if we get only one $\mathcal{O}_{\mathsf{Axis}}(C) = 0$ response, then this means that only one padding byte, namely `0x10` is valid. This implies that the first byte of the plaintext is equal to the "<" character. Similarly, if we get three $\mathcal{O}_{\mathsf{Axis}}(C) = 0$ responses it means that the paddings `0x10`, `0x0F`, and `0x0E` are valid and the "<" character stands on the third position.

For the purpose of getting the number of valid padding bytes we introduce the procedure in Algorithm 3, which collects all the valid padding masks.

Algorithm 3 GetValidPaddingMasks

Input: $IV, C^{(i)}$
Output: A set of valid padding masks $Pset$
1: $Pset := \emptyset$
2: **for** $j := 0x00$ **to** $0x7F$ **do**
3: $IV' := IV \oplus (0^{n-8} || j)$
4: **if** $\mathcal{O}_{\mathsf{Axis}}(IV', C^{(i)}) = 0$ **then**
5: $Pset := Pset \cup IV'_\nu$
6: **end if**
7: **end for**
8: **return** $Pset$

Algorithm 3 computes a set of valid padding masks $Pset$. If $Pset$ contains $\nu = 16$ elements, then this tells us that the block does not include any "<" character. Otherwise, we learn that the "<" character stands on the position $pos := |Pset|$. We can simply change the "<" character by flipping the last bit of the IV_{pos}. We repeat the `GetValidPaddingMasks` procedure and the bit flipping until $|Pset| = \nu$.

After extraction of all the "<" characters, we set the last byte of iv to cause the padding `0x01`. To this end, we introduce another procedure, called `GetIvWithPaddingMask01`. This procedure gets as input IV and $Pset$ of $\nu = 16$ valid padding masks $msk0x01 \ldots msk0x10$. Since the padding mask $msk0x10$ differs from other paddings in the 4-th bit, we can distinguish it from other padding masks. Therefore,

we can simply set the iv to:

$$iv := IV \oplus (0^{n-8}||(msk0x10 \oplus 0x11))$$

The complete procedure is depicted in Algorithm 4.

Algorithm 4 FindIV

Input: A ciphertext $C = (C^{(i-1)}, C^{(i)})$
Output: iv
1: $IV := C^{(i-1)}$
2: **repeat**
3: $Pset := GetValidPaddingMasks(IV, C^{(i)})$
4: $pos := |Pset|$
5: $IV_{pos} := IV_{pos} \oplus$ `0x01`
6: **until** $|Pset| = \nu$
7: $iv := GetIvWithPaddingMask01(PSet, IV)$
8: **return** iv

Extension of FindIV for arbitrary XML data. In the description of the FindIV procedure we had to make some restrictions on the encrypted plaintext. It is possible to overcome these restrictions using very similar techniques. We omit the details due to space limitations.

4.6.2 Procedure FindXbyte

In this section we describe a procedure FindXbyte which takes as input a single-block ciphertext $C^{(i)}$, an initialization vector iv such that $c = (iv, C^{(i)})$ is well-formed, and an index $j \in \{1, \ldots, \nu\}$ (as provided by FindIV). The procedure returns the j-th byte of $x^{(i)} = \mathsf{Dec}(k, C^{(i)})$. Note that we have

$$\mathsf{Dec}(k, C^{(i)}) \oplus iv = \mathsf{Dec}_{\mathsf{cbc}}(k, (iv, C^{(i)})).$$

If $j = \nu$, i.e., the procedure is asked to return the ν-th byte of $x^{(i)}$, then the algorithm can compute $x^{(i)}_\nu$ without even querying the oracle. Recall that we know that the last plaintext-byte contained in c is the single-byte padding `0x01`. Thus we have

$$\begin{aligned}(??, \ldots, ??, \texttt{0x01}) &= \mathsf{Dec}_{\mathsf{cbc}}(k, (iv, C^{(i)})) \\ &= x^{(i)} \oplus iv.\end{aligned}$$

This enables us to recover $x^{(i)}_\nu$ as $x^{(i)}_\nu = \texttt{0x01} \oplus iv_\nu$.

If $j \in \{1, \ldots, \nu - 1\}$, then we need to use the $\mathcal{O}_{\mathsf{Axis}}$ oracle to recover $x^{(i)}_j$. We do this by modifying the initialization iv and evaluating the response behaviour of the oracle. Let us write $iv = (iv_1, \ldots, iv_\nu)$ to denote individual bytes of iv, and $(iv_{j,1}, \ldots, iv_{j,8})$ to denote individual bits of byte iv_j. Similarly, we write $x^{(i)}_j = (x^{(i)}_{j,1}, \ldots, x^{(i)}_{j,8})$ to denote individual bits of byte $x^{(i)}_j$.

Determining the first bit $x^{(i)}_{j,1}$ is easy, since the first bit $m^{(i)}_{1,j}$ of an ASCII-encoded character $m^{(i)}_j$ is always equal to 0. Since we have

$$x^{(i)} = iv^{(i)} \oplus m^{(i)},$$

which implies that $x^{(i)}_{j,1} = iv^{(i)}_{j,1}$ for all $i \in \{1, \ldots, d\}$ and $j \in \{1, \ldots, \nu\}$.

To describe our algorithm to determine the remaining bits, let us divide the ASCII table into blocks, as depicted in Figure 4. The first four bits of an ASCII character determine to which block it belongs. For instance, `0x5A` is a character from Block 5, `0x35` is from Block 3, and so on. This leads us to the following observations on the distribution of Type-A characters.

- Only Blocks 0 to 3 contain Type-A characters.
- There is only one block which *does not* contain any Type-B character, namely Block 1.
- Blocks 2 and 3 contain *exactly one* Type-A character, namely `0x26` in Block 1 and `0x3C` in Block 2.
- The last four bits of `0x26` and `0x3C` are *not equal*.

We use these observations as follows. In order to determine $x^{(i)}_{j,2}, x^{(i)}_{j,3}, x^{(i)}_{j,4}$, we first run Algorithm 5 to compute a set $Aset$ of bit masks. This algorithm initializes set $Aset$ to the empty set (Line 1). Then, by looping through all possible masks $msk \in \{\texttt{0x00}, \texttt{0x10}, \texttt{0x20}, \ldots, \texttt{0x70}\}$, the algorithm modifies the bits of the initialization vector which correspond to the bits $x^{(i)}_{j,2}, x^{(i)}_{j,3}, x^{(i)}_{j,4}$ (Line 4). The algorithm queries $\mathcal{O}_{\mathsf{Axis}}$ to test whether

$$\tilde{m}^{(i)}_j = x^{(i)}_j \oplus (iv_j \oplus msk)$$

is a Type-A character (Line 5). If true, the algorithm stores the corresponding mask msk in $Aset$ (Line 6).

Algorithm 5 Computing the set $Aset$.

Input: $c = (iv, C^{(i)})$, $j \in \{1, \ldots, \nu\}$
Output: Set $Aset \subseteq \{0, \ldots, 7\}$
1: $Aset := \emptyset$
2: **for** R $= 0$ **to** 7 **do**
3: $msk :=$ `0xR0`
4: $iv' := iv \oplus 0^{8(j-1)}||msk||0^{n-8j}$
5: **if** $\mathcal{O}_{\mathsf{Axis}}((iv', C^{(i)})) = 1$ **then**
6: $Aset := Aset \cup \{\texttt{msk}\}$
7: **end if**
8: **end for**
9: **return** $Aset$

We can now observe that the set $Aset$ returned by Algorithm 5 contains always either one or two or three elements. To see this, recall that the last four bits of $m^{(i)}_j$ are never modified. Therefore we have

- $|Aset| = 1$ if and only if the *last* four bits of $m^{(i)}_j$ are equal to `0x?9`, or `0x?A`, or `0x?D` (see the Type-B characters in Block 0).
- $|Aset| = 3$ if and only if the *last* four bits of $m^{(i)}_j$ are equal to `0x?6` or `0x?C` (see the Type-A characters in Block 2 and Block 3).
- $|Aset| = 2$ otherwise.

If $|Aset| = 1$ and $Aset = \{msk\}$, then we learn that $m^{(i)}_j \oplus msk \in \{\texttt{0x19}, \texttt{0x1A}, \texttt{0x1D}\}$. Now observe that there is exactly one mask $msk' \in \{\texttt{0x25}, \texttt{0x26}, \texttt{0x21}\}$ such that $m^{(i)}_j \oplus msk \oplus msk' =$ `0x3C` is a Type-A character. Again we can use the oracle to determine msk'. This gives us an equation

$$x^{(i)}_j \oplus (iv_j \oplus msk) \oplus msk' = \texttt{0x3C}$$

Plaintext size (bytes)	Server requests			Time (seconds)
	FindIV	FindXbyte	Total	
16	32	130	162	0.975
160	449	1688	2137	9.6
1,600	7,453	16,356	23,809	98
16,000	81,155	161,433	242,588	1,039

Figure 5: Summary of Experimental Analysis

where only $x_j^{(i)}$ is unknown. Thus we can recover byte $x_j^{(i)}$. Note that in the case $|Aset| = 1$ we need at most two oracle queries to determine msk'.

If $|Aset| = 2$ or $|Aset| = 3$, then a procedure which applies the same principle as the above, but is slightly more complex, allows us to recover $x_j^{(i)}$, by issuing at most 23 oracle queries in total.

4.7 Attack Variations

A priori knowledge about the plaintext could improve the attack significantly. For instance, knowing the XML Schema which defines the structure of the XML document, one could skip decryption of blocks that contain the known plaintext, like XML element tags, and focus on the blocks that contain unknown contents. In the same line, if the attacker knows *a priori* that an encrypted text is, for instance, a credit card number, he can rule out any potential plaintext character that is not a digit.

Furthermore, it is obvious that knowledge of the $x^{(i)}$ bytes also allows an attacker to encrypt arbitrary messages. To this end, the attacker proceeds "from right to the left", i.e., starting with the last ciphertext block (see also [15]).

5. EXPERIMENTAL ANALYSIS

In order to investigate the feasibility and performance of our approach we developed a proof-of-concept implementation of the algorithms decribed in the previous section. We have implemented slightly optimized variants of the presented algorithms.

We measured the time and the number of server requests sent for different ciphertext sizes. Our implementation uses Java 6 and for the SOAP message handling, using a single thread. As a Web Services server, we used the recent Apache Axis2 Version 1.5.3 with Apache Rampart 1.5 module. Both application and Axis2 server were running on a single machine. For completeness, the machine was equipped with Linux Ubuntu 10.10 and Intel Core2 Duo P9700 processor (2 cores with 2.80 GHz).

SETTING. We implemented a simple Java class and deployed it on the Apache Axis2 server to create a Web Service endpoint. We secured the generated Web Services endpoint with the default XML Encryption setting so that the Axis2 server accepted only the SOAP messages with encrypted SOAP body. We used the AES block cipher with 128-bit key (but everything works the same way with 256-bit key).

We generated messages of various length and structure. The shortest messages consisted of 10 characters, thus fitted in a single AES block, and no '<' characters. Larger SOAP messages contained two '<' characters in each cipher block. The symmetric key was encrypted with the public key of the Axis2 server and put into the header of the SOAP message.

RESULTS. The results of our analysis with messages of size 1, 10, 100, and 1000 blocks are depicted in Figure 5. The first two columns in the table describe the plaintext length. The third column shows the number of requests sent by FindIV and FindXbyte. The overall time for the attack execution is listed in the last column.

ANALYSIS. Our results show the practicality of our attack. A single encrypted block can be decrypted with only 162 requests in less than one second. Moreover, it is also feasible to decrypt larger ciphertexts, decrypting 16,000 bytes takes only 17 minutes.

As we executed the tests on a single machine, the timing results are only approximate. For instance, one has to consider the delay by transporting the SOAP message over the network to the server and back. On the other hand, usage of a more powerful server would speed up the message processing.

In any case the experimental results show that the attack is applicable in practical real world scenarios, not only for very short messages but also for larger plaintexts.

6. COUNTERMEASURES

6.1 XML Signature on Ciphertext

The XML Signature standard [7] describes a method to digitally sign all request messages for ensuring their integrity, either using a public-key digital signature scheme or a secret-key message authentication code. A candidate countermeasure would be to apply XML Signatures in order to ensure the integrity of messages, and thus to thwart our attack. However, in the following we illustrate that unfortunately it is not that trivial in practice.

XML SIGNATURE WRAPPING. It is well-known that many Web Services implementations are vulnerable to *XML Signature Element Wrapping* attacks [10]. These attacks allow to modify a ciphertext without breaking the security of the digital signature scheme or message authentication code.

XML ENCRYPTION WRAPPING. Beyond the classical XML Signature Wrapping attacks, we furthermore developed a novel class of attacks that allow to mount our attack even if signature wrapping attacks are not applicable. We call these attacks *XML Encryption wrapping*. To illustrate the idea, consider a standard scenario where the SOAP body contains data encrypted with a symmetric key. This symmetric key is encrypted with the server's public key and put into the SOAP header. Along with the encrypted key the SOAP header includes a reference list, which tells the server which elements must be decrypted using the symmetric key. The whole SOAP body including the encrypted data is signed. We observed that it is possible to copy the encrypted data to the SOAP header and insert a new element to the encrypted data reference list. This forces the server to process both EncryptedData elements, and thus allows to bypass the XML Signature validation to perform the attack.

ANALYSIS. We have evaluated Apache Axis2 and JBossWS regarding their resistance against XML Signature Wrapping and Encryption Wrapping attacks. Our experiments showed that all Rampart configuration examples and all JBoss standard examples are vulnerable, and conjecture therefore that the majority of all practical applications is vulnerable to this attack.

6.2 Unified 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 responds with a SOAP fault *if and only if* the encrypted data in the SOAP header are incorrect.

6.3 Changing the Mode of Operation

One possibility to avoid our attack is to use a *symmetric* cryptographic primitive that does not only provide confidentiality, but also integrity. This can for instance be achieved by replacing the CBC mode of operation with a mode that provides message integrity. Adequate choices have for instance been standardized in ISO/IEC 19772:2009. We consider this solution as very recommendable for future versions of the XML Encryption standard. Unfortunately, this may bring deployment and backwards compatibility issues.

7. ACKNOWLEDGEMENTS

We would like to thank Meiko Jensen, Jörg Schwenk, Thorsten Holz, Sebastian Schinzel, and Stefan Wigchers for their contributions, and the anonymous reviewers for helpful remarks.

8. REFERENCES

[1] Black, J., and Urtubia, H. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *USENIX Security Symposium* (2002), D. Boneh, Ed., USENIX, pp. 327–338.

[2] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation* (2008).

[3] Degabriele, J. P., and Paterson, K. G. Attacking the IPsec standards in encryption-only configurations. In *IEEE Symposium on Security and Privacy* (2007), IEEE Computer Society, pp. 335–349.

[4] Degabriele, J. P., and Paterson, K. G. On the (in)security of IPsec in MAC-then-encrypt configurations. In *ACM Conference on Computer and Communications Security* (2010), E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM, pp. 493–504.

[5] Duong, T., and Rizzo, J. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *IEEE Symposium on Security and Privacy* (2011).

[6] Eastlake, D., Reagle, J., Imamura, T., Dillaway, B., and Simon, E. XML Encryption Syntax and Processing. *W3C Recommendation* (2002).

[7] Eastlake, D., Reagle, J., Solo, D., Hirsch, F., and Roessler, T. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation* (2008).

[8] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. F. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation* (2003).

[9] JBoss Community. JBoss WS (Web Services Framework for JBoss AS).

[10] McIntosh, M., and Austel, P. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM Press, pp. 20–27.

[11] Mitchell, C. J. Error oracle attacks on cbc mode: Is there a future for cbc mode encryption? In *ISC* (2005), pp. 244–258.

[12] Nadalin, A., Kaler, C., Monzillo, R., and Hallam-Baker, P. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard* (2006).

[13] Paterson, K. G., and Watson, G. J. Immunising CBC mode against padding oracle attacks: A formal security treatment. In *SCN* (2008), R. Ostrovsky, R. D. Prisco, and I. Visconti, Eds., vol. 5229 of *Lecture Notes in Computer Science*, Springer, pp. 340–357.

[14] Paterson, K. G., and Yau, A. Padding oracle attacks on the ISO CBC mode encryption standard. In *Topics in Cryptology – CT-RSA 2004* (Feb. 2004), T. Okamoto, Ed., vol. 2964 of *Lecture Notes in Computer Science*, Springer, pp. 305–323.

[15] Rizzo, J., and Duong, T. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–8.

[16] Robert A. van Engelen. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications.

[17] Thai, T., and Lam, H. *.NET Framework Essentials*. 2001.

[18] The Apache Software Foundation. Apache Axis2.

[19] Vaudenay, S. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In *Advances in Cryptology – EUROCRYPT 2002* (Apr. / May 2002), L. R. Knudsen, Ed., vol. 2332 of *Lecture Notes in Computer Science*, Springer, pp. 534–546.

[20] Yau, A. K. L., Paterson, K. G., and Mitchell, C. J. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In *Fast Software Encryption – FSE 2005* (Feb. 2005), H. Gilbert and H. Handschuh, Eds., vol. 3557 of *Lecture Notes in Computer Science*, Springer, pp. 299–319.