On the insecurity of XML Security

Thesis *in* it - Information Technology · July 2013

DOI: 10.1515/itit-2014-1045

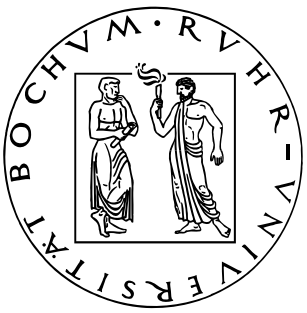
1 author:

Juraj Somorovsky
Ruhr-Universität Bochum
44 PUBLICATIONS **1,126** CITATIONS

SEE PROFILE

On the Insecurity of XML Security

Juraj Somorovsky



Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

Bochum, July 2013

Abstract

XML Encryption and XML Signature describe how to apply encryption and signing algorithms to XML documents. These specifications are implemented in XML frameworks of major commercial and open source organizations like Apache, IBM, Microsoft, or SAP. They are employed in a large number of major web applications, ranging from business communications, eCommerce, and financial services over healthcare applications to governmental and military infrastructures.

This thesis analyzes the security of these specifications and presents several practical and highly critical attacks. First, it describes different classical and novel XML Signature Wrapping (XSW) attack techniques, which allow to break integrity of signed XML documents. The attacks exploit weak interfaces between XML Signature validation and XML processing modules deployed in different frameworks. Their criticality is confirmed by applications to cloud and Single Sign-On interfaces: an attacker was able to use them to gain control over victim's Amazon and Eucalyptus cloud instances, or log in as an arbitrary user in Single Sign-On domains of Salesforce and IBM products.

Second, the thesis describes several practical attacks on XML Encryption. The attacks break confidentiality of RSA PKCS#1 v1.5 encrypted ciphertexts (used for key transport) and CBC encrypted symmetric ciphertexts (used for data encryption). An attacker can decrypt such ciphertexts by sending related ciphertexts to a server processing encrypted messages. He can recover the whole ciphertext by issuing a few hundreds or several thousands of requests, depending on the considered scenario.

The work described in this thesis influenced many XML frameworks and systems, as well as the W3C XML Encryption recommendation. These were updated to prevent the attacks. The thesis summarizes best practices to counter all the described attacks in different practical scenarios that were developed in collaboration with developers and members of standardization groups.

Acknowledgements

I would like to thank many people who have helped me through the completion of this work. First, I would like to thank my supervisor Jörg Schwenk, who gave me the opportunity to work on his chair with a family atmosphere. He always supported me and gave me excellent advices which improved my work. I am also very grateful to my second supervisor Kenny Paterson for providing many helpful comments to the first version of this thesis, for interesting discussions on various security topics, and for our joint work.

I thank to all my colleagues and students for the great collaboration, sharing common activities, and for their contributions to this thesis. Especially, I would like to thank Florian Feldmann for common golf activities; Mathias Herrmann for playing tennis; Tibor Jager for playing tennis, introducing me into the world of crypto attacks and for the excellent joint work, without him XML Encryption would not have been broken; Mario Heiderich for introducing me XSS and whiskey; Meiko Jensen for presenting me the world of Web Services and XML Security, and for cleaning the car in Miami; Florian Kohlar for helping me at the beginning of my work at the chair; Christian Mainka for a great work on WS-Attacker and for the nice trip to Hawaii; Andreas Mayer for the SAML penetration tests and Würth calendars; Christopher Meyer for the windsurfing course, grill parties, and the joint work on TLS attacks; Vladislav Mladenov for cycling tours; and Sebastian Schinzel for countless timing measurements by our attacks on XML Encryption and TLS. It was my pleasure to work with you.

I would also like to thank members of the W3C XML Encryption Working Group, security engineers of the companies mentioned in this thesis, and other researchers. Their helpful comments and contributions have improved this work significantly. Especially, I would like to thank Thomas Alt, Daniel Bleichenbacher, Martijn de Boer, Scott Cantor, Felix Freiling, Colm O hEigeartaigh, Thorsten Holz, David Jorm, Jan Lieskovsky, Martin Rex, Thomas Roessler, Prabath Siriwardena, Alessio Soldano, and Yang Yu.

Finally, my biggest thanks goes to my parents, sisters, and Paulina. They have supported me always and in every situation. Without their support this work would not have been possible.

Contents

1 Introduction

XML [BPSM+08] is a widely deployed, platform-independent data format. It is used in office applications, configuration files, cloud applications, databases, or in Web Services. The wide adoption of XML has resulted in an emergence of numerous extension specifications. Over the last 15 years, OASIS and W3C have established a large family of XML-related standards and recommendations. Those allow for selection [CD99] and transformation [Cla99] of XML contents, remote procedure invocations [MBF+04], Single Sign-On [CKPM05], or access control [Mos05].

Miscellaneous network applications and complex communication standards applying XML have quickly raised security demands. In distributed systems, the most common technology to provide security over a network is Secure Sockets Layer (SSL) / Transport Layer Security (TLS) [DR08]. SSL/TLS protects the transport layer between two communication partners. It secures confidentiality, integrity, and authenticity of all data exchanged on this layer. Even though this technology is widely adopted, it becomes insufficient in some complex scenarios where:

- Only specific message parts have to be encrypted (e.g., credit card numbers or passwords): SSL/TLS encrypts *all* data transfered using the established channel.

- The data has to be transfered over untrusted third parties: SSL/TLS establishes a secure channel only between two communication partners, i.e. offers *point-to-point security*. If a client sends data to a server over a third party (proxy), two SSL/TLS channels are used. The data is first sent over the first encrypted channel to the third party. The third party decrypts it and sends it over the second encrypted channel to the server. Thus, the third party can see all transfered data.

XML Security. The additional security demands are addressed by two further XML standards belonging to the XML Security family: XML Signature [ERS+08] and XML Encryption [ERI+02]. These standards define means to secure integrity, authenticity, and confidentiality of XML messages. To this end, they apply cryptographic algorithms directly *on message level*. They allow to secure *arbitrary* elements within the processed document or even external data. They can secure messages that are stored in untrusted cloud databases or distributed over insecure networks. An attacker cannot modify the signed elements without breaking the signature scheme, and he cannot decrypt it without the corresponding decryption key. Encrypted XML data can be decrypted only by a receiver who is in possession of a decryption key and can be signed only by

a sender with a trustworthy signing key. This way XML Signature and XML Encryption ensure *end-to-end security* and *message-level security*.

Nowadays, XML Signature and XML Encryption are implemented in a wide range of systems and frameworks processing sensitive data, including banking [EBI11, Dan10], eGovernment [HS11, Kan10, VS12], eCommerce, military, and eHealth infrastructures [Cen08, Com10]. The increasing adoption of XML Security in these scenarios is confirmed by a large number of commercially available XML Security Gateways [IBM13, Lay13, Ora11], or cloud applications [AWS13, Euc13] and enterprise software [JBo13, SAP13] supporting these standards. These highly critical scenarios clearly motivate for deep analysis of new security threats.

Contribution. This thesis investigates the security of XML Signature and XML Encryption. First, it analyzes various XML frameworks and systems and their vulnerability to XML Signature Wrapping (XSW) attacks. These attacks allow an attacker to insert new unauthenticated elements in a signed XML message and force the receiver to process them. Thus, an attacker can execute arbitrary content on behalf of the message signer. The practicality and criticality of these attacks are confirmed by attacks on cloud interfaces of Amazon Web Services [AWS13] and Eucalyptus [Euc13], and by attacks on various Single Sign-On frameworks used in governmental or enterprise infrastructures. In these scenarios, a single signed XML message is sufficient to get full control over the victim's cloud machines or log in as an arbitrary user on a system supporting Single Sign-On. In general, the attacks exploit *differences in XML parsing mechanisms* to break integrity of signed messages. They show the necessity for clear interfaces between the XML Signature validation and the XML processing modules deployed in XML Security frameworks. This thesis summarizes such approaches, and presents countermeasures that are practically applicable in various scenarios. In addition, it presents an XSW library for automatic detection of these attacks.

The main results on the XSW attacks were described in the following papers:

- *All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces* published at *ACM CCSW'11* [SHJ+11], and
- *On Breaking SAML: Be Whoever You Want to Be* published at *USENIX Security'12* [SMS+12].

Second, the thesis investigates the security of XML Encryption and its vulnerabilities to adaptive chosen-ciphertext attacks (see Section 4.2 for details on these attacks). In an adaptive chosen-ciphertext attack scenario, the attacker's goal is to decrypt a ciphertext without having a decryption key. To this end, he iteratively creates ciphertexts, which are somehow related to the original ciphertext. He sends the ciphertexts to a receiver and observes its responses. With each response he learns some plaintext information. This helps him to break security of the original ciphertext. This thesis shows how to apply these attacks on servers utilizing XML Encryption. It shows:

1. An attack on symmetric CBC-ciphertexts: This attack generalizes the idea behind Vaudenay's padding oracle attacks [Vau02]. It *exploits XML*

parsing mechanisms and character encodings to perform a highly efficient attack. Using this attack, it is possible to decrypt one byte by issuing 14 server queries on average. The attack is described in Section 4.4.

2. An attack on RSA-PKCS#1 v1.5 encrypted messages: This attack completely breaks confidentiality of the exchanged symmetric keys encrypted with RSA-PKCS#1 v1.5 [Kal98]. It shows *how to adapt* Bleichenbacher's algorithm [Ble98] to attack seemingly secure systems. To this end, the attack exploits timing side-channels, or combination of the PKCS#1 v1.5 and CBC mechanisms. The attack is described in Section 4.5.

3. Backwards Compatibility attacks: These attacks show how an attacker can break security properties of secure encryption schemes in XML Encryption (such as RSA-OAEP [KS98] or AES-GCM [Dwo07]), if the server supports legacy encryption schemes (such as RSA-PKCS#1 v1.5 or AES-CBC). The attacks are described in Section 4.6.

In contrast to the XML Signature Wrapping attacks, these attacks work independently of a processing framework. Thus, as a result of these attacks, the newest version of the XML Encryption recommendation was extended [ERH+13] to explicitly describe countermeasures against them.

The main results on these attacks were described in the following papers:

- *How to Break XML Encryption* published at *ACM CCS'11* [JS11],
- *Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption* published at *ESORICS'12* [JSS12], and
- *One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography* published at *NDSS'13* [JPS13].

The attacks described in this thesis are of general importance. It is likely that they are applicable to other standards as well. One example of another attack target is JSON Object Signing and Encryption [jWG], which specifies application of cryptographic mechanisms to JSON messages. Our attacks on XML Encryption were directly applicable to JSON Web Encryption [JRH12].

Thesis Outline. Chapter 2 gives an overview of the XML technology and related specifications. Chapter 3 introduces general XSW attacks and practical attacks on cloud and Single Sign-On interfaces. It presents practical countermeasures, and our XSW library which is capable of automatic XSW vector generation. Chapter 4 gives background on encryption mechanisms and introduces adaptive chosen-ciphertext attacks. Then, our attacks on XML Encryption and practical countermeasures are presented. The thesis concludes with future research directives in Chapter 5.

2 XML and Web Services – Basics

In the following, we introduce the concepts behind XML, XML Security, and Web Services. They are relevant to this thesis. Readers familiar with these standards and specifications can safely skip this chapter.

2.1 XML

The Extensible Markup Language (XML) [BPSM+08] is a platform-independent language to define new data formats. It has been designed by the W3C Working Group to fulfill the following main goals:

- XML shall be straightforwardly usable over the Internet.
- XML shall support a wide variety of applications.
- XML documents shall be easy to create and process within applications.
- XML shall be human-readable and machine-readable.

Nowadays, XML is used to define various structures for storage (ODF or XHTML), transmission (Web Services), or metadata (XML policies and configuration files).

XML Structure. An XML document consists of characters. Legal characters are tab (`\t`), carriage return (`\r`), line feed (`\n`), and the legal characters of Unicode [Kwa95] and ISO/IEC 10646 [Oht95].

Typically, when processing an XML document (or by processing other tree-based formats such as HTML or XHTML), the document is first represented as a DOM (Document Object Model) [BHH+04]. According to the DOM, the XML document structure consists of the following elements (consider the XML document in Figure 2.1):

- *Document* represents the entire XML document. It provides the primary access to the document's data.

```
<lib:library xmlns:lib="lib-uri">
  <book Id="1">
    <author>Douglas Adams</author>
    <title>The Hitchhiker's Guide to the Galaxy</title>
    <year>1979</year>
  </book>
</lib:library>
```

Figure 2.1: *XML document example.*

- *Element* is a node, which either begins with a start-tag (e.g. `<tag>`) and ends with a matching end-tag (e.g. `</tag>`) or consists only of an empty-element tag (e.g. `<tag/>`). An element can contain a text content or further *child* elements. In our example, we see that the *root* element `library` has one child element `book`. The `book` element contains three child elements.

- *Attribute* is a name-value pair belonging to a specific element. It is placed inside of the start-tag. An example gives the `Id="1"` attribute inside the `book` element.

- *Text* (*character data*) node represents the textual content of an element or attribute. In our document, we see for example the `Douglas Adams` text content within the `author` element.

- *Entity*: In order to display special characters such as brackets (<,>) or ampersands (`&`) in character data, XML specifies entities. Some of the special entities are provided directly by the XML specification, e.g. `&lt;` (<), `&amp;` (&), and `&gt;` (>). Additional entities can be defined directly in the XML document using a Document Type Definition (DTD).

Additionally, XML includes different node types such as processing instructions, document type declarations, comments, or CDATA sections. Please see the XML [BPSM+08] and DOM [BHH+04] specifications for more details about these node types.

XML Well-Formedness. The XML specification summarizes a series of rules that define well-formed XML documents. A well-formed XML document contains one or more elements. It has a correct XML syntax. It contains exactly one root element (also called the document element). Start-tags and end-tags in the document are properly nested. Each of the parsed entities (referenced directly or indirectly within the document) is well-formed. See the XML specification for more details [BPSM+08].

Document Type Definition (DTD). DTD defines the XML document structure with a list of elements and their attributes. DTD is placed *directly* in the XML document, before the first element.

Figure 2.2 defines the structure of the `book` element used in Figure 2.1. This element has three elements containing character data (`author`, `title`, and `year`), and an `Id` attribute. The `Id` attribute is of the type `ID`. The `ID` attribute type ensures that the `book` element can be uniquely referenced by `Id`. Because of this, no two `book` elements can contain the same `Id` value. Otherwise, the document would become invalid.

XML Namespaces. XML namespaces provide unique names for elements and attributes by binding them to a specific URI (Uniformed Resource Identifier) [BLFM98]. Thereby, they allow to prevent conflicts in element and attribute names when working with different XML documents from different sources.

```
<!DOCTYPE library [
  <!ELEMENT book (author,title,year)>
  <!ATTLIST book Id ID>

  <!ELEMENT author (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
]>
```

Figure 2.2: *Document Type Definition defining the structure of a* `book` *element from Figure 2.1: the* `book` *element contains an* `ID` *type attribute and three elements.*

Each XML document has a default namespace. It is possible to define additional prefixed namespaces. In Figure 2.1, we have a namespace with a prefix `lib` defined by `xmlns:lib="lib-uri"`. As the namespace is defined in the root element, each element and attribute could apply the `lib` prefix. In our example, the `library` element comes from the `lib` namespace. The `book` elements and its children are bound to the default namespace.

2.2 XML Schema

The W3C recommendation XML Schema [WF04] is a language to describe the syntax of an XML document. It is a successor of DTD. XML Schema is defined in an *external* XML document and gives the developer more flexibility than DTD. A document is *valid* with respect to a certain schema, if its syntax is compliant with this schema. A schema consists of a content model, a vocabulary, and data types. The content model describes the document structure and the relationship of the items.

Text Definitions. Attribute and element values can be constrained using data types. The specification provides 19 primitive data types to define the content of elements and attributes, e.g. string, boolean, double, or base64Binary. These primitive data types can be used to build derived types (e.g. `ID` or `positiveInteger`).

Simple and Complex Type Definition Components. Type definitions restrict document elements and their structures.

The *simple type* defines elements or attributes containing text values. A simple type element contains no sub-elements. A definition of the `year` element from the previous example is described in Figure 2.3. The value of this element must be higher than 1900.

The content of an element – including its element and text children – can be defined using *complex types*. A complex type typically includes a sequence of different element types (achieved by a `sequence` indicator) and attribute definitions. The example in Figure 2.4 shows the definition of the `book` element. This element contains a sequence of elements `author`, `title` and `year`, and an attribute of type `ID`.

```
<element name="yearType" xmlns="http://www.w3.org/2001/XMLSchema">
  <simpleType>
    <restriction base="integer" >
      <minInclusive value="1900"/>
    </restriction>
  </simpleType>
</element>
```

Figure 2.3: *Simple type definition of a* `year` *element.*

```
<element name="book" xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType>
    <sequence>
      <element name="author" type="string"/>
      <element name="title" type="string"/>
      <element name="year" type="yearType"/>
      <any minOccurs="0" processContents="skip"/>
    </sequence>
  </complexType>
  <attribute name="Id" type="ID"/>
</element>
```

Figure 2.4: *Complex type definition of a* `book` *element.*

Regarding to this thesis there is one important element definition in XML Schema. The `any` element allows the usage of any well-formed XML document in a declared content type. When an XML processor validates an element defined by an `any` element, the `processContents` attribute specifies the level of flexibility. The value `lax` instructs the schema validator to check against the given namespace. If no schema information is available, the content is considered valid. In the case of `processContents="skip"` the XML processor does not validate the element at all. Considering the example from Figure 2.4, the `book` element could contain an arbitrary element from an arbitrary namespace behind the `year` element.

2.3 XPath

XML Path Language (XPath) [CD99] describes paths to traverse XML trees. It is extended with a set of basic arithmetic and string processing functions. Evaluation of an XPath expression can start from an arbitrary element and returns a node set or character data.

XPath defines several axes to move within the document tree, see Figure 2.5. It is possible to select

- sibling elements of the context node using the `following-sibling` and `preceding-sibling` axes,
- child and descendant elements of the context node using the `child` and `descendant` axes, or

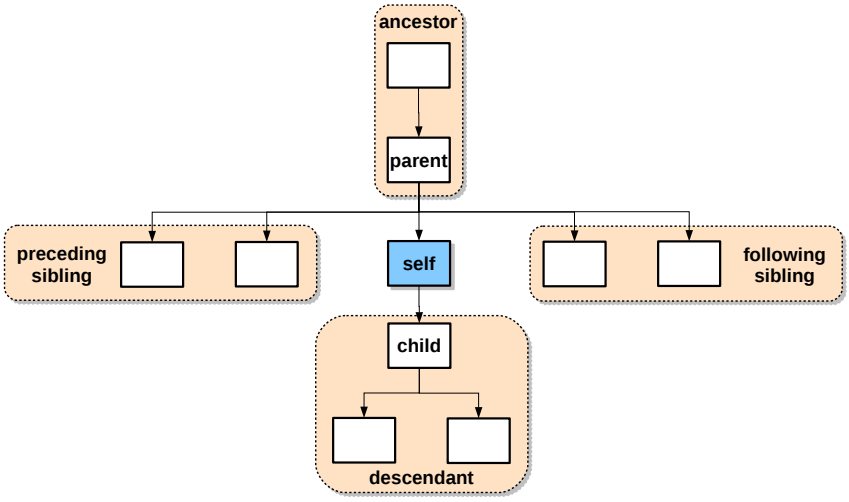


Figure 2.5: *A context node with its ancestor, sibling, and descendant elements.*

- parent and ancestor elements of the context node using the `parent` and `ancestor` axes.

Consider the document in Figure 2.1. The following XPath expressions applied on this document would lead to the following results:

- `/descendant::*/child::book[position()=1]` starts its search from the document root. It searches for a descendant element with an arbitrary name (indicated by `*`) and selects its first `book` element. The result of this expression returns the `book` element.
- `/descendant::author[parent::book]/child::text()` searches for all the `author` elements in the document, which have a `book` parent element. Then, it selects the text values of the found elements. The result is `Douglas Adams`.
- `/descendant::book[position()=2]/attribute::*` searches for the second `book` element in the document and selects all its attributes. As the document contains only one `book` element, the result is an empty set of nodes.
- `/*[local-name()="library" and namespace-uri()="lib-uri"]` starts its search from the document root. It searches for a `library` element from a `lib-uri` namespace. It returns the whole `library` element.

For more details and examples, see the XPath specification [CD99].

XPath is a general construct for selecting data. It is used in different specifications, such as XPointer [MMGW03], XQuery [FFB+09], or XSLT [Cla99].

2.4 XML Processing

Two different programming approaches exist for working with XML documents: tree-based (or also called DOM-based) and streaming-based (or event-based).

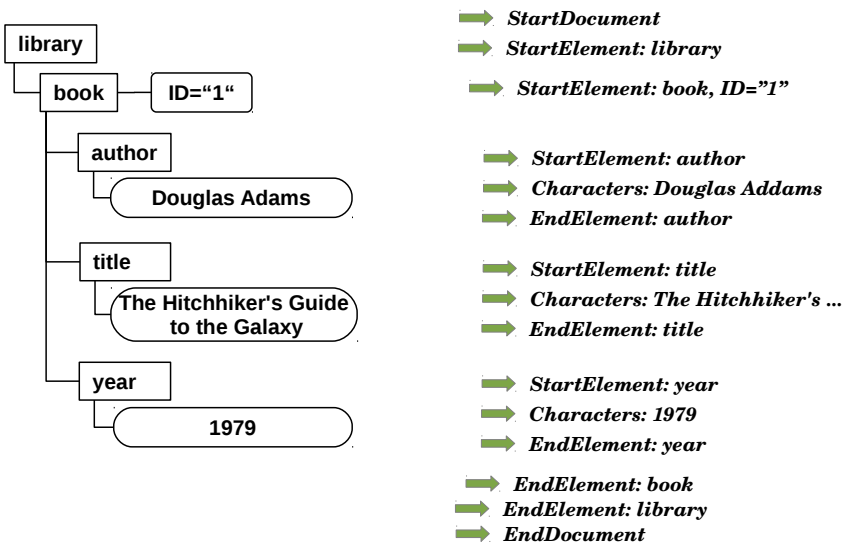


Figure 2.6: *A simplified description of differences between the DOM-based and streaming-based parsing: A developer using the DOM-based approach has access to all the document elements after parsing is complete (left). On the other hand, a developer using the streaming-based approach has only access to specific events while parsing is performed (right).*

DOM-based Processing. An example of a tree-based approach gives the DOM (Document Object Model) [BHH+04]. DOM provides maximum flexibility for developers as they can dynamically access and change every node in a DOM tree. But before doing this, the whole document has to be read and stored in the memory. All the document elements, their relations and properties are mapped into newly instantiated objects. The memory consumption by such a processing is few times higher than the parsed XML document. This leads to a big disadvantage of DOM – high memory consumption when processing large XML documents.

Streaming-based Processing. Streaming-based paradigms – such as the Simple API for XML (SAX) [Mor02] or the Streaming API for XML (StAX) [CC09] – process XML documents in real time. They do not store any data about the processed elements. They can generate output immediately or interrupt parsing. This is much more efficient and thus preferable for high performance applications. On the other hand, each document is processed only once and there is no possibility to go back during parsing. The developer has only access to a list of XML events, which are generated while parsing the XML document tree.

In comparison to the tree-based approach, a developer applying the streaming-based approach uses a totally different code for accessing specific elements in the XML document (see Figure 2.6).

2.5 XML Security

The need for secure message exchange between different XML applications lead to the development of two XML Security specifications: XML Signature [ERS+08] and XML Encryption [ERI+02]. In comparison to SSL/TLS [DR08] or IPsec [VM05] – which secure data *only during the transport* – the XML Security specifications apply security algorithms directly on the *message-level*. XML Security is rather similar to S/MIME [RT10] and OpenPGP [CDF+07] message formats. In comparison to these two specifications, it offers much more flexibility as it allows to secure arbitrary document contents.

2.5.1 XML Signature

The XML Signature specification [ERS+08] defines the syntax and processing rules for creating, representing, and verifying XML-based digital signatures. It allows to protect integrity and authenticity of the XML messages. The specification is very flexible and allows to sign any type of digital data. It is possible to sign a whole XML tree, specific elements, parts of elements, or even arbitrary multiple parts of the XML tree. One XML Signature can cover several local or global resources. Basic structure of an XML Signature is depicted in Figure 2.7.[1]

```
<Signature ID?>
   <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
       </Reference>)+
   </SignedInfo>
   <SignatureValue>
   (<KeyInfo>)?
   (<Object ID?>)*
</Signature>
```

Figure 2.7: *XML Signature data structure taken from [ERS+08] ("?" denotes zero or one occurrence, "+" denotes one or more occurrences, and "*" denotes zero or more occurrences).*

An XML Signature is represented by the `Signature` element. XML Signatures are two-pass signatures: The hash value of the resource (`DigestValue`) along with the used hash algorithm (`DigestMethod`) and the `URI` reference to the resource are stored in the `Reference` element. Additionally, the `Transforms`

[1] For simplicity, namespace declarations and namespace prefixes are omitted in most of the figures. They are explicitly included only when they are important for the attack descriptions.

element specifies the processing steps, which are applied prior to resource hashing. Each signed resource is represented by one `Reference` element in the `SignedInfo` element. Therefore, `SignedInfo` is a collection of hash values and `URI`s. The `SignedInfo` element itself is protected by the signature. The `CanonicalizationMethod` and the `SignatureMethod` element specify the algorithms used for canonicalization and signature creation, and are also embedded in `SignedInfo`. The Base64-encoded value of the computed signature is deposited in the `SignatureValue` element. In addition, the `KeyInfo` element facilitates the transport of signature relevant key management information. `Object` is an optional element that may contain *any* data.

2.5.1.1 XML Canonicalization

Two XML documents containing semantically identical data can differ in their physical byte representation. They can contain different entity structures, character encodings, attribute orderings, or number of whitespaces in the element tags. Computing hash values over such XML documents brings different results. Therefore, before creating or validating XML Signatures, the XML documents (or their signed parts) have to be converted to the canonical form with an appropriate canonicalization method. XML Canonicalization ensures that semantically identical XML documents give identical hash values.

The major changes done by converting to the canonical form are listed below:

- The document is encoded in UTF-8.
- Line breaks are normalized to line feeds (`\n`).
- Empty elements are converted to start-end tag pairs.
- Namespaces and attributes are lexicographically ordered.
- Special characters are encoded as character references.

There are two types of XML Canonicalization: Inclusive [Boy01] and Exclusive [RrB02]. Inclusive XML Canonicalization declares all namespaces used in ancestor elements in the root element of the canonicalized XML fraction. Exclusive XML Canonicalization declares a namespace in an element only if:

1. the element visibly utilizes this namespace (the element or its attribute uses the namespace prefix)
2. and this namespace was not already declared by any ancestor element (that is also canonicalized).

Figure 2.8 shows application of the Inclusive and Exclusive XML Canonicalization methods on the `book` element from Figure 2.1. Inclusive XML Canonicalization declares the `lib` namespace in the `book` element. Exclusive XML Canonicalization omits the `lib` declaration as this namespace is not needed in the canonicalized part.

The Inclusive XML Canonicalization method presents one disadvantage. An XML processor can namely easily destroy a signature over an element, if it

```
<book xmlns:lib="lib-uri" Id="1">
  <author>Douglas Adams</author>
  <title>The Hitchhiker's ...</title>
  <year>1979</year>
</book>
```

```
<book Id="1">
  <author>Douglas Adams</author>
  <title>The Hitchhiker's ...</title>
  <year>1979</year>
</book>
```

Figure 2.8: *Application of the Inclusive (left) and Exclusive (right) XML Canonicalization method on the* `book` *element from Figure 2.1.*

declares a new namespace in one of its ancestors (this would namely lead to inclusion of this namespace in the signed element during the canonicalization process). Thus, Exclusive XML Canonicalization is the *preferred* method for canonicalizing secure XML documents [RrB02, MGMB07].

2.5.1.2 XML Signature Processing

The construction of an XML Signature proceeds as follows. First, the `Reference` elements are created. To this end, each data object to be signed is transformed by the applied transformation algorithms (e.g., by canonicalization or XPath). This transformed data object is used as input for the hash computation and the calculated digest is stored in `DigestValue`. The `URI` attribute, `DigestValue`, `DigestMethod`, and the `Transforms` element are taken together to create a `Reference` element. The `Reference` elements, combined with `CanonicalizationMethod` and `SignatureMethod`, are put into the `SignedInfo` element. The whole `SignedInfo` element is then canonicalized using the algorithm from `CanonicalizationMethod`. The resulting octets are signed using the signature algorithm given by `SignatureMethod`. Finally, the Base64-encoded signature value is stored inside the `SignatureValue` element.

An XML Signature is validated in two steps. First, the references are validated.[2] Thereby, the referenced content of each `Reference` element is retrieved, transformed, and hashed with the specified methods. The calculated hash values are compared with the content of the `DigestValue` elements. Second, the `SignedInfo` element is canonicalized with the specified algorithm from `CanonicalizationMethod`. The output of the canonicalization is used for the signature verification by application of the algorithm specified in `SignatureMethod` (which includes all cryptographic functions involved in the signature operation, e.g. hashing, public key algorithms, HMACs [KBC97], or padding). For this purpose, the verifier uses the key retrieved from the `KeyInfo` element or by other means.

2.5.1.3 XML Signature Types

The `Signature` element can be placed at an arbitrary position in the XML tree. According to the relationship between the position of the `Signature` element

[2] This processing order is recommended by W3C and can be exploited by an XSLT command injection [Hil07]. The author recommends to disable XSLT processing, or to invert the order of signature and reference validation if XSLT processing cannot be disabled.

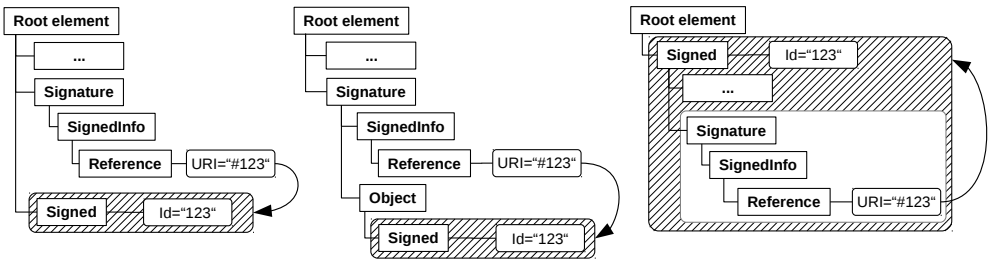


Figure 2.9: *Detached, Enveloping, and Enveloped XML Signature types. The hatched area represents the signed content.*

and the position of the signed data, we differentiate between three types of XML Signatures (see Figure 2.9):

- *Detached* XML Signature protects an element (or an external document) that is neither inside nor a parent of the `Signature` element. A typical application of this signature type finds place in SOAP-based Web Services (see Section 2.6.2).

- *Enveloping* XML Signature protects an element placed inside of the `Signature` element. More concretely, the signed content is placed in the signature's `Object` element.

- *Enveloped* XML Signature protects a `Signature`'s ancestor element (this can also be the document's root element). This specific XML Signature type requires application of an enveloped transformation, which excludes the whole `Signature` element from the `Reference` hash value computation. Thus, even if the root XML element is signed, the document still contains unsigned data within the `Signature` element. This signature type is typically used in SAML messages (see Section 2.7).

2.5.2 XML Encryption

XML Encryption [ERI+02] specifies methods to encrypt XML-based data structures. It specifies two public-key encryption schemes (PKCS#1 in versions 1.5 [Kal98] and 2.0 [KS98]), two symmetric key wrapping schemes (3DES Key Wrap and AES Key Wrap [NIS01a]), and two symmetric ciphers for data encryption (3DES-CBC and AES-CBC [MvV96]). All are mandatory.[3]

XML Encryption defines two basic data structures (see Figure 2.10):

- `EncryptedKey`: This element typically stores an encrypted symmetric key. It can contain up to four components. The `EncryptionMethod` and `KeyInfo` elements provide an information about the encryption method

[3] In addition to these cryptographic primitives, the W3C Working Group included AES-GCM in the latest specification version [ERH+13] in order to prevent a recent attack on CBC-based XML Encryption [JS11], which is described later in this thesis. AES-CBC and 3DES-CBC are still included in the specification, for backwards compatibility reasons.

```
<EncryptedKey ID?>
  <EncryptionMethod/>?
  <KeyInfo/>?
  <CipherData>
    <CipherValue>?
  </CipherData>
  (<ReferenceList>
    <DataReference URI/>*
  </ReferenceList>)?
</EncryptedKey>
```

```
<EncryptedData ID?>
  <EncryptionMethod/>?
  <KeyInfo/>?
  <CipherData>
    <CipherValue>?
    <CipherReference URI?>?
  </CipherData>
</EncryptedData>
```

Figure 2.10: *Structure of data encrypted with XML Encryption taken from [ERI+02] ("?" denotes zero or one occurrence, "+" denotes one or more occurrences, and "*" denotes zero or more occurrences).*

and the (public or symmetric) key used for the encryption process. The `CipherData` element contains the encrypted symmetric key k. `ReferenceList` contains references to all `EncryptedData` elements that can be decrypted with the encapsulated symmetric key.

- `EncryptedData`: This element stores a symmetrically encrypted XML payload. It typically consists of three elements. The `EncryptionMethod` element gives an information about the symmetric encryption method. The `KeyInfo` element stores information about the symmetric encryption key used for payload decryption. This can be a key encapsulated in the `EncryptedKey` element or a reference to a different available key. The encrypted payload is stored in the `CipherData` element.

Since XML is a text data format, all binary data is converted to text data by applying Base64 [Jos06] encoding.

2.5.2.1 XML Encryption Processing

In order to encrypt XML data in common applications, usually *hybrid encryption* is used. That is, encryption proceeds in two steps:

1. The sender chooses a *session key* k. This key is encrypted with a public-key encryption scheme, under the receiver's public-key. The resulting ciphertext is stored in the `EncryptedKey` element.

2. The actual payload data is then encrypted with a symmetric encryption algorithm using the key k. This yields a ciphertext, which is stored in the `EncryptedData` element.

The message decryptor reverses this process in an obvious way. It first decrypts the session key k. Then, it uses k to decrypt the encrypted payload. Finally, the payload data is parsed with an XML parser.

The combination of public-key and symmetric-key encryption schemes is often used in practice. Public-key encryption schemes ensure that two parties can exchange a secret symmetric key without sharing a common secret. However,

such schemes are often more inefficient than typical symmetric-key encryption schemes. The combination of these two schemes provides a system with an efficient data encryption without previous key distribution.

More information on XML Encryption and its decryption processing steps can be found later in Section 4.3.

2.6 Web Services

Web Services are a method for interprocess interactions over networks between different software applications. A Web Service can be implemented using different technologies. Most Web Services apply the REST [FT02] and SOAP [MBF+04] technologies.

In this thesis we consider SOAP-based Web Services. While considering a SOAP-based technology, a Web Service is a general term for a family of standards maintained by W3C and OASIS. These standards extend the basic Web Services idea to provide message-level security, definition of policies, or authentication and authorization mechanisms.

2.6.1 SOAP

SOAP (originally defined as Simple Object Access Protocol) is a W3C specification defining structure of XML messages and a protocol to achieve a machine-to-machine communication [GHM+03]. The communicating applications use SOAP messages. SOAP messages generally consist 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.

An example of a SOAP message is given in Figure 2.11. A server receiving this SOAP message proceeds as follows. First, it searches for a *function name*, which is defined in the `Body` element (the `Body` element contains a `CreditCardPayment` function). Afterwards, it invokes the function and responds to the requester.

```
<Envelope>
  <Header/>
  <Body>
    <CreditCardPayment>
      <Name>John Smith</Name>
      <Number>1234 5678 9000 1234</Number>
      <Issuer>Example Bank</Issuer>
    </CreditCardPayment>
  </Body>
</Envelope>
```

Figure 2.11: *SOAP message example.*

2.6.2 WS-Security

Web Services Security [NKMHB06] – or WS-Security for short – is an OASIS standard defining the application of security primitives to SOAP-based Web Services. In particular, this includes usage of XML Signature, XML Encryption and different security tokens (username token, timestamp, or X.509 certificates [CSF+08]) in XML-based SOAP messages.

A simplified structure of an XML Signature applied to a SOAP message according to WS-Security gives Figure 2.12. The depicted SOAP message includes a function invocation `DeleteUser` 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 signature contains an Id-reference pointing to the SOAP body, which secures integrity of the whole `Body` element.

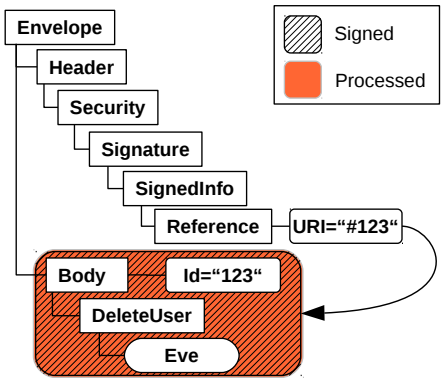


Figure 2.12: *Example of XML Signature applied on the SOAP body.*

The recipient processes such a message as follows. He 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`. If the verification succeeds, he can execute the function *defined in the SOAP body.*

2.6.3 Web Services Policy

A Web Service endpoint can have different requirements, capabilities, or behaviors (e.g. encryption and signature algorithms, usage of security tokens, or specific transport mechanisms). The Web Services Policy (WS-Policy) framework [YHV+07] defines a general construct to specify such Web Service properties and requirements. In WS-Policy, each Web Service property is defined as a policy *assertion*. WS-Policy provides a core set of constructs to combine general policy assertions and describe policies for arbitrary Web Services environments.

Policy assertions are grouped into policy alternatives. A set of policy alternatives gives a WS-Policy definition. For grouping policy assertions two XML tags are used: `All` and `ExactlyOne`. `All` is a logical `AND` and indicates that all child node assertions have to be fulfilled. `ExactlyOne` indicates a logical `XOR` and it

```
<Policy>
  <ExactlyOne>
    <All>
      <EncryptedParts><Body/></EncryptedParts>
      <SignedParts><Body/></SignedParts>
    </All>
    <All>
      <EncryptedParts><Body/></EncryptedParts>
      <SignedParts><Header Name="Timestamp"/></SignedParts>
    </All>
  </ExactlyOne>
</Policy>
```

Figure 2.13: *WS-Security Policy with two policy alternatives. It defines that the SOAP body must be encrypted. Additionally, either the SOAP body or the* `Timestamp` *element must be signed.*

contains assertions, from which *exactly one* has to be fulfilled.[4] An example of a WS-Policy document with two policy alternatives is shown in Figure 2.13.

A WS-Policy can for example be published as a part of a WSDL (Web Services Description Language) [CCMW01] file.

2.6.4 Web Services Security Policy

Web Services Security Policy (WS-Security Policy) is an OASIS framework [LK07]. It defines a set of security-specific policy assertions. The assertions are used together with the WS-Policy specification to express Web Services security constraints and requirements.

An example of a WS-Security Policy document is given in Figure 2.13. This policy definition would enforce the server to process only SOAP messages that contain an encrypted SOAP body. In addition, it is necessary to sign either the `Timestamp` element or the SOAP body element.

2.7 SAML

Security Assertion Markup Language (SAML) is an XML standard for exchanging authentication and authorization statements about *Subjects* [CKPM05] maintained by OASIS. It is typically used in Single Sign-On (SSO) deployments (see Section 3.3).

SAML subjects are described in XML-based assertions. The structure of a SAML assertion is depicted in Figure 2.14. The SAML assertion consists of the following elements. The `Issuer` element specifies the SAML authority that is making the claim(s) in the assertion. The assertion's `Subject` defines the principal about whom all statements within the assertion are made. The `*Statement` elements are used to specify user-defined statements relevant for the context of the SAML assertion.

[4] However, many Web Services implementations handle `ExactlyOne` as a logical `OR` and thus also accept messages with *more* assertions fulfilled.

```
<Assertion Version ID IssueInstant>
  <Issuer>
  <Signature>?
  <Subject>?
  <Conditions>?
  <Advice>?
  <AuthnStatement>*
  <AuthzDecisionStatement>*
  <AttributeStatement>*
</Assertion>
```

Figure 2.14: *SAML assertion structure.*

To protect the integrity of the security claims made by the Issuer, the whole `Assertion` element must be protected with a digital signature following the XML Signature specification. Therefore, the SAML specification [CKPM05] requires that either the `Assertion` element or an ancestor element must be referenced by the `Signature` element, with an *enveloped* XML Signature ([CKPM05], Section 5.4.1). Furthermore, `Id`-based referencing must be used ([CKPM05], Section 5.4.2).

Usage of the SAML assertions in various XML messages is described in the SAML Bindings specification [Sco05]. In REST-based frameworks, the SAML assertion is typically put into an enveloping `Response` element (described by the POST binding). Frameworks applying SOAP insert the SAML assertions into the SOAP header (or the `Security` element inside of the SOAP header). For clarification purposes, consider that the SAML assertions are signed using *enveloped* XML Signatures and are put into some binding root element *R*. These two prevalent SAML bindings are depicted in Figure 2.15.

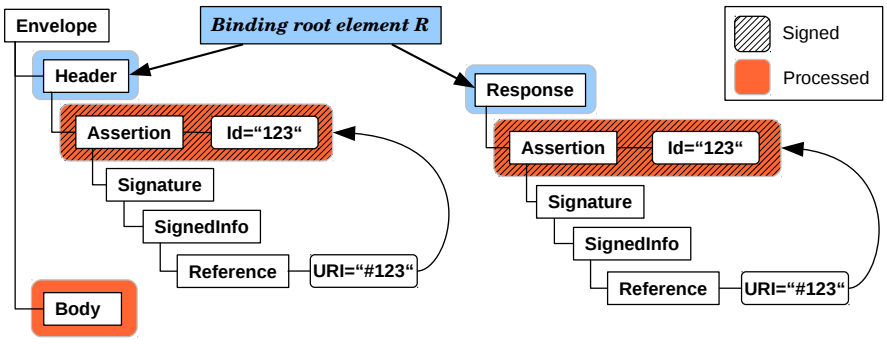


Figure 2.15: *SAML SOAP and HTTP POST Binding: The SAML assertion is put into a binding root element R and signed using an enveloped signature. When signing the SOAP body, an additional detached signature must be used.*

2.8 A Typical Message Flow in a Web Service Framework

A typical Web Service framework supports building Web Service clients as well as deploying robust Web Service server applications. Both client and server should provide a large amount of different functionalities, which result from various Web Service specifications. For example, the framework has to support WS-Security [NKMHB06], WS-Security Policy [LK07], WS-Secure Conversation [NGG+09], or WS-Addressing [GHR06]. These specifications are typically implemented in different independent modules to support a fine-grained message processing.

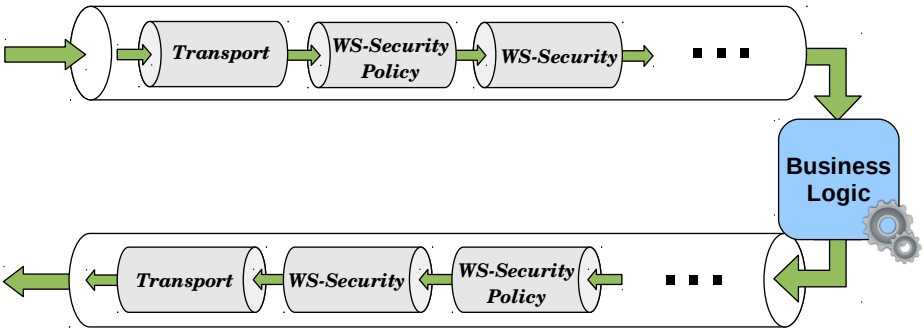


Figure 2.16: *SOAP message flow in a typical Web Service framework.*

The modules are engaged in a 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. An exemplary message flow deployed on a Web Service server is depicted in Figure 2.16. By processing an incoming SOAP message inside of this flow, the server first handles transport specific issues (it includes e.g. processing HTTP headers). Afterwards, the SOAP message context is checked against the WS-Security Policy file. The module verifies, if the message contains all the security specific properties. The security module then processes the security elements in the message. It validates the username token, timestamp, and/or XML Signature. The encrypted elements are decrypted and parsed by an XML parser in order to update the SOAP message context. The decrypted and validated content of the security module is sent to the following modules. At the end, the message is forwarded to the business logic, which executes the function represented by this service. The generated function output is again wrapped in a SOAP message. It is sent through a similar message output flow as a SOAP response, back to the client.

There are two important facts about such a message flow processing, which are important to this thesis:

1. Handlers in the message flow and the business logic can be implemented by different developers and can use completely different parsing mechanisms or even programming languages. It is e.g. common to implement a Web Service logic and place it behind a Web Service Gateway, which

solely processes XML Security elements. It is also common that – due to the complexity of XML Signature and XML Encryption – XML Security modules apply DOM-based XML parsers. On the other hand, business logic processing is simple and can be implemented using streaming-based parsers. These *different processing mechanisms* can result in *different views of an XML document* while processing it within the message flow.

2. Each module in the flow and business logic can stop the SOAP message processing if an error in the message occurs. In that case the message flow through the modules is terminated and the framework responds with an appropriate SOAP fault. The SOAP fault introduces a generalized exception thrown by a module. It does not contain any confidential information. However, after sending an invalid SOAP message to the Web Service server, one can still *detect in which of the handlers the exception was generated*, by evaluating the returned SOAP fault.

2.9 Analyzed Systems and Frameworks

This thesis evaluates various Web Services, SAML, and XML Security interfaces. These interfaces can be a part of different SOAP and SAML libraries or frameworks, XML Security Gateways, or be deployed on real-world Web applications to process authentication and secure Web Services invocations. In the following, the analyzed systems and frameworks are briefly described.

2.9.1 Web Services Frameworks and Gateways

The interfaces described below present frameworks and appliances with WS-* capabilities. They allow to build and run secure SOAP-based Web Services providing XML Security as well as SAML functionalities.

Apache Axis2 [Apa13a] is one of the most popular open source Web Services frameworks. It supports building Web Services clients as well as deploying robust Web Services server applications, and provides the Web Services architects with modules which include many features and functionalities. XML Encryption and XML Signature within this framework are processed in the Rampart module [Apa12a] using the Apache WSS4J library [Apa12b].

Apache CXF [Apa13b] is an open source framework allowing to run Web Services by applying different technologies such as SOAP, XML, or REST. XML Encryption and XML Signature are processed using the Apache WSS4J library [Apa12b].

IBM Datapower XS40 [IBM13] is an XML Security Gateway typically applied in enterprise architectures. It processes the incoming XML messages using a hardware-accelerated XSLT processor [Cla99].

JBossWS [JBo13] is a framework supporting Web Services communication. Its XML Security processing works on the top of the Apache WSS4J library [Apa12b].

2.9.2 SAML Frameworks

In the following, we give an overview of SAML frameworks. These can be for example applied in various Web Services or Web interfaces supporting only SAML HTTP bindings.[5]

Guanxi [gua13] is an open source Java implementation of the SAML specification.

Higgins 1.x [Hig] is an open source Java implementation supporting SAML-based SSO.

Java Open Single Sign-On (JOSSO) [JOS13] is an open source Java implementation of SAML-based SSO. According to JOSSO's website, this framework is for example used by Motorola, NEC, and Redhat.

OIOSAML [OIO13] is an open source Java SAML framework used for example in Danish public sector federations (e.g. eGovernment business and citizen portals).

OpenAM [Ope13a] (formerly known as SUN OpenSSO) is an identity and access management middleware used in major enterprises.

OneLogin [One13] is an open source SAML framework implemented in Ruby, Python, Java, .NET, and PHP. This framework can be used to integrate SAML into various popular open source web applications like Wordpress, Joomla, Drupal, or SugarCRM. Moreover, it is applied by many OneLogin customers (e.g. Zendesk, SAManage, KnowledgeTree, and Yammer) to enable SAML-based SSO.

OpenAthens [Edu13] is a suite of services designed for companies and organizations to establish SSO scenarios. OpenAthens provides the developers a SAML framework implemented in Java and C++. This framework is for example applied in the UK Access Management Federation[6].

OpenSAML [Ope13b] is one of the most widely used SAML libraries. Its source code is open, and the library is implemented in Java and C++. It is for example adopted in Shibboleth [Shi13] and in the SDK of the electronic identity card of Switzerland (SuisseID).

SimpleSAMLphp [Sim13] is a PHP framework implementing SAML and other identity protocols. It is applied for example by Danish eID Federation and different universities.

Microsoft Windows Identity Foundation [Mic13] is a .NET framework providing developers classes for development of identity-based scenarios. This framework is for example used in Microsoft Sharepoint 2010.

[5] Even though some of these frameworks support SOAP bindings and different WS-* scenarios, their primary goal is to support a (SAML-based) identity management.

[6] `www.eduserv.org.uk`

2.9.3 Web Applications

Our work also analyzes real-world Web applications.

Amazon Web Services (AWS) [AWS13] is one of the first cloud services offering its customers virtual machines, cloud storage and many other services. A customer can access these services using different interfaces, including SOAP-based Web Services.

Eucalyptus [Euc13] is a framework designed to run a private cloud on arbitrary servers. Similarly to Amazon Web Services, Eucalyptus also offers its users different interfaces for controlling the cloud machines, including a SOAP-based Web Services interface.

Ping Identity [Ide13] is an identity management solution provider supporting SAML-based SSO [CKPM05]. It provides their customers with products such as PingFederate that can play the role of an Identity Provider or a Service Provider.

Salesforce [Sal13] provides cloud-based customer relationship management (CRM) software. It allows its customers to login using SAML tokens.

SAP NetWeaver [SAP13] is a platform, which supports the composition and management of different (SAP as well as non-SAP) applications across heterogeneous software environments. This platform uses WS-* and SAML specifications.

WSO2 [WSO13] offers their customers many open source products supporting WS-* and SAML specification. In this work, we describe attacks on the SAML implementation applied in WSO2 products. This implementation is for example applied in the WSO2 Stratos cloud or in the WSO2 Enterprise Service Bus.

2.10 Beyond XML Security – JSON Object Signing and Encryption

JavaScript Object Notation (JSON) [Cro06] is a lightweight text-based standard for description and exchange of arbitrary data. JSON structures are typically used in browser-based applications or JSON Web Services. Purposes of the protocols used on the top of JSON are in many cases similar to the XML protocols. Thus, it has been realized that the JSON-formats need standardized security mechanisms. The needs for the security mechanisms fostered standardization of two new security standards. The JSON Web Encryption (JWE) [JRH12] and JSON Web Signature (JWS) [JBS12] standards are maintained by the Javascript Object Signing and Encryption (jose) Working Group. The standards are quite recent, with the first public draft dating to January 2012.

Beyond XML Security, this thesis investigates attacks against these newly developed standards. These attacks are presented in Sections 4.5 and 4.6.

2.10.1 JSON Web Encryption

JSON Web Encryption (JWE) specifies how to apply encryption schemes to JSON data structures. JWE supports different methods for data encryption, using symmetric and public-key encryption algorithms. The current draft 06 of the JWE standard includes the algorithms AES-CBC with HMAC [KBC97], AES-GCM [Dwo07], and AES Key Wrap [NIS01a] as mandatory symmetric ciphers. The mandatory public-key encryption schemes are PKCS#1 v1.5 [Kal98] and v2.0 [KS98] encryption.

A JSON Web Encryption message consists of two components. The *body segment* contains a ciphertext encrypting the payload data. The *header segment* contains information about the algorithms used to encrypt this ciphertext contained in the body. An example of a JWE header segment is given in Figure 2.17. Similarly to XML Encryption, the encryption algorithms are transported in the message. In this example RSA-PKCS#1 v1.5 is used to encapsulate a symmetric key. The actual payload data is encrypted under this key using AES-GCM.

```
{"alg":"RSA1_5",
 "enc":"A256GCM",
  "iv":"__79_Pv6-fg",
 "jku":"https://example.com/p_key.jwk"}
```

Figure 2.17: *JSON Web Encryption header segment example specifying encryption algorithms.*

2.10.2 JSON Web Signature

Different methods to secure integrity and authenticity of JSON messages are provided by the JSON Web Signature (JWS) [JBS12] standard. In order to describe our attacks it is sufficient to know that the JSON Web Signature standard includes the RSA-PKCS#1 v1.5 signature scheme.

3 How to Break XML Signature

XML Signature Wrapping (XSW) attacks are a new class of attacks introduced by McIntosh and Austel in 2005 [MA05]. An attacker executing this attack forces an XML application to process newly inserted bogus contents within a signed XML message. Although XSW attacks are very dangerous because they completely circumvent the integrity protection of XML Security, even seven years after their publication only few practical examples of these attacks [GL09] and incomplete analyses existed [MA05, RMS06]. These facts and the rising popularity of XML Security mechanisms in the real-world applications motivated us to evaluate XML Signature interfaces of various providers and frameworks.

In the following, we first introduce the basic XSW attacks described in the original paper. Afterwards, we present our attacks on SOAP and SAML interfaces, and demonstrate that these attacks are practical and of enormous importance, given the scenarios in which they are executed. Thereby, we also found new and more sophisticated XSW attack types. We discuss countermeasures against XSW attacks. The found attacks motivated us to develop a penetration testing tool for XML Signature interfaces. It is described in the last section.

3.1 XML Signature Wrapping Attacks – Basics

As described in Section 2.5.1, XML Signature validation is a complex process. It involves URI-based dereferencing, XML canonicalization, two-step hash value computation, and evaluation of a cryptographic function. On the other hand, a search for a specific XML element (e.g., an element in the SOAP body) can easily be executed by application of a simple XML parser. This leads to the fact that XML documents containing XML Signatures are typically *processed in two independent steps*: (1) signature validation and (2) function invocation (business logic). The signature validation logic is implemented as an independent XML Security library or an independent XML firewall. The verified document is processed by a custom business logic concentrating on specific parts in the XML document (e.g., the SOAP body in a SOAP message).

If an application processes signed XML documents using different modules having different views on the document, a new class of attacks named *XML Signature Wrapping (XSW)* [MA05] can appear. In these attacks the attacker modifies the message structure by injecting forged elements, which do not invalidate the XML Signature. The goal of this alteration is to change the message in such a way that the application logic and the signature verification module use different parts of the message: The receiver should successfully verify the XML Signature, but the application logic should process the bogus element. The at-

tacker thus circumvents the integrity protection and the origin authentication of the XML Signature and can inject arbitrary content.

For explanation purposes, assume that user Bob communicates with a Web Service using signed SOAP messages. Attacker Eve eavesdrops one of Bob's messages, which executes the function `DeleteUser`. Her goal is to modify this message to execute the function `AddAdmin` with parameter `Eve`. Figure 3.1 shows a simple XSW attack example on the eavesdropped SOAP message executed by Eve. In this example, Eve moves the original SOAP body containing `DeleteUser` element to a `Wrapper` element in the SOAP header. Afterwards, she creates a SOAP body with a new `Id="attack"` and defines a new function `AddAdmin`. A vulnerable Web Service processes this message as follows. First, it attempts to verify the signature over an element with the `Id="123"`. The signature verification module can successfully find a referenced `Body` element in the `Wrapper` element. As the Id of the referenced `Body` element stays the same and as it is not altered, the signature can be successfully verified. The business logic searches for the `Body` element placed directly in the `Envelope` root element (this is a typical proceeding in the Web Services implementations). Thus, it finds the newly created SOAP body and invokes the newly inserted function `AddAdmin`. We refer to this attack as a *classical XSW attack*.

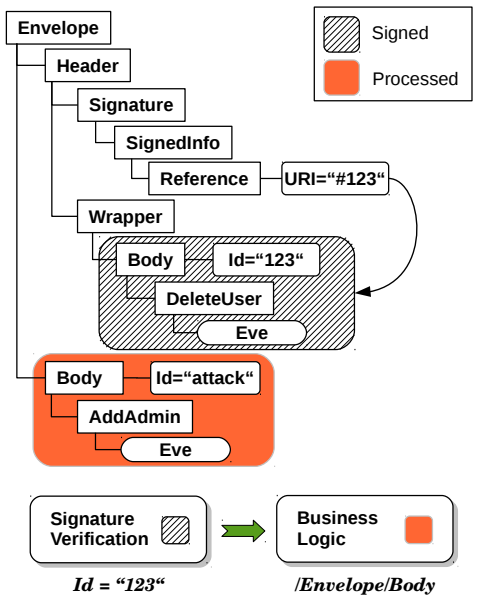


Figure 3.1: *A simple XSW attack: The attacker moves the original signed content to a newly created* `Wrapper` *element. Afterwards, she creates an arbitrary content with a different* `Id`*, which is invoked by the business logic.*

Remark: In the following sections, suppose that the `Id` attributes are of type `ID` (see Section 2.1 for more details on `ID` type attributes).

Attack Prerequisites. XSW attacks can be executed by an attacker with far fewer resources than the classical network based attacker from cryptography: The attacker can succeed even if he does not control the network. He does not

need realtime eavesdropping capabilities. The prerequisites to execute an XSW attack are:

1. The attacker needs access to the receiving endpoint. XML Signatures are typically applied to secure Web Services and SAML applications. Interfaces of these applications are typically public.

2. The attacker needs one signed message valid for the attacked domain or interface. He can gain this message by different means: He can eavesdrop it, find it on the Internet (e.g. in a forum), or he can register as a valid user on the domain and use XSW to impersonate a different user. We describe these scenarios in the following sections.

In general, the XSW attack is possible due to separation of security processing and business logic. This prerequisite is given in most of the practical systems and frameworks applying XML Signatures. On the one hand, the developers apply libraries and frameworks to process complex XML Security processing steps. On the other hand, they implement a simple business logic evaluation on their own.

3.2 All Your Clouds Are Belong to Us – Security Analysis of Cloud Management Interfaces

The cloud computing paradigm offers users cost-effective storage resources and computational services. The users can take advantage of "unlimited" computation power and on-demand reserve new storages or virtual machines. The cloud services can be managed using different interfaces, e.g. Web interfaces, XML or JSON Web Services. These interfaces allow to execute the whole communication between the user and the cloud provider (e.g. start, stop or upload new virtual machines, or upload and download stored data). Thus, a vulnerability in a cloud interface could be sufficient to compromise a customer's account.

In this section, we present XSW attacks on the authentication mechanisms used in Amazon EC2 and Eucalyptus SOAP control interfaces. We demonstrate that these control interfaces were highly vulnerable to several new and classical variants of XSW attacks. For these attacks, knowledge of a single signed SOAP message is sufficient to attain a complete compromise of the security within the customer's account. The reason for this easiness is that one can generate arbitrary SOAP messages accepted by this interface from only one valid signature. To make things even worse, in one attack variant, knowledge of the (public) X.509 certificate alone enabled a successful execution of an arbitrary cloud control operation on behalf of the certificate owner.

Contribution. The contribution of this section can be enumerated in the following main points:

1. We propose to view the Cloud control interface security as an important and challenging research topic, additionally marked by its high impact factor.

2. We show that XSW attacks are a serious threat, as they are yet to be resolved or understood.

3. We devise a methodology of investigating "black box" Web Services implementations by making claims as to how SOAP message verification works in the Amazon EC2 cloud.

Responsible disclosure. All the vulnerabilities found throughout our research were reported to the Amazon and Eucalyptus security teams. We worked closely with both security teams and put forward the solutions for fixing the issues that we identified. Subsequently, we monitored the countermeasures as they were being implemented.

Paper. This section is based on the paper *All Your Clouds Are Belong to Us – Security Analysis of Cloud Management Interfaces* published at the ACM Cloud Computing Security Workshop [SHJ+11]. In addition to the analysis of XSW attack possibilities, Mario Heiderich managed to mount different XSS attacks on the Amazon and Eucalyptus cloud interfaces. These attacks are out of scope of this thesis and can be found in [SHJ+11].

My contribution to this research lay in the XSW attacks' development, their practical execution on Amazon EC2 cloud interfaces and their analysis. Moreover, I supervised a thesis by Xiaofeng Lou [Xia11], who investigated XSW attacks on Eucalyptus cloud interfaces.

3.2.1 Cloud Security – Related Work

Cloud security is an emerging research topic, already addressed in many academic and research-based publications. A good overview of cloud security issues is given by Molnar and Schechter who investigated advantages and disadvantages of storing and processing data by the public cloud provider with regards to security [MS10]. The authors detail the new kinds of technological, organizational, and jurisdictional threats resulting from the cloud usage, as they also provide a selection of countermeasures.

Ristenpart et al. analyzed the physical placement of new allocated virtual machines in Amazon EC2 [RTSS09]. They showed that an attacker can iteratively allocate new instances until one is placed on the same physical machine as the victim's instance. Afterwards, the attacker can exploit data from the victim's running instance using cross-VM side-channel attacks. In this regard, the authors propose the usage of blinding techniques to make cross-VM side-channel attacks unfeasible.

XSW attacks were first described in 2005 [MA05]. However, until the year of our publication, only one research paper analyzed the practical impact of XSW attacks on real-world applications. In 2009, Gruschka and Lo Iacono examined the security of the Amazon EC2 cloud's interfaces [GL09]. They showed how XSW attacks can be performed to attack Amazon's EC2 service. They presented a vulnerability that enabled an attacker in possession of a signed control message from a legitimate user to execute any operation on the cloud control interface. However, the authors did not apply their attack on the `Timestamp` elements

included in control messages. Thus, the attack only worked with fresh messages not older than five minutes.

3.2.2 Cloud Control Interfaces

From a conceptual standpoint, cloud services need some form of cloud control which enables users to manage and configure the service, whilst also preserving access to the stored data. In IaaS (Infrastructure As A Service) clouds, the control interface allows to instantiate machines, as well as to start, pause, or stop them. Machine images can be created or modified, and the links to persistent storage devices must be configured. It is therefore quite undebatable that the security of a cloud service highly depends on robust and effective security mechanisms for the cloud control interfaces.

3.2.2.1 Amazon EC2 and Eucalyptus Cloud Control Interfaces

One of the most prominent cloud computing platforms is Amazon Web Services (AWS) [AWS13]. It furnishes an array of products, e.g. computation services, content delivery, databases, messaging, payments, storage, and others, all made available to arbitrary companies and end-users.

Amazon EC2 is a service that provides users with scalable computation capacity. Across a certain time period, the users can run their own virtual instances with customizable (virtual) hardware and operating system properties. Upon starting an instance using the EC2 cloud control, the user can for example access the instance over SSH (for Linux/Unix machines). Cryptographic keys for the SSH login may be similarly generated via the EC2 cloud control.

Two main interfaces are primarily responsible for EC2 services' control. The first one is a browser-based web application (AWS Management Console). Using this interface, the user can check the status of the instances, run new instances, generate keys for communication with the running instances over SSH, or generate keys and certificates for controlling the cloud over SOAP- and REST-based Web Services. The web application control interface is not intended for customers who own a huge number of machines that are dynamically started and stopped according to the computation power and storage needs. For this reason, AWS offers a complementary Web Services interface that allows users to control their cloud over SOAP and REST-based services. Communication with both interfaces can be automated.

The SOAP interface provides users with the same functionality as the AWS Management Console. The structure of a SOAP message used for the communication with the EC2 SOAP interface is depicted in Figure 3.2. As can be seen, the depicted SOAP message contains an XML Signature applied according to the WS-Security specification. The `Timestamp` element includes the message expiration date and thereby ensures its freshness. `BinarySecurityToken` [HBKMN07] includes a Base64 encoded X.509 certificate that identifies the user. The `Signature` element contains an XML Signature [ERS+08] authenticating the message issuer and protecting the integrity of the `Timestamp` and `Body` elements. The `MonitorInstances` element indicates the (sample) operation to be

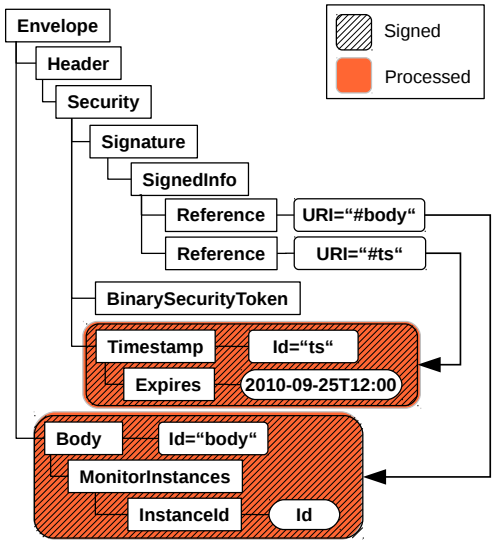


Figure 3.2: *SOAP request sent to the EC2 interface.*

called on the AWS interface. This operation is used to gather status information on the user's EC2 virtual machine instances.

In addition to the EC2 SOAP interface described above, AWS provides three other types of Web Services interfaces: S3 SOAP Web Services interface with custom signature validation, AWS REST-based Web Services interface, and AWS XQuery Web Services interface. We exclude them from the discussion in this thesis as they are not involved in the attacks we are covering.

While Amazon Web Services operates as a public cloud provider, the need for private cloud environments fostered the development of freely available open source implementations of cloud systems. Among other advancements, the Eucalyptus cloud implementation [Euc13] gained a lot of public attention and made its way into the well-known Ubuntu operating system (Ubuntu Server Edition).

As far as functionality is concerned, the cloud management interfaces of Eucalyptus were designed to copy the Amazon cloud control interface in order to support a switch from the prominent pre-existent Amazon EC2 cloud to a Eucalyptus cloud. Every Eucalyptus installation by default provides almost the exact same interfaces as the Amazon EC2 cloud. Nevertheless, it must be stressed that the functionality and security mechanisms were implemented independently.

3.2.3 AWS SOAP Interface Attacks

Within the scope of our security analysis of Amazon's EC2 cloud control interfaces (performed at the end of 2010), we carried out an investigation of the SOAP message processing of the cloud control with respect to the applicability of XSW attacks.

3.2.3.1 Vulnerability Analysis

We found out that the overall structure of incoming SOAP messages – defined by the XML Schema [WF04] – was not checked at all. Therefore, it became possible to add, remove, duplicate, nest, or move arbitrary XML fragments within the SOAP request message, without the message's validity being affected. We performed a set of SOAP requests that exploited this flexibility in SOAP message design. Since the Amazon EC2 SOAP interface replied with quite meaningful SOAP fault messages in case of an error, we were able to easily test the Amazon EC2 SOAP interface for its XSW resistance.

3.2.3.1.1 AWS Duplicated-Id XSW Attack Variant Type 1. The starting point for our security analysis was derived from the previous work done by Gruschka and Lo Iacono in 2009 [GL09]. Their attack used a forged SOAP request with a duplication of the signed SOAP body. Likewise, we duplicated the SOAP body of the `MonitorInstances` message, changing the operation in the first SOAP body to `CreateKeyPair`. We sent the forged message to the EC2 SOAP interface for verification. The message was successfully validated, and a new key pair for SSH access to an EC2 instance was created. Conclusively, the EC2 SOAP interface validated the XML Signature only for the second SOAP body (which was not modified and hence verified successfully), but it used the first SOAP body for determining operation and parameter values. Supplementary tests with other operation names indicated that an attacker could use this technique to trigger arbitrary operations. Still, all attacks had to be performed within the five minute time frame enforced by the timestamp.

A slight attack variant circumvents the timestamp verification, and therefore extends the attack to be independent of the time passing. Having duplicated the `Timestamp` element in the security header – the same approach used for the SOAP body before – we observed a similar behavior of the verification component: The first timestamp was compared to the current time, the second timestamp was verified for integrity. To sum up, this attack variant (shown in Figure 3.3) could be performed using arbitrary signed SOAP messages, even when their timestamp had already expired. The variant described above clearly breaks the timing constraints mechanism used in the EC2 SOAP interface, proving its potential for being used for execution of arbitrary operation invocation.

It is important to mention that the Id attributes of both, *wrapped* and *executed*, elements needed to be identical as otherwise the message was rejected.

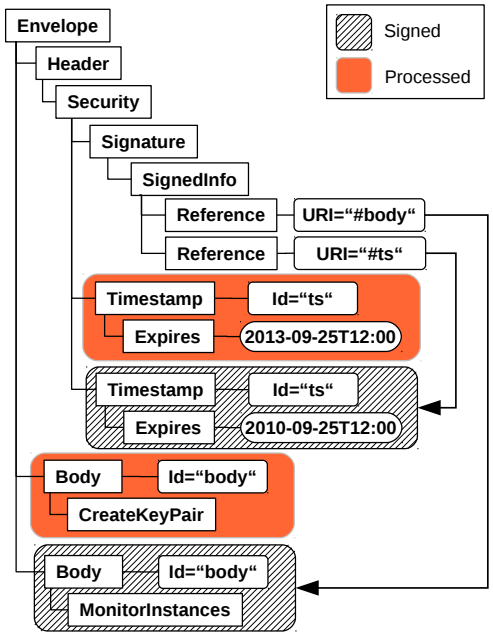


Figure 3.3: *Duplicated-Id XSW attack type 1.*

3.2.3.1.2 AWS Duplicated-Id XSW Attack Variant Type 2. After reporting the first variant to the AWS security team, we were informed about the provision of a fix that disallowed duplications of the timestamp element. From this point forward, all the SOAP messages with duplicated timestamps *in the SOAP message's security header* were refused. However, it was still possible to have several `Body` elements with the same `Id` attribute value within one SOAP message. For this reason, we continued our analysis focusing on moving the signed timestamp element to other positions within the document tree.

Figure 3.4 illustrates the first adapted XSW attack on the EC2 SOAP interface. As it was no longer possible to duplicate the timestamp within the security header, we created three different `Body` elements, and moved the originally signed timestamp element into the second body. Sending this forged SOAP message to the EC2 SOAP interface revealed that this attack technique indeed worked. The timestamp in the second body and the whole third body were checked by the signature verification component. The timestamp in the security header was attested for expiration, and the first body was interpreted as determining the operation and parameter value.

We also exposed other attack variants. For example, it was possible to duplicate the full SOAP security header. The first header included the timestamp that would be validated for its recency, and the timestamp in the second security header was verified by the signature validation component. Again, the first `Body` element was executed, and the last one was verified for integrity. When compared to the type 1 vulnerabilities, the same prerequisites and the same impact characterized the type 2 class.

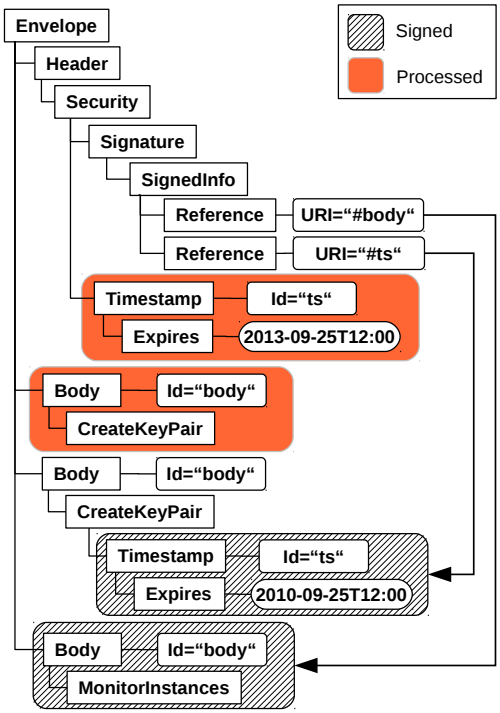


Figure 3.4: *Duplicated-Id XSW attack type 2.*

3.2.3.1.3 AWS Signature Exclusion Bug. The prerequisite for the above described XSW attacks is that an attacker manages to obtain (e.g., eavesdrop, copy from a log file, etc.) a SOAP message with a valid XML Signature. Although this seems like a rather small obstacle (see also Section 3.2.3.2), we detected another vulnerability with even less prerequisites: In the absence of an XML Signature, the signature verification component did not monitor any XML Signature at all, but nevertheless treated the message as validly signed. The task of user identification and authorization took place in other components relying solely on the X.509 certificate data from the `BinarySecurityToken` element, which can be present even if there is no signature. Hence, that SOAP request message was authorized to trigger operations on behalf of the owner of the X.509 certificate. For completeness, the message is depicted in Figure 3.5.

To conclude, while performing an arbitrary SOAP request for any of the EC2 SOAP interface operations, an attacker needs only the public X.509 certificate of the victim. Since X.509 certificates are by definition considered to constitute public data, harvesting them from the Internet is not a major challenge for an attacker. Moreover, Mario Heiderich described in our original paper [SHJ+11] a download link XSS vulnerability that could have allowed us to gather valid certificates.

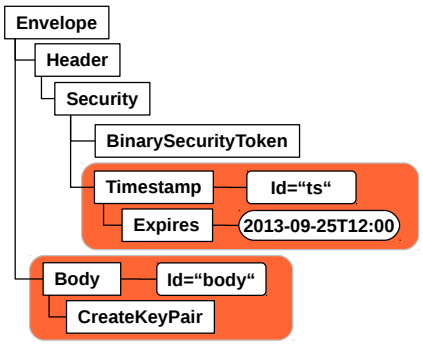


Figure 3.5: *Signature Exclusion bug in the AWS control interface allowed us to execute an arbitrary function on behalf of our victim, by knowing only the victim's public certificate.*

3.2.3.2 Attack Preparation

Based on the attack techniques highlighted so far, we continued our security analysis of the EC2 cloud control SOAP interface by surveying the degree of difficulty it takes for an attacker to get to the point where he can perform a successful XSW attack.

Knowledge of a single validly signed SOAP request message remains the only prerequisite for an XSW attack. Gathering such a SOAP message turned out to be quite an easy endeavor: Many AWS developers seeking assistance post their SOAP requests on the AWS forums, which turned out to be a convenient source for signed SOAP messages. During the first attempt, we immediately recovered about 20 SOAP requests from multiple users of `solutions.amazonwebservices.com` and `developer.amazonwebservices.com`. A slightly more sophisticated search would very likely lead to even more results.

In addition, it turned out that an attacker could also execute Man-in-the-Middle attacks by claiming himself as being the AWS control interface. This could have been possible since the AWS developer framework did not correctly check SSL certificates. This observation was made one year after our publication by Georgiev et al. [GIJ+12], who investigated validation of SSL certificates in a large number of non-browser applications, including the AWS framework.

3.2.3.3 Analysis of the AWS Security Framework

Based on the attack findings described above, we performed an extensive security analysis of the Amazon EC2 cloud control SOAP Interface. By sending SOAP messages with different types of errors for different processing components of the AWS framework, we tried to determine the general architecture that Amazon uses for its SOAP interface services. Relying on publicly known best practices, we assumed the AWS interface consisted of a set of modules that perform specific tasks for every SOAP message received at the service interface. The order of these modules, and the amount of verification performed therein usually is an important parameter of whether and how a typical Web Service specific attack

can be accomplished. Our goal was to gain as much information on this internal topology as possible, for a full view on the EC2 SOAP interface implementation.

Through sending hand-crafted SOAP messages to the EC2 interface, we effectuated a series of SOAP-based tests. Each of these SOAP messages was carrying a different type of fault, causing the SOAP server implementation to raise diverse errors and respond with different types of SOAP fault messages. For instance, upon processing a SOAP message that contained a basic syntactical fault in the SOAP message's XML structure (e.g. a missing `'>'` character in the XML syntax) we received a SOAP fault message with a general XML structure as illustrated in Figure 3.6 (left). Please note the way the XML tag names are equipped with prefixes (e.g. `"SOAP-ENV"`). Though usually there is no semantic relevance for the choice of these namespace prefixes, they nevertheless tend to change for different XML frameworks, hence allowing differentiation of a SOAP fault message's origin.

A second test was performed with the use of SOAP messages with correct XML syntax but faults on the semantic level. As a result, the EC2 SOAP interface responded with a SOAP fault message as well, but this time there was a remarkable difference in the way the XML data was serialized. Figure 3.6 (right) shows an example of such a SOAP fault, received in reply to a SOAP request with an expired timestamp. Note the differences in how the XML namespaces are chosen (here: `"soapenv"`). Hence, it is reasonable to assume that both SOAP fault messages were generated by different SOAP modules or frameworks.

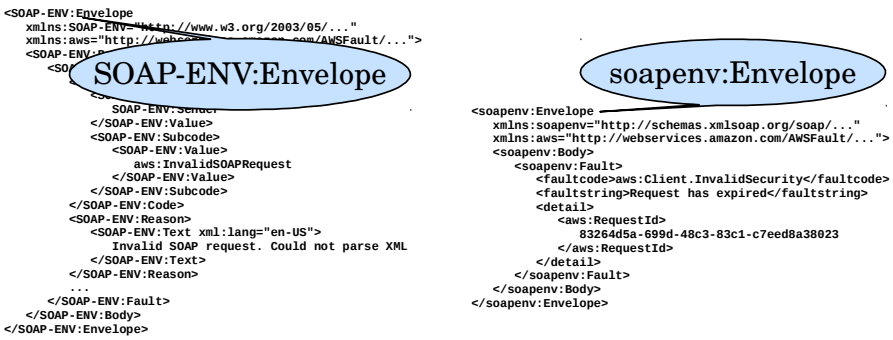


Figure 3.6: *SOAP fault messages for a SOAP request with a syntactical (left) and semantic fault (right).*

Similarly, we used SOAP messages containing other types of faults, such as data type violations in operation parameters, invalid XML Signatures, or untrusted X.509 certificates. We also performed tests with SOAP messages that contained two or more of these faults at the same time, in order to see which fault the EC2 SOAP interface complained about first. This way, we managed to identify the order in which the particular tasks were performed, and the ways in which they accessed the XML data from the SOAP messages.

The results of this analysis are depicted in Figure 3.7. As can be seen, the AWS SOAP interface processes the incoming SOAP messages in (at least) four separate logical steps, implemented by separate modules.

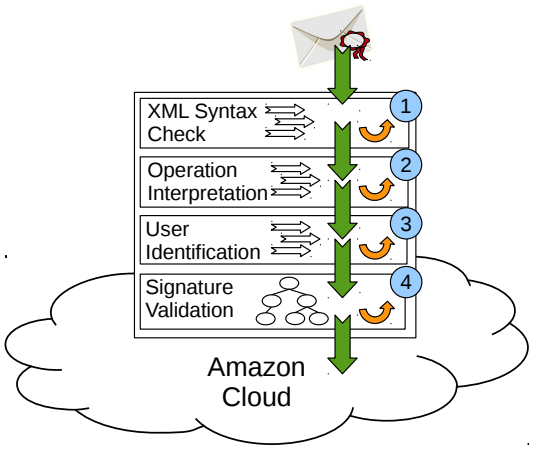


Figure 3.7: *Amazon EC2 SOAP message processing architecture consists of four separate logical steps.*

XML Syntax Check. In a first step, the XML parser performs an XML syntax check for *well-formedness*. If even a single one of the XML tags is not properly closed or a namespace declaration is missing, the interface returns a SOAP fault. This step is most probably done by an independent XML parser, as the namespaces and the XML structure in the SOAP responses differed from the SOAP responses that were returned after processing of well-formed SOAP requests (see above).

Operation Interpretation and Time Constraints. In a second step, the XML processor reads and interprets the content of the SOAP request. First, it validates the time given within the `Timestamp` element. Then, it reads the `Body` element, validating the contained operation name (e.g. `MonitorInstances`) and the number of its parameters. In all probability, this is obtained by using a *streaming* XML parser (such as SAX or StAX), since on duplication of the `Timestamp` or `Body` elements only the first occurrence of that element is interpreted. This can be deemed as typical behavior for implementations that use streaming-based XML processing approaches, since these tend to interrupt message parsing immediately after having processed the first occurrence of the particularly interesting XML element.

As can be seen by all the XSW variants, the `Id` attributes of the wrapped and executed elements needed to stay *equal*. Therefore, we assume that the Ids of processed elements are extracted and passed to the further XML Signature verification step.

User Identification and Authorization. A third step attempts to identify the user by processing the X.509 certificate contained in the `BinarySecurityToken` element. The certificate determines the customer account of the AWS user, thus performing solely the SOAP request's authorization task.

XML Signature Verification. The last step before the operation in the SOAP message is executed, comprises of XML Signature verification. The `URI` attributes of the XML Signature are dereferenced, i.e. the XML processor

searches for XML elements that contain an `Id` attribute with the same identifier string value as indicated in the `URI` attribute of the `Reference` element. Hence, for regular SOAP requests, this search returns the `Timestamp` and `Body` elements as determined within the second processing step. Then, hash value calculation and signature verification are performed for those elements. If this task fails, the SOAP message gets rejected, otherwise the operation determined in the step two component is performed on the Amazon EC2 cloud system for the user identified in step three.

In addition to accommodating verification of signature and digest values, this step checks if the elements being validated include the same `Id` attributes as the elements being processed in step two. This grants the approval for the communication between the modules for *Operation interpretation* and *Signature validation*, which were there to attempt prevention of the XSW attacks. However, allowing for multiple equal `Id` attributes in the SOAP message has opened possibilities for new variants of XSW attacks.

For the XML processing model of the last step we suppose that the `URI` dereferencing and processing the signed elements is embedded in a tree-based XML Parser. Tree-based parsers are typically applied by validating complex XML Signatures. We suppose that this processing step was performed at the end due to the inefficient memory model behind the tree-based XML parsers. Thus, invalid SOAP requests could have been filtered out by an efficient streaming-based XML parser in the preceding steps.

3.2.4 Eucalyptus SOAP Interface Attacks

To analyze the Cloud control interface of Eucalyptus, we used a default cloud installation of the Ubuntu Server Edition, which provides an extended version of the original Eucalyptus framework [Euc13].

3.2.4.1 Vulnerability Analysis

During our investigation, we determined that XSW attack techniques could be successfully applied to Eucalyptus. However, the techniques applied in the Amazon case were not functional, since Eucalyptus detected multiple identical `Id` attribute values, and rejected such SOAP messages. More precisely, in our analysis we discovered that an attacker could use a slightly modified *classical* XSW attack technique to execute an arbitrary function without a time limitation. We give an example of a SOAP message of that sort in Figure 3.8.

As the Eucalyptus SOAP interface validated the format of incoming SOAP messages against an XML Schema, the attacker could not duplicate the SOAP `Body` element or copy the signed elements directly to the SOAP header. For the attack to be feasibly executed, signed elements had to be copied to a newly created deeper-nested element. For this purpose, we chose a duplicated security header element that does not violate the SOAP message XML Schema. Through this process, the attacker could move the signed body and the timestamp elements to this newly allocated place.

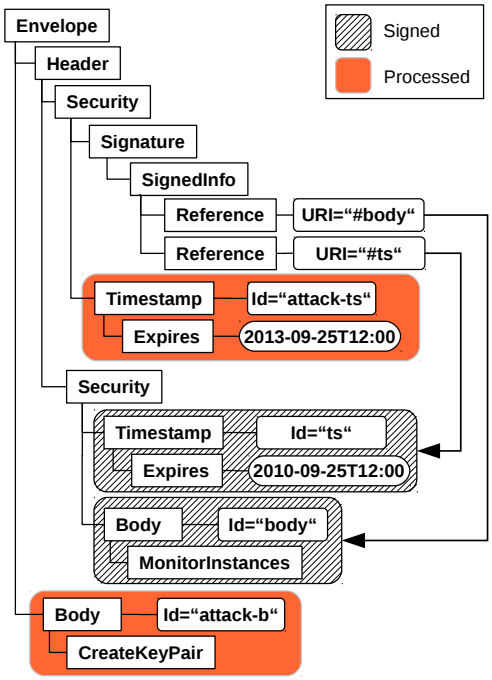


Figure 3.8: *Successful XSW attack on the Eucalyptus SOAP interface.*

Remark: This should be seen as a proof that XML Schema validation alone does not protect against XSW attacks.

In addition to the SOAP message structure, the Eucalyptus validation framework checked for duplicated `Id` attribute values in the XML document. Conversely, it did not check if the processed data items have the same `Id` values as the signed data. Therefore, it was possible to use different `Id` attributes for the *executed* `Body` and `Timestamp` elements, which then had a potential to convey arbitrary content.

3.2.4.2 Attack Prerequisites

To execute an attack on Eucalyptus, an attacker had to be in possession of a single validly signed SOAP message of the victim. It must be stressed once again that SSL does not prevent such attacks, since the SOAP messages in question can be retrieved in many different ways besides the network sniffing, see our considerations in Section 3.2.3.2.

3.2.4.3 Analysis of the Eucalyptus Security Framework

Eucalyptus Framework is an open source private cloud provider. Therefore, there was no need for an extensive ”black box” analysis. After analyzing the source code we found out that Eucalyptus uses Apache Rampart [Apa12a] – the security module of a widely used Apache Axis2 Web Services Framework. Further tests of the Rampart module using various deployment properties proved its vulnerability to XSW attacks.

3.3 On Breaking SAML: Be Whoever You Want to Be

Typical Internet users have many identities for different websites and web services. This leads to the fact that users choose weak passwords for their authentication, forget their passwords, or even their identities. To overcome this problem, Single Sign-On (SSO) was developed. In this approach, the users authenticate only once to a trustworthy Identity Provider (*IdP*). After a successful login, the *IdP* issues security tokens on demand. These tokens are used to authenticate to Relying Parties (*RP*).

A simplified SSO scenario is depicted in Figure 3.9. In this setting, a user logged-in by the *IdP* first visits the desired *RP* (1). The *RP* issues a token request (2). This token is sent to the user (3), who forwards it to the *IdP* (4). The *IdP* issues a token response for the user including several claims (e.g. his access rights or expiration time). In order to protect the authenticity and integrity of the claims, the token is signed (5). Subsequently, the token is sent to the user (6), who forwards it to the *RP* (7). The *RP* validates the signature and afterwards grants access to the protected service or resource, if the user is authorized (8). This access control decision is based on the claims in the validated token.

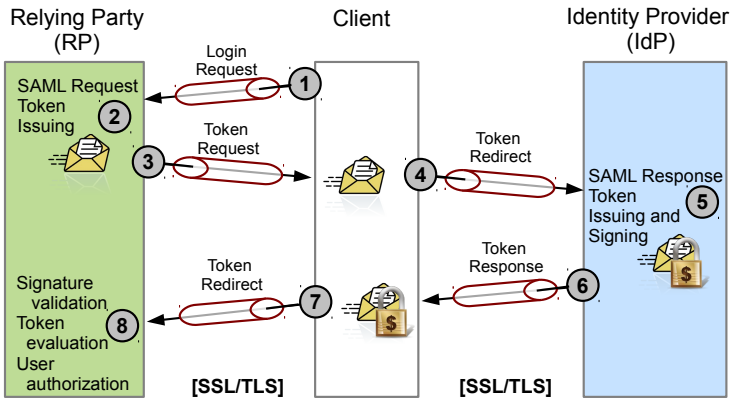


Figure 3.9: *A typical Single Sign-On (SSO) scenario: The user visits the RP, which generates a request token. He redirects this token to the IdP. The issued token is sent to the user and forwarded to the RP. Even though the channel is secured by SSL/TLS, the user still can see the token.*

SAML Providers and Frameworks. The above described authentication process can be applied and is implemented in different scenarios, e.g. in Web Services or browser-based SSO. One of the standards used for definitions of tokens in these scenarios has become the XML standard SAML [CKPM05]. Protection of authenticity and integrity in the SAML assertions is realized by application of XML Signatures [ERS+08]. Therefore, breaking the XML Signature application leads also to breaking the token's integrity.

In this section, we show the results of our practical XML Signature application analysis in SAML executed on frameworks and systems summarized in Table 3.1.

Framework	Language	Application
Apache Axis2	Java	WSO2 Web Services
Guanxi	Java	Sakai Project (`www.sakaiproject.org`)
Higgins 1.x	Java	Identity project
IBM XS40	XSLT	Enterprise XML Security Gateway
JOSSO	Java	Motorola, NEC, Redhat
WIF	.NET	Microsoft Sharepoint 2010
OIOSAML	Java, .NET	Danish eGovernment (e.g. `www.virk.dk`)
OpenAM	Java	Enterprise-Class Open Source SSO
OneLogin	Java, PHP, Ruby, Python	Joomla, Wordpress, SugarCRM, Drupal
OpenAthens	Java, C++	UK Federation (`www.eduserv.org.uk`)
OpenSAML	Java, C++	Shibboleth, SuisseID
Salesforce	—	Cloud Computing and CRM
SimpleSAMLphp	PHP	Danish eID Federation (`www.wayf.dk`)
WSO2	Java	WSO2 products (Carbon, ESB, ...)

Table 3.1: *Analyzed SAML frameworks and providers: The columns give information about programming language (if known) and application in concrete products or frameworks.*

See Section 2.9 for more details on these frameworks and systems.

Contribution. In this section, we present an in-depth analysis of 14 SAML frameworks and systems. During this analysis, we found critical XSW vulnerabilities in eleven of these frameworks. This result is alarming given the importance of SAML in practice. Our attacks present new classes of XSW attack. We show that these attacks could also be applied if the whole document is signed or if specific countermeasures are applied.

Second, these vulnerabilities are exploitable by an attacker with far fewer resources than the classical network based attacker from cryptography: A single signed SAML assertion is sufficient to completely compromise a SAML issuer/Identity Provider. Using SSL/TLS to encrypt SAML assertions, and thus to prevent adversaries from learning assertions by intercepting network traffic, does not help either: The attacker may e.g. register as a regular customer at the SAML issuer, and may use his own assertion to impersonate other customers.

Last, our results confirm that XSW vulnerabilities constitute an important and broad class of attack vectors. There is no easy defense against XSW attacks: Contrary to common belief, even signing the whole document does not necessarily protect against them.

Responsible Disclosure. All vulnerabilities found during our analysis were reported to the responsible security teams. Accordingly, in many cases, we closely collaborated with them in order to patch the found issues.

Paper. This section is based on the paper *On Breaking SAML: Be Whoever You Want to Be* presented at the USENIX Security Symposium [SMS+12].

The original idea of attacking SAML-based frameworks stemmed from Meiko Jensen. My responsibility lay in a practical analysis of the following SAML frameworks: Apache Axis2, Guanxi, OpenAthens, OpenSAML[1], Salesforce, and WSO2. Moreover, I supervised theses written by Marco Kampmann [Mar11a,

[1]The analysis was performed in an email communication between the OpenSAML developer Scott Cantor and me.

Mar11b], who investigated security of IBM XS40 and JOSSO, and developed the first version of our XSW penetration testing tool, which was used to reevaluate security of WSO2 and Salesforce SAML interfaces. Analysis of IBM XS40 was supervised by Meiko Jensen. The remaining frameworks were analyzed by Andreas Mayer. Jörg Schwenk formally analyzed countermeasures against XSW attacks (see our original paper for this analysis).

3.3.1 SAML and Single Sign-On – Related Work

Since SAML offers very flexible mechanisms to make claims about identities, there is a large body of research on how SAML can be used to improve identity management (e.g. [HJK08, YsJ10]), and other identity-related processes like payment on the Internet [LS10, TFP+06]. In all these applications, the security of all SAML standards is assumed.

In 2003, T. Groß initiated the security analysis of SAML [Gro03] from a Dolev-Yao point of view, which was formalized in [BG05]. He found, together with B. Pfitzmann [GP06], deficiencies in the information flow between the SAML entities. Their work influenced a revision of the standard.

In 2008, Armando et al. [ACC+08] built a formal model of the SAML 2.0 Web Browser SSO protocol and analyzed it with the model checker SATMC. By introducing a malicious *RP* they found a practical attack on the SAML implementation of Google Apps. Another attack on the SAML-based SSO of Google Apps was found in 2011 [ACC+11]. Again, a malicious *RP* was used to force a user's web browser to access a resource without approval. Thereby, the malicious *RP* injected malicious content in the initial unintended request to the attacked *RP*. After successful authentication on the *IdP* this content was executed in the context of the user's authenticated session.

The fact that SAML protocols consist of multiple layers was pointed out in [Cha06]. In this paper, the *Weakest Link Attack* enabled adversaries to succeed at all levels of authentication by breaking only at the weakest one.

In recent years, many researches pointed out the importance of SSO protocols [WCW12, SB12, BML+13]. The authors analyzed the security quality of commercially deployed SSO solutions applying OpenID [Fou07] or OAuth [Har12]. They showed serious logic and implementation flaws in high-profile *IdP*s and *RP*s (such as OpenID, Facebook, or JanRain), which allowed an attacker to sign in as the victim user. SAML-based SSO systems were not analyzed.

3.3.2 Attack Theory

In this section, we first characterize the assumed SAML threat model. Second, we describe the basic attack principle that underlies our analysis of the 14 frameworks. In contrast to Section 3.2, in this section we first theoretically analyze different attack permutations, which are later applied practically to the analyzed SAML frameworks.

3.3.2.1 Threat Model

As a prerequisite the attacker requires an arbitrary signed SAML message. This could be a single assertion A or a whole document with an embedded assertion, and its lifetime can be expired. After obtaining such a message, the attacker modifies it by injecting evil payload content. We call this content evil assertion, EA.

In our model, we assume two possibilities to get a valid signed SAML assertion A:

1. The attacker registers as a user of an Identity Provider IdP. He then receives, through normal interaction with IdP, a valid signed SAML assertion A making claims about the attacker. The attacker now adds additional claims EA about any other subject S, and submits the modified document to RP.

2. The attacker retrieves SAML assertions from the Internet (similarly to the attacker described in Section 3.2.3.2). This can be done either by accessing transmitted data directly from unprotected networks (sniffing), or in an "offline" manner by analyzing proxy or browser caches. Since SAML assertions should be worthless once their lifetime expired, they may even be posted in technical discussion boards, where the attacker may access them.

3.3.2.2 Basic Attack Principle

As described in Section 2.7, XML Signatures can be applied to SAML assertions in different ways and placed in different elements. The only prerequisite is that the `Assertion` element or the protocol binding element (ancestor of `Assertion`) is signed using an enveloped signature with `Id`-based referencing. In this section, we analyze the usage of SAML assertions in different frameworks and the possibilities of inserting malicious content. Generally, SAML assertions and their signatures are implemented as depicted in Figure 3.10:

1. The first possible usage of signatures in SAML assertions is to insert the XML Signature *S1* as a child of the SAML assertion *A1* and sign only the `Assertion` element *A1*. This type can be used independently of the underlying binding (SOAP or HTTP POST).

2. The second type of signature application in SAML signs the whole protocol binding element R. The XML Signature can be placed into the SAML assertion *A1* or directly into the protocol binding root element R. This kind of signature application is used in different SAML HTTP bindings, where the whole `Response` element is signed.

3. It is also possible to use more than one XML Signature. The third example shows this kind of signature application: the inner signature *S1* protects the SAML assertion and the outer signature S additionally secures the whole protocol message. This kind of signature application is used by the SimpleSAMLphp framework, for example.

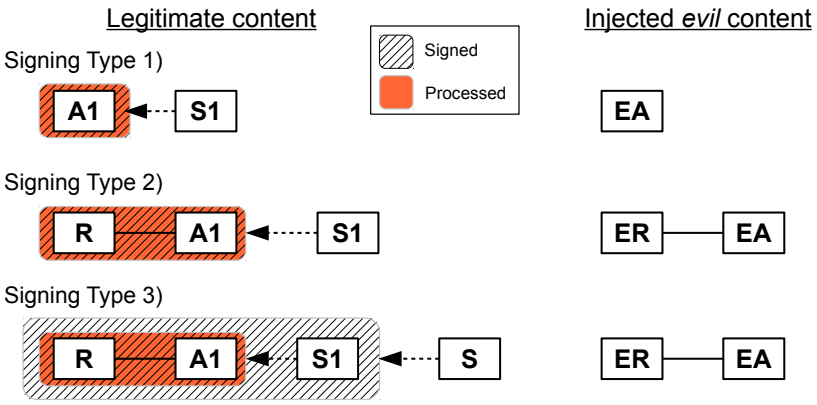


Figure 3.10: *Types of signature applications on SAML assertions on the left. The new malicious content needed to execute the attacks depicted on the right, accordingly.*

In order to apply XSW attacks to SAML assertions, the basic attack idea stays the same: The attacker has to create new malicious elements and force the assertion logic to process them, whereas the signature verification logic verifies the integrity and authenticity of the original content. In applications of the first signature type, the attacker only has to create a new *evil assertion* EA. In the second and third signing types, he also has to create the whole *evil root* ER element including the *evil assertion*.

3.3.2.3 Attack Permutations

The attacker has many different possibilities where to insert the malicious and the original content. To this end, he has to deal with these questions:

- At which level in the XML message tree should the malicious content and the original signed data be included?
- Which `Assertion` element is processed by the assertion logic?
- Which element is used for signature verification?

By answering these questions we can define different attack patterns, where the original and the malicious elements can be permuted (Figure 3.11). We thus get a complete list of attack vectors, which served as a guideline for our investigations.

For the following explanations we only consider signing type 1 defined in Figure 3.10. In this signing type only the `Assertion` element is referenced. The attack permutations are depicted in Figure 3.11. In addition, we analyze their SAML standard conformance and the signature validity:

1. Malicious assertion, original assertion, and signature are left on the same message level: This kind of XML message can have six permutations. None of them is SAML standard conformant, since the XML Signature

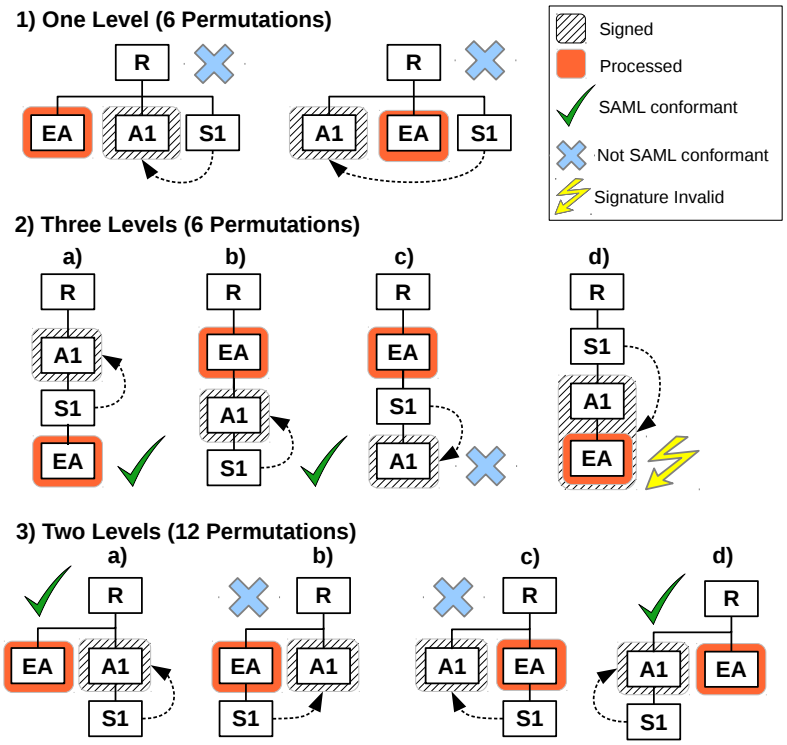


Figure 3.11: *Possible variants for XSW attacks applied on messages with one signed SAML assertion divided according to the insertion depth of the evil assertion EA, the original assertion A1 and the signature S1. The various permutations are labeled according to their validity and SAML-conformance.*

does not sign its parent element. The digest value over the signed elements in all the messages can be correctly validated. We can use this type of attack messages if the server does not check SAML conformance.

2. All the three elements are inserted at different message levels, as child elements of each other, which again results in six permutations: Messages 2-a and 2-b show examples of SAML standard conforming and cryptographically valid messages. In both cases the signature element references its parent – the original assertion *A1*. Message 2-c illustrates a message that is not SAML standard conformant as the signature signs its child element. Nevertheless, the message is cryptographically valid. Lastly, message 2-d shows an example of an invalid message since the signature would be verified over both assertions. Generally, if the signature is inserted as the child of the root element, the message would also be either invalid or not SAML standard conformant.

3. For the insertion of these three elements we use two message levels: Message 3-a shows an example of a valid and SAML conformant document. By constructing message 3-b, the signature element was moved to the new malicious assertion. Since it references the original element, it is still valid, but does not conform to the SAML standard.

The analysis shown above can similarly be applied to messages with different signing types (see Figure 3.10).

3.3.3 Practical Evaluation

We evaluated the above defined attacks on real-world systems and frameworks, which were introduced in Table 3.1. In this section, we present the results.

3.3.3.1 Signature Exclusion Attacks

We start the presentation of our results with the simplest attack type called *Signature Exclusion attack*. This attack relies on poor implementation of a server's security logic, which checks the signature validity only if the signature is included. If the security logic does not find the `Signature` element, it simply skips the validation step. Evaluation of this attack was motivated by our previous results described in Section 3.2.3, which showed that the AWS interface was vulnerable to Signature Exclusion attacks.

The evaluation showed that three SAML-based frameworks were vulnerable to these attacks: Apache Axis2 Web Services Framework, JOSSO, and the Java-based implementation of SAML 2.0 in Eduserv (other versions of SAML and the C-implementation in Eduserv were not affected).

By applying this attack on JOSSO and Eduserv the attacker had to remove the `Signature` element from the message, since if it was found, the framework tried to validate it. On the other hand, the Apache Axis2 framework did not validate the `Signature` element over the SAML assertion at all, even if it was included in the message. Apache Axis2 validated only the signature over the SOAP body and the `Timestamp` element. The signature protecting the SAML assertion, which is included separately in the `Assertion` element, was completely ignored.

3.3.3.2 Refined Signature Wrapping

Ten out of 14 systems were prone to refined XSW attacks.

Classified on the three different signature application types given in Figure 3.10, five SAML-based systems failed in validating Type 1 messages, where only the assertion is protected by an XML Signature. Figure 3.12 depicts the XML tree-based illustration of the found XSW variants. Higgins, Apache Axis2, and the IBM XS 40 Security Gateway were outfoxed by two attack variants. In the first variant it was sufficient to inject an evil assertion with a different `Id` attribute in front of the original assertion. As the SAML standard allows to have multiple assertions in one protocol element, the XML Schema validation still succeeds by this type of message. The second attack type embedded the original assertion as a child element into the evil assertion EA. In both cases the XML Signature was still standard conformant, as enveloped signatures were applied. This was broken in the case of OIOSAML by using detached signatures. In this variant the original texttttSignature element was moved into the EA, which was inserted before the legitimate assertion. The last shown permutation was applicable to the cloud services of Salesforce and the OpenAM

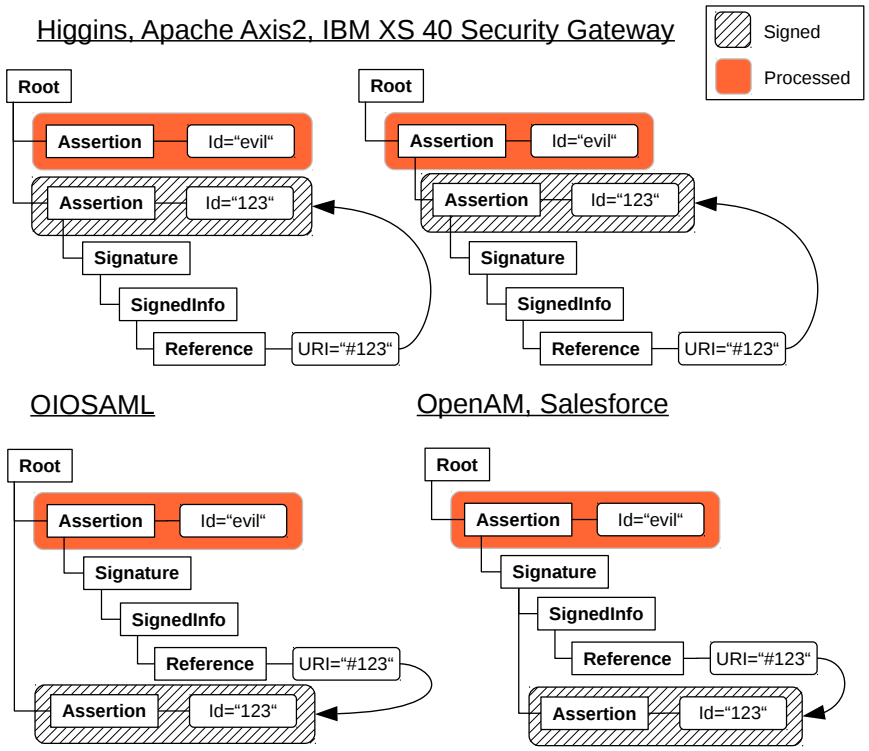


Figure 3.12: *XML tree-based illustration of refined XSW attacks found in Type 1 signature applications.*

framework. In this case, the original assertion was placed into the `Signature` element. As both implementations apply XML Schema for validating the schema conformance of a SAML message, this was done by injecting them into the `Object` element, which allows arbitrary content. Again, this is not compliant to the SAML standard because this mutation transforms the enveloped to an enveloping signature. Finally, the OneLogin Toolkits were prone to all shown attack variants as they did not apply XML Schema, validated the XML Signature independently of its semantic occurrence and used a fixed reference to the processed SAML claims (`/samlp:Response/saml:Assertion[1]`).

We found three susceptible implementations, which applied Type 2 messages, where the whole message is protected by an XML Signature. We depict the attacks on these implementations in Figure 3.13. In the Guanxi and JOSSO implementations the legitimate root element was inserted into the `Object` element in the original `Signature`. The `Signature` node was moved into the *ER* element, which also included the new evil assertion. In the case of WSO2, it was sufficient to place the original root element into the *ER* object. Naturally, someone would expect that enforcing full document signing would eliminate XSW completely. The both given examples demonstrate that this does not hold in practice. Again, this highlights the vigilance required when implementing complex standards such as SAML.

Finally, we did not find vulnerable frameworks that applied Type 3 messages,

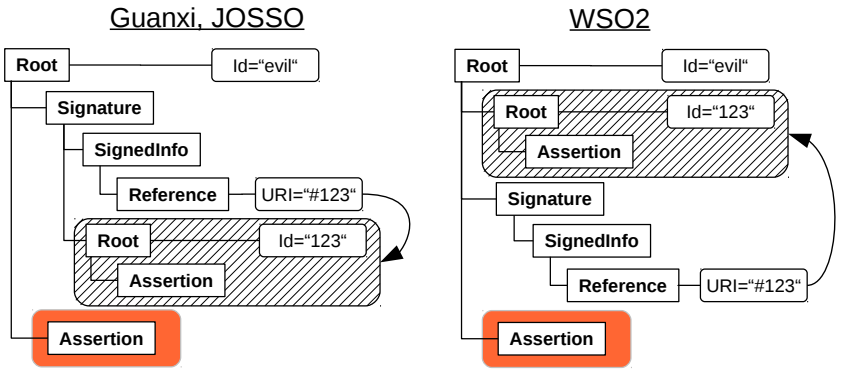


Figure 3.13: *XML tree-based illustration of refined XSW attacks found in Type 2 signature applications.*

where both the root and the assertion are protected by different signatures. Indeed, one legitimate reason is that most SAML implementations do not use Type 3 messages. In our practical evaluation, only SimpleSAMLphp applied them by default. Nevertheless, this does not mean that XSW is not applicable to this message type in practice.

3.3.3.3 OpenSAML Vulnerability

The attack vectors described above did not work against the prevalently deployed OpenSAML library. The reason was that OpenSAML compared the `Id` used by the signature validation with the `Id` of the processed assertion. If these identifiers were different (based on a string comparison), the signature validation failed. Additionally, XML messages including more than one element with the same `Id` were rejected. Both mechanisms are handled in OpenSAML by using the Apache Xerces library and its XML Schema validation method [The13]. Nevertheless, it was possible to overcome these countermeasures with a more sophisticated XSW attack.

As mentioned before, in OpenSAML the Apache Xerces library performs a schema validation of every incoming XML message. Therefore, the `Id` of each element can be defined by using the appropriate XML Schema file. This allows the Xerces library to identify all included `Ids` and to reject messages with `Id` values which are not unique (e.g. duplicated). However, a bug in this library caused XML elements defined with `any` content to be incorrectly processed. More concretely, the content of the elements defined as `<any processContents="lax">` were not checked using the defined XML Schema. Therefore, it was possible to insert elements with arbitrary – also duplicated – `Ids` inside an XML message. This created a good position for our wrapped content.

It is still the question which of the extensible elements could be used for the execution of our attacks. This depends on two processing properties:

1. Which element is used for assertion processing?
2. Which element is validated by the security module if there are two elements with the same `Id`?

Interestingly, the two existing implementations of Apache Xerces (Java and C++) handled element dereferencing *differently*. For C++, the attacker had to ensure that the original signed assertion was copied before the evil assertion. In the Java case, the legitimate assertion had to be placed within or after the evil assertion. In summary, if two elements with the same `Id` values occurred in an XML message, the XML security library detected only the first (for C++) or the last (for Java) element in the message. This property gave the attacker an opportunity to use e.g. the `Extensions` element for the C++ library, whose XML Schema is defined in Figure 3.14. The `Extensions` element is not the only possible position for our wrapped content. The schemas of SAML and XML Signature allow more locations (e.g. the `Object` element of the `Signature`, or the `SubjectConfirmationData` and `Advice` elements of the `Assertion`).

```
<element name="Extensions" type="samlp:ExtensionsType"/>
<complexType name="ExtensionsType">
  <sequence>
    <any namespace="##other" processContents="lax"
                 maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Figure 3.14: *XML Schema definition of the* `Extensions` *element.*

The previously described behavior of the XML Schema validation forced OpenSAML to use the wrapped original assertion for signature validation. In contrast, the application logic processed the claims of the evil assertion. In Figure 3.15, we present the concrete attack messages of this novel XSW variant.

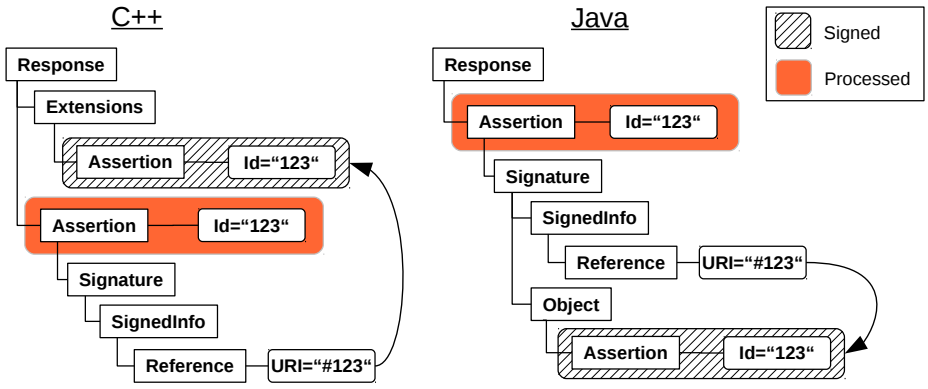


Figure 3.15: *XSW attack on OpenSAML library.*

The successful attack on OpenSAML shows that countering the XSW attack can become more complicated than expected. Even if several countermeasures are applied, the developer should still consider vulnerabilities in the underlying libraries. Namely, one vulnerability in the XML Schema validating library can lead to the execution of a successful XSW attack.

3.3.3.4 Salesforce SAML Interface Revisited

After reporting the XSW vulnerability to Salesforce, the security response team developed a simple and promising countermeasure: The SAML interface solely accepted messages containing *one* `Assertion` element.[2]

On request of the Salesforce security team, we investigated the fixed SAML interface with handcrafted messages containing wrapped contents in different elements. Our manual analysis did not reveal any new attack vectors. Every message containing more than one `Assertion` element was automatically rejected. Therefore, we first considered this interface to be secure.

A few months later, Marco Kampmann finished the development of his XSW penetration test tool [Mar11b]. We decided to use this tool to verify the countermeasures applied on the Salesforce interface. Surprisingly, the automated penetration test tool revealed a new successful attack variant by inserting the wrapped content into the `Audience` element – a descendant of the `Conditions` element. This element typically contains a `URI` constraining the parties that can consume the issued assertion. The wrapped message is depicted in Figure 3.16. As can be seen in the figure, both `Assertion` elements needed to contain the same `Id` attribute.

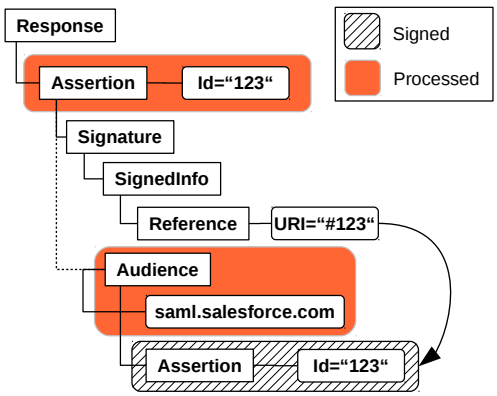


Figure 3.16: *A successful XSW attack performed against the patched Salesforce SAML interface.*

This scientifically interesting attack vector stayed unanalyzed as the Salesforce security team did not expose any concrete information about their SAML interface. However, it showed again how complex the development of secure XSW countermeasures is.

3.3.3.5 Various Implementation Flaws

While reviewing the OneLogin Toolkit, we discovered another interesting flaw: The implementation did not care about what data was actually signed. Therefore, any content signed by the *IdP* was sufficient to launch an XSW attack.

[2] This countermeasure is not standard conformant as one message can generally contain several assertions.

In our case we used the metadata of the IdP[3] and created our own self-made response message to successfully attack OneLogin.

Besides the fact that a SAML system has to check what data is signed, it is also essential to verify by whom the signature was created. In an early version of SimpleSAMLphp, which applied Type 3 messages, we observed that an attacker could forge the outer signature of the response message with any arbitrary key. In short, the SimpleSAMLphp RP did not verify if the included certificate in the `KeyInfo` element is trustworthy at all. The key evaluation for the signed assertion was correctly handled.

3.3.3.6 Secure Frameworks

In our evaluation of real-world SAML implementations we observed that Microsoft Sharepoint 2010 and SimpleSAMLphp were resistant to all applied test cases. Based on these findings the following questions arise: How do these systems implement signature validation? In which way do signature validation and assertion processing work together? Due to the fact that the source code of Sharepoint 2010 is not publicly available, we were only able to analyze SimpleSAMLphp.

According to this investigation the main signature validation and claims processing algorithm of SimpleSAMLphp performs the following five steps to counteract XSW attacks:

1. **XML Schema validation:** First, the whole response message is validated against the applied SAML schemas.
2. **Extract assertions:** All included assertions are extracted. Each assertion is saved as a DOM tree in a separate variable. The following steps are only applied on these segregated assertions.
3. **Verify what is signed:** SimpleSAMLphp checks, if each assertion is protected by an enveloped signature. In short, the XML node addressed by the `URI` attribute of the `Reference` element is compared to the root element of the same assertion. The XML Signature in the assertion is an enveloped signature if and only if both objects are identical.
4. **Validate signature:** The verification of every enveloped signature is exclusively done on the DOM tree of each corresponding assertion.
5. **Assertion processing:** The subsequent assertion processing is solely done with the extracted and successfully validated assertions.

When not considering the signature exclusion bug found in the OpenAthens implementation and its Java-based assertions processing, this framework was also resistant to all the described attacks. The analysis of its implementation showed that it processes SAML assertions similarly to the above described SimpleSAMLphp framework.

[3]The SAML Metadata [CMPM05] describes properties of SAML entities in XML to allow easy establishment of federations. Typically, the metadata is signed by the issuer and publicly available.

3.3.3.7 Summary

We evaluated 14 different SAML-based systems. We found eleven of them vulnerable to XSW attacks. One prevalently used framework (OpenSAML) was vulnerable to a new, more subtle, variant of this attack vector. In addition, three of the tested frameworks were vulnerable to Signature Exclusion attacks. We found two implementations which were resistant against all test cases. The results obtained from our analysis are summarized in Table 3.2.

Frameworks / Providers	Signing type	Signature exclusion	Refined XSW	Sophisticated XSW	Not vulnerable
Apache Axis2	1)	X	X		
Guanxi	2)		X		
Higgins 1.x	1)		X		
IBM XS40	1)		X		
JOSSO	2)	X	X		
WIF	1)				X
OIOSAML	1)		X		
OpenAM	1)		X		
OneLogin	1)		X		
OpenAthens	1)	X			
OpenSAML	1)			X	
Salesforce	1)		X		
SimpleSAMLphp	3)				X
WSO2	2)		X		

Table 3.2: *Results of our practical evaluation show that a majority of the analyzed frameworks were vulnerable to the refined wrapping techniques.*

3.4 Further Related Work

Starting from 2005, XSW attacks have become a research topic considered in many scientific publications. In the following, we give an overview of the major publications. We present also two additional XSW attacks, which can be executed even if XPath-based referencing is used.

3.4.1 Security of XML Signature

XSW attacks were first presented by McIntosh and Austel [MA05] and Bhargavan et al. [BFG04]. McIntosh and Austel [MA05] also discussed receiver-sided security policies. They demonstrated the complexity of the problem by showing sophisticated XSW attacks bypassing such policies.

Rahaman, Schaad and Rits [RSR06, RMS06, RS07] refrained from policy-driven approaches and introduced an *inline* approach. The authors proposed to embed an `Account` element into the SOAP header. This element contains partial information about the structure of the SOAP message and the neighborhood of the signed element(s). The information preserves the structure of the data to

be signed. However, Gajek et al. showed that this approach does not prevent XSW attacks [GLS07].

Jensen et al. [JMSS11] analyzed the effectiveness of XML Schema validation in terms of fending off XSW attacks in Web Services. Thereby, they used manually hardened XML Schemas. The authors concluded that XML Schema validation is capable of fending off XSW attacks, at the expense of two important disadvantages: For each application, a specific hardened XML Schema without extension points must be carefully created. Moreover, document validation according to a hardened XML Schema entails severe performance penalties.

XML Signatures allow the application of HMAC verification algorithm [MvV96]. The specification offers an `HMACOutputLength` parameter when applying HMACs. The parameter specifies the number of HMAC output bits that must be verified. If the receiver fully trusts the `HMACOutputLength` parameter, the attacker can force the receiver to verify only one HMAC bit [Roe09]. A surprisingly high number of implementations were vulnerable to this attack. The XML Signature specification now explicitly mentions that at least 80 bits of the HMAC output must be verified.

3.4.2 XSW Attacks on XML Signatures with XPath Referencing

XPointer [MMGW03] and XPath Filter [RBH02] are specified as referencing mechanisms in the XML Signature specification. An example of a message signed using the XPointer specification is shown in Figure 3.17. The signed `Body` element is referenced using the `/Envelope/Body` XPath expression.

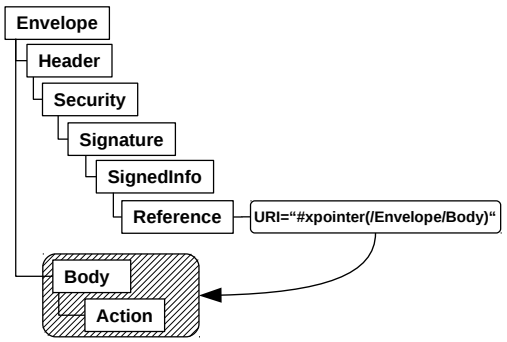


Figure 3.17: *Example of an XPath-based XML Signature applied on the SOAP body.*

One could think that usage of XPath expressions could mitigate XSW attacks because these expressions fix the positions of the signed elements. However, Gajek, Jensen, Liao and Schwenk showed that the XPath expressions have to be chosen very carefully, otherwise new XSW vulnerabilities could appear [GJLS09, JLS09]. These attacks were not exploited practically.

3.4.2.1 Imprecise XPath Expressions

Gajek et al. [GJLS09] evaluated the effectiveness of XPath mechanisms to mitigate XSW attacks in the SOAP context. They showed that many imprecise forms of XPath expressions still offer possibilities for XSW attacks.

3.4.2.1.1 Identifier Referencing. Each `Id`-based reference can be transformed into an XPath expression. Consider an element with `Id="ts"`, which is referenced using `URI="#ts"`. This element can also be simply referenced by an XPath expression `//*[@Id="ts"]`, where "`//`" is a short form for `descendant-or-self`. As the XPath expression equals to the Id-based referencing, it is obvious that its application could lead to equivalent XSW attacks. An example of such an XSW attack on the `Timestamp` element is depicted in Figure 3.18.

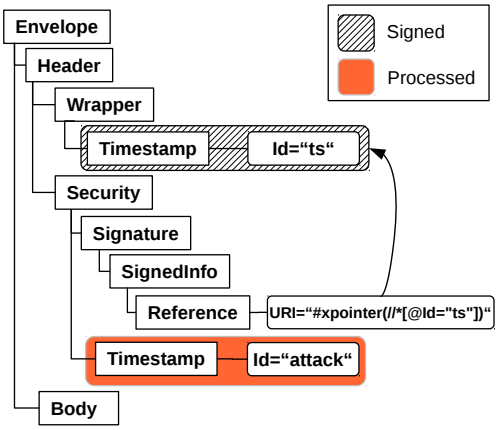


Figure 3.18: *Imprecise usage of XPath referencing could also lead to XSW attacks.*

3.4.2.1.2 Referencing with `descendant-or-self` (`//`). "`//`" selects the context node and all its descendants. "`//`" can occur also in the middle of the XPath expression. An example of such an XPath expression gives `/Envelope[1]/Header[1]//Timestamp[1]`. However, this expression does not secure the exact position of the `Timestamp` element. It finds namely the first `Timestamp` element within the SOAP header. If we assume that the application logic evaluates a `Timestamp` element in the `Security` SOAP header element, the same XSW attack as depicted in Figure 3.18 could be applied.

In addition to `descendant-or-self`, also the usage of different axes such as `descendant`, `ancestor`, or `next-sibling` could in general be exploited.

3.4.2.1.3 Proposed Countermeasures. Gajek et al. [GJLS09] proposed a lightweight variant called *FastXPath* defininig usage of precise XPath expressions. FastXPath starts its search from the root element. In each step, only one child element is selected. It is explicitly indicated by its name and position. An example of a FastXPath expression gives `/Envelope[1]/Header[1]/Security[1]/Timestamp[1]`. See Section 3.6.3 for more details.

3.4.2.2 XML Namespace Injection

The FastXPath approach by Gajek et al. [GJLS09] tries to fix the exact position of an element in the XML tree. However, Jensen et al. [JLS09] showed that a

naive usage of XML namespaces in XPath expressions could result in a *Namespace Injection attack*: By clever manipulation of XML namespace declarations within a signed document, XSW attacks could successfully be mounted even against XPath referenced resources. The attack exploits resolution of newly defined XML namespaces. It can be applied only if XML Signature utilizes the Exclusive XML Canonicalization method.

In the following attack description, we will use the complete XML structure including its namespaces.

Attack. For the description of the basic attack idea please consider Figure 3.19. This figure contains a SOAP message with an operation's content. The content is referenced using the XPath expression `/soap:Envelope/soap:Body/op:Operation/Content`. The XPath engine searches for such an XML element using namespace URIs (not namespace prefixes). Thus, the namespace prefixes are first resolved. The XPath engine traverses the document to the top and collects all the namespace URIs: `soap="ns-soap"` and `op="ns-op"`. Afterwards, it searches for the element `/{ns-soap}:Envelope/{ns-soap}:Body/{ns-op}:Operation/Content`.[4] The `Content` element is then canonicalized and signed. Note that the Exclusive XML Canonicalization method *omits* the `op="ns-op"` namespace declaration as it is not needed in the `Content` element. Thus, this namespace is not protected.

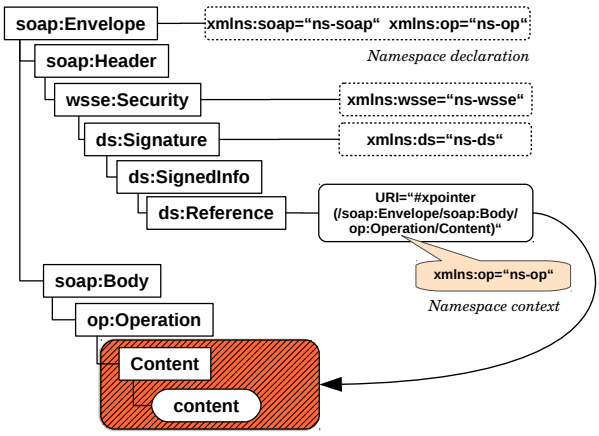


Figure 3.19: *XPath-based referencing securing the content of* `op:Operation`.

Figure 3.20 shows a simplified example of an XSW attack on the message from the previous figure. The attacker proceeds by the message modification as follows. He first declares a new `xmlns:op="ns-attack"` namespace in one of the ancestors of the `ds:SignedInfo` element (in the figure `wsse:Security` is used). Afterwards, he duplicates the `op:Operation` element. The first element declares the original namespace `xmlns:op="ns-op"` and the second element declares `xmlns:op="ns-attack"`. The attacker inserts his content into the first `op:Operation` element.

The signature verification logic processes this message as follows. It finds the

[4] `{ns-soap}:Envelope` refers to an `Envelope` element with the namespace URI `ns-soap`.

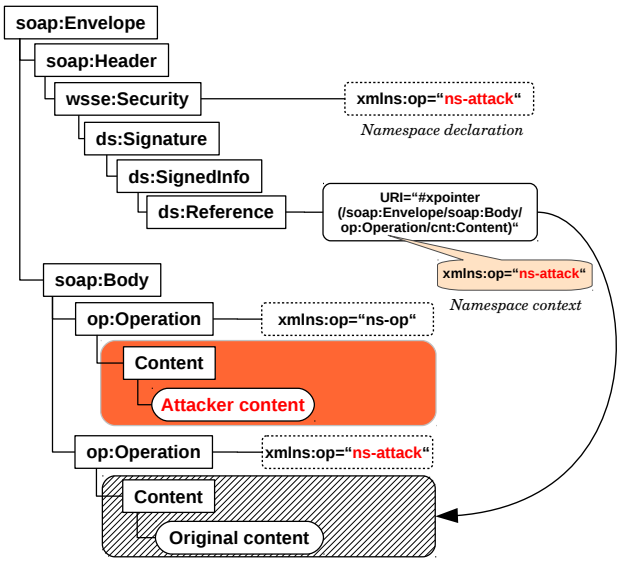


Figure 3.20: *XML Namespace Injection XSW technique applied on the message from the previous figure: The signature verification logic resolves the* `xmlns:op="ns-attack"` *namespace (injected by an attacker) and thus verifies a wrong element.*

`Reference` element and resolves the namespace context. As the attacker declared the `xmlns:op="ns-attack"` namespace in the `wsse:Security` element, it resolves the following URIs: `soap="ns-soap"` and `op="ns-attack"`. Afterwards, it searches for `/{ns-soap}:Envelope/{ns-soap}:Body/{ns-attack}:Operation/Content`. Thus, this XPath expression returns the second (original) `op:Operation` element and the signature is successfully verified. The business logic executes the first (modified) `op:Operation` element.

Proposed Countermeasures. Jensen et al. proposed several countermeasures against these attacks [JLS09]. Two of them are practically applicable in the current XML Security frameworks. The first countermeasure includes explicit namespaces in the `SignedInfo` element, which results in a hash value computation over additionally defined namespaces. The second countermeasure proposes a new XPath syntax. The syntax identifies each element explicitly by its element name (by using the `local-name()` function) and its namespace context (by using the `namespace-uri()` function). This ensures that the referenced elements belong to the correct namespaces.

Both approaches are described in more detail in Section 3.6.3.

3.5 Summary of XSW Attacks

Throughout this chapter we described several XSW attacks containing various properties. The attacks were applicable on different interfaces. In the following, we summarize these attacks and divide them into four main categories.

***Classical* XSW Attack.** The classical XSW attack was defined by McIntosh

and Austel [MA05] (see Figure 3.1). A major prerequisite for executing this attack is that the signature verification logic and the business logic do not communicate with each other. Thus, the business logic does not know which element was verified by the XML Signature. This allows an attacker to place the original signed content into an arbitrary element and create a new evil element with a *different* `Id`. As the business logic does not know which element with which `Id` was verified, it processes the newly created evil element.

***Duplicated-Id* XSW Attack.** We showed that several systems and frameworks tried to establish communication between the business logic and the security verification modules. Amazon Web Services and OpenSAML developers tried to achieve this by providing the business logic with the `Id` attribute of the signed element. The business logic can then check if the processed element has the same `Id` as the signed element. This basic countermeasure can be bypassed by using identical `Id` attributes for the signed and processed evil element. Practical attacks were executed against OpenSAML and AWS interfaces.

When executing this type of attack, the attacker has to consider the following properties:

- XML Schema validation: XML Schema does not allow the use of two identical `Id` elements in one document. The attack can thus only be applied against an interface that (1) does not validate XML Schema (see e.g. the attack on AWS in Section 3.2.3), or (2) applies a vulnerable parser allowing two elements with identical `Id` attributes (see e.g. the attack on OpenSAML in Section 3.3.3.3).

- Order of signed and processed elements: When an XML document contains more elements with identical `Id` attributes referenced by an XML Signature, the signature verification logic typically verifies the first (see e.g. the attack on OpenSAML C++ in Section 3.3.3.3) or the last element (see e.g. the attacks on AWS and OpenSAML Java in Sections 3.2.3 and 3.3.3.3). Thus, the attacker has to decide if he puts the original signed element before or behind the new evil element.

Imprecise XPath Referencing. Usage of imprecise XPath expressions can also give an attacker a possibility to attack XML Signatures with XPath-based referencing (see Section 3.4.2.1). The imprecision of an XPath expression can be caused by the use of an identifier referencing (e.g. `//*[@Id="ts"]`), or by the use of XPath axes not correctly fixing the positions of signed elements (e.g. `descendant-or-self` or `ancestor`).

In general, if the attacker can move the signed content without invalidating the XPath expression, the XSW attacks are still possible.

XML Namespace Injection. The XML Namespace Injection technique of Jensen et al. [JLS09] can basically be executed if two prerequisites are fulfilled. First, the XPath expression does not explicitly define namespace URIs used in the path of the signed element. Second, the Exclusive XML Canonicalization method is used. This gives the attacker an opportunity to redefine XML namespaces: He can, in specific messages, change the namespace context of the signed

element without invalidating the original signature. He thus forces the signature validation logic to verify the original XML content placed in an element with a different namespace. The business logic processes new elements from the original namespace (see Section 3.4.2.2).

3.6 Countermeasures

In Section 3.3.3.6 we analyzed message processing of SimpleSAMLphp. This framework was resistant against all XSW attacks. One could therefore ask the question: Why do we need further countermeasures and why is it not appropriate to apply the security algorithm of SimpleSAMLphp in every system?

We want to make clear that SimpleSAMLphp offers both critical functionalities in one framework: signature validation and SAML assertion evaluation. These two methods are implemented using the same libraries and processing modules. After parsing a document, the elements are stored within a document tree and can be accessed directly. This allows the security developers to conveniently access the same elements used in signature validation and assertion evaluation steps. However, there exist scenarios (e.g., in enterprise environments) that force the developers to separate these two steps into different modules or even different systems, for example:

- **Using a signature validation library:** Before evaluating the incoming XML document, the developer uses a DOM-based signature library, which returns `true` or `false` according to the message validity. However, the developer does not exactly know which elements were validated. If the business logic applies a different parsing approach (e.g. streaming-based SAX or StAX approach) or another DOM-library, the message processing could become error-prone.
- **XML Security Gateways:** XML Security Gateways can validate XML Signatures and are configured to forward only validated XML documents. If the developer evaluates a validated document in his application, he has no explicit information about the position of the signed element. Synchronization of signature and assertion processing components in this scenario becomes even more complicated if the developer has no information about the implementation of the Security Gateway (e.g. IBM XS40).

These two examples show that convenient access to the same XML elements is not always given. We present three general countermeasures applicable in systems that split security processing and business logic processing into different modules.

3.6.1 See What Is Signed

A countermeasure referred to as *see what is signed* is constituted by the fact that the application logic is only able to notice the XML content that was digitally signed. This can be achieved using two techniques. First, the signature

verification logic can forward only the signed elements. Second, the signature verification logic returns next to a boolean value some sort of position of the signed data. A disadvantage of the countermeasures is that they require serious changes in XML processing modules on the recipient side. In the following, we present these countermeasures in more detail. They were described in [GLS07] and formally analyzed in [SMS+12].

3.6.1.1 Strict Filtering

The core idea of this countermeasure is to forward only those elements to the business logic module that were validated by the signature verification module. This is not trivial as extracting the unsigned elements from the message context could make the further message processing in some scenarios impossible. Therefore, we propose a solution that excludes only the unsigned elements, which do not contain any signed descendants. We give an example of such a message processing in Figure 3.21. This way, the claims and message processing logic would get the whole message context: In case of SOAP it would see the whole `Envelope` element. The main advantage of this approach is that the message processing logic does not have to search for validated elements because it can assume that all forwarded elements are valid.

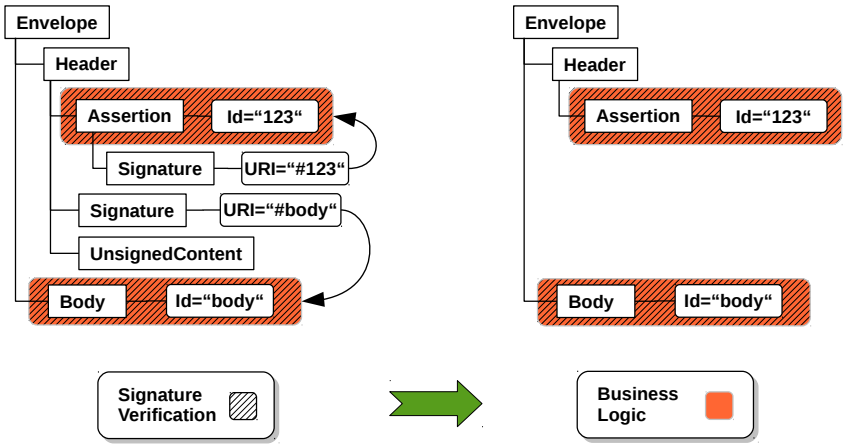


Figure 3.21: *The strict filtering approach applied to SOAP message processing: After successful signature verification, the signature verification module excludes all the unsigned elements and forwards the message to the module processing assertion security claims and the business logic.*

This idea was already discussed by Gajek et al. [GLS07]. However, no XML Signature framework currently implements this countermeasure. It could be applied especially in the context of SAML HTTP POST bindings because the unsigned elements within the SAML response do not contain any data needed in the business logic. We consider this countermeasure in these scenarios as appropriate because the SAML standard only allows the usage of `Id`-based referencing, exclusive canonicalization, and enveloped transformation. This countermeasure would not work if the XML Signature were to use specific XSLT or XPath transformations.

3.6.1.2 Unique Identification (Tainting) of Signed Data

The second countermeasure represents another form of the *see what is signed* approach. The basic idea is to uniquely identify the signed data in the signature verification module and forward this information to the following modules. This could be done by generating a random value r, sending it to the next processing module (or as an attribute in the document root element), and attaching it as a new attribute to all the signed elements. We give an example of this countermeasure applied to a SOAP message in Figure 3.22. The main drawback of this countermeasure is that XML Schemas do not allow the inclusion of new attributes. Therefore, the XML Schema validation would fail. For general application of this idea the XML Schemas need to be extended.

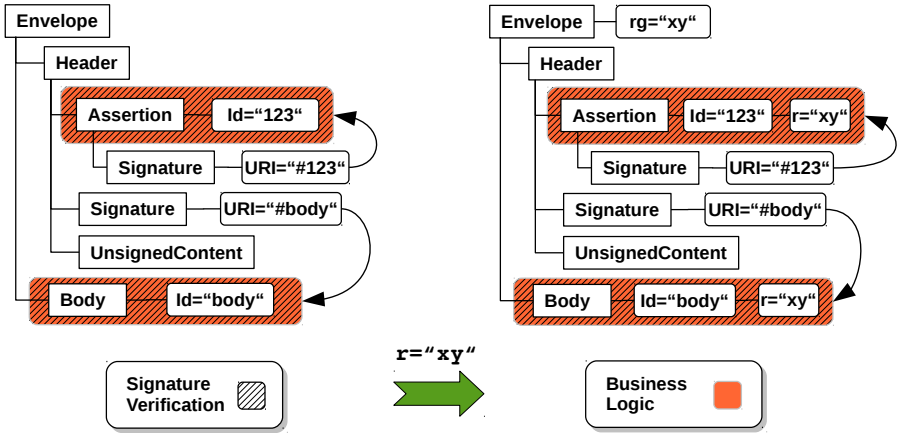


Figure 3.22: *Unique identification of signed data applied on a SOAP message including two signed elements: The signature verification module uniquely identifies the signed elements with a random value r and forwards this information along with the whole XML message.*

Another possibility to implement this countermeasure is to use XML node types that do not violate the XML Schema, but are visible to the XML processors. For example, processing instructions [BPSM+08], which are intended to carry instructions to the application belong to this group. They can be placed anywhere in the document without invalidating the XML Schema. Additionally, they can be conveniently found by processing XML trees with streaming and DOM-based parsers. Therefore, the presence of these XML nodes would help to find the validated data and thus allows to mitigate XSW attacks.

3.6.2 See Which `Id` Is Signed

A simple approach to counter XSW attacks while using `Id`-based referencing is to check if the processed XML content contains the same `Id` as the signed element. We saw this approach in SOAP-based as well as SAML-based interfaces (see description of Amazon and OpenSAML interfaces in Sections 3.2.3 and 3.3.3.3). The advantage of this countermeasure is that it does not need any changes in the XML Signature verification logic. Even though this countermeasure looks

straightforward, its application requires careful XML parsing and processing.

As described in Section 2.1, `Id` or id attributes are not automatically handled as attributes of type `ID`. The attributes must explicitly be defined as `ID` type attributes. This can be done using different approaches:

- The developer provides the parser an XML Schema identifying attributes of type `ID`.
- The developer indicates to the DOM in the code that certain attributes are handled as `ID` type attributes. This can e.g. be done by invoking `setIdAttribute()` method on the element containing the attribute.
- The `ID` attributes are defined directly in the XML message prolog using a DTD [BPSM+08].

First, DTDs give the message *sender* an opportunity to define `ID` type attributes. Thus, *DTDs must be forbidden* on the recipient side. Otherwise, they would allow an attacker for definition of new `ID` type attributes and thus confuse the signature verification and business logic modules.

Second, by applying this countermeasure, the message receiver must ensure that the signature verification logic and the business logic process elements with the same `Id` attributes. XML messages containing duplicated `Id` values must be rejected. Otherwise, the attacker could apply a *duplicated-Id* XSW attack. To achieve this, the developer could use one of these two approaches:

- XML Schema validation: XML Schema automatically invalidates XML documents containing duplicated `Ids`. However, the developer should verify that the used XML parser is reliable and that it detects duplicated `Ids`. Moreover, if the XML application consists of different modules using different parsers in the signature verification and business logic module, the developer should ensure that both modules identify the same attributes as `ID` type attributes.
- The XML application can carefully identify specific element attributes in the DOM as `ID` type attributes. The number of the `ID` type attributes must not be larger than the total amount of `Reference` elements within the XML Signatures in the document. If the XML application is separated into two distinct modules (signature verification and business logic), the developer should check that his application identifies the same elements and indicates their elements as `ID` type attributes.

If one of these two approaches is applied, the business logic module can safely check if the processed element has the same `Id` as the element referenced by the XML Signature.

3.6.3 Fixing Positions of Signed Elements

The previously described countermeasures showed how to apply secure XML Signature processing on the receiver side. In the following, we summarize two practical countermeasures enforcing secure XML Signature processing on the

sender side by fixing the positions of signed XML elements. The only prerequisites are that the message receiver correctly processes XPath expressions and that the business logic module processes elements on fixed positions (e.g. a specific function in the SOAP body or a `Timestamp` element in the SOAP security header). If the positions of the signed elements are fixed, the attacker cannot move them into different document parts.

Both countermeasures can be used with XPointer [MMGW03] and XPath Filter [RBH02] referencing mechanisms. They were investigated by Jensen et al. [JLS09] and are based on the FastXPath grammar. This grammar prescribes that a signed element can be referenced only by an XPath that

- starts its search from the root element,
- uses only `child` axes (no `descendant` or `ancestor` axes are allowed), and
- precisely defines the element name.

In order to thwart XML Namespace Injection attacks – which allow an attacker to place the signed content into an element with a newly defined namespace (see Section 3.4.2.2) – the FastXPath grammar is extended with precise namespace declarations.

In the following, assume that the sender signs an `op:Operation/Content` element in the SOAP body (see Figure 3.19).

Explicitly Embedding Namespaces in the Hashed XML content. The first proposed approach is to embed the relevant namespaces into the `InclusiveNamespaces` element. In the case of our message, the sender needs to define `soap` and `op` namespaces (see Figure 3.23). This enforces that the hash value is computed over the `Content` element as well as explicitly over the `soap` and `op` namespaces. An attacker moving the `Content` element into an element from a different namespace context would thus invalidate the message.

```
<Reference URI="#xpointer(
        /soap:Envelope[1]/soap:Body[1]/op:Operation[1]/Content[1])">
  <Transforms>
    <Transform Algorithm=".../xml-exc-c14n#">
      <InclusiveNamespaces PrefixList="soap op"></InclusiveNamespaces>
    </Transform>
  </Transforms>
  <DigestMethod Algorithm=".../xmldsig#sha1"></DigestMethod>
  <DigestValue>yc17yWXGca510flwlu4BzHuZOIU=</DigestValue>
</Reference>
```

Figure 3.23: *`InclusiveNamespaces` enforces that the hash value is computed over the `soap` and `op` namespace declarations.*

Prefix-free XPath. The second countermeasure approach defines namespaces directly in the XPath expressions. The proposed XPath expressions explicitly reference elements by their names (using the `local-name()` function) and namespace URIs (using the `namespace-uri()` function). An example of such an XPath expression referencing the `Content` element gives:

```
/*[local-name()="Envelope" and namespace-uri()="ns-soap"]/
 *[local-name()="Body" and namespace-uri()="ns-soap"]/
 *[local-name()="Operation" and namespace-uri()="ns-op"]/
 *[local-name()="Content" and namespace-uri()="ns-default"]
```

This expression contains the namespace context of each element. Thus, if the attacker would move the original element into an element with a different namespace context, the original element would not be found. This approach is currently e.g. used in IBM appliances [IBM13].

It is important to mention that even if the signature mechanism verifies a correct element, the user still has to ensure that this element is also processed by the business logic. Considering the above given XPath expression evaluated by the signature validation logic, an XSW attack could still be applied if the business logic would process elements according to their namespace prefixes.

3.7 XSW Attack Library

Our crucial findings and the vast amount of possible permutations motivated us to develop the first fully automated library for XSW attacks. The goal of this library is to take an arbitrary signed XML message and generate a list of XSW attack vectors according to this XML message. The first library was implemented by Marco Kampmann and intended for SAML-based XML messages [Mar11b, SMS+12]. Christian Mainka generalized Marco's approach so that it is now possible to automatically create XSW attack messages from (1) arbitrary XML messages (2) using Id-based as well as XPath-based XML Signatures [Chr12].

In this section, we briefly describe basic requirements, design, and implementation decisions for this library. The library can easily be integrated in penetration testing applications. Currently, it is integrated in WS-Attacker[5] – a framework offering automatic pentesting of SOAP-based Web Services endpoints [MSS12].

More information on this topic can be found in the work of Christian Mainka [Chr12].

3.7.1 Design and Algorithms

According to the theoretical and practical analysis of different frameworks and systems applying XML Signatures, we gained the following general knowledge about executing XSW attacks:

- XML Schema validation: Some of the frameworks check message conformance to the underlying XML Schema. Therefore, it is necessary to use XML Schema extension points (identified by `any` elements in the XML Schema document) for placing the wrapped content.

[5] `http://ws-attacker.sourceforge.net`

- Order and position: The order and position of signed and executed elements in the message tree can force the different processing modules to have inconsistent data views.

- Processing `Id`s: Several frameworks explicitly check, if the `Id` attribute in the XML content processed by the business logic is also used in the XML Signature.

- XPath-based referencing: Usage of imprecise XPath expressions can introduce new attack risks. Attacks based on imprecise axes definitions as well as on namespace rewriting should be considered (see Sections 3.4.2.1 and 3.4.2.2).

- Signature exclusion: A bug in a framework implementation can cause the signature validation step to be omitted.

- Untrusted signatures: It is essential to check that the signature was created with a trustworthy key.

Based on this knowledge, we developed a library that allows for systematic generation of a vast number of different XSW attack vectors.

3.7.1.1 XSW Attack Complexity

The general idea of an XSW library is very simple: Take a message, find a position for a wrapper element, and insert new evil content defined by the user. However, the number of possible XSW messages grows very fast based on the number of signed elements in the attacked message and on the different attack methods, see Figure 3.24.

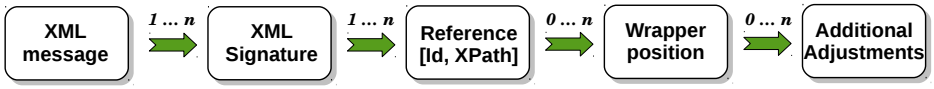


Figure 3.24: *XSW attack complexity: the number of generated XSW attack vectors depends on many factors ("`0...n`" indicates a zero-to-many relationship, "`1...n`" indicates a one-to-many relationship).*

In general, a single XML message can contain one or more XML Signatures. Each XML Signature can contain one or more `Reference` elements. A reference can apply an `Id`-based or XPath-based referencing mechanism. If XPath referencing is used, the XSW attack algorithm has to check for imprecise XPath expressions as well as Namespace Injection techniques. The referenced contents can be placed in a large number of wrapper positions resulting in many attack vectors. Additionally, each attack vector can be adjusted, e.g. the newly generated content can contain identical or different `Id` attributes.

These XSW attack properties show that a large number of attack messages can be generated. It is impossible to cover all the possibilities without automation.

3.7.1.2 Transforming `Id` References to XPath Expressions

An element can be referenced using an `Id` or XPath. As mentioned in the previous sections, each `Id`-based reference can be transformed into an XPath expression. For example, an element with an `Id="123"` can be referenced by the XPath expression `//*[@Id="123"]`. Thus, `Id`-based referencing can be considered as a subgroup of XPath referencing mechanisms.

In order to treat both referencing mechanisms generally – before searching for new positions of signed data – each `Id` reference is internally transformed into an XPath expression. Instead of developing two distinct XSW attack algorithms for XPath and `Id` referencing, only one algorithm for the XPath referencing is needed.

3.7.1.3 Handling Timestamps

Many XML messages applying XML Security contain `Timestamp` elements. When executing an XSW attack, the timestamps included in the message have to be updated. Otherwise, the message receiver would automatically reject all provided XML messages.

For convenience, the XSW library automatically detects a `Timestamp` element and adjusts it. An example of a `Timestamp` structure is depicted in Figure 3.25. The library updates the `Created` and `Expires` elements inside the timestamp. Thereby, the difference between the values of `Expires` and `Created` (timestamp lifetime) remains equal.

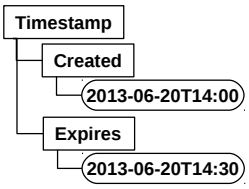


Figure 3.25: *XSW library automatically detects `Timestamp` elements and updates their contents.*

3.7.1.4 Analyzing XML Schema

The signed content can theoretically be wrapped into an arbitrary element inside the XML message. This can result in a huge number of possible positions. The number of positions can be reduced by considering only the extensible elements. These elements are defined in the XML Schema by an `any` element and can contain an arbitrary content – which makes them ideal for placing the signed elements.

It is notable that the applied algorithm does not only search for extensible elements in the XML message, but also in the XML Schema document. If it finds an extensible element in the XML Schema and this element is not contained in the XML message, the XML message is explicitly extended with this element. An example of this approach gives processing of the the `Signature` element

(see Figure 3.26). This element typically contains two children: `SignedInfo` and `SignatureValue`. However, it can also contain a `KeyInfo` or an `Object` element. The `Object` element is an extensible element (as could be seen in the previous sections, it is an ideal element for placing the wrapped content). Thus, the algorithm automatically extends the `Signature` element and includes a new `Object` element serving as a wrapper. This way it is possible to construct a large number of valid positions for the wrapped XML content.

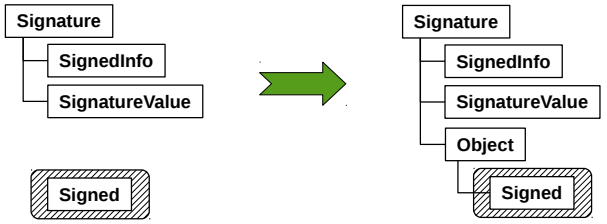


Figure 3.26: *According to the XML Schema, XSW library automatically extends the original XML message with extensible elements. The extensible elements can then be used as a placeholder for signed contents.*

3.7.1.5 XPath Weakness Algorithms

Sections 3.4.2.1 and 3.4.2.2 gave an overview of different XSW attacks against XPath-based referencing. The XSW library implements attacks abusing the weaknesses of XML referencing (identifier referencing using an attribute value and referencing with `/descendant-or-self` axes) and XML namespace redefinition (XML Namespace Injection attacks).

In general, by executing an attack, the library first carefully analyzes the included XPath expression. It searches for the signed content. Then, it divides the XPath expression into processing steps and tries to find new positions within the XML message tree used for the inclusion of the signed content.

Concrete algorithm descriptions are given in [Chr12].

3.7.1.6 Beyond XSW – XML Signature Forging and Exclusion

In summary, in four analyzed frameworks and systems (AWS, Apache Axis2, JOSSO, and OpenAthens) implementation bugs caused the signature validation step to be omitted. Based on the relevance of this attack, we included this attack vector in the XSW library.

It is also essential to check whether the signature was created with a trustworthy key. Otherwise, the attacker can forge a signature with an arbitrary key and embed the corresponding certificate in the `KeyInfo` element.

3.7.2 Implementation

The above described algorithms have been implemented and provided in the XSW library. The library takes a signed XML message as input and returns

valid XSW attack vectors. Thus, it can be embedded in an arbitrary penetration testing tool.

The XSW library consists of three major parts:

- *XML Signature Manager* analyzes an XML message and identifies the signed elements.
- *XPath parser* analyzes an XPath expression and represents it in an object-oriented form.
- *Wrapping Oracle* accepts a signed XML message, a new evil payload, and XML Schema documents. It analyzes the XPath expressions and the XML Schema documents, and finds the extension possibilities within the provided XML message. The found positions are numerated, for each position different variations of the accepted evil payload are generated, and the processing state is stored.[6] The developer can then query the XSW library iteratively to generate XSW attack vectors. See Figure 3.27.

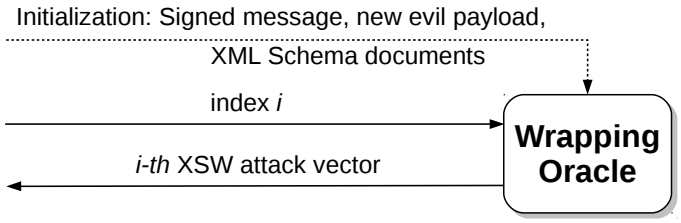


Figure 3.27: *The Wrapping Oracle is initialized by a signed XML message, a new evil payload, and XML Schema documents. Afterwards, it can be queried for XSW attack vectors.*

3.7.3 Integration in WS-Attacker

WS-Attacker is a penetration testing framework for SOAP-based Web Services [MSS12]. It provides a user with the ability to communicate with Web Services. Developers can extend its functionality with new modules covering additional attacks.

The XSW library was integrated in the WS-Attacker framework. This enables automatic testing for XSW attacks against SOAP-based Web Services. The XSW attack execution proceeds in the following steps (see Figure 3.28): First, the user chooses the Web Service endpoint. He also provides WS-Attacker with a signed XML message and with a new evil payload that should be executed with a successful XSW attack. WS-Attacker analyzes the provided XML message and the payload content. This results in a number of XSW attack vectors. The XSW attack vectors are then iteratively generated by the Wrapping Oracle

[6] Processing an XML message can result in a few thousands of XSW attack possibilities. For performance reasons, the attack messages are not stored in the memory. They can be generated on the fly by using the attack message index `i`. See Figure 3.27.

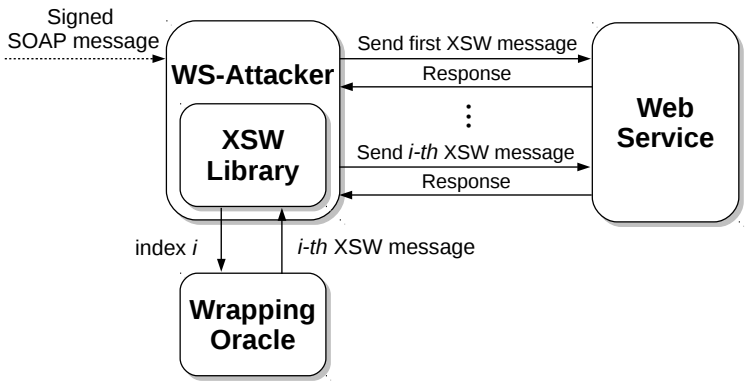


Figure 3.28: *The XSW library with its Wrapping oracle was integrated into the WS-Attacker framework.*

and sent to the Web Service interface. If the Web Service responds with an appropriate message, the XSW attack is successful.[7]

We are currently in the process of extending the WS-Attacker to provide communication with arbitrary SAML interfaces.

Remark: If WS-Attacker does not find a successful XSW attack vector, it does not automatically mean that the Web Service is secure. There are many XML frameworks containing specific properties and slightly different parsing mechanisms. Specific attack vectors can be overlooked or indicated as false negatives. Thus, in addition to this automatic penetration testing with WS-Attacker, manual testing should be performed.

3.8 Conclusion

In this chapter we showed that a large majority of systems and frameworks exhibit security insufficiencies in their XML Security interfaces. These insufficiencies lead to critical XSW vulnerabilities. Application of XSW attacks on SSO systems and cloud management interfaces can result in disastrous scenarios: An attacker in possession of a single signed XML message can at any time steal the identity of an arbitrary user or get control over the user's cloud system. This confirms that securing such interfaces is of crucial importance.

We presented a deep practical analysis of closed and open source SOAP and SAML interfaces. Our analysis revealed new classes of XSW attacks which worked even if specific countermeasures were applied. We showed that secure application of XML Signatures heavily depends on the underlying XML processing system (i.e. different XML libraries and parsing types). If the system processes XML messages in different modules, *it has to be ensured that all these modules have the same view on the processed XML messages.* Otherwise, XSW attacks could arise and an attacker could force critical modules to process unsigned XML contents.

[7] In most of the cases, for recognizing a successful XSW attack, it is sufficient to search for a specific string in the response message, e.g. ‘‘`successful login`’’. This string is provided by the user.

Based on our attacks, we developed a new XSW library for automatic generation of *all* XSW attack vectors described in this chapter. The complexity and large variety of the attacks motivates us for further development of automatic penetration testing tools in this area. We believe that attacks similar to XSW can also be executed against different interfaces processing different data formats.

4 How to Break XML Encryption

In this chapter, we describe how to perform Bleichenbacher's attack [Ble98] and a novel chosen-ciphertext attack (which is related to Vaudenay's attack [Vau02]) on XML Encryption, and thus break the confidentiality of the exchanged XML ciphertexts. Our attacks use different side-channels: errors from padding and parsing mechanisms, or timing differences for processing valid and invalid messages. We describe how to use these side-channels to decrypt AES-CBC [MvV96] and RSA-PKCS#1 v1.5 [Kal98] encrypted ciphertexts. Afterwards, we introduce another novel attack class, Backwards Compatibility attacks. These attacks allow us to also break AES-GCM [Dwo07] and RSA-OAEP [KS98] ciphertexts.

In order to describe our attacks, we first give cryptographic background on symmetric and asymmetric encryption mechanisms, and a high-level overview of Bleichenbacher's and Vaudenay's attacks. We describe the steps necessary for decryption of encrypted XML messages. Then, we move to the description of our adaptive chosen-ciphertext attacks.

Notation. Throughout this chapter we use the following notation:

- $\ell_a = |a|$: byte-length of a byte string a
- $\{0,1\}^n$: set of all bit strings of bit-length n ($n = 8\ell$)
- $a \oplus b$: bit-wise XOR of strings a and b
- $a||b$: concatenation of strings a and b
- m: plaintext
- C: ciphertext
- ν: block cipher block-length in bytes
- ξ: number of plaintext / ciphertext blocks of length ν
- $m = (m_1, \ldots, m_\nu)$: individual bytes of m in a block
- $m_{j,k}^{(i)}$: the k-th bit of the byte j from the i-th plaintext block

4.1 Cryptographic Background

This section presents symmetric and asymmetric encryption schemes relevant to this thesis and to the attacks on XML Encryption. Readers familiar with the CBC and GCM modes of operations and the PKCS#1 standard can skip this section.

4.1.1 Symmetric Encryption

4.1.1.1 Block Ciphers

The XML Encryption specification specifies Triple-DES (3DES) [Nat99] and AES [AES01] as mandatory block ciphers. The attacks described in this thesis do not exploit specific properties of these algorithms. They exploit rather a weakness in the used mode of operation and padding scheme (see below), and work with any cipher in a similar way. Thus, 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 $k \in \{0,1\}^t$ and an ν-byte plaintext $m \in \{0,1\}^n$, where $n = 8\nu$, 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 $m \in \{0,1\}^n$.

Since AES and 3DES are block ciphers, they allow the processing of data whose length is $\nu = 16$ or $\nu = 8$ bytes, respectively. In order to apply these algorithms to data of arbitrary length, the data has to be *padded* and processed using a *mode of operation*. In the following we describe two modes of operation relevant to this thesis: Cipher Block Chaining (CBC) [MvV96] and Galois Counter Mode (GCM) [Dwo07].

4.1.1.2 Padding Scheme

4.1.1.2.1 Padding Scheme in XML Encryption. Suppose a byte string m of arbitrary length is to be encrypted with a block cipher. The string m must first be padded in order to achieve a length ℓ, which is an integer multiple of the block size ν of the selected block cipher. XML Encryption specifies the following padding scheme π:

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

For instance, the example given in [ERI+02] considers a three-byte message $m = \texttt{0x61 62 63}$ and a block cipher with a block size of $\nu = 8$ bytes. In this case, we have

$$\pi(m) = m' = \texttt{0x61 62 63}\ ??\ ??\ ??\ ??\ \texttt{05},$$

where ?? is an arbitrary byte value, chosen randomly. To remove the padding, one simply reads the last byte of m' and removes the required number of bytes from m' to obtain m.

4.1.1.2.2 Padding Scheme in PKCS#5. Note that there exist different padding schemes. For example, PKCS#5 [Kal00] defines a padding scheme in which the processor also appends *padlen* bytes to m, but the padded bytes all have the length value *padlen* (instead of an arbitrary randomly chosen value). By applying such a padding scheme, the message $m = \texttt{0x61 62 63}$ would be padded as follows:

$$\pi(m) = m' = \texttt{0x61 62 63 05 05 05 05 05}.$$

4.1.1.3 Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) [MvV96] is the most popular block cipher mode of operation in practice. Its functionality with the XML Encryption padding scheme is depicted in Figure 4.1.

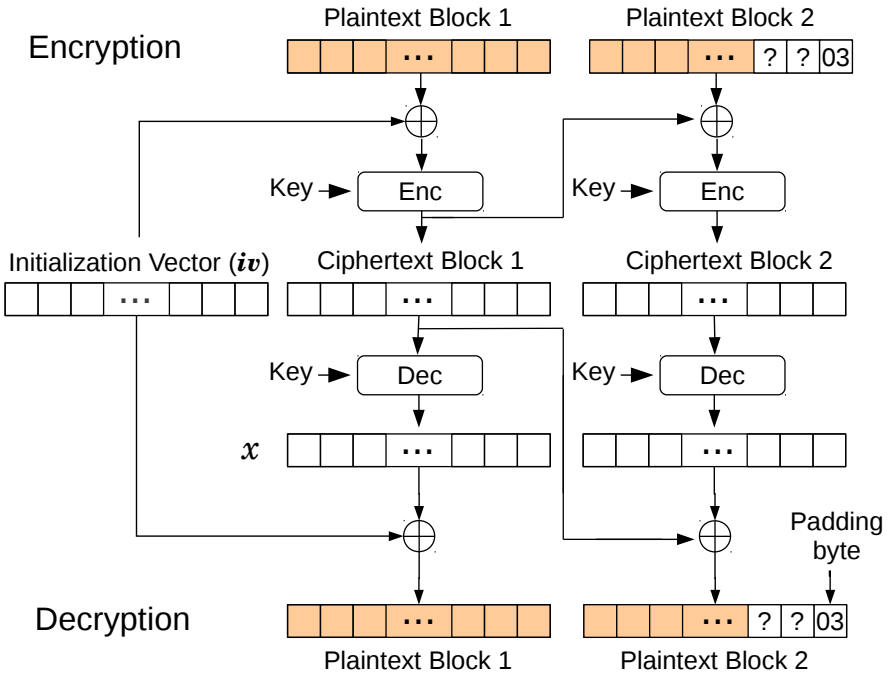


Figure 4.1: *CBC mode of operation with the XML Encryption padding scheme.*

Using a padding scheme we get m', whose length is a multiple of ν. Now, we can split m' into blocks of length ν: $m' = (m'^{(1)}, \ldots, m'^{(\xi)})$. These blocks are processed in CBC as follows:

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

$$x^{(1)} := m'^{(1)} \oplus iv, \qquad C^{(1)} := \mathsf{Enc}(k, x^{(1)}). \tag{4.1}$$

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

$$x^{(i)} := m'^{(i)} \oplus C^{(i-1)}, \qquad C^{(i)} := \mathsf{Enc}(k, x^{(i)}) \tag{4.2}$$

  for $i = 2, \ldots, \xi$.

- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(\xi)})$.

The decryption procedure reverts this process in the obvious way:

- The first plaintext block $m'^{(1)}$ is computed as

$$x^{(1)} := \mathsf{Dec}(k, C^{(1)}), \qquad m'^{(1)} := iv \oplus x^{(1)}.$$

- The subsequent plaintext blocks $m'^{(2)}, \ldots, m'^{(\xi)}$ are computed as

$$x^{(i)} := \mathsf{Dec}(k, C^{(i)}), \qquad m'^{(i)} := C^{(i-1)} \oplus x^{(i)}$$

  for $i = 2, \ldots, \xi$.

The last decrypted block $m'^{(\xi)}$ is treated specifically. After the decryption process, it is unpadded to recover the last plaintext block

$$m^{(\xi)} = \pi^{-1}(m'^{(\xi)}).$$

In the sequel we will write

$$C = \mathsf{Enc}_{\mathsf{cbc}}(k, m') \quad \text{and} \quad m' = \mathsf{Dec}_{\mathsf{cbc}}(k, C)$$

to denote encryption and decryption in CBC mode.

4.1.1.4 Galois Counter Mode (GCM)

Galois Counter Mode (GCM) [Dwo07] is a block cipher mode of operation, which provides both high efficiency and strong security in the sense of authenticated encryption [BN00]. In particular, GCM provides security against chosen-ciphertext attacks, like padding oracle attacks [Vau02], for instance. GCM is therefore an attractive choice for a replacement of CBC.

In the sequel let us assume a block cipher $(\mathsf{Enc}, \mathsf{Dec})$, consisting of an encryption algorithm Enc and a decryption algorithm Dec, with 128-bit block size[1] (like AES [AES01]). Let k be the symmetric key used for encryption and decryption. Let $m = (m^{(1)}, \ldots, m^{(\xi)})$ be a message consisting of $(\xi - 1)$ 128-bit blocks $(m^{(1)}, \ldots, m^{(\xi-1)})$ and one block $m^{(\xi)}$, where $m^{(\xi)}$ contains maximum of 128 bits and $\xi < 2^{32}$. The number of blocks $\xi_{max} = (2^{32} - 1)$ is the maximal number of blocks in a GCM plaintext according to [Dwo07]. The reason is that the counter *cnt* can only have $(2^{32} - 1)$ possible values. Longer messages must be split and encrypted separately.

A message is encrypted with $(\mathsf{Enc}, \mathsf{Dec})$ in GCM-mode as follows (see Figure 4.2).

- A 96-bit *initialization vector* $iv \in \{0,1\}^{96}$ is chosen at random. A counter cnt is initialized to $\mathsf{cnt} := iv||0^{31}||1$, where 0^{31} denotes the string consisting of 31 0-bits.

[1] In [Dwo07] GCM is specified only for 128-bit block ciphers.

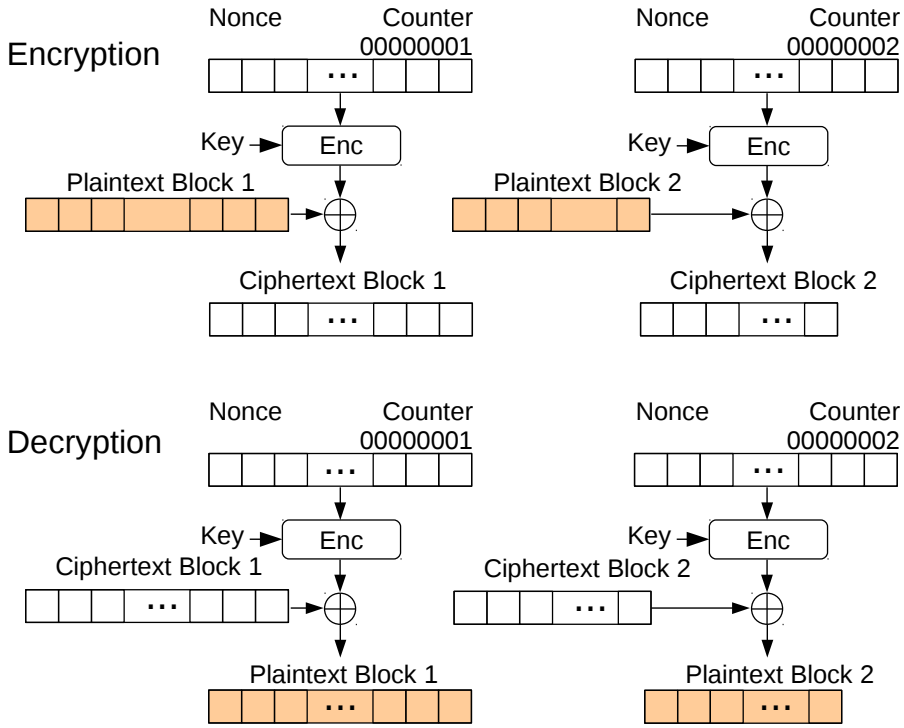


Figure 4.2: *Counter mode encryption and decryption processing (we omit Galois field computation details as it is not relevant to our attacks).*

- For $i \in \{1, \ldots, \xi\}$, the i-th message block[2] $m^{(i)}$ is encrypted by computing the i-th ciphertext block $C^{(i)}$ as

$$C^{(i)} := \mathsf{Enc}(k, \mathsf{cnt} + i) \oplus m^{(i)}.$$

- In parallel, an authentication tag τ (a message authentication code) is computed using arithmetic over a binary Galois field. The details of this computation are not relevant for our attacks. Without τ, GCM would become a simple stream cipher (Counter mode of operation) without integrity and authenticity protection.

- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(\xi)}, \tau)$.

The decryption procedure inverts this process in the obvious way. The ciphertext $C = (iv, C^{(1)}, \ldots, C^{(\xi)}, \tau)$ is decrypted as follows:

- The 96-bit *initialization vector* iv is used to initialize the counter cnt: $\mathsf{cnt} := iv||0^{31}||1$.

- For $i \in \{1, \ldots, \xi\}$, the i-th message block $m^{(i)}$ is decrypted as

$$m^{(i)} := \mathsf{Enc}(k, \mathsf{cnt} + i) \oplus C^{(i)}.$$

[2]Note that $i < 2^{32}$.

- In parallel, an authentication tag τ' is computed and compared with τ. If $\tau' = \tau$, the ciphertext is authenticated and the plaintext m can be processed.

4.1.2 Asymmetric Encryption

Let (N, e) be an RSA public key, where N has byte-length ℓ ($|N| = \ell$), with corresponding secret key $d = 1/e \bmod \phi(N)$.

4.1.2.1 PKCS#1 v1.5 Encryption Padding

The basic task of the PKCS#1 v1.5 encryption padding scheme [Kal98] is to prepend to a message k (typically a symmetric session key) a random padding string PS ($|PS| > 8$), and then apply the RSA encryption function:

1. The encryptor takes a message k and chooses a random byte string PS, where $|PS| > 8$, and $|PS| = \ell - 3 - |k|$, and $\texttt{0x00} \notin \{PS_1, \ldots, PS_{|PS|}\}$.

2. It sets encryption block $m = 00||02||PS||00||k$. By interpreting this string as an integer, $m < N$.

3. It computes the ciphertext as $C = m^e \bmod N$.

By decrypting such a ciphertext, the decryptor first computes $m = C^d \bmod N$. Afterwards, it checks whether the decrypted message m has a correct PKCS#1 v1.5 format. We say that the ciphertext C and the decrypted message $m = m_1||m_2||...||m_\ell$ are PKCS#1 v1.5 conformant if:

$$\begin{aligned} m_1 &= \texttt{0x00} \\ m_2 &= \texttt{0x02} \\ \texttt{0x00} &\notin \{m_3, \ldots, m_{10}\} \\ \texttt{0x00} &\in \{m_{11}, \ldots, m_\ell\} \end{aligned}$$

If this holds, it searches for the first value $i > 10$ such that $m_i = 0x00$. Then, it extracts $k = m_{i+1}|| \ldots ||m_\ell$. Otherwise, the ciphertext is rejected.

In case of SSL/TLS, PKCS#1 v1.5 is for example used for encapsulation of the `pre_master_secret` exchanged during the handshake [DR08]. Thus, k is interpreted as the `pre_master_secret`. In case of XML Encryption, k is interpreted as a key that is directly used for an AES/3DES computation. Figure 4.3 gives an example of a 16-byte long symmetric key padded to be encrypted with a 1024-bit RSA key. If the size of k would be incorrect, the application could not use k in further processing and should apply specific steps to thwart Bleichenbacher's attack [Ble98]. We describe the attack and the countermeasures against this attack in Sections 4.2.2 and 4.5.

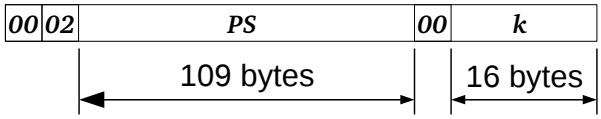


Figure 4.3: *PKCS#1 v1.5 padding applied to a 16-byte long symmetric key padded to be encrypted with a 1024-bit RSA key.*

4.1.2.2 RSA-OAEP

In RSA-OAEP [BR94] (aka. PKCS#1 v2.0 [KS98] or 2.1 [JK03][3]) a much more complex padding scheme is used. Let us describe the padding in detail. In the sequel let $\ell_G, \ell_H, \ell_k, \ell_0 \in \mathbb{N}$ be integers such that $\ell = 2+\ell_G+\ell_H$ and $\ell_0 = \ell_G-\ell_k$. Moreover, let $G : \{0,1\}^{8\ell_H} \to \{0,1\}^{8\ell_G}$ and $H : \{0,1\}^{8\ell_G} \to \{0,1\}^{8\ell_H}$ be cryptographic hash functions.[4]

A message k of byte-length ℓ_k is encrypted as follows (see Figure 4.4).

1. Choose a random padding string $PS \in \{0,1\}^{8\ell_H}$.
2. Compute values $s \in \{0,1\}^{8\ell_G}$ and $t \in \{0,1\}^{8\ell_H}$ as

$$s := k||0^{8\ell_0} \oplus G(PS) \quad \text{and} \quad t := PS \oplus H(s).$$

3. Set $m := 00||s||t$. Interpret m as an integer such that $0 < m < N$.
4. Compute the ciphertext as $C = m^e \bmod N$.

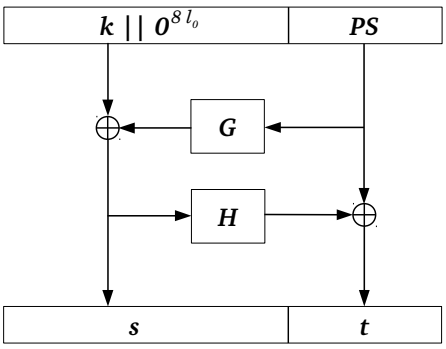


Figure 4.4: *The OAEP padding process.*

[3]PKCS#1 v2.1 introduces "multi-prime" RSA (where the modulus may have more than two prime factors) and the RSASSA-PSS signature. They are not relevant for this work.

[4]The message encryptor can choose between SHA-1, SHA-256, SHA-384, and SHA-512 [EJ01]. The default hash function is SHA-1.

4.1.2.3 RSA-PKCS#1 v1.5 Signatures

In the sequel let $H : \{0,1\}^* \to \{0,1\}^{8\ell_H}$ be a cryptographic hash function (e.g. SHA-1) with ℓ_H-byte output length. A digital signature over message m according to RSA-PKCS#1 v1.5 is computed in three steps.

1. Compute the hash value $H(data)$.
2. Prepend $H(data)$ (from right to left) with
   - a 15-byte ASN.1 string α, which identifies the hash function H,
   - one `0x00`-byte,
   - $\ell - \ell_H - 17$ copies of the `0xFF`-byte, and
   - the `0x01`-byte,

   to obtain a padded message string m of the form

$$m = \texttt{0x01}||\texttt{0xFF}||\ldots||\texttt{0xFF}||\texttt{0x00}||\alpha||H(data).$$

3. Compute the signature σ as

$$\sigma := m^d \bmod N.$$

4.2 Adaptive Chosen-Ciphertext Attacks

Over time, cryptography and security research introduced different classes of attacks breaking different security properties. One of these attack classes is *adaptive chosen-ciphertext attacks*.

Chosen-Ciphertext Attacks. In cryptographic theory, *adaptive chosen-ciphertext attacks* (also known as CCA2 attacks, in contrast to the non-adaptive counterpart known as CCA1 attacks [KL07]) are a class of attacks where the attacker receives as input a ciphertext C from a challenger, and is allowed to query a decryption oracle $\mathcal{O}$. $\mathcal{O}$ decrypts any ciphertext under key k except for C. The goal of the attacker is to obtain some non-trivial information about C (e.g., decrypt C or break indistinguishability [KL07]). See Figure 4.5. A detailed description of these attacks is out of scope of this thesis, we refer to [KL07] for more details.

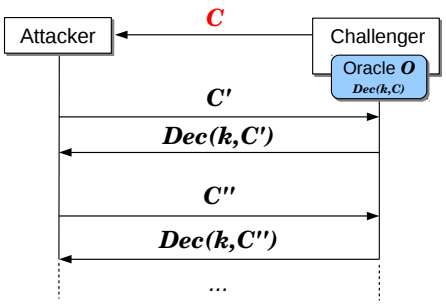


Figure 4.5: *Adaptive chosen-ciphertext attack in theory.*

Chosen-Ciphertext Attacks in Practice. While theoretically useful, this setting does not completely fit to practice. In practice, the attacker has only access to a message receiver (e.g. a web server). If he would have access to a decryption oracle, then he could simply use this oracle to decrypt the given ciphertext C. Typical examples of chosen-ciphertext attacks in practice, like the ones considered in this chapter, do not need a powerful decryption oracle. Instead, they turn a receiver into a weaker oracle, which, for example, only allows to distinguish valid from invalid ciphertexts. An attacker having access to such an oracle iteratively chooses ciphertexts derived from the original ciphertext, sends them to the oracle, and evaluates the responses. With each response, he learns some information about the plaintext. He repeats these steps until he achieves his goal. See Figure 4.6.

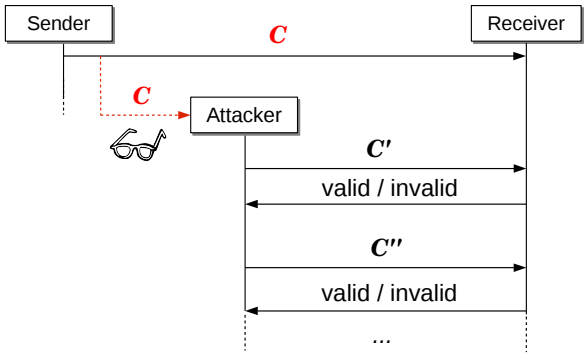


Figure 4.6: *Adaptive chosen-ciphertext attack in practice.*

Two major examples of these attacks are Vaudenay's attack on CBC-based symmetric encryption [Vau02] and Bleichenbacher's attack on RSA-PKCS#1 v1.5-based public-key encryption [Ble98]. We describe them in this section.

Side-Channel Attacks. In practice, the oracle is provided by information gained from the implementation of a cryptosystem, rather than a weakness of the algorithm itself. This includes, for instance, error messages returned from a server, distinguishable timing behavior of the receiver, power consumption, etc. Such additional information is called "side-channel information", and attacks exploiting such information are "side-channel attacks".

4.2.1 Vaudenay's Padding Oracle Attack

At Eurocrypt 2002 Serge Vaudenay presented an adaptive chosen-ciphertext attack on protocols applying CBC mode of operation [Vau02]. The follow-up works by Canvel et al. [CHVV03] and Paterson et al. [DP07, DP10, AP13] showed that the attack is practically applicable to IPSec [TDG98] or TLS [DA99], for example.

One ingredient to Vaudenay's attack is that ciphertexts encrypted in CBC mode (see Section 4.1.1.3) can be modified by an attacker such that the resulting ciphertext is related to the original ciphertext in a certain way. This works for *any* block cipher.

Suppose a ciphertext $C = (iv, C^{(1)}, \ldots, C^{(\xi)})$ encrypting a message $m = (m^{(1)}, \ldots, m^{(\xi)})$ 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 (4.1) in Section 4.1.1.3. Similarly, Equations (4.2) show that the ciphertext

$$(C^{(i-1)} \oplus msk, C^{(i)})$$

is a valid encryption of the message $m^{(i)} \oplus msk$ for all $i \in \{2, \ldots, \xi\}$. Here we use that the decryption algorithm interprets $C^{i-1} \oplus msk$ as an initialization vector, if the ciphertext starts with this value. This property of the CBC mode of operation allows an attacker to *flip* arbitrary bits in the plaintext.

Vaudenay applied an adaptive chosen-ciphertext attack on ciphertexts using the PKCS#5 padding scheme [Kal00]. Recall from Section 4.1.1.2 that this padding scheme is more restrictive than the padding scheme used in XML Encryption as described in Section 4.1.1.2. A padding string PS according to the PKCS#5 scheme can have the following values, if a block cipher of block size $\nu = 8$ is used:[5]

$$\begin{aligned} PS &= \texttt{0x01} \\ PS &= \texttt{0x02 02} \\ PS &= \texttt{0x03 03 03} \\ &\ldots \\ PS &= \texttt{0x08 08 08 08 08 08 08 08}. \end{aligned}$$

If the decrypted CBC ciphertext does not end with one of these values, the ciphertext is invalid.

Vaudenay showed how to use the malleability of CBC and the strict PKCS#5 padding scheme to execute padding oracle attacks. These attacks require an oracle that responds with two types of messages according to the padding validity:

$$\mathcal{O}(C) = \begin{cases} 1 & \text{if } CBC \text{ decryption of } C \text{ ends with a valid padding} \\ 0 & \text{otherwise.} \end{cases}$$

Suppose an attacker is in possession of an encrypted message $C = (iv, C^{(1)})$. The attacker applies Vaudenay's padding oracle attack as follows. He first randomizes the initialization vector and obtains $iv' \in \{0,1\}^{8\nu}$. If he sends the message $(iv', C^{(1)})$ to the oracle $\mathcal{O}$, it most likely answers with 0 since the decryption of a randomized ciphertext results in a plaintext with an invalid padding. Next, the attacker modifies the message until $\mathcal{O}(iv', C^{(1)}) = 1$. He proceeds from the last byte in the initialization vector. He generates values $msk \in \{\texttt{0x00}, \ldots \texttt{0xFF}\}$ and sets

$$iv'' = iv' \oplus 0^{\nu-1} || msk,$$

[5] SSL/TLS[DR08] uses a similar padding scheme, where $PS \in \{\texttt{0x00}, \texttt{0x01 01}, \texttt{0x02 02 02}, \ldots\}$.

where ν is the block cipher block size in bytes (thus, he generates initialization vectors iv'' with different values of the last byte iv''_ν). He sends $(iv'', C^{(1)})$ to the oracle. If $\mathcal{O}(iv'', C^{(1)}) = 1$, he knows that he the decryption of $(iv'', C^{(1)})$ results in a plaintext with a valid padding $PS = \texttt{0x01}$ with a high probability.[6] Thus, the attacker can compute

$$\begin{aligned} x^{(1)}_\nu &= iv''_\nu \oplus \texttt{0x01} \\ &= iv_\nu \oplus msk \oplus \texttt{0x01}, \end{aligned}$$

where $x^{(1)} = \mathsf{Dec}(k, C^{(1)})$. See also Figure 4.7.

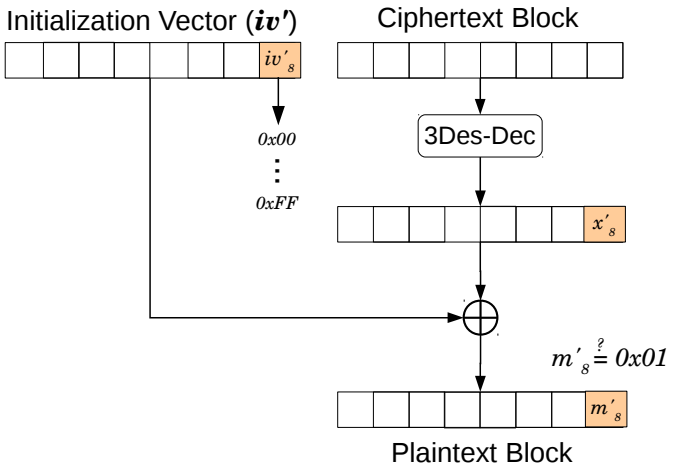


Figure 4.7: *The malleability of CBC and the padding scheme allow an attacker to recover x_8.*

As the attacker now knows the byte $x^{(1)}_\nu$, he can generate a ciphertext resulting in a valid plaintext ending with $\texttt{0x02}$: $iv''_\nu = x^{(1)}_\nu \oplus \texttt{0x02}$. Now, he can repeat the previous procedure for the $(\nu-1)$th byte. He iteratively generates new values of $iv''_{\nu-1} = iv'_{\nu-1} \oplus msk$ and sends them to $\mathcal{O}$. He knows that if $\mathcal{O}(iv'', C^{(1)}) = 1$, the decrypted plaintext ends with $PS = \texttt{0x02 02}$. Thus, the attacker can compute the second last byte

$$x^{(1)}_{\nu-1} = iv''_{\nu-1} \oplus \texttt{0x02}.$$

This way, the attacker can iteratively decrypt all the $x^{(1)}$ bytes. For decryption of one byte, he needs 255 queries in the worst-case, and about 128 queries on average. Knowledge of $x^{(1)}$ allows him to decrypt the ciphertext

$$m^{(1)} = iv \oplus x^{(1)}.$$

To recover a ciphertext consisting of more blocks $C = (C^{(0)}, C^{(1)}, \ldots, C^{(\xi)})$, the attacker simply splits C into ξ block pairs $(C^{(i-1)}, C^{(i)})$, where $i \in \{1, \ldots, \xi\}$.

[6] It is also possible that the attacker guessed $PS = \texttt{0x02 02}$ or another valid padding string correctly. The attacker can however with one additional oracle query find out, if $PS = \texttt{0x01}$. He sets $iv''_{\nu-1} = iv''_{\nu-1} \oplus \texttt{0x80}$. If $\mathcal{O}(iv'', C^{(1)}) = 1$, he can be sure that $PS = \texttt{0x01}$.

He uses each block pair as a pair of an initialization vector and one ciphertext block $(iv, C^{(1)}) = (C^{(i-1)}, C^{(i)})$, and queries $\mathcal{O}$ to recover $m^{(i)}$. See the original paper for more details [Vau02].

The attack became popular and was applied to different standards and implementations in the recent years [Vau02, RD10, DR11a, AP12].

4.2.2 Bleichenbacher's Million Message Attack

In 1998 Daniel Bleichenbacher presented an adaptive chosen-ciphertext attack on protocols implementing RSA-PKCS#1 v1.5 encryption standard [Ble98]. He applied his attack exemplarily on the SSL v3.0 protocol. By turning the SSL server into a decryption oracle $\mathcal{O}$, he was able to decrypt the encrypted SSL `pre_master_secret`, from which a symmetric SSL session key is derived [DR08].

Bleichenbacher's attack allows an attacker to recover the encrypted plaintext m from the ciphertext C. For the attack execution the attacker uses an oracle $\mathcal{O}$ that decrypts C and responds with 1 or 0 according to a conformity of the message. More precisely, $\mathcal{O}$ gives the attacker a hint if the decrypted message m starts with `0x00 02` or not:[7]

$$\mathcal{O}(C) = \begin{cases} 1 & \text{if } m = C^d \bmod N \text{ starts with } \texttt{0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

Bleichenbacher's attack uses such an oracle to invert the RSA encryption function $m \mapsto m^e \bmod N$. The algorithm is based on the malleability of the RSA encryption scheme. Assume that a PKCS#1 v1.5 conformant $C = m^e \bmod N$ is given. Then, $m = C^d \bmod N$ lies in the interval $[2B, 3B)$, where $B = 2^{8(\ell-2)}$ and N has byte-length ℓ ($|N| = \ell$). Bleichenbacher's algorithm proceeds as follows. It chooses a small integer s, computes

$$C' = (C \cdot s^e) \bmod N = (ms)^e \bmod N,$$

and queries the oracle with C'. If $\mathcal{O}(C') = 0$, the algorithm increments s and repeats the previous step. Otherwise, the algorithm learns that

$$2B \leq ms - rN < 3B,$$

for some r. This allows the attacker to reduce the set of possible solutions to

$$\frac{2B + rN}{s} \leq m < \frac{3B + rN}{s}.$$

By iteratively choosing new s, querying the oracle $\mathcal{O}$, and computing new r values, the attacker reduces the possible solutions m, until only one is left. On average, the attacker needs to issue about 215,000 oracle queries when he knows that the searched m is PKCS#1 v1.5 conformant.[8]

[7] This oracle does not strictly verify the PKCS#1 v1.5 conformity as defined in Section 4.1.2.1. The oracle verifies only the first two message bytes. We describe later in this section, what attack performance impact has a strict PKCS#1 v1.5 conformity verification.

[8] The original paper estimates about one million queries, thus the attack has also been named "The million message attack". This estimation assumes that the attacker decrypts an arbitrary ciphertext, which is not necessarily PKCS#1 v1.5 conformant.

Very recently, Bardou et al. [BFK+12] improved the original attack. Their algorithm is about four times faster on average than the original. Their technique allows division to be used to manipulate encrypted PKCS#1 v1.5 messages (and not only multiplication as in the original algorithm). We refer to the original papers [Ble98, BFK+12] for details.

Impact of Oracle Type on Attack Performance. The oracle $\mathcal{O}$ needed for the attack execution can be provided, for example, by a server responding with different error messages if the message is PKCS#1 v1.5 conformant or not. Bleichenbacher tested his attack against an SSL server, which strictly checked the PKCS#1 v1.5 format. However, the attack performance varies. It heavily depends on the restrictiveness of the oracle $\mathcal{O}$ by validating the PKCS#1 v1.5 message format. Bleichenbacher's algorithm relies on the knowledge that the first two message bytes are equal to `0x00 02`. Bardou et al. analyzed different oracle types and their impact on the attack performance [BFK+12]. They characterize the oracles by three Boolean values (see also Section 4.1.2.1 for the PKCS#1 v1.5 format description). A Boolean value signifies whether the oracle responds with 1 for plaintexts that contain:

1. *no* `0x00` bytes at all after the first byte: $\texttt{0x00} \notin \{m_1, \ldots, m_\ell\}$.
2. `0x00` in the first eight bytes of mandatory non-zero padding: $\texttt{0x00} \in \{m_3, \ldots, m_{10}\}$.
3. `0x00` at a wrong position (i.e. the oracle for example does not check whether the unwrapped symmetric key is of correct length).

These three Boolean values can be used to define specific oracles, for example:

- `TTT`: The oracle verifies only the first two bytes of the decrypted message and responds with 1 each time the message starts with `0x00 02`. This makes it very helpful (i.e. "strong") for an attacker in executing the attack.
- `TFT`: This oracle responds with 1 when the decrypted message starts with `0x00 02` and the first eight bytes of mandatory padding contain no `0x00`. This oracle is also very strong.
- `FFF`: Additionally to `TFT`, this oracle checks whether the `0x00` byte is placed at the correct position so that the unwrapped key is of the correct size. Such a behavior leads to many false negatives (since many messages starting with `0x00 02` are indicated as invalid), which slows down the attack performance. This oracle is very restrictive and thus very "weak".

Restrictiveness of an oracle can be measured according to its ability to respond with 1 in case that the decrypted message starts with `0x00 02`. Suppose $P(A)$ defines a probability that the first two bytes of the decrypted message are `0x00 02`. $P(1|A)$ is a probability that the oracle answers with 1, in case that the decrypted message starts with `0x00 02`. For `TTT`, this probability is 1. For `TFT`, the probability can be computed as:

$$P_{TTT}(1|A) = \left(\frac{255}{256}\right)^8 \approx 0.969.$$

If we use a 1024 bit long RSA key to unwrap a 128 bit long symmetric key, the padding string PS has to contain 109 non-zero bytes and is followed by a `0x00` byte (see Figure 4.3). The probability for the `FFF` oracle strictly checking this structure can be computed as:

$$P_{FFF}(1|A) = \left(\frac{255}{256}\right)^{109} \cdot \left(\frac{1}{256}\right) \approx 0.0025.$$

Lower probabilities drastically slow down the attack performance. For example, for a 1024 bit long RSA key, the improved Bleichenbacher attack algorithm needs about 10,000 queries on average using a `TTT` oracle. It needs about 18,000,000 queries using an `FFF` oracle [BFK+12].

4.3 Decryption of Encrypted XML Messages

Recall from Section 2.5.2 that in most scenarios *hybrid* encryption is used. In the following we give a precise description of how a Web Service server processes an encrypted XML message. This example is necessary for the description of our attacks in this chapter.

Figure 4.8 gives an example of a SOAP message containing a hybrid ciphertext. This message consists of the following parts.

1. The `EncryptedKey` element (C_{pub}) with an encrypted session key k stored in the `CipherValue` element.

2. The `EncryptedData` element (C_{sym}) with an encrypted payload data stored in the `CipherValue` element.

A Web Service receiving this XML document processes it as follows. It parses the document to locate the C_{pub} part. It locates the `EncryptionMethod` and `KeyInfo` elements within this part to retrieve the used algorithm and decryption key. The server assumes PKCS#1 v1.5 padding and decrypts the content of the `CipherValue` element. If the resulting plaintext m is PKCS#1 v1.5 conformant, the session key k is extracted from m as a byte string following the second `0x00` byte (see Section 4.1.2.1).

Afterwards, the server searches for the C_{sym} part according to the URI in the `DataReference` element. It determines the needed symmetric algorithm from the `EncryptionMethod` element and decrypts the content of the `CipherValue` element with the session key k. Finally, the decrypted payload data is *parsed*, if well-formed, and put back into the XML document tree. This enables the processing of the whole unencrypted XML document in subsequent steps, and the server can finally respond to the sender.

If an error occurs during the decryption or parsing process, this error is propagated to the message handler, and the message handler responds to the sender with an *error message*. Otherwise, the decrypted XML document can be correctly processed and the sender receives a *legitimate response* from the Web Service server.

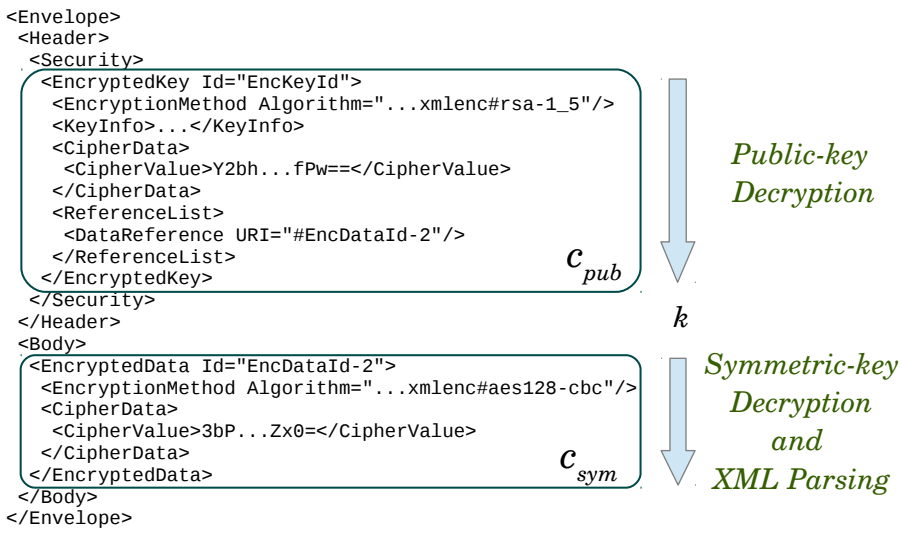


Figure 4.8: *Decryption process of a SOAP message with a hybrid ciphertext.*

4.4 Attacking CBC Ciphertexts in XML Encryption

In this section, we present an attack technique that enables an attacker to decrypt arbitrary data was 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 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 to the Apache Axis2 XML framework. 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 alternative XML frameworks and possibly even other systems beyond XML Encryption as well. For instance, we verified that the attack works against Red Hat JBoss, and SAP systems 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 [ERS+08] 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 [MA05] techniques, or by using a new attack technique that we call *Encryption Wrapping*.

Responsible disclosure. The attack described in this section 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), Red Hat Linux (JBoss), IBM, Microsoft, and a governmental CERT. All acknowledged the validity of our attack. It received the CVE Identifier CVE-2011-1096.

As a countermeasure against our attacks, the newest XML Encryption specification version now includes AES-GCM. AES-CBC and 3DES-CBC are still in the specification for backwards compatibility reasons. Our security considerations are summarized in the specification [ERH+12, Section 6.1.1].

Paper. This section is based on the paper *How to Break XML Encryption* published at the ACM Conference on Computer and Communications Security [JS11] written together with Tibor Jager.

The initial idea of applying adaptive chosen-ciphertext attacks on AES-CBC in XML Encryption came from Tibor. He noticed that it is possible to use the parsing mechanism in XML as an additional side-channel needed to construct new attacks. I analyzed the decryption mechanisms in XML Encryption frameworks and developed a practical attack exploiting the XML parsing mechanisms. I used different performance optimizations, which made our attack about ten times faster than Vaudenay's padding oracle attack [Vau02]. Afterwards, I investigated the side-channels provided by the XML Encryption frameworks that allow the application of this attack.

4.4.1 Related Work

It is well-known that the CBC encryption mode is malleable unless additional methods for ensuring integrity are applied. This was exploited by Vaudenay [Vau02], 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 [Vau02], for instance to other padding schemes and modes of operations [BU02, PY04], random or secret initialization vectors [YPM05], attacks on real world systems like IPSec [DP07, DP10] and ASP.NET, JSF CAPTCHA, the Ruby on Rails framework, and an OWASP security system [RD10, DR11a]. Duong and Rizzo [RD10, DR11a] 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 possible to describe padding schemes which are secure against padding oracle attacks [BU02, PW08], a corresponding formal security model was given by Paterson and Watson [PW08].

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

specification specifies a different padding scheme. By the application of a typical padding oracle attack, the attacker decrypts bytes in a ciphertext block one after another, starting from the last one. According to the oracle responses, he modifies iv' and successively searches for $(iv', C^{(1)})$ combinations resulting in valid plaintexts ending with $PS_1 = \texttt{0x01}, PS_2 = \texttt{0x02 02}, PS_3 = \texttt{0x03 03 03}$, and so on. See Figure 4.9 (left). In XML Encryption this is *not* possible. The XML Encryption padding scheme strictly defines only the value of the *last* padding byte. This gives the attacker a possibility to construct an oracle responding with 1 or 0 according to the validity of the *last* message byte. However, all the preceding padding bytes can have *arbitrary* values. Changing these bytes does *not influence* the oracle response. The attacker has no possibility to find out whether these bytes are valid or not. See Figure 4.9 (right).

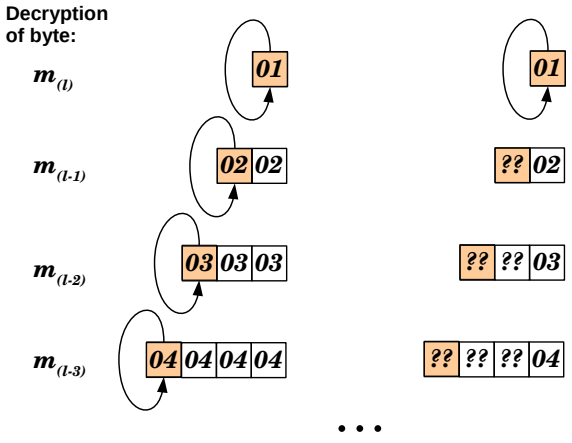


Figure 4.9: *A PKCS#5 padding validity oracle allows an attacker to successively construct valid paddings:* $PS \in \{\texttt{0x01}, \texttt{0x02 02}, \ldots\}$ *(left). This allows him to decrypt* $m_\ell, m_{\ell-1}$*, etc. An XML Encryption padding validity oracle checks only the last padding byte (right). Thus, it is not possible to decrypt* $m_1, \ldots, m_{\ell-1}$*.*

In contrast to padding oracle attacks, we use the *encoding* of data as a side-channel that allows us to attack encryption schemes using a weak mode of operation, which allows to exploit the in most cases inevitable fact that an attacker 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 independently of the padding scheme, and thus potentially also in scenarios where padding oracle attacks are not applicable. Mitchell [Mit05] 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 Encryption ciphertexts. For comparison, Vaudenay's padding oracle attack or the related attack of Rizzo and Duong [RD10] issue 128 oracle queries per byte on average.

4.4.2 Basic Idea of the Attack – A Toy Example

In this section, we describe a simple attack on ciphertexts encrypted in CBC mode, which allows one to recover the plaintext message, if a certain *oracle* (to be described below) is given. The actual attack on XML Encryption from Section 4.4.3 is based on the same idea, but in addition handles some technical obstacles that arise when the theoretical concept is adapted 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 TypeAset and TypeBset. We say that TypeAset contains "Type-A" characters, and TypeBset contains "Type-B" characters. In this toy example we assume that TypeAset $= \{w\}$ 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.

$$\mathcal{O}(C) = \begin{cases} 1 & \text{if } m = \mathsf{Dec}_{\mathsf{cbc}}(k, C) \text{ contains } \textit{only} \text{ Type-B characters.} \\ 0 & \text{otherwise.} \end{cases}$$

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 TypeAset, the probability is equal to

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

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 $x = \mathsf{Dec}(k, C^{(1)})$. 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

$$\begin{aligned} m &= \mathsf{Dec}_{\mathsf{cbc}}(k, (iv'', C^{(1)})) \\ &= iv'' \oplus \mathsf{Dec}(k, C^{(1)}) \\ &= iv'' \oplus x \end{aligned}$$

contains a character from $\mathsf{TypeAset} = \{w\}$. 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. See Algorithm 4.1.

Algorithm 4.1 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: $msk := \mathtt{0x00}$
2: **repeat**
3: $\quad msk := msk + 1$
4: $\quad iv'' := iv' \oplus 0^{8(j-1)} || msk || 0^{n-8j}$
5: **until** $(\mathcal{O}(iv'', C^{(0)}) = 0)$
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 4.10.

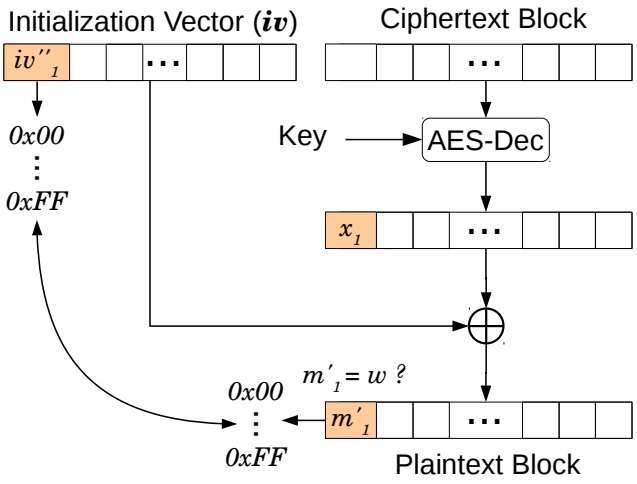


Figure 4.10: *Using the malleability of CBC and the oracle $\mathcal{O}$ to recover x_1.*

4.4.3 Attacking XML Encryption

In this section, we show how to apply the attack described above on a real world Web Service framework. We chose Apache Axis2, since it is one of the major frameworks. However, the attack is in general applicable to an arbitrary framework utilizing XML Encryption. We start our description with some basics on character encodings. Then, we describe how we can use 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.

4.4.3.1 Character Encondings

The XML Encryption specification 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 many more) plus many special symbols (e.g. from mathematics or music). Most characters specified in UTF-8 are rather exotic, and thus only rarely used. 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 4.11. 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 4.11 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.

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

Figure 4.11: *ASCII Character Encoding Table and Classification of Characters.*

4.4.3.2 Axis2 Security Faults

After sending an invalid SOAP message to the Axis2 Web Service server, one can receive different SOAP faults based on the error origin (e.g. invalid security processing, malformed XML document, or an invalid function parameter in the SOAP body). We exploit this to construct our oracle $\mathcal{O}_{\mathsf{Axis}}$. 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`.

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 characters 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.[9]

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

4.4.3.3 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 4.11. We denote with $\mathsf{TypeAset}$ the set of all Type-A characters, and with $\mathsf{TypeBset}$ the set of all Type-B characters. Observe that $\mathsf{TypeAset}$ 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 4.4.2, 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.

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

As described in the previous section, the Axis2 server will return no `security fault`, if

- the decryption $m' = \pi(m) = iv \oplus \mathsf{Dec}(k, C^{(1)})$ 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`.

[9] Please note that an invalidly placed right bracket "`>`" does not cause a parsing error when it is positioned in a text element. In this case, the "`>`" character is escaped and the parsing process proceeds.

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 4.4.2.

4.4.3.4 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 $C = (iv, C^{(1)}, \ldots, C^{(\xi)})$. Note that C may consist of multiple blocks (i.e., $\xi \geq 1$). Due to the rather complex structure of the set $\mathsf{TypeAset}$ 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 implemented the optimized algorithms. We will furthermore make the (in practice reasonable) assumption that the plaintext 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. $m_\nu =$ `0x01`) and consists only of Type-B characters (i.e. $m_j \in \mathsf{TypeBset}$ for all $j \in \{1, \ldots, \nu - 1\}$).

The algorithm is a composition of two sub-procedures, which we call FindIV and $\mathsf{FindXbyte}$.

- The FindIV procedure prepares the ciphertext for our attack. It takes as input a multi-block ciphertext $C = (iv, C^{(1)}, \ldots, C^{(\xi)})$ and an index $i \in \{1, \ldots, \xi\}$, and returns an initialization vector iv such that the ciphertext $C = (iv, C^{(i)})$ is well-formed.
- The $\mathsf{FindXbyte}$ 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 4.2 recovers the plaintext m contained in C. The algorithm loops through all ξ 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 $\mathsf{FindXbyte}$ 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 4.2 Using $\mathcal{O}_{\mathsf{Axis}}$ to recover plaintexts.

Input: $C = (C^{(0)} = iv, C^{(1)}, \ldots, C^{(\xi)})$
Output: $m = (m^{(1)}, \ldots, m^{(\xi)})$
1: **for** $i = 1$ **to** ξ **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** $(m^{(1)}, \ldots, m^{(\xi)})$

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

4.4.3.4.1 Procedure FindIV. In this section, we describe the FindIV procedure. This procedure takes as input a ciphertext $C = (C^{(0)}, C^{(1)}, \ldots, C^{(\xi)})$ 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 specification. 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 TypeBset. 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 TypeBset (no "<" character), then $\mathcal{O}_{\mathsf{Axis}}$ returns exactly $\nu = 16$ responses such that $\mathcal{O}_{\mathsf{Axis}}(C) = 1$. Otherwise, the number of $\mathcal{O}_{\mathsf{Axis}}(C) = 1$ responses *depends on the position* of the first "<" character in the block. For example, if we get only one $\mathcal{O}_{\mathsf{Axis}}(C) = 1$ 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 $\mathcal{O}_{\mathsf{Axis}}(C) = 1$ responses it means that the paddings `0x10`, `0x0F`, and `0x0E` are valid and the "<" character stands in the third position.

For the purpose of getting the number of valid padding bytes we introduce the procedure in Algorithm 4.3, which collects all the valid padding masks.

Algorithm 4.3 GetValidPaddingMasks

Input: $iv, C^{(i)}$
Output: A set of valid padding masks $Pset$

```
1: Pset := ∅
2: for j := 0x00 to 0x7F do
3:    iv' := iv ⊕ (0^{n-8} || j)
4:    if O_Axis(iv', C^(i)) = 1 then
5:       Pset := Pset ∪ iv'_ν
6:    end if
7: end for
8: return Pset
```

Algorithm 4.3 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 in 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

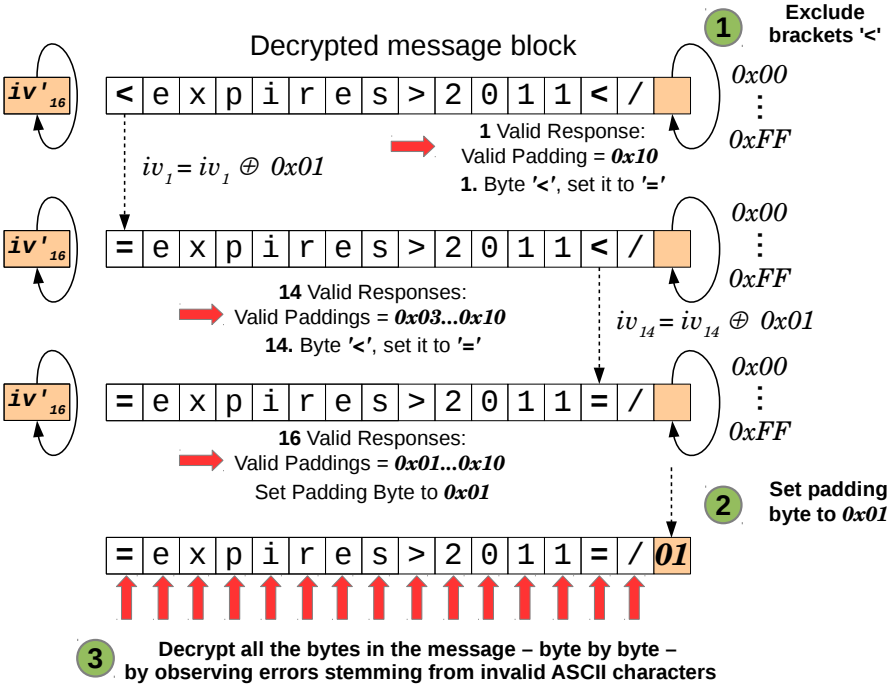


Figure 4.12: *Example of* FindIV *processing: it ensures that* $(iv, C^{(i)})$ *results in a plaintext with a* `0x01` *padding, which contains only Type-B characters.*

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 4.4. In Figure 4.12 we give an example showing how a block is processed in the FindIV procedure.

Algorithm 4.4 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 ⊕ 0x01
6: until |Pset| = ν
7: iv := GetIvWithPaddingMask01(PSet, iv)
8: return iv
```

4.4.3.4.2 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 $x_\nu^{(i)}$ 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 $x_\nu^{(i)}$ as $x_\nu^{(i)} = \texttt{0x01} \oplus iv_\nu$.

If $j \in \{1, \ldots, \nu - 1\}$, then we need to use the $\mathcal{O}_{\mathsf{Axis}}$ oracle to recover $x_j^{(i)}$. We do this by modifying the initialization vector iv and evaluating the response behavior 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 $x_j^{(i)} = (x_{j,1}^{(i)}, \ldots, x_{j,8}^{(i)})$ to denote individual bits of byte $x_j^{(i)}$.

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

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

we know that $x_{j,1}^{(i)} = iv_{j,1}^{(i)}$ for all $i \in \{1, \ldots, \xi\}$ 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 4.11. 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 2 and `0x3C` in Block 3.
- The last four bits of `0x26` and `0x3C` are *not equal*.

We use these observations as follows. In order to determine $x_{j,2}^{(i)}, x_{j,3}^{(i)}, x_{j,4}^{(i)}$, we first run Algorithm 4.5 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 $x_{j,2}^{(i)}, x_{j,3}^{(i)}, x_{j,4}^{(i)}$ (Line 4). The algorithm queries $\mathcal{O}_{\mathsf{Axis}}$ to test whether

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

Algorithm 4.5 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** $\mathtt{R} = 0$ **to** 7 **do**
3: $msk := \mathtt{0xR0}$
4: $iv' := iv \oplus 0^{8(j-1)} || msk || 0^{n-8j}$
5: **if** $\mathcal{O}_{\mathsf{Axis}}((iv', C^{(i)})) = 0$ **then**
6: $Aset := Aset \cup \{\mathtt{msk}\}$
7: **end if**
8: **end for**
9: **return** *Aset*

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

We can now observe that the set *Aset* returned by Algorithm 4.5 contains always either one or two or three elements. To see this, recall that the last four bits of $m_j^{(i)}$ are never modified. Therefore, we have:

- $|Aset| = 1$ if and only if the *last* four bits of $m_j^{(i)}$ 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 $m_j^{(i)}$ 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 $m_j^{(i)} \oplus msk \in \{\mathtt{0x19}, \mathtt{0x1A}, \mathtt{0x1D}\}$. Now observe that there is exactly one mask $msk' \in \{\mathtt{0x25}, \mathtt{0x26}, \mathtt{0x21}\}$ such that $m_j^{(i)} \oplus msk \oplus msk' = \mathtt{0x3C}$ is a Type-A character. Again we can use the oracle to determine msk'. This gives us an equation

$$x_j^{(i)} \oplus (iv_j \oplus msk) \oplus msk' = \mathtt{0x3C}$$

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 that 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. Due to the complexity of the procedure and its similarity to the procedure for $|Aset| = 1$, we omit further details.

4.4.3.5 Attack Variations

Since FindIV executes its attack on each cipher block independently, the attack can easily be parallelized so that each block preparation following by a byte decryption is made in a separate thread or on a different machine.

A priori knowledge about the plaintext could improve the attack significantly. For instance, knowing the XML Schema [WF04], 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. Since this kind of knowledge is commonly published at the Web Service endpoint as a Web Service Description Language (WSDL) [CCMW01], we expect such types of optimization to speed up an attack run.

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 [RD10]).

4.4.4 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 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. As a Web Service 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 running at 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 Service 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 lengths and structures. The shortest messages consisted of 10 characters, thus fitted in a single AES block, and no '<' characters. Larger SOAP messages contained two '<' characters (i.e. one validly closed XML element) in each plaintext 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 Table 4.1. 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.

Plaintext size (bytes)	**Server requests** FindIV	FindXbyte	Total	**Time** (seconds)
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

Table 4.1: *Summary of Experimental Analysis.*

Figure 4.13 shows that the number of server requests and the running time of the attack grow linearly with the size of the ciphertext.

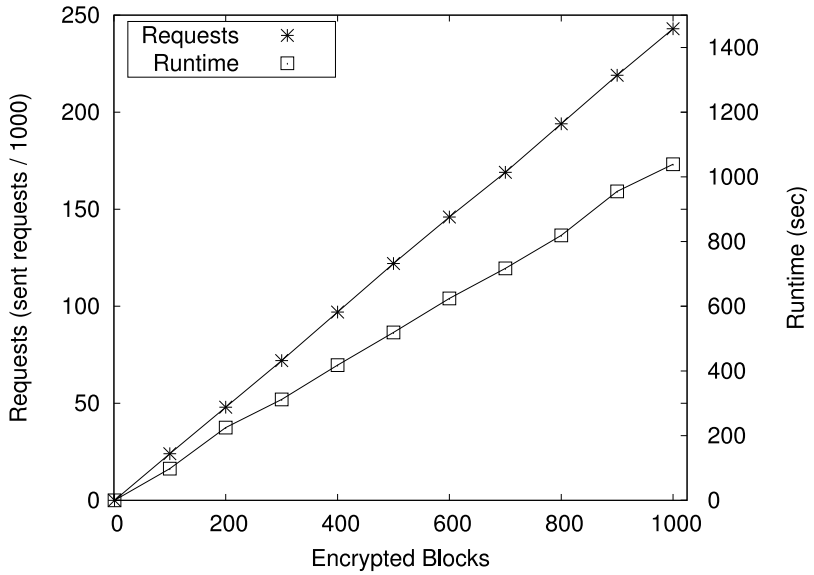


Figure 4.13: *Relation between ciphertext size, number of server requests, and attack duration.*

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 in 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.

4.4.5 Countermeasures

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

4.4.5.1 XML Signature

Application of XML Signatures on ciphertexts can ensure their authenticity and integrity. This specification 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 the described attack *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 ensures these two points, the attack can 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 4.14. In this message the SOAP body contains an encrypted payload, which is signed using an XML Signature and Id-based referencing.

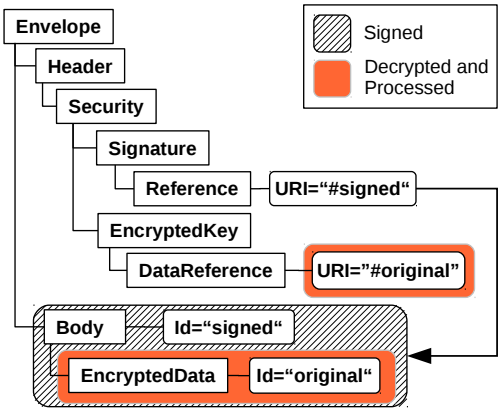


Figure 4.14: *XML Signature applied on the encrypted payload in a SOAP message.*

4.4.5.1.1 Attacker Able to Create Validly Signed Messages. The first problem with 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.

4.4.5.1.2 XML Signature Wrapping. Another problem with application of XML Signatures on ciphertexts is the XML Signature Wrapping (XSW) attack. This attack affects public-key as well as secret-key XML Signatures. Practical XSW attacks were presented in the previous chapter.

Figure 4.15 gives an example of XSW 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 validation component is able to verify 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.

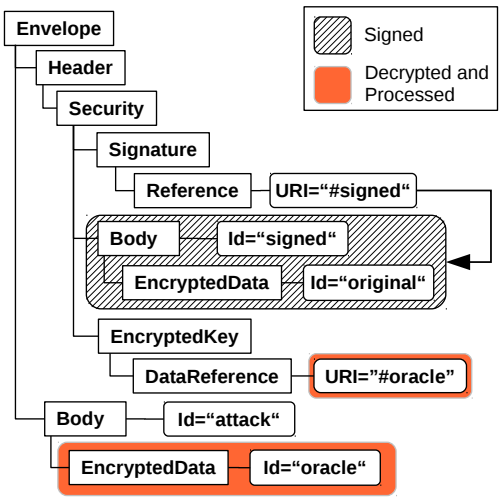


Figure 4.15: *XML Signature Wrapping moves the signature validation to the SOAP header and thereby offers a new possibility for mounting of the XML Enryption attack.*

The server would process the depicted message as follows. It first validates the XML Signature. Afterwards, it decrypts and parses the content of the newly defined SOAP body. If this step fails, the server returns an error. Otherwise, if the data is successfully decrypted and parsed, the payload is forwarded to the business logic. Business logic processing most probably fails, since decryption of a modified ciphertext provides a payload which cannot be processed. Thus, by applying this attack, the attacker must rely on differences between fault messages coming from the *decryption processing* and the *business logic*.

4.4.5.1.3 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 XSW attacks are not applicable. We call these attacks *XML Encryption Wrapping*. To illustrate the idea, consider the message from Figure 4.14, 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 `DataReference` list. A simple attack message is depicted in Figure 4.16. This forces the server to process both `EncryptedData` elements, and thus allows to bypass the XML Signature validation to perform the attack.

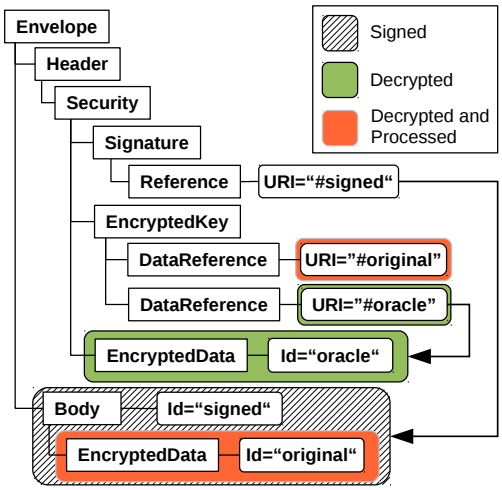


Figure 4.16: *XML Encryption Wrapping copies the encrypted payload to an unsigned document part and thereby offers an attacker construction of a server oracle.*

By application of this attack, the server validates the signature over the original SOAP body. It can also decrypt and correctly process the original SOAP body, as its content is unchanged. On the other hand, the newly inserted `EncryptedData` element is only decrypted. It is not processed by the business logic, since it is hidden in the SOAP header. The server only responds with a fault message if the new ciphertext cannot be decrypted or parsed. Thereby, the attacker creates a server oracle, which *responds with* 0 *or* 1 *depending solely on the validity of the new ciphertext.*

4.4.5.2 Unifying Error Messages

The possibly most obvious countermeasure to our attack consists of 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.

Firstly, 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.

4.4.5.3 Other Countermeasures

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

4.4.5.3.1 Revocation of Session Keys. By application of the attack on one `EncryptedData` element, the attacker uses the same symmetric key for each server request. Revocation of symmetric keys could be considered as a valid countermeasure. However, this countermeasure causes 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. For example, this is the case with messages including boolean values ("yes"/"no") or credit card numbers.

4.4.5.3.2 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 4.4.5.1: dishonest clients having server access would be blocked after sending invalid messages.

4.4.5.3.3 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 4.4.5.1.

4.4.5.3.4 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.

4.4.5.3.5 Decryption only of Signed Elements. WS-Security Policy defines security properties of elements contained in the processed message. For example, it defines that specific elements have to be signed or encrypted. However, it does not explicitly allow to define which elements must not be decrypted. This makes it possible to apply XML Encryption Wrapping attacks. We propose inclusion of new policy mechanisms allowing to decrypt the element only if it is signed.

We communicated with the Apache WSS4J developers. The Apache WSS4J library now includes a configuration parameter called `REQUIRE_SIGNED_ENCRYPTED_DATA_ELEMENTS`. If this is set to `true`, then any symmetrically encrypted `EncryptedData` elements which are are not signed are rejected without processing. In the default configuration, this parameter is set to `false`. The developers considered to default this parameter to `true` for `EncryptedData` elements secured with the CBC mode encryption in the next framework release. However, they decided against this modification as it would break many existing use-cases.

Please note that application of this countermeasure still would not be useful, if the attacker were able to create valid signatures as described in Section 4.4.5.1.1.

4.4.5.4 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 [KBC97] (see [MvV96]) 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) [MvV96], for instance.

The XML Security Working Group already included this mode of operation into the new specification version.

4.4.5.4.1 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. If the encrypted text is long enough, the attacker could observe the server response time differences. Longer response time would indicate correct payload and failure of the MAC verification. Shorter response time would indicate an incorrect payload and a failure of its parsing.

This is a valid assumption also when applying standard DOM-based (tree-based) parsers [BHH+04]. Namely, some DOM-parsers include an underlying streaming-based parser, which is used for preprocessing of the incoming elements. An example is given by `org.apache.xerces.parsers.DOMParser`, which is included as the default parser in JDK. Therefore, developers should pay attention when implementing modes of operations including MAC verification: The encrypted part must always be completely processed and parsed and the MAC must be validated afterwards. Otherwise, different side-channels could appear.

4.5 Breaking PKCS#1 v1.5 in XML Encryption

In 1998 Bleichenbacher [Ble98] published an algorithm for an adaptive chosen-ciphertext attack on the RSA-based PKCS#1 v1.5 encryption scheme. For instance, the algorithm enabled to attack popular implementations of the SSL protocol. These implementations were fixed immediately using a workaround patch, which until today seems to be sufficient to provide security in the context of SSL/TLS. Nonetheless, Bleichenbacher's attack sheds serious doubt on the security of PKCS#1 v1.5.

In spite of these negative results, in 2002, four years after publication of the Bleichenbacher attack, the W3C consortium published the XML Encryption specification [ERI+02], in which PKCS#1 v1.5 encryption is specified as a *mandatory* key transport mechanism. The decision to use PKCS#1 v1.5 despite the known criticisms on its security may be partly due to the fact that the *ad hoc* countermeasures against Bleichenbacher's attack employed in SSL seem to work well – at least for protocols of the SSL family. However, one must not ignore that SSL and XML Encryption are fundamentally different protocols, running in different settings, using a different combination of cryptographic primitives, and providing different side-channels. *Does the use of PKCS#1 v1.5 make XML Encryption vulnerable to attacks*?

Contributions. We describe different attacks on the key transport mechanism of XML Encryption, which is based on PKCS#1 v1.5. Our goal is to turn a given Web Service into a "Bleichenbacher oracle" that allows us to mount the Bleichenbacher attack [Ble98].

First, we show that it is possible to execute Bleichenbacher's attack *in a straightforward way* against some widely-used Web Services implementations, such as Apache Axis2 or Red Hat's JBossWS. This is noteworthy, given that Bleichenbacher's attack has received much attention in the computer security community.

Second, and from a theoretical point of view more interesting, we show that it is possible to conduct practical attacks even against Web Services implementations that seem not vulnerable (e.g. since they implement the classical countermeasure against Bleichenbacher's attack, which we describe below). To this end, we exploit two properties of the XML Encryption specification:

- *The attacker can choose the ciphertext size.* The basic idea is that a larger ciphertext increases the running time of the decryption process. We will show that this allows the attacker to perform very powerful timing attacks, which work even in networks where such attacks can usually not be executed in practice, e.g., in networks with a substantial amount of jitter.

- *A weak mode-of-operation.* XML Encryption uses CBC mode of operation. As described in Section 4.4, CBC exhibits a weakness [Vau02] that allows an attacker to make modifications to the encrypted plaintext, by XOR-ing arbitrary bit strings to the plaintext. We show that it is possible to use this weakness as an alternative way to determine whether a PKCS#1 v1.5 ciphertext is "valid" or not.

  Besides CBC mode, the updated version of the XML Encryption specification allows to use the GCM mode of operation. This mode was introduced to prevent the attacks from Section 4.4. Interestingly, the attack we describe in this section *allows to decrypt GCM ciphertexts, too* — if the receiving Web Service *is able to* decrypt CBC ciphertexts, which is mandatory for any standard compliant implementation. This is due to the fact that we use the PKCS#1 v1.5 weakness in combination with the CBC weakness only to decrypt the session key. After we obtained this session key, we can decrypt an arbitrary ciphertext, regardless of whether it is encrypted using CBC, GCM, or any other mode-of-operation.

  A classical countermeasure against Bleichenbacher's attack is to let the decryption algorithm return a random key if decryption fails. Then the system proceeds with this random key. We stress that the CBC-based attack described in this section *can not be prevented* by this countermeasure.

We verify our attacks by experimental analyses. Because of the very detailed error messages of JBossWS, we found that for certain ciphertexts (a 1/80 fraction of all valid ciphertexts) the straightforward implementation of Bleichenbacher's attack takes less than 30 minutes to recover the symmetric key. Apache Axis2 was used to test the timing-based and CBC-based attacks. The timing-based attack takes 200 minutes on the localhost and less than one week when performed over the Internet. The CBC-based attack takes less than

five days. These attacks are applicable to other systems as well, as we describe below. We stress that all figures are derived using "good" ciphertexts, a property that we describe more precisely in Section 4.5.3, and which holds for (heuristically) one out of 80 ciphertexts (see Section 4.5.3.1). We also note that the recent improvements to Bleichenbacher's algorithm by Bardou et al. [BFK+12] apply in our case as well.

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 [ERS+08] to ensure integrity. However, this is not true, since chosen-ciphertext attacks on XML Encryption can be applied even if either public-key or secret-key XML Signatures over the ciphertext are used.

Further Applications. In close cooperation with SAP AG, Germany, we furthermore verified that all attacks worked also against the implementation of XML Encryption in Version 7.03 of the SAP ABAP stack.

Beyond XML Encryption, the recent JSON Web Encryption (JWE) specification [JRH12] prescribes RSA-PKCS#1 v1.5 as a mandatory encryption scheme. This specification is under developement and at the time of writing there only existed one implementation following this specification [Nim13]. We verified that this implementation was vulnerable to our attacks. We cooperated with the developers, who implemented countermeasures against our attacks.

Responsible disclosure. In June 2011 we disclosed our attack to the W3C XML Encryption Working Group, several developers of well-known Web Services frameworks, and a governmental CERT. All acknowledged the validity of the attack. The attack was assigned CVE-2011-2487.

The W3C XML Encryption Working Group added a remark to the updated specification [ERH+13, Section 6.1.2], which addresses our attack and recommends to use PKCS#1 v2.1 (aka. RSA-OAEP) instead. However, PKCS#1 v1.5 is still contained in the specification, and mandatory for any specification compliant implementation.

Paper. This section is based on the paper *Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption* published at the 17th European Symposium on Research in Computer Security [JSS12]. The paper was written together with Tibor Jager and Sebastian Schinzel.

The idea of applying Bleichenbacher's attack on XML Encryption came from Tibor and me. We designed the high-level attacks. In the implementation phase, Tibor concentrated on the Bleichenbacher's attack and its evaluation. I was responsible for the analysis of the used Web Services frameworks and practical attack applications. Sebastian investigated and implemented the timing-based attacks.

4.5.1 Related work

Bleichenbacher's attack [Ble98] on PKCS#1 v1.5 [Kal98] was published at CRYPTO 1998. This attack was applied by Klima et al. to popular real world implementations of the SSL protocol by incorporating an additional side-channel: a version number check over PKCS#1 plaintext [KPR03]. In [BFK+12]

Bardou et al. describe several ways to improve the efficiency of Bleichenbacher's attack. They demonstrate the attacks on various security devices including smartcards, USB security tokens, and HSMs (Hardware Security Modules).

At Crypto 2001 Manger [Man01] presented an attack on Version 2.0 of PKCS#1 (RSA-OAEP) [KS98], which is very similar to Bleichenbacher's attack, and applicable to the current Version 2.1 [JK03] as well. Manger's attack is considered as rather theoretical, since it requires that a specific side-channel oracle is given. We are not aware of any practical application, since the required side-channel information is usually not given in practice. Bauer et al. [BCN+10] showed that PKCS#1 v1.5 is insecure in two non-standard (but realistic) settings, namely broadcast encryption and IND-CPA security in presence of a plaintext validity checking oracle.

A result with many similarities to our work was published by Smart [Sma10], who shows how to apply a Bleichenbacher-style attack to break RSA-based PIN encryption, if a certain side-channel oracle is given. Thus, like our work, Smart points out the danger of using legacy cryptosystems, and suggests to replace them with new ones. Very recently, Degabriele et al. [DLP+12] provided another Bleichenbacher-style attack that allows to forge signatures in an EMV transaction. Both these attacks are rather theoretical, since it is unlikely that the required oracle is given in practice.

In [Res02] it was noted that valid (symmetric-cipher) padding may lead to a side-channel that allows an attacker to mount Bleichenbacher's attack, but without additionally exploiting the plaintext-malleability of the symmetric cipher or giving any concrete application. In contrast, we obtain an oracle which is able to determine whether a given ciphertext is PKCS#1 v1.5 conformant with probability 1 in at most 256 steps, and show that this attack is practically relevant.

Generally, we give a truly practical attack which is directly applicable to a vast number of real world systems. This shows that using legacy cryptosystems is extremely dangerous, and makes a very strong case for replacing them.

4.5.2 Attacks

4.5.2.1 Axis2 Security Faults

We performed our tests using the Apache Axis2 framework. When receiving a SOAP message containing encrypted data, Axis2 locates C_{pub} and C_{sym} in the XML document structure. In order to decrypt C_{pub}, Axis2 performs the PKCS#1-validity checks described in Section 4.1.2.1. In addition, Axis2 tests whether the resulting session key k has a length equal to 16, 24, or 32 bytes. If this fails, then the SOAP error message `security processing failed` is returned. Otherwise, key k is used to decrypt C_{sym}, which yields the payload data m. Finally, m is parsed as an XML message. If this parsing fails, a `security processing failed` SOAP error message (i.e., *the same error message* that is returned if decryption of k fails) is returned. Otherwise, it is forwarded to the next module in the processing chain or to the business application

Now, assume we are given a ciphertext (C_{pub}, C_{sym}), and we modify the key

encapsulation part C_{pub} (this is necessary to mount Bleichenbacher's attack). Then we obtain a modified ciphertext (C'_{pub}, C_{sym}). If we send this ciphertext to the Web Service, then we will receive a `security processing failed` error message, since either processing of C'_{pub} or parsing of the payload m contained in C_{sym} will fail (except for a negligibly small probability). Thus, we are not able to distinguish whether C'_{pub} is a valid or an invalid ciphertext. This seems to thwart Bleichenbacher's attack at first sight. However, in the following, we will describe techniques for exploiting side-channels allowing us to determine the validity of C'_{pub}.

Remark: Though we analyze mainly Apache Axis2, and thus strictly speaking all our experimental results are only valid for Axis2, we stress that the attacks described below are in principle applicable to other frameworks as well (as we verified for SAP, for instance). Moreover, as we describe in Section 4.5.3.4 in detail, it turns out that exploiting certain additional framework-specific side-channels may even lead to dramatically more efficient attacks.

4.5.2.2 Basic Ideas

Imagine an attacker who intercepts a message transferred to the Web Service server and whose goal is to decrypt C_{sym}. In order to gain the session key k needed for data decryption, the attacker can apply Bleichenbacher's attack on C_{pub}. In this section, we describe two ways to obtain a side-channel that allows to determine whether a given ciphertext is valid (PKCS#1 v1.5 conformant), *even though* the server does not respond with error messages allowing to distinguish valid from invalid ciphertexts. Thus, we turn a seemingly secure Web Service server into an oracle $\mathcal{O}$ responding with 1, if the decrypted k is valid, or 0 otherwise.

See Figure 4.17 for the description of this scenario. Bleichenbacher's attacker sends an adapted ciphertext C'_{pub} to $\mathcal{O}$. $\mathcal{O}$ inserts C'_{pub} into an XML document, sends it to the Web Service server, and evaluates its response. $\mathcal{O}$ repeats this step until it knows if C'_{pub} is valid or not. Afterwards, it responds to the attacker with 1 or 0 according to the message validity. When constructing the oracle $\mathcal{O}$ we have to face the following challenges:

1. $\mathcal{O}$ must not respond with false positives: ciphertexts falsely identified as valid cause that Bleichenbacher's algorithm execution ends up in a wrong internal state from which the algorithm cannot recover.

2. $\mathcal{O}$ should respond with as few false negatives as possible: valid ciphertexts falsely identified as invalid slow down the attack performance.

3. The number of requests and the amount of data sent between $\mathcal{O}$ and the Web Service server should be as small as possible.

The goal of this work is to show how to construct an oracle $\mathcal{O}$. If such an oracle is given, the attacker is able to query $\mathcal{O}$ with further ciphertexts $C''_{pub}, C'''_{pub}, \ldots$ to execute Bleichenbacher's attack.

Let us first sketch our ideas on a high level. The first idea is to exploit the fact that the server decrypts and parses the payload data C_{sym} if and only if C_{pub} is valid. Therefore, the time between sending the ciphertext and receiving

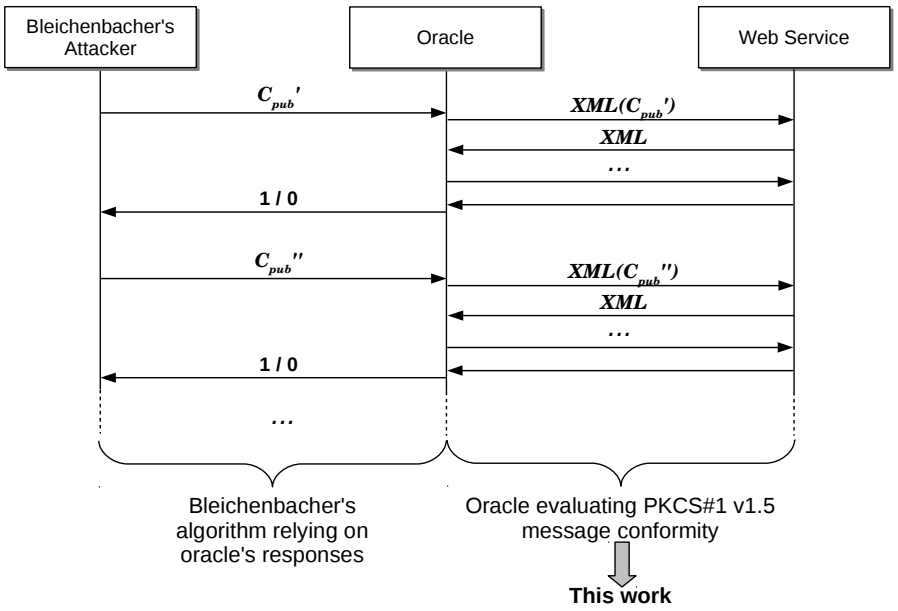


Figure 4.17: *Bleichenbacher's attack algorithm relies on an oracle that returns 1 or 0 according to the message validity. In this section we describe how to construct such an oracle by communicating with a Web Service server.*

the error message depends on the validity of C_{pub}. Therefore, we can create a Bleichenbacher oracle by measuring this response time. In practice, this does not always form a practically useful side-channel, since timing measurements in real networks contain jitter introduced by network latency or server workload. However, here it comes in handy, that the attacker can set C_{sym} to any bit string whose length is a multiple of the block size of the block cipher. Thus, by increasing the length of C_{sym}, the attacker can also increase the timing gap between a valid and an invalid C_{pub}. The challenge is to keep C_{sym} as small as possible (to keep the attack efficient), but as large as necessary (to get distinguishable timing results).

In certain scenarios, the timing approach may become inefficient, for instance if the server workload is extremely unbalanced, or the network connection is not reliable. Therefore, we describe a second idea, which exploits a weakness of the CBC mode. Consider a ciphertext encrypting a single (padded) payload data block $m^{(1)}$. Recall that such a ciphertext consists of an iv and a ciphertext block $C^{(1)} := \mathsf{Enc}(k, x)$, where $x := m^{(1)} \oplus iv$. Thus, by flipping bits in iv, we can implicitly flip bits in the plaintext $m^{(1)}$. In particular, we can modify the last byte of $m^{(1)}$, which contains the number of padding bytes. The crucial observation is now that there exists one modified iv' such that the last byte of $m^{(1)'} = x \oplus iv'$ equals the block-length of the block cipher. In this case, $(iv', C^{(1)})$ corresponds to an encryption of the empty string, and XML parsing of the empty string does *not* fail. We use this property to distinguish a valid from an invalid C_{pub}.

In the following sections, we describe how to use these ideas to construct an oracle $\mathcal{O}$ telling whether a given C_{pub} is valid. This oracle can then be used to mount Bleichenbacher's attack.

4.5.2.3 Timing Attack

In this section, we describe a timing oracle $\mathcal{O}_t$ that determines if a given C_{pub} is valid. Our observation is that the analyzed Web Service only then decrypts C_{sym} if C_{pub} is valid. Furthermore, parsing of the clear text does not start until C_{sym} was fully decrypted, i.e. filling C_{sym} with random data will yield a parsing error *after* the decryption has completed, except for some negligible probability. Another observation is that a larger C_{sym} leads to measurably longer decryption times as depicted in Figure 4.18. This combination makes our attack well suited for timing attacks across noisy networks, because the attacker can increase the timing differences by changing the size of C_{sym}. Note that the actual content of C_{sym} is irrelevant, only the size is important for the timing delay. In our experiments we enforced Axis2 to decrypt C_{sym} using AES-CBC. Note that 3DES-CBC would bring even larger timing differences because the decryption process in 3DES is less efficient than AES. This would make our attack easier.

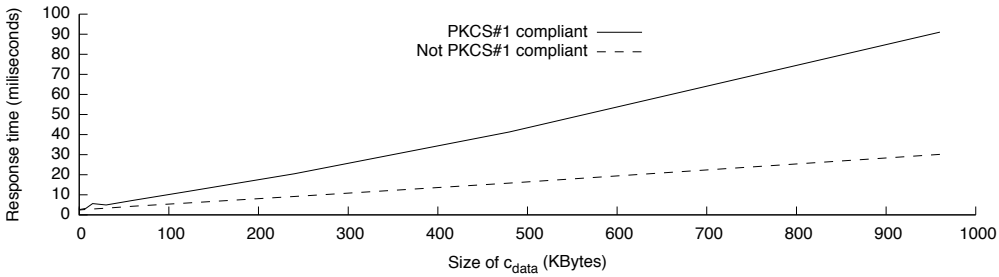


Figure 4.18: *Timing difference of valid C_{pub} and invalid C_{pub} in relation to the size of C_{sym}, which was decrypted using AES-CBC.*

By nature, the timing measurements in an adaptive chosen-ciphertext attack need to be evaluated during the attack because subsequent requests depend on the answer of the timing oracle of the previous request. By executing Bleichenbacher's adaptive chosen-ciphtertext attack, it is very important that the given oracle does not respond with 1 when a ciphertext is not PKCS#1 v1.5 conformant (an oracle request should not result in a *false positive*). If the this would happen, Bleichenbacher's algorithm computes a wrong interval and gets into a non-recoverable state. On the other hand, *false negatives* (PKCS#1 v1.5 conformant ciphertexts falsely indicated as invalid) slow down Bleichenbacher's attack execution. We propose a new algorithm that takes into account these properties. The algorithm exploits the facts that valid keys have a longer processing time than invalid keys, and that any noise in the form of random delays that occur in networks and busy systems is strictly additive. Intuitively, the algorithm determines the minimum response time t_{min} for valid keys. Any measured response time $t < t_{min}$ must be from an invalid key. We call a key a *candidate* for a valid key if the associated response time is above t_{min} . To make sure that this candidate is not actually an invalid key with the random noise pushing it above the timing boundary, we repeat the timing measurement with this key i times, resulting in a set of measurements $T_{C_{pub}} = \langle t_1, t_2, \ldots, t_i \rangle$. If

any of the repeated measurements is below the boundary, the key is marked as invalid. This algorithm was designed by Sebastian Schinzel. Our later measurements confirmed that it was well-suited for our purposes and for omitting false positives which break Bleichenbacher's algorithm. Note that the attacker can freely choose the size of the timing differences of valid and invalid keys by adjusting the size l of C_{sym}. Equation 4.3 formally defines the timing oracle.

$$\mathcal{O}_t(C_{pub}, l) = \begin{cases} 1 & \text{if } min(T_{C_{pub}}) \geq t_{min}, \\ 0 & \text{if } \exists t \in T_{C_{pub}} : t < t_{min}, \end{cases} \tag{4.3}$$

The algorithm is split into two phases: First, there is a calibration phase, where the particular timing conditions of the system are determined. The result of this phase is t_{min}, which is fed to the timing oracle in the second phase.

4.5.2.3.1 Calibration Phase The oracle can determine if a given C_{pub} is valid by measuring the response time of a request that uses this particular key. Thus, the oracle must be calibrated so that it can distinguish the response time of a valid C_{pub} from an invalid C_{pub}. For this, we perform n requests with a valid C_{pub} and record the set of timings $T_{valid} = \langle t_1, t_2, \ldots, t_n \rangle$. Note that the attacker already has one valid C_{pub} from the message he listened in to. Let $t_{min} = min(T_{valid}) - \epsilon$ where ϵ accounts for the fact that $min(T_{valid})$ is only an approximation for the actual minimum response time t'_{min} of valid keys, because $t'_{min} \leq t_{min}$.

We assume at this stage that the response times for valid and invalid keys remain stable during the attack phase, i.e. t_{min} remains the lower boundary for response times with valid keys for the duration of the attack. If this assumption does not apply for a given system, the attacker can regularly repeat the calibration phase to address fluctuations of t_{min}.

4.5.2.3.2 Attack Phase. Now that $\mathcal{O}_t$ is calibrated, the attacker can apply the Bleichenbacher algorithm as described in Section 4.2.2. Figure 4.19 describes the procedure of $\mathcal{O}_t$. The Bleichenbacher algorithm calls $\mathcal{O}_t$ and passes C_{pub} as a parameter. The oracle copies C_{pub} in a SOAP message, sends it to the server, and measures the response time t. The oracle answers with 0 if $t < t_{min}$. It repeats the measurement n times if $t \geq t_{min}$ to confirm that C_{pub} is indeed valid.[10] The oracle answers with 1 if all measurements resulted in greater response times than t_{min}.

[10] We used $n = 100$ in our measurements.

```
def is_valid(C_{pub}, n):
        do n times:
                start = now()
                request(C_pub, l)
                end = now()

                t = end - start
                if t < t_min:
                        return 0 // "invalid"
         return 1 // "valid"
```

Figure 4.19: *Pseudo code sketching the validation routine of candidates of valid keys.*

4.5.2.4 Exploiting a Weakness of CBC

In this section, we describe another attack on C_{pub}, which is based on the properties of the CBC mode of operation. As described in the previous sections, Axis2 processes XML Encryption as follows. It first decrypts C_{pub}. Afterwards, it uses the decrypted session key k to decrypt C_{sym}. If an error during the decryption occurs, Axis2 returns an error message that reads `security processing failed`. There are several possible causes for this error:

- C_{pub} decryption: the decrypted C_{pub} was invalid.
- C_{sym} decryption: the decrypted data from C_{pub} was valid, but the C_{sym} decryption or padding processing failed.
- *data* parsing: C_{sym} was correctly decrypted and padded, but it contained non-printable characters (e.g. `NULL` or `vertical tab`) or a badly placed special character (`<` or `&`).

Thus, if the attacker receives a `security processing failed` error message, he does not know in which of these three steps the message processing failed (and thus if C_{pub} is valid or not). The attacker only then knows that C_{pub} is valid if all steps including parsing completed successfully. Therefore, the attacker must find a way to construct well-formed data that will be parsed successfully.

To construct well-formed data, we create C_{sym} consisting of two randomly generated 16 bytes long blocks $C_{sym} = (\text{iv}, C^{(1)})$. Then we submit the ciphertext (C_{pub}, C_{sym}) to the Web Service, claiming that C_{sym} is generated in CBC mode. The latter is possible by simply adjusting the `Algorithm` attribute value in the `EncryptionMethod` element (see Figure 4.8). The decryption module first decrypts the $C^{(1)}$ block resulting in: $x = Dec_k(C^{(1)})$. The result of decryption x is afterwards XORed with the initialization vector iv, so that the plaintext block becomes $m^{(1)} = \text{iv} \oplus x$. The last byte of $m^{(1)}$ is taken as a padding byte and the padding is checked. Again, if the padding byte is not valid or the unpadded bytes result in non-printable characters, an error is returned.

To overcome this problem one can iterate over all the byte values in the last byte of the initialization vector iv and construct 256 different iv' values (see

Figure 4.20). As flipping a bit in iv implicitly changes the corresponding bit in the $m^{(1)}$ block, one can iteratively modify the value of the last byte in $m^{(1)\prime}$. Thereby exactly one pair $(iv', C^{(1)})$ results in a valid padding byte $0x10$, which pads the whole plaintext block. As this special plaintext is empty (0 bytes in length), parsing always succeeds. In this case, the message is passed to the next module in the Axis2 processing chain. Note that errors in other modules result in different error messages.

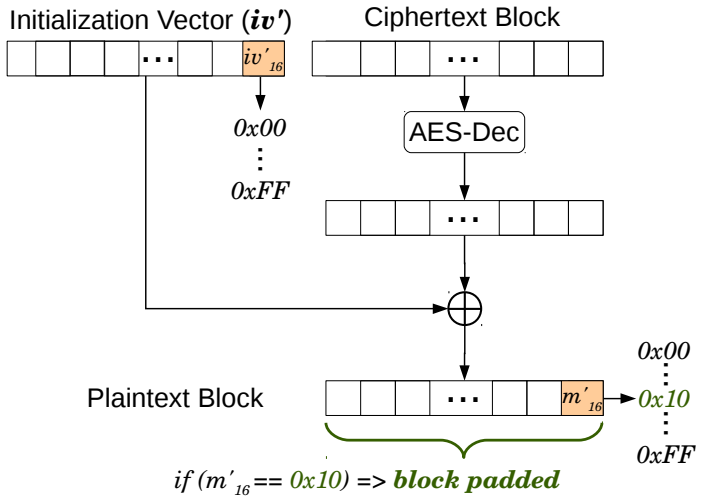


Figure 4.20: *Iterating over all the values of the last iv' byte results in one plaintext with the padding byte $m'_{16} = \texttt{0x10}$. This ensures that the whole 16-bytes long block is padded and the unpadded plaintext is of length 0.*

We can use these observations for constructing an oracle, which returns 1 or 0, depending on the validity of the given C_{pub}. For each tested C_{pub}, the CBC-oracle $\mathcal{O}_{cbc}$ needs to send at most 256 requests with different iv' values[11]. As shown in Equation 4.4, if Axis2 responds with a `security processing failed` error for a given C_{pub} and all possible values of iv, then $\mathcal{O}_{cbc}$ returns that C_{pub} was invalid.

$$\mathcal{O}_{cbc}(C_{pub}) = \begin{cases} 1 & \text{if } \exists iv_{16} \in \{0,1,\ldots,255\} : Dec(C_{pub}, iv) = \text{“no error”} \\ 0 & \text{if } \forall iv_{16} \in \{0,1,\ldots,255\} : Dec(C_{pub}, iv) = \text{“error”} \end{cases} \tag{4.4}$$

[11] We want to mention that the parsing error could be omitted if the server would be forced to handle the decrypted bytes as binary data. This would be possible by forcing the server to process MTOM encrypted binary data [MGRN05]. It would improve our attack by factor of 16 as *each* plaintext containing a valid padding would be valid, independently of the unpadded content. However, as the encryption application on the binary data is not supported by the analyzed frameworks, we do not investigate it further.

4.5.3 Experimental Analysis

In this section, we describe the results of our practical experiments. The timing-based and padding-based attacks were carried out using "good" ciphertexts, see Section 4.5.3.1 for a description of this property. We did this to speed up our experiments, which was necessary due to limited computational resources. However, a heuristical analysis shows that it is very likely that a random ciphertext (e.g., encrypting a cryptographic key with correct padding) meets this property: For a 1024-bit modulus a fraction of about 1/80 of all ciphertexts is good in the above sense.

We stress that all timing figures derived from our experiments are valid only for this 1/80 fraction of all PKCS#1 ciphertext, which is however still a significant number. We also note that Bleichenbacher's attack in principle allows to decrypt any ciphertext, but for a 79/80 fraction the running time of the attack will be longer. However, we stress that it is possible to test whether a given ciphertext is good, by issuing at most $N/(3B) - N/(2B) = N/(6B) \approx 10,000$ oracle queries.

Bad performance of the described timing-based and padding-based attacks is caused by high restrictiveness of the oracle that was provided by the tested Apache Axis2 implementation. This implementation can also be characterized by a variant of an `FFF` oracle type introduced by Bardou et al. [BFK+12] (see Section 4.2.2). Apache Axis2 strictly checks whether the unwrapped key is of correct length. It accepts three possible key lengths: 16, 24, or 32 bytes. Later in this section we show how to perform Bleichenbacher's attack against JBossWS using direct error messages. Using error messages of JBossWS results in a permissive oracle, and thus in better attack performance.

In order to evaluate our attacks, we deployed a Web Service secured with XML Encryption and generated a valid SOAP message containing C_{pub} in the SOAP header. This element included a symmetric key for C_{sym} decryption, encrypted with a 1024-bit RSA key. The results of the timing-based and padding-based attacks shown here were all performed against Axis2. Please note that we also got similar results when testing our attack against the other mentioned XML Encryption implementations and other RSA key sizes.

4.5.3.1 Probability of "good" ciphertexts

The first step of Bleichenbacher's algorithm searches for an integer s such that $m \cdot s \bmod N$ is PKCS#1 v1.5 conformant. Note that $m \cdot s \bmod N$ can only be PKCS#1 v1.5 conformant if

$$\frac{i \cdot N}{3B} \leq s \leq \frac{i \cdot N}{2B}$$

for some $i \in \mathbb{N}$. Therefore, the Bleichenbacher algorithm starts with $s = N/3B$ and increments this value until a suitable s is found. Clearly, this procedure finds s quickly if m has the property that there exists an s such that

$$\frac{1 \cdot N}{3B} \leq s \leq \frac{1 \cdot N}{2B}$$

and $m \cdot s \bmod N$ is PKCS#1 v1.5 conformant. Moreover, in our application we will only be able to learn that a ciphertext $C = (ms)^e \bmod N$ is PKCS#1 v1.5 conformant if $ms \bmod N$ has the form

$$ms \bmod N = 00||02||PS||00||k$$

where PS contains only non-zero bytes and the byte-length of k is equal to 16, 24, or 32. In the sequel, we will say that a ciphertext is a *good* ciphertext if it satisfies these properties.

In order to save computation time, all our experiments were executed with random *good* ciphertexts. Thus, all our experimental results are meaningful only if the probability that a honestly generated ciphertext meets the above property is sufficiently high. This leads us to the question *what is the probability that a real world ciphertext is* good?

We ran some additional experiments in order to determine the probability that a random ciphertext is *good*. To this end, the algorithm depicted in Figure 4.21 was implemented. This algorithm generates a random 1024-bit RSA modulus. Then it generates ℓ random padded plaintexts, and counts the number of plaintexts such that there exists a suitable $s \in [N/3B, N/2B]$ with $m \cdot s \bmod N$ being PKCS#1 v1.5 conformant.

1. Generate a random 1024-bit RSA modulus N. Set $c = 0$.
2. For i from 1 to ℓ do:
   - Choose a random bit string k
   - Pad k according to PKCS#1 v1.5, such that
     $$m = 00||02||PS||00||k$$
   - If there exists $s \in [N/3B, N/2B]$ such that
     - $m \cdot s \bmod N$ is PKCS#1 v1.5 conformant,
     - $ms \bmod N = 00||02||PS||00||k$, with $|k| \in \{16, 24, 32\}$,

     then set $c = c + 1$.

Figure 4.21: *Experimental analysis of the distribution of "good" ciphertexts.*

We repeated this algorithm 100 times, i.e., we generated 100 random moduli, and tried $\ell = 1,000$ padded plaintexts for each modulus, such that in total 100,000 plaintexts where tested. Among these 100,000 plaintexts there were 1,543 padded plaintext that lead to *good* ciphertexts. Thus, about one in 80 ciphertexts is *good*.

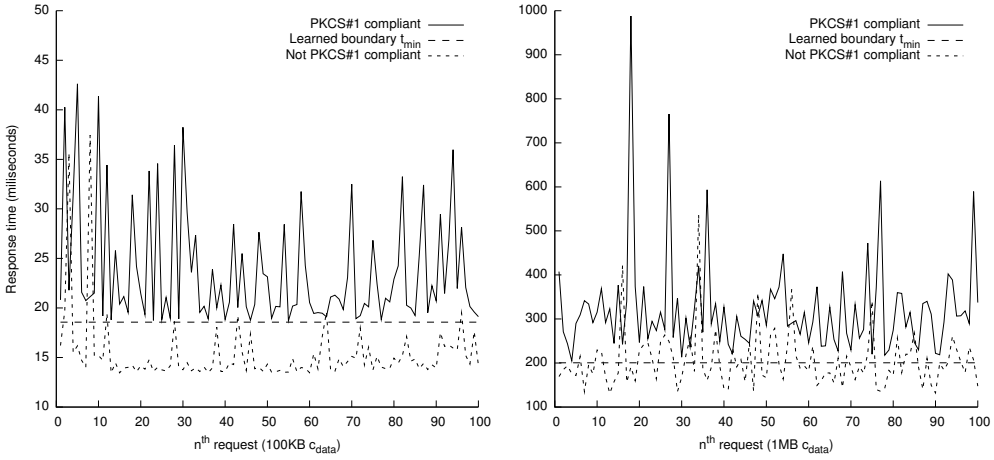
Note also that in general *all* ciphertexts are vulnerable, even though the attack execution might take some more time (i.e., more server requests) to decrypt.

4.5.3.2 Timing-based Attack

In this section, we show the results of the empirical evaluation of $\mathcal{O}_t$ proposed in Section 4.5.2.3. We used the RDTSC assembler instruction of recent Intel Pentium processors to measure the timings with below nanosecond accuracy. In the following, we describe the results of the timing oracle evaluation of two different attacker models.

4.5.3.2.1 Attack on Local Machine. In this measurement setup, we run the Axis2 server and the attack script on the same computer. This is a very practical attack scenario, e.g. in cloud computing and especially in a *Platform as a Service*, where it is feasible for an attacker to rent a virtual machine that is co-located on the same physical hardware [RTSS09] as the victim.

The measurement computer had 2 Intel XEON 2.4 GHz processors. Figure 4.22a shows the response times measured during the calibration phase with 100 KB C_{sym} ciphertext and a C_{pub} encrypted with an 1024-bit RSA key. The solid line denotes valid requests, the dashed horizontal line marks the learned boundary, and the dotted line indicates invalid requests. When compared to the learned timing boundary t_{min}, it becomes clear that most invalid requests are below t_{min}. Any request above t_{min} is treated as a candidate for a valid request and repeated $n = 100$ times for confirmation. The figure suggests that only few invalid requests slipped above t_{min} leading to a repetition of the request. Nevertheless, our oracle responded all the queries correctly (no false negatives and no false positives appeared). As a result, C_{pub} could be reconstructed successfully in 200 minutes. Overall, the 321,870 oracle queries resulted in 398,123 queries in our measurement setup, i.e. the oracle needs to perform 1.24 actual Web Service requests per oracle query. On our hardware, we could perform on average 37 Web Service requests per second.



(a) Localhost **(b)** Internet

Figure 4.22: *Response times with valid and invalid C_{pub}.*

4.5.3.2.2 Attack through Internet. Additionally, we evaluated the effectiveness of the timing oracle for a remote attacker who attacks the Web Service through the Internet. For this measurement setup, we chose two Planetlab nodes at universities. The nodes were seven hops apart from each other and the round trip time was approximately 22 milliseconds.

We calibrated the valid/invalid boundary of the timing oracle as shown in Figure 4.22b and used 1,000KB of random data as C_{sym}. In this configuration, the oracle correctly answers approximately 2,000 queries per hour and needs to perform approximately 2,400 actual Web Service requests to the server. Thus, an attacker can decrypt C_{pub} across practical networks in less than one week.

4.5.3.3 Padding-based Attack

As the padding-based attack does not depend on the network connection, we tested its functionality on a localhost so the Web Service client and server did not communicate over the Internet. The used machine had 2 Intel XEON 2.4 GHz processors.

The whole attack execution lasted less than five days. Thereby, the attacker sent about 322,000 oracle queries, which resulted in 82,180,000 ($\approx 256 * 322,000$) total server requests.

Note that other ciphertexts could lead to different attack executions with different number of oracle queries. Our practical attack targeting one ciphertext could be seen as a proof that the constructed CBC-oracle $\mathcal{O}_{CBC}$ is sufficient to mount Bleichenbacher's attack.

4.5.3.4 Exploiting JBossWS PKCS#1 Processing

All our attacks presented so far are also applicable to the XML Encryption implementation of JBossWS [JBo13]. In addition, we discovered another side-channel in JBossWS 6.0 that allows us to mount Bleichenbacher's attack *directly*, by adapting it slightly to the XML Encryption setting. We do not even need to consider *good* ciphertexts.

In the sequel, let us assume a 1024-bit modulus is used. Given a ciphertext C_{pub}, JBossWS first decrypts C_{pub} and obtains a padded plaintext $m = (m_1, \ldots, m_{128})$ consisting of 128 bytes m_i, $i \in \{1, \ldots, 128\}$. Then it performs the following checks:

1. Test whether $(m_1, m_2) = (0x00, 0x02)$. If true, proceed.
2. Test whether there exists $i \in \{3, \ldots, 10\}$ such that $m_i = 0x00$. If false, proceed.
3. Test whether there exists $i \in \{11, \ldots, 128\}$ such that $m_i = 0x00$. If true, proceed.

If any of these tests fails, an `internal WS-Security error` SOAP fault message is generated and returned. Otherwise, JBossWS tries to proceed. This might also fail, for instance if the decrypted key has incorrect length or if the parsing of

encrypted payload fails. However, in this case a *different* error message, namely a `Decryption failed` SOAP fault, is returned.

This leads to the following side-channel leakage. If the attacker does *not* receive the `internal WS-Security error` SOAP fault, then it learns that the first two bytes $(m_1, m2)$ of the plaintext contained in the submitted ciphertext were equal to $(0x00, 0x02)$. This suffices to mount Bleichenbacher's attack directly. The oracle provided by this implementation can also be characterized by the `FFT` oracle type introduced by Bardou et al. [BFK+12].

We applied the Bleichenbacher attack on JBossWS using a 1024-bit key. We measured the execution time on a machine with two 2.8 GHz processors. It turns out that it is possible to decrypt a given ciphertext within less than 30 minutes, by issuing about $250,000$ server request. Again, different ciphertexts could lead to different attack executions with different number of oracle queries.

4.5.3.5 Exploiting Additional Side-Channels in Apache Axis2

As described in Section 4.3, SOAP messages containing encrypted data typically consist out of two parts: C_{pub} and C_{sym}. In order to reference the C_{sym} part from the C_{pub} part, the `DataReference` element is used. Using `DataReference`, the message interceptor can locate the part dedicated for symmetric decryption.

By modifying the C_{pub} ciphertexts in the *original* SOAP messages, Axis2 in comparison to JBossWS always correctly responded with the same error message. Thus, we tried to find additional side-channels to mount the straightforward Bleichenbacher attack. By analyzing the Axis2 framework we found out that removing the `DataReference` elements from the C_{pub} part reveals a new side-channel: When the decrypted message was *not* PKCS#1 v1.5 conformant, the server responded with an obvious security error (`security processing failed`). In case of a PKCS#1 v1.5 conformant message the server correctly decrypted a session key. However, as there was no `DataReference` element, the server security module skipped the C_{sym} decryption and forwarded the document to further processing modules responding with *different* error messages. This way we were able to provoke new error responses leading to a direct application of Bleichenbacher's attack.

This interesting result shows that secure looking systems can reveal unexpected side-channels coming from the communication between different processing layers – in this case: XML layer processing XML Encryption structure and the underlying crypto layer processing PKCS#1.5. Interfaces communicating with the underlying libraries should be analyzed more deeply in order not to reveal details leading to cryptographic side-channels.

4.5.4 Countermeasures

In this section, we discuss several countermeasures against the described attacks and analyze their security.

4.5.4.1 XML Signature

XML Signatures can be used to secure authenticity and integrity of arbitrary document elements, including the elements that contain PKCS#1 v1.5 ciphertexts. In general, by using XML Signatures to counter these attacks, the same problems could appear as by thwarting the attacks from Section 4.1.1.3: An attacker could be registered as an honest client and issue valid XML Signatures (see Section 4.4.5.1), or he could apply XML Signature Wrapping attacks. In the following, we describe only the attacks on secret-key XML Signatures and XML Encryption Wrapping attacks as they work slightly differently.

Suppose a SOAP document includes one XML Signature securing a symmetric ciphertext C_{sym}. See Figure 4.14.

4.5.4.1.1 Secret-Key XML Signatures. The XML Signature specification allows to apply HMACs. HMACs use the symmetric key encrypted in C_{pub}. This binds the encryption and signature processing together. The server handling this message would first decrypt C_{pub}, then verify the signature over the ciphertext C_{sym}, and decrypt the encrypted block.

However, as the signature validation needs as input the symmetric key encapsulated in the PKCS#1 v1.5 structure for HMAC computation, the attacker would possibly get a new timing side-channel. The server processing would actually last longer if the HMAC value over C_{sym} has to be computed. This side-channel would have to be prevented by generating a random key as described later in Section 4.5.4.2, which could introduce new implementation problems.

4.5.4.1.2 XML Encryption Wrapping. An attacker being in possession of a signed SOAP message could also execute *XML Encryption Wrapping* attacks to turn a Web Service into an oracle. He proceeds as follows. He copies the original `EncryptedData` element $C_{sym-original}$ to the SOAP header and changes its `Id`. We call this element $C_{sym-oracle}$. Afterwards, he simply duplicates the original `EncryptedKey` element $C_{pub-original}$ containing the asymmetric ciphertext and makes it point to the newly created $C_{sym-oracle}$ (see Figure 4.23). Thereby, he forces the server to process both `EncryptedKey` elements and both `EncryptedData` elements. As $C_{pub-original}$ and $C_{sym-original}$ are left unchanged, the signature stays valid and the decryption of these elements produces no error. The attacker can perform the attack using $C_{pub-oracle}$ and $C_{sym-oracle}$.

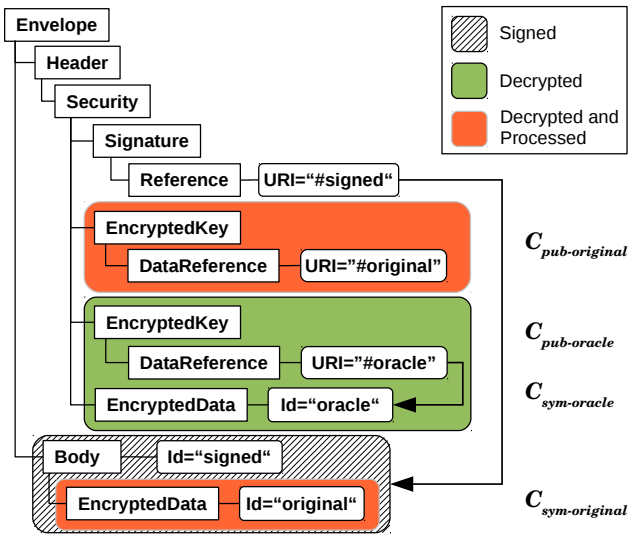


Figure 4.23: *XML Encryption Wrapping: The attacker copies the original encrypted payload to an unsigned document part ($C_{pub-oracle}$ and $C_{sym-oracle}$). He modifies this payload and uses the server as an oracle. The original payload ($C_{pub-original}$ and $C_{sym-original}$) is left unchanged.*

4.5.4.2 Generating Random Symmetric Keys

The classical countermeasure against Bleichenbacher's attack is to let the decryption algorithm return a random key k if C_{pub} is invalid, and then to proceed as if C_{pub} was valid [Res02, DR08].

A first obvious drawback of this countermeasure is that the system has to proceed with the random key *even if it knows that this key is invalid*. This may lead to data inconsistencies at the receiver side.

Even worse, it turns out that this countermeasure cannot prevent our CBC-based attack. Note that if C_{pub} is valid, then among all 256 initialization vectors chosen by the attacker there *must exist* at least one *iv* such that $C_{sym} = (\text{iv}, C^{(1)})$ returns no error. In particular, if the attacker submits a ciphertext C_{sym} that decrypts to well-formed XML repeatedly to the Web Service, then it will always respond that the ciphertext is valid. In contrast, if C_{pub} is invalid, and a random key k_0 is chosen by the Web Service for further processing, then even if the Web Service responds once that the tuple $C = (C_{pub}, C_{sym})$ is decrypted into well-formed XML for k_0, then the attacker can resubmit the same C to the Web Service. Again, another random key $k_1 \neq k_0$ will be chosen for further processing, and it is unlikely that the same C will decrypt to well-formed XML for k_0 and k_1 simultaneously. By repeating this procedure, the attacker can easily determine whether C_{pub} is valid with probability close to 1.

To summarize, this countermeasure effectively thwarts the presented timing-based attacks. However, we stress that it works against the presented CBC-based attacks *only* if the attacker has *no possibility* to distinguish valid from invalid ciphertexts C_{sym}. This could be e.g. ensured, if the application would process *signed* symmetric ciphertexts, see Section 4.4.5.2 for more details.

4.5.4.3 RSA-OAEP

The XML Encryption specification allows to choose between two different versions of the PKCS#1 encryption standard, namely Version 1.5 [Kal98] (which is vulnerable to Bleichenbacher's attack) and Version 2.0 [KS98], which is also known as RSA-OAEP [BR94]. In contrast to v1.5, it is known [FOPS04] that v2.0 (and its successor v2.1 [JK03]) meets the strong notion of *chosen-ciphertext security* (see e.g. [BDPR98]). In particular, Version 2.0 is not vulnerable to Bleichenbacher's attack.

A valid countermeasure to our attack might therefore be to use Version 2.0 instead of Version 1.5. This requires modifications on *both* the client and the server. We stress that the server must reject any PKCS#1 v1.5 ciphertext. Otherwise, an attacker could exploit the server's PKCS#1 v1.5 processing vulnerability to decrypt PKCS#1 v2.0 ciphertexts. We present these attacks in the next section in more detail.

We note also that Manger presented at Crypto 2001 [Man01] that there exists a variant of Bleichenbacher's attack that allows to break Version 2.0 as well. This however only works for implementations leaking a certain intermediate value during decryption. If a cryptography library correctly implements the decryption process, the developer can safely use this library in his implementation (e.g., in his XML Security library, JSON Security library, or SAML provider). There is no possibility for the attacker to decrypt PKCS#1 v2.0 ciphertexts as the attacks against PKCS#1 v2.0 are *implicitly thwarted on the cryptography library* level. On the other hand, if the developer supports PKCS#1 v1.5 in his implementation, he always needs to consider countermeasures against Bleichenbacher's attack, no matter how secure the applied cryptography library is. Attacks on PKCS#1 v1.5 *cannot be directly thwarted in a cryptography library*. Thus, they force the developers to apply countermeasures on higher levels (e.g., in XML Security or JSON Security libraries).

4.6 Backwards Compatibility Attacks

Complexity is often portrayed as being the enemy of security: The more complex a system is, the harder it is to analyze, and the harder it is to eliminate all possible attack vectors. One source of complexity in real world security systems stems from the desire to maintain backwards compatibility between new and old versions of systems.

It is obvious that introducing a new system whilst maintaining backwards compatibility with old versions having known weaknesses undermines security: If a system or a protocol *can* be configured into an insecure state, then some users *will* do so. In this section, we show something a little less obvious. Namely, that even if users have the best of intentions to use only the most up-to-date, vulnerability-free version of a system, the mere *existence* of support for old versions can have a catastrophic effect on security. We show this in the context of systems employing cryptography, introducing what we term *backwards compatibility (BC)* attacks.

As a taster of our attacks to follow, consider a situation where, for backwards compatibility reasons, a system still allows the use of CBC mode encryption, but where Galois Counter Mode (GCM) is the preferred secure encryption scheme. The reason to switch to GCM may be that the CBC mode is vulnerable to one of the several attacks presented in the previous sections. Now what happens if users select GCM as their preferred mode? Then an attacker who can modify messages so that they are decrypted using CBC mode instead of GCM can use the *old* attack to decrypt the ciphertexts as if they were CBC encrypted. Here we assume that the *same* key is used, irrespective of the mode. Then, this CBC decryption capability can be quickly and efficiently turned into a distinguishing attack against GCM. In a public key scenario, similar attacks can be used to break confidentiality of RSA-OAEP ciphertexts or to forge server signatures.

This situation is not purely hypothetical. As we described in the previous sections, our attacks on CBC and PKCS#1 v1.5 applied in XML Encryption influenced this specification. This has lead to a specification update and to a recommendation of GCM and RSA-OAEP encryption schemes. However, CBC and PKCS#1 v1.5 schemes were retained in the specification for backwards compatibility reasons. These legacy schemes typically apply the same keys as GCM and RSA-OAEP. Finally, a man-in-the-middle attacker can easily manipulate XML attributes so that an insecure mode for decryption is indicated. So all the pre-conditions for our attack are met.

Contributions. First, we demonstrate working BC attacks against the W3C XML Encryption [ERH+13] and XML Signature [ERS+08] specifications. In the secret key setting, we describe a practical BC attack that allows to break (i.e., to distinguish plaintexts of) GCM-based encryption in XML Encryption, based on a weakness of CBC. The basic idea of this generic attack is described in Section 4.6.2. Furthermore, in Section 4.6.5.2 we apply a significantly more efficient variant of this attack, which exploits specific weaknesses of XML Encryption, exemplarily to the widely-used Apache *Web Services Security for Java (WSS4J)* library. In the public-key setting, we show how the well-known attack of Bleichenbacher [Ble98] gives rise to a BC attack that allows an attacker to decrypt ciphertexts of PKCS#1 v2.0 encryption in both XML Encryption [ERH+12], and to forge signatures for arbitrary messages in XML Signature [ERS+08]. In addition to the XML Security specifications, we show that our attacks are applicable against the current draft of JSON Web Encryption [JRH12] and Web Signature [JBS12]

Second, we discuss countermeasures against the BC attacks. Obviously, usage of appropriate key separation ensuring that different keys are used in "weak" and "strong" cryptographic algorithms would be a correct countermeasure. However, this apparent simplicity is deceptive. Our experience is that developers sometimes fail to appreciate this requirement, or understand the requirement but fail to provide key separation. Moreover, in the context of public key cryptography, the most common data format for transporting public keys, the X.509 certificate, does not by default contain a field that limits the cryptographic algorithms in which a public key and its corresponding private key can be used. This lack of precision opens up the possibility of BC attacks in the public key setting.

Responsible Disclosure. We informed the W3C Working Group of the attacks presented in this section in July 2012. They acknowledged the attack and extended the specification with security considerations addressing BC attacks [ERH+13, Section 6.1.3]. We informed the JOSE Working Group, which is in charge of JSON Web Encryption and JSON Web Signature, of our BC attack on RSA-OAEP and PKCS#1 v1.5 in April 2012. Their standards are still under development at the time of writing.

We also communicated with several vendors applying XML Signature and XML Encryption. We highlight the steps they used to counter our attacks in Section 4.6.6.3. The attacks received the CVE identifier CVE-2012-5575.

Paper. This section is based on the paper *One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography* published at the Network and Distributed System Security Symposium (NDSS) [JPS13]. The paper was written together with Tibor Jager and Kenneth G. Paterson.

My contribution lay in the investigation of the BC attacks' application on XML Encryption and JSON Web Encryption. I designed and implemented the attacks, and contributed to the practical section. Tibor and Kenny analyzed attacks' theory and implications, and investigated general countermeasures against these attacks.

4.6.1 Related Work

Wagner and Schneier [WS96] described *version rollback attacks* on Version 2.0 of the SSL protocol. Speaking generally, version rollback attacks target cryptographic protocols where cryptographic algorithms and parameters are negotiated interactively between communication partners at the beginning of a protocol execution. The attacker modifies messages exchanged in this negotiation phase, in order to lure both communication partners into using weak cryptography, such as for instance legacy export-weakened algorithms.

Backwards compatibility attacks can be seen as a variant of version rollback attacks that apply to non-interactive protocols. An essential difference is that version rollback attacks on two-party protocols can be prevented by either party, if that party simply uses *exclusively* strong state-of-the-art cryptography.[12] In contrast, in this section we describe attacks that can not be prevented if one party is only prepared to use strong cryptography: The willingness of the other party to use weak cryptography suffices to foil security.

Kelsey et al. [KSW97] described *chosen-protocol attacks*. These consider a scenario where a victim executes a cryptographic protocol Π, and an attacker is able to trick this victim into executing an additional *maliciously designed* cryptographic protocol Π', too. This helps the attacker to break the security of Π. Clearly such attacks require a very strong attacker, and are only applicable if potential victims can be seduced into executing malicious protocols. In contrast, in typical backwards compatibility attacks, no adversarial control over the protocols executed by honest parties is needed.[13]

[12] In presence of an attacker the negotiation might then fail, which reduces the version rollback attack to a denial-of-service attack.

[13] Even worse, in the examples of BC attacks described in this section honest parties are *forced* to execute weak cryptographic algorithms, in order to remain standard compliant.

The attack described by Kaliski Jr. [Kal02] assumes an attacker that is able to register new *hash function identifiers*, and can thus be seen as a special case of chosen-protocol attacks.

Gligoroski et al. [GAK08] emphasize the need for key separation when using different modes of operation of a block cipher, and criticize some ISO and NIST standards for failing to make this point explicitly. However, they do not present any concrete attacks against deployed protocols.

Barkan et al. [BBK03] showed that the key separation principle is violated in the GSM mobile telecommunications system, and exploited this in what can be seen as a BC attack on the GSM encryption mechanism: In their attack, an active attacker fools the receiver into using a weak encryption algorithm (A5/2), extracts the key by cryptanalysis, and then uses the same key to decrypt traffic protected by the stronger A5/1 algorithm. Thus, the continued presence of a weak algorithm enables the enhanced security provided by a stronger algorithm to be bypassed. This is the only previous concrete example of a BC attack (of the specific type we explore in this section) that we know of.

A cryptographic primitive with the property that different instantiations can securely share the same key is called *agile* [ABBC10]. In a sense, the attacks presented in this section provide evidence that block cipher modes of operation and public-key schemes are not agile, and show how this property leads to relatively efficient practical attacks on important web standards. Another line of work, related to agility, concerns *joint security*, wherein a single asymmetric key pair is used for both signatures and encryption. An up-to-date overview of work in this area is provided in [PSST11].

4.6.2 Breaking GCM with a CBC Weakness

In this section, we describe a BC attack on symmetric encryption. We show how to break the expected security of ciphertexts encrypted in Galois Counter Mode (GCM) by exploiting a weakness of the cipher-block chaining (CBC) mode.

This attack provides just one concrete example of a BC attack. We decided to describe this particular case in detail because we will show the practical applicability of exactly this attack in Section 4.6.5.2. Similar attacks can applied on different modes of operations.

4.6.2.1 An abstract view on attacks on CBC

Starting with Vaudenay's padding-oracle attacks [Vau02], several efficient attacks exploiting the malleability of CBC-encrypted ciphertexts were published [DR11a, JS11, AP12] (see Section 4.2.1). These attacks are the main reason why CBC is phased out in new standards and replaced with modes of operation like GCM that provide security against chosen-ciphertext attacks.

Two properties that all these attacks have in common will be important for us: They allow to *decrypt* ciphertexts encrypted in CBC-mode, and they are *efficient*.

Thus, from an abstract point of view, the attacks provide an *efficient CBC decryption oracle* $\mathcal{O}_{\mathsf{CBC}}$. This oracle takes as input a CBC-encrypted ciphertext

$C = (iv, C^{(1)}, \ldots, C^{(n)})$ encrypting a message $(m^{(1)}, \ldots, m^{(n)})$, and returns

$$(m^{(1)}, \ldots, m^{(n)}) = \mathcal{O}_{\mathsf{CBC}}(C)$$

4.6.2.2 The Backwards Compatibility Attack

In this section, we describe a generic backwards compatibility attack on GCM, which is based on a weakness of CBC. We will first describe an abstract application scenario, which is practically motivated by the recent development of the XML Encryption specification. Then we describe the attack that allows an attacker to determine whether a ciphertext contains a certain message, and discuss the relevance of such distinguishing attacks. Finally, we sketch optimizations of the generic attack, which lead to significant efficiency improvements.

4.6.2.2.1 Application Scenario. In the sequel let us consider a scenario (an example application) in which encrypted messages are sent from senders $S_1, \ldots, S_\ell$ to a receiver R. Each ciphertext C received by R consists of two components $C = (C_{pub}, C_{sym}^{CBC})$, where

- C_{pub} is a public-key encryption of an ephemeral session key k under R's public-key, and
- C_{sym}^{CBC} encrypts the actual payload data under key k, using a block cipher in CBC-mode.

Suppose that $S_1, \ldots, S_\ell$ and R use this application, until it eventually turns out that it is susceptible to a chosen-ciphertext attack (CCA) which allows an attacker to decrypt ciphertexts in CBC-mode. For example, this may involve a padding oracle attack.

The application is immediately updated. The update replaces CBC-mode with GCM-mode, because GCM-mode provides provable CCA-security [MV04]. It is well-known that if the public-key encryption scheme used to encrypt the session key k is CCA-secure too,[14] then this combination forms a CCA-secure encryption scheme. Therefore, senders using this combination of algorithms may expect that their data is protected against chosen-ciphertext attacks.

After the update the receiver R remains *capable* of decrypting CBC-mode ciphertexts for backwards compatibility reasons, since it is infeasible to update the software of all senders $S_1, \ldots, S_\ell$ simultaneously. However, at least those senders that are using GCM instead of CBC may expect that their data is sufficiently protected.

We show that the latter is not true. The sole *capability* of R being able to decrypt CBC ciphertexts significantly undermines the security of GCM ciphertexts.

[14] For instance, RSA-OAEP [BR94], standardized in RSA-PKCS#1 v2.1 [JK03], is a widely used public-key encryption algorithm that provably meets this security property [FOPS04].

4.6.2.2.2 A Distinguishing Attack on GCM. We describe a *distinguishing attack*, which allows the attacker to test whether a GCM ciphertext contains a particular message. The attack exploits the CBC decryption capability of R. It can be applied block-wise to each ciphertext block, which enables the attacker to employ a "divide-and-conquer" strategy that in many scenarios is equivalent to a decryption attack. See Section 4.6.2.2.3 for further discussion of why distinguishing attacks matter.

The attack consists of two key ingredients.

1. We show that the availability of the CBC decryption attack allows the attacker not only to decrypt arbitrary ciphertexts in CBC-mode, but also to invert the block cipher used within CBC at arbitrary positions. That is, we show that a CBC decryption oracle implies a block cipher decryption oracle.

2. We show that this block cipher decryption oracle can be used to mount a distinguishing attack on GCM.

CBC-Decryption implies Block Cipher Inversion. Due to the availability of the CBC decryption attack, R involuntarily provides an efficient CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$, which takes as input a tuple $C = (C_{pub}, C_{sym}^{CBC})$, and returns the decryption of C_{sym}^{CBC} under the key k contained in C_{pub}.

We show that this oracle $\mathcal{O}_{\mathsf{CBC}}$ can be turned into a new oracle $\mathcal{O}_{\mathsf{Dec}}$ that inverts the block cipher used in CBC-mode. Oracle $\mathcal{O}_{\mathsf{Dec}}$ takes as input a tuple $C = (C_{pub}, C')$, and returns the block cipher decryption $m' = \mathsf{Dec}(k, C')$ of C' under the key k contained in C_{pub}.

Oracle $\mathcal{O}_{\mathsf{Dec}}$ proceeds on input (C_{pub}, C') as follows.

1. It chooses an arbitrary initialization vector iv'.

2. It queries the CBC decryption oracle on input

$$(C_{pub}, (iv', C')).$$

   Note that (iv', C') is a valid CBC ciphertext consisting of an initialization vector iv and a single ciphertext block C'. Therefore, oracle $\mathcal{O}_{\mathsf{CBC}}$ will return the CBC decryption

$$m = \mathsf{Dec}(k, C') \oplus iv$$

   of (iv', C').

3. Finally, $\mathcal{O}_{\mathsf{Dec}}$ computes and outputs $m' = m \oplus iv'$.

It is straightforward to verify that $m' = \mathsf{Dec}(k, C')$.

Distinguishing GCM Ciphertexts. Consider an attacker who eavesdrops an encrypted message $C = (C_{pub}, C_{sym}^{GCM})$ sent from a sender S to receiver R. Ciphertext C_{pub} encrypts a key k, and $C_{sym}^{GCM} = (iv, C^{(1)}, \ldots, C^{(n)}, \tau)$ encrypts a message $m = (m^{(1)}, \ldots, m^{(n)})$ in GCM-mode with key k.

Assume the attacker has access to an oracle $\mathcal{O}_{\mathsf{Dec}}$ which takes as input a tuple $C = (C_{pub}, C')$ where C' is a single ciphertext block, and returns the block cipher decryption of C' under the key k contained in C_{pub}.

The attacker can use this oracle to test whether the i-th encrypted message block $m^{(i)}$ contained in the eavesdropped ciphertext block $C^{(i)}$ is equal to a certain message m'. It proceeds as follows.

1. The attacker queries oracle $\mathcal{O}_{\mathsf{Dec}}$ by submitting the ciphertext
$$\tilde{C} := (C_{pub}, C^{(i)} \oplus m').$$
2. If the decryption oracle $\mathcal{O}_{\mathsf{Dec}}$ responds with
$$\mathcal{O}_{\mathsf{Dec}}(\tilde{C}) = iv||0^{31}||1 + i, \tag{4.5}$$
then the attacker concludes that $m' = m^{(i)}$.

To see that this indeed allows the attacker to determine whether $C^{(i)}$ encrypts m', note that in GCM-mode
$$\mathsf{Dec}(k, C^{(i)} \oplus m^{(i)}) = iv||0^{31}||1 + i$$
holds if and only if
$$C^{(i)} = \mathsf{Enc}(k, iv||0^{31}||1 + i) \oplus m^{(i)}.$$
Because $(\mathsf{Enc}, \mathsf{Dec})$ is a block cipher, $\mathsf{Enc}(k, \cdot)$ is a permutation, and $\mathsf{Dec}(k, \cdot) = \mathsf{Enc}^{-1}(k, \cdot)$ is its inverse. Thus, if Equation (4.5) holds, then it must hold that $m^{(i)} = m'$.

4.6.2.2.3 Why Distinguishing Attacks Matter. Practitioners are prone to dismissing distinguishing attacks as being only of theoretical interest. However, we caution against this viewpoint, for two reasons. Firstly, such attacks are readily converted into plaintext recovery attacks when the plaintext is known to be of low entropy. We will demonstrate this in practice in Section 4.6.5.2. Secondly, such attacks are indicative of problems that tend to become more severe with time. The recent example of TLS1.0 provides a good example of this phenomenon: As early as 1995, Rogaway [Rog95] pointed out that CBC encryption is vulnerable to a chosen plaintext distinguishing attack when the initialization vectors used are predictable to the attacker. This vulnerability was addressed in TLS1.1, but TLS1.0 support remained widespread. Then in 2011, the Duong and Rizzo BEAST attack [DR11b] showed how to extend Rogaway's original observation to produce a full plaintext recovery attack. Their attack applies to certain applications of TLS in which there is some adversarially-controllable flexibility in the position of unknown plaintext bytes. The resulting scramble to update implementations to avoid the Rogaway/BEAST attack could easily have been avoided had the distinguishing attack been given more credence in the first place.

4.6.2.2.4 Optimizations. We based our description of the GCM distinguishing attack on the availability of an abstract CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$. This oracle can be provided *somehow*, that is, by an arbitrary attack on CBC-mode encryption. The distinguishing attack uses the $\mathcal{O}_{\mathsf{CBC}}$ oracle naively as a black-box, without taking into account, which specific weaknesses of CBC-encryption and the target application are exploited to implement $\mathcal{O}_{\mathsf{CBC}}$. While on the positive side this implies that the GCM distinguishing attack works in combination with *any* CBC decryption attack, we also note that an attack making naive usage of the $\mathcal{O}_{\mathsf{CBC}}$ oracle is potentially not optimally efficient.

For instance, in practice the CBC decryption oracle is usually given by a padding oracle attack. A typical padding oracle attack requires on average between 14 [JS11] and 128 [Vau02, DR11a] chosen-ciphertext queries to recover one plaintext byte. If the CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$ is used naively as a black-box, without further consideration of which particular attack is performed by $\mathcal{O}_{\mathsf{CBC}}$, then this complexity is inherited by the attack on GCM. Thus, in order to test whether a particular GCM-encrypted ciphertext block $C^{(i)}$ contains a particular message m' (in case of a 16-byte block cipher like AES [AES01]), one expects that between $14 \cdot 16 = 224$ and $128 \cdot 16 = 2048$ chosen-ciphertext queries are required to perform one test.

We note that the GCM distinguishing attack can be improved dramatically by exploiting specific properties of the provided CBC padding oracle and the application. Jumping a bit ahead, our implementation of the GCM distinguishing attack (as described in Section 4.6.5.2) uses an optimized version of the naive attack. This optimized attack takes into account specific details of the target application, like formatting of valid plaintexts and padding, which allows for much more efficient attacks. For the optimized attacks on GCM in XML Encryption and JOSE detailed in Section 4.6.5.2, only 2 queries are already sufficient to mount our distinguishing attack.

4.6.3 Further BC Attacks on Symmetric Cryptography and Generic Countermeasures

The principle of backwards compatibility attacks on symmetric encryption schemes is of course not limited to CBC and GCM. We described this special case in the previous section as a first example because it represents a reasonable practical scenario, which nicely matches the practical attacks described in Section 4.6.5.2. In this section, we discuss further BC attacks on symmetric encryption schemes and generic countermeasures.

4.6.3.1 BC Attacks on Other Modes of Operation

There exists a large number of block cipher modes of operation defined by various organizations in various standards. For instance, popular unauthenticated modes of operation are ECB, CBC, OFB, and CTR [NIS80, NIS01b]. Widely used authenticated modes of operation are OCB [RBBK01], EAX [BRW04], and CCM [NIS04].

For any authenticated mode of operation, one can select a suitable unauthenticated mode of operation and describe a backwards compatibility attack which allows an attacker to distinguish encrypted messages, or even to decrypt high-entropy ciphertexts. Since of course most combinations of modes of operation and attack scenarios are not of practical relevance, and the additional theoretical contribution over the attack from Section 4.6.2.2.2 is limited because the attack principle is always the same, we do not describe all possible attacks in detail.

We note only that different modes of operation have very different properties and characteristics w.r.t. backwards compatibility attacks. For example:

1. Some modes use the encryption algorithm $\mathsf{Enc}(k, \cdot)$ of the block cipher for encryption, and the decryption algorithm $\mathsf{Dec}(k, \cdot)$ for decryption. Examples for such modes are ECB and CBC.

2. Some modes use the encryption algorithm $\mathsf{Enc}(k, \cdot)$ of the block cipher for *both* encryption and decryption. Examples of this type are OFB and "counter"-modes, like CTR and GCM, where the block cipher is turned into a stream cipher by encrypting an incrementing counter value.

The type of oracle provided by an attack on a mode of operation depends strongly on such characteristics. For instance, a CBC decryption attack provides a block cipher *decryption* oracle that allows an attacker to compute the block cipher decryption function $\mathsf{Dec}(k, \cdot)$. In contrast, a decryption attack on OFB mode would provide a block cipher *encryption* oracle $\mathsf{Enc}(k, \cdot)$.

In Section 4.6.2.2.2 we showed that the block cipher decryption oracle $\mathsf{Dec}(k, \cdot)$ provided by the attack on CBC is sufficient to mount a distinguishing attack on GCM. In turn, this allows the decryption of low-entropy ciphertexts by exhaustive search over all possible plaintexts. If instead an *encryption* oracle was given, then this would even allow the decryption of high-entropy GCM ciphertexts, since this oracle essentially computes the block cipher operation performed in the GCM-decryption algorithm.

In a different application scenario, with a different combination of algorithms, a block cipher decryption oracle may also lead to a full-fledged decryption attack. For example, AES Key Wrap [NIS01a] is a NIST-specified symmetric key transport mechanism designed to encapsulate cryptographic keys. AES Key Wrap is used, for instance, in XML Encryption. Indeed, the block cipher decryption oracle provided by attacks from Section 4.4 allows to *decrypt* even high-entropy keys encrypted with the AES Key Wrap scheme.

4.6.3.2 Generic Countermeasures

There are a number of obvious countermeasures which would prevent our symmetric BC attacks. The cleanest approach is to fully embrace the principle of *key separation*, which dictates that different keys should be used for different purposes. Extending this principle would mean using completely different keys for different algorithms serving the same purpose. Of course, the required keys

may not be readily available, and making them available might require significant re-engineering of other system components. This approach does not sit well with maintaining backwards compatibility.

A compromise position would be to take the existing key and ensure that distinct, algorithm-specific keys are derived from it using suitable algorithm identifiers. For example, we could set $k^{prime} = \mathsf{PRF}(k, \text{"Algorithm Identifier"})$ where now the original key k is used as a key to a pseudo-random function supporting key derivation. Suitable pseudorandom functions can be implemented based on block ciphers or hash functions, which are readily available in most cryptographic libraries.

4.6.4 BC Attacks on Public-Key Cryptography

In this section, we recall the well-known attack of Bleichenbacher [Ble98] on RSA-PKCS#1 v1.5 encryption [Kal98]. We discuss its applicability to RSA-OAEP encryption [BR94] (as standardized in RSA-PKCS#1 v2.0 [KS98] and v2.1 [JK03]) and to RSA-PKCS#1 v1.5 signatures [JK03].

Essentially, Bleichenbacher's attack allows to invert the RSA-function $m \mapsto m^e \bmod N$ without knowing the factorization of N. This fact gives rise to obvious attacks on RSA-based encryption and signature schemes. Therefore, the fact that Bleichenbacher's attack may in certain applications give rise to backwards compatibility attacks is not very surprising. We stress that we consider the contribution of this part therefore not in demonstrating this relatively obvious fact, but rather in showing that such attacks are indeed applicable in practice.

4.6.4.1 The Power of Bleichenbacher's Attack

As already noted in [Ble98], the attack of Bleichenbacher allows not only to decrypt PKCS#1 v1.5 ciphertexts. Instead, it uses the PKCS#1 validity oracle to invert the RSA function $m \mapsto m^e \bmod N$ on an *arbitrary* value (not necessarily a PKCS#1 v1.5 conformant ciphertext).

Therefore, Bleichenbacher's attack can potentially also be used to decrypt RSA-OAEP ciphertexts, or to forge RSA-based signatures, whenever the following two requirements are met.

1. The PKCS#1 v1.5 encryption scheme and the attacked cryptosystem (like RSA-OAEP encryption or RSA-signatures) use the same RSA-key (N, e).

2. A PKCS#1 v1.5-validity oracle is given, in order to mount Bleichenbacher's attack.

We will show that these two requirements are indeed met in certain practical applications, where PKCS#1 v1.5 encryption is available due to backwards compatibility reasons.

4.6.4.2 Attacking RSA-OAEP

Note that in order to decrypt an OAEP-ciphertext it suffices to be able to invert the RSA encryption function $m \mapsto m^e \bmod N$, since the message encoding

and decoding steps are unkeyed. Thus, if the RSA public key (N, e) is used for OAEP-encryption and an oracle $\mathcal{O}$ is available which tells whether a given ciphertext is PKCS#1 v1.5 conformant w.r.t. (N, e), then one can use this oracle to decrypt OAEP-ciphertexts by mounting Bleichenbacher's attack.

4.6.4.3 Attacking RSA-PKCS#1 v1.5 Signatures

In order to forge an RSA-PKCS#1 v1.5 signature it suffices to be able to invert the RSA encryption function. Thus, if the RSA public key (N, e) is used for RSA-PKCS#1 v1.5 signatures and an oracle $\mathcal{O}$ is available that tells whether a given ciphertext is PKCS#1 v1.5 conformant w.r.t. (N, e), then one can use this oracle to forge RSA-PKCS#1 v1.5 signatures by mounting Bleichenbacher's attack on a suitably randomized version of the encoded message M.

This attack possibility is mentioned in Bleichenbacher's original paper [Ble98]. A variant of the attack was recently explored in [DLP+12] in the context of EMV signatures (where the same RSA key pair may be used for both signature and encryption functions).

4.6.4.4 Countermeasures and the Difficulty of Key Separation with X.509 Certificates

Key separation means to use different (independent) keys for different algorithms. In theory this principle provides a simple solution to prevent backwards compatibility attacks. As described in Section 4.6.3.2, key separation is very easy to enforce in the symmetric setting, for instance by a suitable application of a pseudorandom function before using the symmetric key.

In principle, key separation in the public-key setting is almost as easy to enforce as in the symmetric setting. One could simply generate different keys for different purposes. For instance, one RSA-key (N_0, e_0) is generated exclusively for PKCS#1 v1.5 encryption, another independent RSA-key (N_1, e_1) exclusively for PKCS#1 v1.5 signature, and yet another independent RSA-key (N_2, e_2) only for RSA-OAEP encryption. Each public-key should then be published together with some information (included in the certificate, for instance) that specifies for which algorithm this key can be used. Accordingly, each secret key should be stored together with this additional information. Cryptographic implementations should check whether the provided key is suitable for the executed algorithm.

Unfortunately this theoretically sound solution is not easy to implement in practice. This is because common data formats for public keys do not provide this additional information as part of the basic standard. For example, the X.509 standard for public-key certificates defines a popular data format for public keys. While an X.509 certificate does include algorithm identifiers for the signing algorithm used to create the certificate itself, these certificates do not necessarily include any information about with which algorithms the certified public key can be used. It is possible to extend X.509 certificates with such a field, the Subject Public Key Info field (see RFC 5280 [CSF+08] and more specifically RFC 4055 [SKH05] for naming conventions for RSA-based algorithms), but supporting this

field is not mandatory and would require major changes to implementations and libraries. In view of BC attacks, we consider this to be a big handicap of X.509 certificates. We suggest that algorithm identifiers for certified keys be included by default in future revisions of X.509.

The importance of key separation still seems to be not very well understood in practice. For instance, a large cloud identity security provider even *suggested* the use of RSA keys for both digital signatures and encryption [Pin12], while RFC 4055 [SKH05] permits the same RSA key pair to be used for more than one purpose (see specifically Section 1.2 of RFC 4055). There is limited theoretical support for this kind of key reuse (see [PSST11] and the references therein), but in general, as our attacks show, it is a dangerous practice.

4.6.5 Attacking XML Encryption and JSON Web Encryption

In this section, we demonstrate the vulnerability of current versions of XML Encryption [ERI+02] and JSON Web Encryption [JRH12] to BC attacks. We describe optimized versions of the BC attacks illustrated in previous sections. Then, we discuss practical countermeasures, their applicability, and propose changes to the algorithms and security considerations in the analyzed specifications.

4.6.5.1 Platforms for our Experimental Analyses

As described in the previous sections, for execution of the backwards compatibility attacks two prerequisites have to be given. First, the server implementation has to support both secure and insecure algorithms using the same key. Second, the attacker has to be able to lure the server into processing the ciphertexts with a different algorithm. XML Encryption supports RSA-PKCS#1 v1.5 and AES-CBC, and JWE supports RSA-PKCS#1 v1.5. In order to force the server to process a ciphertext with a desired algorithm, the client specifies the algorithm directly in the message (he changes the `Algorithm` attribute in `EncryptionMethod` in XML Encryption message or the `alg` attribute in the JWE header, see Figures 2.17 and 4.8). This makes XML Encryption and JWE suitable for application of our attacks.

We analyze the practicality and performance of our attacks on XML Encryption and XML Signature by applying them to the Apache Web Services Security for Java (Apache WSS4J) library. This is a widely used library providing Web Services frameworks with implementations of XML Encryption and XML Signature. It is used in several major Web Services frameworks, including JBossWS [JBo13], Apache CXF [Apa13b], and Apache Axis2 [Apa13a].

The practicality and performance of our attacks on JWE and JWS are investigated by applying these attacks to the Nimbus-JWT library [Nim13]. Nimbus-JWT is a Java implementation of JSON Web Encryption (JWE) and JSON Web Signature (JWS), developed by NimbusDS to support their Cloud Identity management portfolio.[15]

[15] Even though Nimbus-JWT claims to implement version 02 of the JWE standard draft, at the time of writing it still supported usage of AES-CBC (without MAC), which was

Note that we test our attacks in this section at the library level, not against actual applications. It may therefore be possible that applications implement specific countermeasures to prevent these attacks. However, we stress that preventing most attacks on higher application layers is extremely difficult or even impossible, as we describe later in this section.

4.6.5.2 Breaking AES-GCM

In this section, we describe our practical attacks breaking indistinguishability of the AES-GCM ciphertexts in XML Encryption. We first describe a performant variant of the attack from Section 4.6.2. Then we present the results of our experimental evaluation, executed against Apache WSS4J and against the Nimbus-JWT library.

4.6.5.2.1 Plaintext Validity Checking. A symmetric XML Encryption ciphertext is processed as follows. It is first decrypted with a decryption key. Then the padding is removed, and the decrypted plaintext is parsed as XML data. If any of these steps fails, the process returns a processing failure.

Padding and parsing mechanisms in XML Encryption were already described in Sections 4.1.1.3 and 4.4.3. In the following we describe how these mechanisms influence our BC attacks.

In the sequel let us assume that XML Encryption is used with a block cipher of block size $\nu = 16$ byte, like AES, for instance.

Padding in XML Encryption. For the description of our attacks, it is important to rephrase that the last byte in the plaintext indicates the padding byte. When using a block cipher of block size $\nu = 16$, there exist 16 valid padding byte values.

Based on this fact, observe that a randomly generated plaintext block is valid according to the XML Encryption padding scheme with a probability of $P_{pad} = 16/256$ (if a 16-byte block cipher is used, as we assume), since there are 16 possible values for the last byte that yield a valid padding.

XML Parsing. Valid XML plaintexts have to consist of valid characters and have a valid XML structure. Parsing XML data that are not well-formed or contain invalid characters leads to parsing errors (see Section 4.4.3 for more details). For simplicity, let us assume in the following that an XML plaintext consists only of ASCII characters. The ASCII code allows to encode $2^7 = 128$ different characters.

The ASCII table contains two sets of characters: parsable and non-parsable. Parsable characters include letters, numbers, or punctuation marks. About a 25% of ASCII characters are non-parsable. This includes, for example, the `NUL`, `ESC`, and `BEL` characters. If any of these is contained in an XML document, then this will lead to a parsing error.

available in version 01, but not in version 02 or any subsequent versions. This made the Nimbus-JWT library suitable for testing our symmetric key scenarios.

Thus, P_{parse}, the probability that a random byte corresponds to a parsable character, is about $1/2 \cdot 3/4 = 3/8$.

Probability of valid XML ciphertexts. The fact that an XML processor responds with an error message if the padding or the plaintext format of a decrypted message is invalid allows us to determine whether a given CBC-encrypted ciphertext is valid or not. This allows us to construct an XML decryption validity oracle $\mathcal{O}_{\mathsf{CBCxml}}$, which takes as input an AES-CBC ciphertext $\tilde{c} = (\tilde{iv}, \tilde{C}^{(1)})$, decrypts it, and responds with 1 if the plaintext is correct, and 0 otherwise.

In particular, a randomly generated ciphertext $(\tilde{iv}, \tilde{C}^{(1)})$ consisting of an initialization vector and one ciphertext block leads to a decryption error with high probability. The probability that a random ciphertext is valid is only

$$P_{valid} = \sum_{i=0}^{15} (1/256)(3/8)^i \approx 0.0062$$

This low probability that a random ciphertext is valid is one of the key ingredients to our attack.

Plaintext Validity Checking in JWE. The JWE standard applies a different padding scheme, namely PKCS#5. This padding scheme has a more restrictive padding validity check, such that random ciphertexts are rejected with even higher probability. This improves the success probability of our attack. In the context of JWE we thus obtain a plaintext validity oracle $\mathcal{O}_{\mathsf{CBCjwe}}$, which is similar to $\mathcal{O}_{\mathsf{CBCxml}}$ but has an even smaller false positive rate.

4.6.5.2.2 Optimized Algorithm

Distinguishing Plaintexts. Let us now describe our optimized attack. Consider an attacker who eavesdrops an AES-GCM ciphertext

$$C = (iv, C^{(1)}, \ldots, C^{(n)}, \tau).$$

His goal is to determine whether the i-th ciphertext block $C^{(i)}$ encrypts a particular message m'. The attacker proceeds as follows (see Figure 4.24):

1. He computes a CBC ciphertext by setting the first ciphertext block equal to $\tilde{C} = m' \oplus C^{(i)}$. (If he guessed m' correctly, then this sets $\mathsf{Dec}(k, \tilde{C}) = cnt = iv||0^{31}||1 + i$.)

2. He selects a valid XML plaintext $\tilde{m}$ and a CBC-mode initialization vector $\tilde{iv}$, such that
$$\tilde{m} = \tilde{iv} \oplus cnt$$

3. The ciphertext $(\tilde{iv}, \tilde{C})$ is then sent to the CBC validity checking oracle.

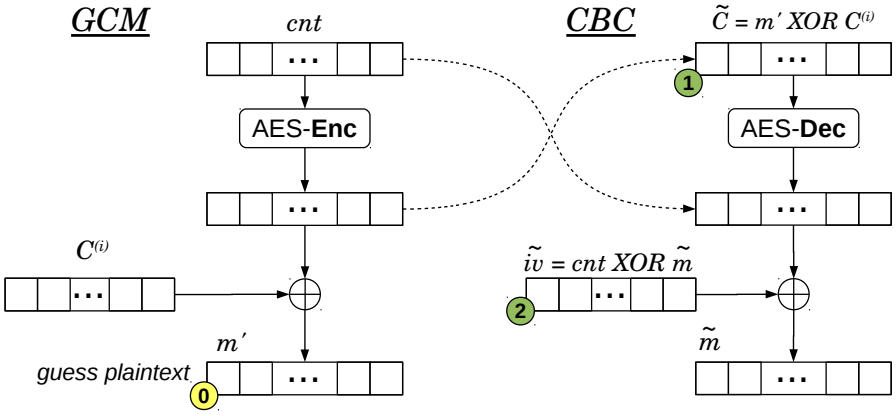


Figure 4.24: *Breaking indistinguishability of AES-GCM with AES-CBC.*

If the CBC validity checking oracle accepts this as a valid ciphertext, then the attacker most likely guessed m' correctly (with a probability of $P_{m'} = 1 - P_{valid} \approx 0.9938$). Otherwise, he certainly guessed wrongly. This test can be repeated a few times with distinct values of $\tilde{m}$ to decrease the probability of a false positive.

Recovering Plaintext Bytes. The distinguishing attack can also be used to decrypt low-entropy plaintexts. For our experiments, we consider an attacker that *a priori* knows the complete plaintext except for one plaintext byte. We also assume that the attacker reduces the number of false positives by one additional oracle query for each positive response.

The attack procedure for recovering one plaintext byte is depicted in Algorithm 4.6. The algorithm iterates over all the $n = 256$ possibilities for byte b. The performance of this step can be improved significantly if the attacker is able to narrow the number of possible values for b, for instance if the target application accepts only ASCII characters, only letters, only integers, etc.

The algorithm can easily be extended to decrypt larger numbers of unknown bytes in one block. To decrypt x unknown bytes, the attacker needs to issue about n^x oracle queries.

4.6.5.2.3 Evaluation. We evaluated performance of our attacks against both WSS4J and Nimbus-JWT. We first used the libraries to generate valid messages containing AES-GCM ciphertexts. Then we modified the algorithm parameters in the messages, forcing the receiver to process the ciphertexts using AES-CBC, and executed the attack described in Algorithm 4.6. The required ciphertext validity oracles were based on error messages generated by the libraries.

Table 4.2 reports the results of our evaluation, with figures obtained by averaging over 50 executions. We include results for ciphertext blocks containing 1, 2, and 3 unknown bytes. We restricted the possible character set to a group of alphabetic and numeric characters. Thus, in this setting the attacker needs to test $n = 64$ possibilities for each byte.

Algorithm 4.6 Recovering a single plaintext byte b from an AES-GCM ciphertext using an $\mathcal{O}_{\mathsf{CBCxml}}$ oracle.

Input: Plaintext block m' containing one unknown byte b. Position p of the unknown byte b. AES-GCM ith ciphertext block $C^{(i)}$ and initialization vector iv.

Output: Plaintext byte b.

1: $\tilde{m}_{valid1} := \texttt{0x00}||\texttt{0x00}||\ldots||\texttt{0x00}||\texttt{0x10}$
2: $\tilde{m}_{valid2} := \texttt{0x40}||\texttt{0x40}||\ldots||\texttt{0x40}||\texttt{0x01}$
3: $cnt := iv||0^{31}||1+i$
4: $n := 256$
5: **for** $b = 0 \rightarrow (n-1)$ **do**
6: $\quad m'_p := b$
7: $\quad \tilde{C} := m' \oplus C^{(i)}$
8: $\quad \tilde{iv} := cnt \oplus m_{valid1}$
9: $\quad$ **if** $\mathcal{O}_{\mathsf{CBCxml}}(\tilde{iv}, \tilde{C}) = 1$ **then**
10: $\quad\quad \tilde{iv} := cnt \oplus m_{valid2}$
11: $\quad\quad$ **if** $\mathcal{O}_{\mathsf{CBCxml}}(\tilde{iv}, \tilde{C}) = 1$ **then**
12: $\quad\quad\quad$ **return** b
13: $\quad\quad$ **end if**
14: $\quad$ **end if**
15: **end for**

Number of unknown bytes	Guessed m' plaintexts	$\mathcal{O}_{\mathsf{CBCxml}}$ queries	$\mathcal{O}_{\mathsf{CBCjwe}}$ queries
1	36	37	37
2	2,130	2,145	2,139
3	142,855	143,811	143,409

Table 4.2: *Attack results applied on ciphertext blocks containing* 1, 2, *and* 3 *unknown bytes from a group of alphabetic and numeric characters.*

As expected, the attack performs well if the target ciphertext blocks contain a large number of known plaintext bytes. The number of oracle queries needed increases exponentially with the number of unknown plaintext bytes.

While the number of guessed m' plaintexts is constant for both libraries, the number of total oracle queries varies. The different numbers of queries is caused by different plaintext validation models being used in the XML Encryption and JWE specifications: The validation model in JWE employs a stricter verification for the padding, which results in less oracle queries being needed by the attacker.

Extension to a Full Plaintext Recovery Attack. Our evaluation shows that an attacker is able to efficiently decrypt ciphertexts with a large number of known bytes in the plaintext. We note that an attacker who is able to control parts of the plaintext is also able to recover high-entropy plaintexts, by employing the technique from Duong and Rizzo's BEAST attack [DR11b].

Let us sketch the basic idea of this technique. The attacker uses its control over the plaintext to prepend the unknown high-entropy plaintext with $n-1$ known bytes, where n is the block size of the block cipher in bytes. Thus, only the last byte of the first block is unknown to the attacker, and can be recovered relatively efficiently with the above procedure. In the next step, the attacker prepends the high-entropy plaintext with $n-2$ known bytes. Since the first byte of the plaintext is already recovered, there is again only one unknown byte in the resulting plaintext. By executing Algorithm 4.6 repeatedly with this divide-and-conquer strategy, the attacker is able to recover the full high-entropy plaintext.

4.6.5.3 Practical Examples of BC Attacks on Public-Key Cryptography

As described in Sections 2.5 and 2.10.1, both XML Encryption and JWE specify public-key encryption according to PKCS#1 v1.5 and v2.0 as being mandatory. Similarly, both XML Signature and JWS specify PKCS#1 v1.5 signatures as being mandatory.

Recall from Section 4.6.4.1 that the known attack of Bleichenbacher on PKCS#1 v1.5 can be used to decrypt PKCS#1 v2.0 ciphertexts or to forge RSA-signatures if two requirements are met:

1. The application allows the RSA public-key (N, e) used for PKCS#1 v2.0 encryption (or RSA-signatures) to be also used for PKCS#1 v1.5 encryption; and

2. the application provides a PKCS#1 v1.5 validity oracle.

In Section 4.5 we observed that both XML Encryption and JWE inherently provide PKCS#1 v1.5 validity oracles. Thus, Property 2 is met by XML Encryption and JWE. It remains to show that Property 1 is also met. Indeed, neither specification distinguishes between keys for PKCS#1 v2.0 encryption, PKCS#1 v1.5 encryption, and PKCS#1 v1.5 signatures (as noted before, some providers even recommend re-use of RSA-keys across different algorithms).

Let (N, e) be the RSA public key of a receiver. A ciphertext according to PKCS#1 (regardless of v1.5 or v2.0), consists of a single integer y modulo N. Thus, in order to apply the correct decryption algorithm to y, the receiver needs additional information, namely the version (v1.5 or v2.0) of PKCS#1 according to which the ciphertext C was encrypted by the sender. In both XML Encryption and JWE, this information is provided in metadata[16] accompanying the ciphertext. These metadata are (typically) not integrity-protected. Thus, an attacker can change them arbitrarily.

This enables an attacker to trick the receiver into applying the PKCS#1 v1.5 decryption algorithm to an arbitrary value y modulo N. In combination with the PKCS#1 v1.5 validity oracle from Section 4.5 and Bleichenbacher's attack [Ble98], this suffices to invert the RSA-function $m \mapsto m^e \bmod N$ on an arbitrary value y. This in turn allows to decrypt PKCS#1 v2.0 ciphertexts or to forge RSA-signatures with respect to key (N, e), as explained in Section 4.6.4.1.

[16] The `EncryptedKey` element in XML Encryption, the *header segment* in JWE.

4.6.5.3.1 Experimental Results. In order to assess the practicability and performance of the attack, we implemented Bleichenbacher's attack and applied it to the Nimbus-JWT library. The PKCS#1 v1.5 validity oracle was provided by different error messages returned by this library.[17]

The experiment was repeated 10,000 times, each time with a fresh 1024-bit RSA-key, which was generated using the standard Java key pair generator.[18] Decrypting a random PKCS#1 v2.0 ciphertext took about 171,000 oracle queries on average. Forging a JSON Web Signature for an arbitrary message took about 218,000 queries on average. See Table 4.3 for details.

	Mean	Median	Maximum # of queries	Minimum # of queries
PKCS#1 v2.0 Ciphertext	171,228	59,236	142,344,067	4,089
PKCS#1 v1.5 Signature	218,305	66,984	395,671,626	20,511

Table 4.3: *Experimental results of BC attacks on PKCS#1 v2.0 ciphertexts and PKCS#1 v1.5 signatures.*

Executing the attacks with 2048 and 4096-bit RSA-keys resulted in only a slightly higher number of requests.

For our evaluation purposes, we used the original attack algorithm. Bardou et al. [BFK+12] described significantly improved variants of Bleichenbacher's attack. We did not implement these optimizations yet, but since the improvements in [BFK+12] are very general, we expect that they lead to much more efficient BC attacks, too.

4.6.6 Countermeasures

In this section, we discuss why several seemingly simple countermeasures (see Sections 4.6.3.2 and 4.6.4.4) are hard to employ in practice.

4.6.6.1 Disallowing Legacy Algorithms

An obvious countermeasure would be to disallow all legacy algorithms and to use only state-of-the-art cryptosystems. Unfortunately, this countermeasure would also destroy interoperability for all parties that are only capable of running older algorithms. This is a real issue: For example, the attack on XML Encryption from Section 4.4 showed the insecurity of CBC in XML Encryption. Therefore, GCM is now available as an additional option in the specification. Even though the attack was published almost one year ago (and was disclosed to vendors and developers several months earlier), users applying important Web

[17] In practice one could instead use the more elaborate attack techniques from Section 4.5 to determine whether a given ciphertext is PKCS#1 v1.5 valid.

[18] `java.security.KeyPairGenerator`.

Services frameworks like Apache Axis2 [Apa13a] or SAML-based Single Sign-On [CKPM05] frameworks like Shibboleth [Shi13] cannot adapt GCM as the platforms these frameworks are running on do not support GCM.

In the case of XML Encryption and Web Services one may also apply WS-Security Policy [LK07]. This specification allows the definition of policies forcing usage of specific algorithms in client-server communication. However, it is still questionable how strictly these policy restrictions are implemented. We present some details about the implementation of this specification in Apache CXF in Section 4.6.6.3.

4.6.6.2 Key Separation

4.6.6.2.1 Symmetric Algorithms. The *key separation* countermeasure proposed in Section 4.6.3.2 is simple and effective. As the JWE standard is still in a draft version, we strongly recommend to consider application of this principle in the final version of JWE.

4.6.6.2.2 Asymmetric Algorithms. The key separation principle can also prevent BC attacks on public-key schemes like PKCS#1 v2.0. Unfortunately, it seems that the importance of this principle is not well-understood in practice. For instance, the WS-Security Policy standard [LK07] explicitly mentions in Section 7.5 that it is possible to use the same RSA key pair for encryption and signature processing. Moreover, some providers *suggest* their users to use the same RSA key pair for different cryptographic algorithms [Fus12, Pin12].

4.6.6.3 Communication with Developers

We discussed our attacks with developers of several frameworks. In this section, we summarize some approaches that developers followed to counter them.

The most recent draft of XML Encryption which includes AES-GCM is not widely adopted yet (note that the first public draft version dates to March 2012). The only framework we are aware of that currently supports this version is Apache CXF [Apa13b]. It utilizes the tested Apache WSS4J library [Apa12b].

4.6.6.3.1 Apache CXF and WSS4J. One possibility to restrict the list of algorithms that can be used by Web Services is provided by the WS-Security Policy standard [LK07]. This standard allows the server to define specific algorithms that clients must use. Apache CXF supports the WS-Security Policy standard and correctly checks the algorithms used in the encrypted XML messages. However, the specific design of the Apache CXF framework means that algorithms used for data decryption are checked *after* the message is decrypted. More precisely, the WS-Security Policy module checking the algorithms used in the XML processing pipe (as described in Section 2.8), is placed *after* the XML Encryption processing module. This means the attacker is able to force the server to decrypt arbitrary data with arbitrary cryptographic algorithms, which in turn allows to use the server as an plaintext/ciphertext validity oracle, as required for our attacks.

As a response to our attacks, the Apache CXF developers redesigned Apache WSS4J and Apache CXF implementations to check the used security algorithms *before* ciphertexts are decrypted.

4.6.6.3.2 Ping Identity. Ping Identity [Ide13] provides its customers with products acting as an Identity Provider as well as a Service Provider. Both products enable users to apply XML Encryption.

In its documentation, Ping Identity suggested that its users could use the same asymmetric key pair for signature as well as encryption processing [Pin12]. We notified the framework developers. The Ping Identity website was updated immediately and the suggestion removed. Moreover, we cooperated with the developers and evaluated XML Encryption processing in their Service Provider and Identity Provider implementations. We found that our BC attacks were applicable to the Service Provider implementation in all the provided settings. The BC attacks against the Identity Provider implementation could be executed for specific settings where XML Signatures are not applied.[19]

The Ping Identity developers changed their implementation such that only *signed* XML ciphertexts will be decrypted. Furthermore, the RSA-PKCS#1 v1.5 algorithm will be disabled by default *for message creators* (senders). For interoperability reasons receivers will still need to support RSA-PKCS#1 v1.5. Even though the latter still enables BC attacks, this is a step towards phasing out RSA-PKCS#1 v1.5.

4.6.6.3.3 Shibboleth. Shibboleth [Shi13] is a SAML-based framework supporting federated identity management deployments. Decryption of XML messages is supported only in the Service Provider implementation. XML Encryption is enabled by default in the Shibboleth deployments.

After we communicated the attacks to the framework developers, they decided to blacklist RSA-PKCS#1 v1.5 by default in the newest Service Provider version (Shibboleth 2.5.0).

4.7 Summary of Countermeasures and Best Practices

In the following we summarize best practices to counter attacks described in this section. The presented countermeasures are of general use and work for any application in general scenarios. We stress that other countermeasures are most likely to fail in specific scenarios. More information can be found in previous Sections 4.4.5, 4.5.4, and 4.6.6.

Key Separation. An application using different security primitives should always apply proper key separation (see Section 4.6.6.2). This principle should be applied even if the application exclusively applies provably secure ciphers. Namely, it is possible that a security library leaks a side-channel information

[19] The attacks against the Identity Provider are significant, since they allow an attacker to forge Identity Provider signatures for arbitrary SAML tokens when the same key pair for signature and encryption processing is used.

making the provably secure cipher vulnerable, or that a cipher becomes vulnerable to a new practical attack. This would influence cryptographic algorithms using the same key.

The key separation principle should be applied for public key as well as symmetric key algorithms:

1. Implementations using public key algorithms should always use distinct public key pairs for distinct public key algorithms. For example, if an implementation uses PKCS#1 v1.5 and PKCS#1 v2.0 encryption algorithms, and PKCS#1 v1.5 signatures, the implementation should force the user to generate three distinct RSA key pairs, each for a different algorithm.

2. Implementations using symmetric keys should not use the same key material for different algorithms, even if serving the same purpose. Key derivation based on a single key and the algorithm identifier can be used to accomplish this. For example, if the implementation uses CBC, GCM, and HMAC, it could derive distinct keys for these algorithms from a given master key k using a cryptographic hash function hash:

$$k_{\mathsf{CBC}} = \mathsf{hash}(k||\text{"}\mathsf{CBC}\text{"})$$
$$k_{\mathsf{GCM}} = \mathsf{hash}(k||\text{"}\mathsf{GCM}\text{"})$$
$$k_{\mathsf{HMAC}} = \mathsf{hash}(k||\text{"}\mathsf{HMAC}\text{"}).$$

We encourage developers to enforce these principles directly in their libraries so that the library users are forced to apply them.

Thwarting Adaptive Chosen-Ciphertext Attacks. We propose two general ways to protect implementations against the described adaptive chosen-ciphertext attacks:

1. The first possible countermeasure is to restrict usage to secure algorithms (i.e. authenticated encryption schemes). Application of this countermeasure could cause interoperability problems for all parties that are only capable of running older algorithms. However, in specific scenarios this is applicable. The countermeasure could be applied in different ways. Web Services servers can be configured to accept only specific algorithms via WS-Security Policy documents. Other applications can be extended with specific configuration files to blacklist insecure algorithms. It is important that by application of this countermeasure, documents containing insecure algorithms are *rejected without further processing* (the application must not attempt to decrypt the ciphertext encrypted with an insecure algorithm).

2. It is also possible to maintain applications that support PKCS#1 v1.5 and CBC. In that case, the developers should apply specific countermeasures:

   - CBC ciphertexts should be decrypted only if they are authenticated (e.g., in XML messages by XML Signatures). Unauthenticated CBC

ciphertexts should be rejected without further processing. This can be achieved by implementing specific measures directly in the security libraries (see Section 4.4.5.3).[20]

- PKCS#1 v1.5 encrypted messages should be decrypted and their format should be verified. If a decrypted message is not PKCS#1 v1.5 conformant or if the unwrapped key is of invalid length, the implementation should use a new random key (see Section 4.5.4.2). We stress that this countermeasure does not work if the application accepts unauthenticated CBC ciphertexts.

4.8 Conclusion

In this chapter we showed different ways to break confidentiality of encrypted XML messages. We first presented applications of Bleichenbacher's attack and a novel adaptive chosen-ciphertext attack on XML Encryption. We showed how specific properties of XML Encryption and behavior of Web Services servers can improve the attacks.

In case of the attacks on CBC, we showed a general attack that exploits properties of the encoding of encrypted data in combination with the inevitable response behavior of a server. We extended the ideas of Vaudenay [Vau02] and related work [BU02, PY04, YPM05, RD10, Mit05] by making the observation that character encoding patterns can lead to successful attacks even if padding oracles are not available. Interestingly, the encoding patterns speed up the original attack.

By investigating Bleichenbacher's attack, we found new side-channels allowing the execution of the attack. Our attacks exploit timing behavior of the server and combination of PKCS#1 v1.5 with CBC.

We used the attacks against PKCS#1 v1.5 and CBC as the basis for exploration of backwards compatibility attacks. The backwards compatibility attacks show that the mere presence of these insecure options (e.g. PKCS#1 v1.5 and CBC) can adversely affect the security of state-of-the-art algorithms (such as AES GCM or RSA-OAEP), which would otherwise be invulnerable to adaptive chosen-ciphertext attacks. Thus, our work makes another case for using encryption schemes secure against chosen-ciphertext attacks and for application of proper key separation.

We applied all the attacks practically against different common frameworks adopting XML Encryption and JSON Web Encryption. However, our attack ideas are generic and can probably be applied to other specifications as well.

[20] Note that application of public key signatures as a countermeasure does not work if an attacker is registered as a valid Web Service client able to sign XML requests. In that case, he can take a message signed by a different client, remove the original signature, adapt the ciphertext and sign the message with his own key.

5 Conclusions and Future Work

This work presented several critical vulnerabilities in XML Security interfaces of different frameworks and systems.

General Research Impact. We showed various practical classical and novel XSW attacks. The attacks prove that the complexity of XML Security interfaces creates a large seedbed of potential vulnerabilities. These vulnerabilities show that, in order to construct secure systems, it is not sufficient to rely solely on strong cryptography. It is furthermore inevitable to *analyze the interaction between cryptographic primitives and application logic thoroughly.* Based on our collection of different practical attacks, we summarized requirements for XSW attacks, and designed an XSW library which systematically constructs new XSW attack vectors.

We furthermore presented several adaptive chosen-ciphertext attacks on XML Encryption. From a scientific point of view, the attacks are interesting due to novel side-channels and properties they exploit: character encoding errors, timing differences measurable over network, or combinations of weaknesses in different encryption schemes. These side-channels confirm that a system should *not* rely on legacy cryptographic algorithms with *ad-hoc* countermeasures. Even if the system is secure today, it does not mean it will stay secure in the future. There is still a possibility that future research discovers new side-channels or more performant attacks. These attacks can then affect security of the whole system, including other cryptographic schemes the system uses. Thereby, our research motivates the *application of provably secure cryptographic schemes and proper key separation.*

Practical Research Impact. Our work influenced many frameworks and systems, which were updated in order to prevent the presented attacks. In many cases, we closely cooperated with the developers responsible for the affected systems. Moreover, the newest version of the XML Encryption recommendation was extended with a description of specific countermeasures that prevent our attacks.

During this research, I cooperated with the W3C XML Security Working Group to define practical countermeasures, and with several developers who also deployed specific countermeasures to our attacks in these systems and frameworks: Amazon Web Services (AWS) [AWS13], Apache Axis2 [Apa13a], Apache CXF [Apa13b], Eucalyptus [Euc13], IBM Datapower [IBM13], JBossWS [JBo13], OpenAthens [Edu13], OpenSAML [Ope13b], Ping Identity [Ide13], Salesforce [Sal13], SAP [SAP13], and WSO2 [WSO13].

Future Work. We learned that the areas of theoretical cryptography and practical security are separated by a huge research gap. This provides possibilities for new attacks and researches. Combination of these two areas should be

followed more closely by theoretical researchers as well as practical developers. This is the way for establishing a good base for secure cryptographic systems and frameworks.

We applied the presented attacks to specifications of the XML (and partly JSON) family. The attacks are however of general importance and most likely applicable to other systems as well. Research on practical cryptographic attacks is needed to motivate practitioners to remove insecure cryptographic algorithms from their systems and specifications.

Bibliography

[ABBC10] Tolga Acar, Mira Belenkiy, Mihir Bellare, and David Cash. Cryptographic agility and its relation to circular encryption. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 403–422, French Riviera, May 30 – June 3, 2010. Springer, Berlin, Germany.

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In Vitaly Shmatikov, editor, *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*. ACM, Alexandria and VA and USA, 2008.

[ACC+11] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, Giancarlo Pellegrino, and Alessandro Sorniotti. From multiple credentials to browser-based single sign-on: Are we more secure? In Jan Camenisch, Simone Fischer-Hübner, Yuko Murayama, Armand Portmann, and Carlos Rieder, editors, *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 68–79. Springer Boston, 2011. 10.1007/978-3-642-21424-0_6.

[AES01] Advanced encryption standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[AP12] Nadhem AlFardan and Kenneth G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, February 2012.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[Apa12a] Apache Software Foundation. Apache Rampart – Axis2 Security Module. URL: `http://axis.apache.org/axis2/java/rampart` (Retrieved: April 2013), 2012.

[Apa12b] Apache Software Foundation. Apache WSS4J – Web Services Security for Java. URL: `http://ws.apache.org/wss4j` (Retrieved: April 2013), 2012.

[Apa13a] Apache Software Foundation. Apache Axis2. URL: `http://axis.apache.org/axis2/java/core` (Retrieved: April 2013), 2013.

[Apa13b] Apache Software Foundation. Apache CXF. URL: `http://cxf.apache.org` (Retrieved: April 2013), 2013.

[AWS13] Amazon Web Services. URL: `http://aws.amazon.com` (Retrieved: April 2013), 2013.

[BBK03] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 600–616, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Berlin, Germany.

[BCN+10] Aurélie Bauer, Jean-Sébastien Coron, David Naccache, Mehdi Tibouchi, and Damien Vergnaud. On the broadcast and validity-checking security of PKCS#1 v1.5 encryption. In Jianying Zhou and Moti Yung, editors, *ACNS 10*, volume 6123 of *LNCS*, pages 1–18, Beijing, China, June 22–25, 2010. Springer, Berlin, Germany.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Berlin, Germany.

[BFG04] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verifying policy-based security for Web Services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 268–277, New York, NY, USA, 2004. ACM Press.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In Ran Canetti and Rei Safavi-Naini, editors, *Advances in Cryptology – CRYPTO*, 2012.

[BG05] Michael Backes and Thomas Groß. Tailoring the dolev-yao abstraction to web services realities. In Ernesto Damiani and Hiroshi Maruyama, editors, *SWS*, pages 65–74. ACM, 2005.

[BHH+04] 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`.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo

Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Berlin, Germany.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), August 1998. Obsoleted by RFC 3986, updated by RFC 2732.

[BML+13] Guangdong Bai, Guozhu Meng, Jike Lei, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jinsong Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Network and Distributed System Security Symposium (NDSS)*, February 2013.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545, Kyoto, Japan, December 3–7, 2000. Springer, Berlin, Germany.

[Boy01] John Boyer. Canonical XML version 1.0. W3C recommendation, W3C, March 2001. `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`.

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 2008.

[BR94] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111, Perugia, Italy, May 9–12, 1994. Springer, Berlin, Germany.

[BRW04] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX mode of operation. In Bimal K. Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 389–407, New Delhi, India, February 5–7, 2004. Springer, Berlin, Germany.

[BU02] 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.

[CC09] Larry Cable and Thorick Chow. JSR 173: Streaming API for XML, 2009.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Recommendation*, March 2001.

[CD99] James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[CDF+07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[Cen08] Centers for Disease Control and Prevention. Public Health Information Network (PHIN) – Secure Message Transport Guide, July 2008. Version 2.0.

[Cha06] Yuen-Yan Chan. Weakest link attack on single sign-on and its case in saml v2.0 web sso. In Marina Gavrilova, Osvaldo Gervasi, Vipin Kumar, C. Tan, David Taniar, Antonio Laganá, Youngsong Mun, and Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, volume 3982 of *Lecture Notes in Computer Science*, pages 507–516. Springer Berlin / Heidelberg, 2006. 10.1007/11751595_54.

[Chr12] Christian Mainka. Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services, May 2012. Master thesis supervised by Jörg Schwenk and Juraj Somorovsky.

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 583–599, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Berlin, Germany.

[CKPM05] 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`.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[CMPM05] Scott Cantor, Jahan Moreh, Rob Philpott, and Eve Maler. Metadata 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-metadata-2.0-os.pdf`.

[Com10] Committee IT-014. Australian Technical Specification – E-health XML secured payload profiles, March 2010.

[Cro06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746.

[Dan10] Danske Bank / Sampo Pankki. Encryption, Signing and Compression in Financial Web Services, May 2010. Version 2.4.1.

[DLP+12] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. On the joint security of encryption and signature in EMV. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2012.

[DP07] 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.

[DP10] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *ACM Conference on Computer and Communications Security*, pages 493–504, 2010.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878.

[DR11a] 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.

[DR11b] Thai Duong and Juliano Rizzo. Here come the ⊕ Ninjas. Unpublished manuscript, 2011.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. In *NIST Special Publication 800-38D, November 2007, National Institute of Standards and Technology. Available at http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf*, 2007.

[EBI11] Specification EBICS (Electronic Banking Internet Communication Standard), May 2011. Version 2.5.

[Edu13] Eduserv. OpenAthens. URL: `http://www.openathens.net` (Retrieved: April 2013), 2013.

[EJ01] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFC 4634.

[ERH+12] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. XML Encryption Syntax and Processing 1.1. *W3C Candidate Recommendation*, 2012. `http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018`.

[ERH+13] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. XML Encryption Syntax and Processing 1.1. *W3C Recommendation*, 2013. `http://www.w3.org/TR/xmlenc-core1`.

[ERI+02] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.

[ERS+08] Donald Eastlake, Joseph Reagle, David Solo, Frederick Hirsch, and Thomas Roessler. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation*, 2008.

[Euc13] Eucalyptus. URL: `http://www.eucalyptus.com` (Retrieved: April 2013), 2013.

[FFB+09] Mary F. Fernández, Daniela Florescu, Scott Boag, Jonathan Robie, Don Chamberlin, and Jérôme Siméon. XQuery 1.0: An XML query language (second edition). W3C proposed edited recommendation, W3C, April 2009. http://www.w3.org/TR/2009/PER-xquery-20090421/.

[FOPS04] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, March 2004.

[Fou07] OpenID Foundation. OpenID Authentication 2.0 - Final. Technical report, December 2007.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[Fus12] Fuse services framework documentation. Providing Encryption Keys and Signing Keys, July 2012. URL: `http://fusesource.com/docs/framework/2.4/security/MsgProtect-SOAP-ProvideKeys.html#MsgProtect-SOAP-ProvideKeys-SpringConfig`.

[GAK08] Danilo Gligoroski, Suzana Andova, and Svein J. Knapskog. On the importance of the key separation principle for different modes of operation. In Liqun Chen, Yi Mu, and Willy Susilo, editors, *ISPEC*, volume 4991 of *Lecture Notes in Computer Science*, pages 404–418. Springer, 2008.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.

[GHR06] Martin Gudgin, Marc Hadley, and Tony Rogers. Web Services Addressing 1.0 - SOAP Binding. *W3C Recommendation*, May 2006.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[GJLS09] Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jörg Schwenk. Analysis of signature wrapping attacks and countermeasures. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 575–582, 2009.

[GL09] 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.

[GLS07] Sebastian Gajek, Lijun Liao, and Jörg Schwenk. Breaking and fixing the inline approach. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services*, pages 37–43, New York, NY, USA, 2007. ACM.

[GP06] Thomas Groß and Birgit Pfitzmann. SAML artifact information flow revisited. In *In IEEE Workshop on Web Services Security (WSSS)*, pages 84–100, Berkeley, May 2006. IEEE.

[Gro03] Thomas Groß. Security Analysis of the SAML SSO Browser/Artifact Profile. In *ACSAC*, pages 298–307, 2003.

[gua13] Guanxi. URL: `http://codebrane.com/brane/node/7` (Retrieved: April 2013), 2013.

[Har12] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[HBKMN07] Phillip Hallam-Baker, Chris Kaler, Ronald Monzillo, and Anthony Nadalin. Web Services Security X.509 Certificate

Token Profile. W3C recommendation, W3C, Jun. 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626.

[Hig] Higgins 1.x. URL: `http://www.eclipse.org/higgins` (Retrieved: April 2013).

[Hil07] Bradley W. Hill. Command Injection in XML Signatures and Encryption. *Information Security Partners*, 2007.

[HJK08] P. Harding, L. Johansson, and N. Klingenstein. Dynamic security assertion markup language: Simplifying single sign-on. *Security Privacy, IEEE*, 6(2):83 –85, march-april 2008.

[HS11] Moritz Horsch and Martin Stopczynski. The German eCard-Strategy, 2011. Technical Report.

[IBM13] IBM. WebSphere DataPower SOA Appliances. URL: `http://www-01.ibm.com/software/integration/datapower` (Retrieved: April 2013), 2013.

[Ide13] Ping Identity. PingFederate. URL: `https://www.pingidentity.com` (Retrieved: April 2013), 2013.

[JBo13] JBoss Community. JBoss Projects. URL: `http://www.jboss.org/projects` (Retrieved: April 2013), 2013.

[JBS12] M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS) – draft-ietf-jose-json-web-signature-06, October 2012. `http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-06`.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[JLS09] 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.

[JMSS11] M. Jensen, C. Meyer, J. Somorovsky, and J. Schwenk. On the effectiveness of xml schema validation for countering xml signature wrapping attacks. In *Securing Services on the Cloud (IWSSC), 2011 1st International Workshop on*, pages 7 –13, September 2011.

[Jos06] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[JOS13] Java Open Single Sign-On (JOSSO). URL: `http://www.josso.org` (Retrieved: April 2013), 2013.

[JPS13] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography. In *Network and Distributed System Security Symposium (NDSS)*, February 2013.

[JRH12] M. Jones, E. Rescorla, and J. Hildebrand. JSON Web Encryption (JWE) – draft-ietf-jose-json-web-encryption-06, October 2012. `http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-06`.

[JS11] Tibor Jager and Juraj Somorovsky. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[JSS12] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In Sara Foresti and Moti Yung, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings*, LNCS. Springer, 2012.

[jWG] IETF jose Working Group. Javascript Object Signing and Encryption (jose). `http://datatracker.ietf.org/wg/jose/`.

[Kal98] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), March 1998. Obsoleted by RFC 2437.

[Kal00] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.

[Kal02] Burton S. Kaliski Jr. On hash function firewalls in signature schemes. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 1–16, San Jose, CA, USA, February 18–22, 2002. Springer, Berlin, Germany.

[Kan10] Kantara Initiative. Kantara Initiative eGovernment Implementation Profile of SAML V2.0, June 2010. Version 2.0.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[KPR03] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 426–440, Cologne, Germany, September 8–10, 2003. Springer, Berlin, Germany.

[KS98] B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437 (Informational), October 1998. Obsoleted by RFC 3447.

[KSW97] John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 1997.

[Kwa95] Peter Kwan. Unicode: a universal character set. *Language International*, Volume 7(4), 1995.

[Lay13] Layer7 Technologies. Layer7 XML Firewall. URL: `http://www.layer7tech.com/products/xml-firewall` (Retrieved: April 2013), 2013.

[LK07] Kelvin Lawrence and Chris Kaler. WS-SecurityPolicy 1.2. OASIS Standard, July 2007.

[LS10] D.J. Lutz and B. Stiller. Combining identity federation with payment: The saml-based payment protocol. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 495 – 502, april 2010.

[MA05] 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.

[Man01] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 230–238, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.

[Mar11a] Marco Kampmann. Analyse SAML-basierter Single Sign-On Frameworks, April 2011. Thesis supervised by Jörg Schwenk and Juraj Somorovsky.

[Mar11b] Marco Kampmann. Implementierung eines Signature Wrapping Penetration Test Tools für SAML-basierte SSO-Frameworks, December 2011. Diploma thesis supervised by Jörg Schwenk and Juraj Somorovsky.

[MBF+04] Francis McCabe, David Booth, Christopher Ferris, David Orchard, Mike Champion, Eric Newcomer, and Hugo Haas. Web services architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[MGMB07] Michael McIntosh, Martin Gudgin, K. Scott Morrison, and Abbie Barbir. Basic Security Profile Version 1.0. *Web Services Interoperability Organization (WS-I)*, 2007.

[MGRN05] Noah Mendelsohn, Martin Gudgin, Hervé Ruellan, and Mark Nottingham. SOAP message transmission optimization mechanism. W3C recommendation, W3C, January 2005. http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/.

[Mic13] Microsoft. Windows Identity Foundation. URL: `http://msdn.microsoft.com/en-us/library/hh291066.aspx` (Retrieved: April 2013), 2013.

[Mit05] Chris J. Mitchell. Error oracle attacks on cbc mode: Is there a future for cbc mode encryption? In *ISC*, pages 244–258, 2005.

[MMGW03] Jonathan Marsh, Eve Maler, Paul Grosso, and Norman Walsh. XPointer framework. W3C recommendation, W3C, March 2003. http://www.w3.org/TR/2003/REC-xptr-framework-20030325/.

[Mor02] Rajiv Mordani. JSR 63: JavaTM API for XML Processing 1.1, 2002.

[Mos05] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. *OASIS Standard*, 2005.

[MS10] David Molnar and Stuart E. Schechter. Self hosting vs. cloud hosting: Accounting for the security impact of hosting in the cloud. In *WEIS*, 2010.

[MSS12] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration testing tool for web services security. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.

[MV04] David A. McGrew and John Viega. The security and performance of the Galois/counter mode (gcm) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355, Chennai, India, December 20–22, 2004. Springer, Berlin, Germany.

[MvV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.

[Nat99] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. October 1999. supersedes FIPS 46-2.

[NGG+09] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. WS-SecureConversation 1.4. *OASIS Standard*, 2009.

[Nim13] Nimbus Directory Services. Nimbus JSON Web Token. URL: `https://bitbucket.org/nimbusds/nimbus-jose-jwt` (Retrieved: April 2013), 2013.

[NIS80] NIST. Cipher block chaining. NIST FIPS PUB 81, U.S. Department of Commerce, 1980.

[NIS01a] NIST. AES key wrap specification, 2001.

[NIS01b] NIST. Recommendation for block cipher modes of operation. Special Publication 800-38A, 2001.

[NIS04] NIST. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. Special Publication 800-38C, 2004.

[NKMHB06] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard*, 2006.

[Oht95] M. Ohta. Character Sets ISO-10646 and ISO-10646-J-1. RFC 1815 (Informational), July 1995.

[OIO13] OIOSAML. URL: `http://www.ohloh.net/p/oiosaml` (Retrieved: April 2013), 2013.

[One13] OneLogin. URL: `http://www.onelogin.com` (Retrieved: April 2013), April 2013.

[Ope13a] OpenAM. URL: `http://forgerock.com/openam.html` (Retrieved: April 2013), 2013.

[Ope13b] OpenSAML. URL: `http://opensaml.org` (Retrieved: April 2013), 2013.

[Ora11] Oracle. Securing SOA and Web Services with Oracle Enterprise Gateway, April 2011. Technical Report.

[Pin12] Ping Identity product documentation. PingFederate 6.6, Selecting a Decryption Key (SAML), July 2012. URL: `http://documentation.pingidentity.com/display/PF66/Selecting+a+Decryption+Key+(SAML)`.

[PSST11] Kenneth G. Paterson, Jacob C. N. Schuldt, Martijn Stam, and Susan Thomson. On the joint security of encryption and signature, revisited. In *ASIACRYPT 2011*, LNCS, pages 161–178. Springer, Berlin, Germany, December 2011.

[PW08] Kenneth G. Paterson and Gaven J. Watson. Immunising CBC mode against padding oracle attacks: A formal security treatment. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 340–357. Springer, 2008.

[PY04] Kenneth G. Paterson and Arnold Yau. Padding oracle attacks on the ISO CBC mode encryption standard. In Tatsuaki Okamoto, editor, *CT-RSA 2004*, volume 2964 of *LNCS*, pages 305–323, San Francisco, CA, USA, February 23–27, 2004. Springer, Berlin, Germany.

[RBBK01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM CCS 01*, pages 196–205, Philadelphia, PA, USA, November 5–8, 2001. ACM Press.

[RBH02] Joseph Reagle, John Boyer, and Merlin Hughes. XML-signature XPath filter 2.0. W3C recommendation, W3C, November 2002. http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/.

[RD10] Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[Res02] E. Rescorla. Preventing the Million Message Attack on Cryptographic Message Syntax. RFC 3218 (Informational), January 2002.

[RMS06] Mohammad Ashiqur Rahaman, Rits Marten, and Andreas Schaad. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.

[Roe09] Thomas Roessler. HMAC truncation vulnerability. *CVE-2009-0217*, 2009.

[Rog95] Phil Rogaway. Problems with proposed IP cryptography. Unpublished manuscript, 1995. `http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt`.

[RrB02] Joseph Reagle, Donald E. Eastlake 3rd, and John Boyer. Exclusive XML canonicalization version 1.0. W3C recommendation, W3C, July 2002. http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/.

[RS07] Mohammad Ashiqur Rahaman and Andreas Schaad. Soap-based secure conversation and collaboration. In *ICWS*, pages 471–480, 2007.

[RSR06] Mohammad Ashiqur Rahaman, Andreas Schaad, and Maarten Rits. Towards secure soap message exchange in a soa. In *Workshop on Secure Web Services*, 2006.

[RT10] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), January 2010.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[Sal13] Salesforce. URL: `http://www.salesforce.com` (Retrieved: April 2013), 2013.

[SAP13] SAP. SAP Netweaver. URL: `http://scn.sap.com/community/netweaver` (Retrieved: April 2013), 2013.

[SB12] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. Aug 2012.

[Sco05] Scott Cantor and Frederick Hirsch and John Kemp and Rob Philpott and Eva Maler. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf`, March 2005.

[Shi13] Shibboleth Consortium. Shibboleth. URL: `http://shibboleth.net` (Retrieved: April 2013), 2013.

[SHJ+11] 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.

[Sim13] SimpleSAMLphp. URL: `http://simplesamlphp.org` (Retrieved: April 2013), 2013.

[SKH05] J. Schaad, B. Kaliski, and R. Housley. Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 4055 (Proposed Standard), June 2005. Updated by RFC 5756.

[Sma10] Nigel P. Smart. Errors matter: Breaking RSA-based PIN encryption with thirty ciphertext validity queries. In Josef Pieprzyk, editor, *CT-RSA 2010*, volume 5985 of *LNCS*, pages 15–25, San Francisco, CA, USA, March 1–5, 2010. Springer, Berlin, Germany.

[SMS+12] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On Breaking SAML: Be Whoever You Want to Be. In *21st USENIX Security Symposium*, Bellevue, WA, August 2012.

[TDG98] R. Thayer, N. Doraswamy, and R. Glenn. IP Security Document Roadmap. RFC 2411 (Informational), November 1998.

[TFP+06] H. Tschofenig, R. Falk, J. Peterson, J. Hodges, D. Sicker, and J. Polk. Using saml to protect the session initiation protocol (sip). *Network, IEEE*, 20(5):14 –17, sept.-oct. 2006.

[The13] The Apache Software Foundation. Apache Xerces. URL: `http://xerces.apache.org` (Retrieved: April 2013), 2013.

[Vau02] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 534–546, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Berlin, Germany.

[VM05] J. Viega and D. McGrew. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP). RFC 4106 (Proposed Standard), June 2005.

[VS12] Kristi Uukkivi Veiko Sinivee. Encrypted DigiDoc Format Specification, June 2012. Version 1.1.

[WCW12] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy (Oakland), IEEE Computer Society*, May 2012.

[WF04] Priscilla Walmsley and David C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[WSO13] WSO2. WSO2 Enterprise Service Bus. URL: `http://wso2.com/products/enterprise-service-bus` (Retrieved: April 2013), 2013.

[Xia11] Xiaofeng Lou. Überprufung von Sicherheitsvorgaben für Cloud Computing nach OSSTMM, July 2011. Diploma thesis supervised by Aaron Brown, Jörg Schwenk and Juraj Somorovsky.

[YHV+07] Prasad Yendluri, Maryann Hondo, Asir S Vedamuthu, Frederick Hirsch, David Orchard, Ümit Yalçinalp, and Toufic Boubez. Web services policy 1.5 - framework. W3C recommendation, W3C,

September 2007. http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[YPM05] 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, *FSE 2005*, volume 3557 of *LNCS*, pages 299–319, Paris, France, February 21–23, 2005. Springer, Berlin, Germany.

[YsJ10] Zhang Yong-sheng and Yang Jing. Research of dynamic authentication mechanism crossing domains for web services based on saml. In *Future Computer and Communication (ICFCC), 2010 2nd International Conference on*, volume 2, pages V2–395 –V2–398, may 2010.