

Bleichenbacher’s Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption

Tibor Jager,¹ Sebastian Schinzel,^{2*} and Juraj Somorovsky^{3**}

¹ Institut für Kryptographie und Sicherheit, Karlsruhe Institute of Technology, Germany
tibor.jager@kit.edu

² Lehrstuhl für Informatik 1 - IT-Sicherheitsinfrastrukturen,
Universität Erlangen-Nürnberg, Germany
sebastian.schinzel@cs.fau.de

³ Horst-Görtz Institute for IT Security, Ruhr-University Bochum, Germany
juraj.somorovsky@rub.de

Abstract. We describe several attacks against the PKCS#1 v1.5 key transport mechanism of XML Encryption. Our attacks allow to recover the secret key used to encrypt transmitted payload data within a few minutes or several hours, depending on the considered scenario.

The attacks exploit differences in error messages and in the timing behavior of XML frameworks. We show how to attack seemingly invulnerable implementations, by exploiting additional properties of the XML Encryption standard that lead to new side-channels. An interesting novelty of one of our attacks is that it combines a weakness of a public-key scheme (transporting an ephemeral session key) with a different weakness of a symmetric encryption scheme (which transports the payload data, encrypted with the session key).

Recently the XML Encryption standard was updated, in response to an attack presented at CCS 2011. The attacks described in this paper work even against the updated version of XML Encryption. Our work shows once more that legacy cryptosystems have to be used with extreme care, and should be avoided wherever possible, since they may lead to practical attacks.

Keywords: PKCS#1 v1.5, Bleichenbacher’s attack, XML Encryption, chosen-ciphertext attacks.

1 Introduction

In 1998 Bleichenbacher [3] published a chosen-ciphertext attack on the RSA-based PKCS#1 v1.5 encryption scheme specified in RFC 2313 [15]. This attack exploits the availability of an “oracle” that allows to test whether a given ciphertext is PKCS#1 v1.5 conformant. Due to its high relevance, Bleichenbacher’s algorithm was well noticed. For instance, it enabled practical attacks on popular implementations of the SSL protocol [17]. 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 particular in scenarios where an adversary may issue chosen-ciphertexts to a server and observe the response.

In spite of these negative results, in 2002, four years after publication of the Bleichenbacher attack, the W3C consortium published the XML Encryption standard [6], in which PKCS#1 v1.5 encryption is specified as a *mandatory* key transport mechanism. This standard is implemented in XML frameworks of major commercial and open-source organizations like Apache, redhat, IBM, Microsoft, and SAP and employed world-wide in a large number of major web-based and cloud-based applications, ranging from business communications, e-commerce, and financial services over healthcare applications to governmental and military infrastructures.

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

* Sebastian Schinzel was supported by Deutsche Forschungsgemeinschaft (DFG) as part of SPP 1496 “Reliably Secure Software Systems”.

** Juraj Somorovsky was supported by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

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 [3].

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 [28] or redhat’s JBossWS [11]. 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 standard:

1. *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.
2. *A weak mode-of-operation.* XML Encryption allows the usage of block ciphers in the *cipher-block chaining* (CBC) mode-of-operation. CBC exhibits a weakness [29] that allows an adversary to make modifications to the encrypted plaintext, by XORing 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 [10]. Interestingly, the CBC-attack we describe in this paper *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 have 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 paper *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 (an 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. We compare these two attacks and give two realistic scenarios where each attack performs especially well. 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 5, and which holds for (heuristically) one out of 80 ciphertexts (see Section 5.1). We also note that the recent improvements to Bleichenbacher’s algorithm by Bardou et al. [1] 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 [7] to ensure integrity. (Note that XML Signature specifies not only classical public-key signatures, but also “secret-key signatures”, i.e., message authentication codes.) However, 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, see [10, 26] for a detailed description.

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. SAP is currently in the process of fixing this issue.

Beyond XML Encryption, the recent JSON Web Encryption (JWE) specification [12] prescribes PKCS#1 v1.5 as a mandatory cipher. This specification is under development and at the time of writing there existed only one implementation following this specification¹. We verified that this implementation was vulnerable to two versions of the Bleichenbacher’s attack: the direct attack based on error messages and the timing-based attack.

RELATED WORK. At CCS 2011 [10] an attack on XML Encryption was described which allows to extract the plaintext contained in a given ciphertext. This attack breaks the *symmetric* encryption scheme of XML Encryption (AES-CBC or 3DES-CBC) by submitting modified ciphertexts to a Web Service and observing its response. The attack requires on average $14 \cdot \ell$ chosen-ciphertext queries, where ℓ is the byte-length of the recovered plaintext. Even though this is very efficient, the complexity grows linearly with the size of the plaintext, thus may become infeasible if the attacker has to decrypt long plaintexts. The W3C has responded to the attack of [10] by updating the XML Encryption standard. Now it recommends the GCM mode instead of CBC, which prevents chosen-ciphertext attacks against the symmetric cipher.

Let us compare the attack of [10] to our work. For efficiency reasons, a typical XML Encryption ciphertext consists of two components. The first component is a public key encryption c_{key} of an ephemeral session key under the public key of the receiver. The second component is a symmetric encryption c_{data} of the actual plaintext data (see Section 3.2 for a detailed description). Jager and Somorovsky’s attack directly decrypts the c_{data} component of the ciphertext to obtain the plaintext. In contrast, the attacks presented in this paper break the public-key encryption part c_{key} , to recover the ephemeral key first. The ephemeral key can then be used to decrypt c_{data} with the symmetric decryption algorithm. This novel approach has two interesting features. First, it is *independent of the symmetric cipher*, so it can also be used to attack XML Encryption ciphertexts that, according to the updated specification, are generated in GCM mode. Second, the attack complexity is *independent of the size of c_{data}* , and thus becomes more efficient than [10] for large c_{data} . Finally, it allows to recover the *session key* instead of only the plaintext, which may in certain scenarios be more serious.

Bleichenbacher’s attack [3] on PKCS#1 v1.5 [15] has been published at CRYPTO 1998. This attack has been 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 [17]. In [1] Bardou et al. describe several ways to improve the efficiency of Bleichenbacher’s attack. At Crypto 2001 Manger [18] has presented an attack on Version 2.0 of PKCS#1 (RSA-OAEP) [16] which is very similar to Bleichenbacher’s attack, and applicable to the current Version 2.1 [13] 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. [2] have shown 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 has been published by Smart [25], 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. [4] gave 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 [21] it was noted that valid (symmetric-cipher) padding may lead to a side-channel that allows 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.

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 acknowl-

¹ Nimbus-JWT: <https://bitbucket.org/nimbusds/nimbus-jwt>

edged the validity of the attack. The W3C XML Encryption working group added a remark to the updated standard [5, 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 standard, and mandatory for any standard-compliant implementation.

We also informed the developers of the JWE implementation and the whole JOSE (JSON Object Signing and Encryption) working group about the possible threats.² They acknowledged our attack and are reconsidering exclusion of PKCS#1 v1.5 from the standard.

2 Bleichenbacher’s Attack

Bleichenbacher’s attack [3] on version 1.5 of the PKCS#1 encryption standard [15] exploits properties of the encoding of messages. It requires an attacker who has gained access to an encrypted message and who can send chosen ciphertexts to the intended receiver of the message as shown in Figure 1. Furthermore, it requires that an “oracle” is given that allows to distinguish between “valid” (PKCS#1 conformant) and “invalid” (not PKCS#1 conformant) ciphertexts. Such an oracle may in practice be given for instance by a server responding with appropriate error messages. When referring to PKCS#1 in the sequel, then we mean version 1.5, unless specified otherwise. We let (N, e) be an RSA [23] public key, with corresponding secret key d . We denote with ℓ the byte-length of N , thus, we have $2^{8(\ell-1)} < N < 2^{8\ell}$.

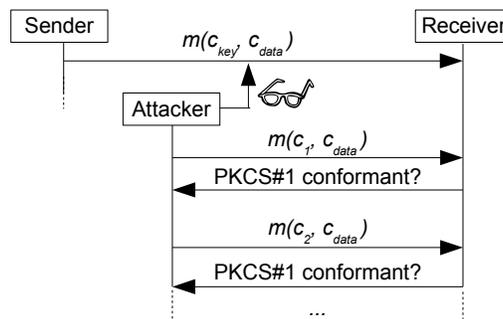


Fig. 1: Attack scenario.

2.1 PKCS#1 v1.5 Padding and Encryption

The basic idea of PKCS#1 v1.5 is to take a message k (a bit string), concatenate this message with a random padding string PS , and then apply the RSA encryption function $m \mapsto m^e \bmod N$.

Let us describe the padding in more detail. In the following, let $a||b$ denote the concatenation of two bit strings a and b . Suppose a message k of byte-length $|k| \leq \ell - 11$ is given. This string is encrypted as follows.

1. Choose a random padding string PS of length $\ell - 3 - |k|$, such that PS contains no 00-byte. Note that the byte length of PS is at least $|PS| \geq 8$.
2. Set $m := 00||02||PS||00||k$. Interpret m as an integer such that $0 < m < N$.
3. Compute the ciphertext as $c = m^e \bmod N$.

The decryption algorithm computes $m' = c^d \bmod N$ and interprets integer m' as a bit string. It tests whether m' has the correct format, i.e., $m' = 00||02||PS||00||k$. If true, it returns k , otherwise it rejects the ciphertext.

In this paper we say that a ciphertext $c \in \mathbb{Z}_N$ is *valid* (PKCS#1 conformant), if the $m = c^d \bmod N$ has the format $m = 00||02||PS||00||k$. Note that this implies in particular that $2B \leq (c^d \bmod N) < 3B$, where $B = 2^{8(\ell-2)}$.

² See <http://www.mail-archive.com/jose@ietf.org/msg00157>

2.2 A Ciphertext-Validity Oracle

The only necessary prerequisite to execute Bleichenbacher’s attack is that an oracle \mathcal{O} is given which tells whether a given ciphertext is valid (PKCS#1 conformant) w.r.t. the target public key (N, e) . This oracle takes as input a ciphertext c and responds as follows.

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } c \text{ is PKCS\#1 conformant w.r.t. } (N, e), \\ 0 & \text{otherwise.} \end{cases}$$

Such an oracle may be given in many practical scenarios, for instance by a web server responding with appropriate error messages. We will show how to construct such an oracle based on properties of XML Encryption.

2.3 Bleichenbacher’s Algorithm

In this section we sketch the idea of Bleichenbacher’s algorithm, which uses the PKCS#1 validity oracle to invert the RSA encryption function $m \mapsto m^e \bmod N$. We give only a high-level description of the attack, and refer to the original paper [3] for details.

Suppose $c = m^e \bmod N$ is given. We assume that c is PKCS#1 conformant. Thus, $m = c^d \bmod N$ lies in the interval $[2B, 3B)$. Bleichenbacher’s algorithm proceeds as follows. It chooses a small integer s (see [3] for details on how s is chosen), computes

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

and queries the oracle with c' . If $\mathcal{O}(c') = 1$, then the algorithm learns that $2B \leq ms - rN < 3B$, for some r , which is equivalent to

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

Thus, m must lie in the interval $m \in [(2B + rN)/s, (3B + rN)/s)$. By iteratively choosing new s , the adversary reduces the possible solutions m , until only one is left.

For a 1024-bit modulus and a random ciphertext, the analysis in [3] shows that the attack requires about one million oracle queries to recover a plaintext, plus a small amount of additional computations. Therefore, Bleichenbacher’s attack became also known as the “Million Question Attack”. The most time-consuming step of the algorithm is to find the first value s such that $\mathcal{O}((c \cdot s^e) \bmod N) = 1$.

We note that very recently Bardou et al. described improvements to Bleichenbacher’s algorithm by Bardou et al. [1], which are applicable in our case as well.

3 Web Services

This section summarizes the fundamentals of XML, XML Security, and Web Services, which are relevant to our paper. The reader familiar with these concepts can safely skip this section.

3.1 XML and Web Services

Web Services is a W3C standard [9] developed to support interoperable interactions over networks between different software applications. Thereby, the communicating applications use SOAP messages [8]. SOAP messages are XML-based messages generally consisting of *header* and *body*. The header element includes message-specific data (e.g. timestamp, user information, or security data). The body element contains function invocation and response data, which are mainly addressed to the business logic processors.

As the XML documents often contain data whose confidentiality and integrity must be protected, the W3C consortium developed standards describing the XML syntax for applying cryptographic primitives to XML data. These are specified in the XML Encryption [6] and XML Signature [7] standards.

3.2 XML Encryption

In order to encrypt XML data, in most scenarios *hybrid encryption* is used, i.e. encryption proceeds in two steps.

1. The encryptor chooses a *session key* k . This key is encrypted using a public-key encryption scheme.
2. The actual payload data is then encrypted with a symmetric cipher.

The XML Encryption standard [6] specifies two public-key encryption schemes, namely PKCS#1 in Versions 1.5 and 2.0. Both are mandatory. Furthermore, the updated version of the standard allows to choose between three symmetric ciphers, namely AES-CBC, AES-GCM, and 3DES-CBC.

```
<Envelope>
  <Header>
    <Security>
      <EncryptedKey Id="EncKeyId">
        <EncryptionMethod Algorithm="...xmlenc#rsa-1_5"/>
        <KeyInfo>...</KeyInfo>
        <CipherData>
          <CipherValue>Y2bh...fPw==</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI="#EncDataId-2"/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </Header>
  <Body>
    <EncryptedData Id="EncDataId-2">
      <EncryptionMethod Algorithm="...xmlenc#aes128-cbc"/>
      <CipherData>
        <CipherValue>3bP...Zx0=</CipherValue>
      </CipherData>
    </EncryptedData>
  </Body>
</Envelope>
```

C_{key}

C_{data}

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

Figure 2 gives an example of a SOAP message containing such a hybrid ciphertext. This message consists of the following parts.

1. The **EncryptedKey** part (c_{key}). The **CipherValue** element (inside the **CipherData** element) contains the encrypted session key. **ReferenceList** contains references to all **EncryptedData** elements that can be decrypted with the session key.
2. The **EncryptedData** part (c_{data}). The **CipherValue** element contains the payload data, encrypted using the key encapsulated in c_{key} . The symmetric cipher is specified in the **EncryptionMethod** element.

Since XML is a text data format, all binary data are converted to text data by applying Base64 [14] encoding. DECRYPTION PROCESSING AND PARSING. A Web Service receiving such an XML document processes it as follows. It parses the document to locate c_{key} and c_{data} . It decrypts c_{key} to obtain the session key k . Then it uses k to decrypt c_{data} to obtain the payload data. Finally, the payload data is parsed as an XML document. PADDING IN CBC. XML Encryption prescribes usage of block ciphers, namely AES or 3DES. Therefore the payload *data* being encrypted needs to be padded to achieve a length which is a multiple of the cipher's block-size bs of the applied block cipher. XML Encryption specifies the following padding scheme:

1. Compute the smallest integer $p > 0$ such that $|data| + p$ is an integer multiple of bs .
2. Append $(p - 1)$ random bytes to *data*.

3. Append one more byte to $data$, whose integer value equals p .

Let us give an example. Suppose a block-size of $bs = 8$ and payload data consisting of $|data| = 5$ bytes, e.g.

$$data = 0x0101010101.$$

Then we have $p = 8 - 5 = 3$. Thus, the padded payload data would be equal to

$$data = 0x0101010101????03,$$

where the ?? are arbitrary random bytes.

CIPHER BLOCK CHAINING. *Cipher-block chaining* (CBC) [20] is the most popular block cipher mode-of-operation in practice. The XML Encryption standard allows to choose between CBC and GCM mode, both are mandatory. For our application it suffices to describe CBC, but we stress again that both attacks that we present in this paper apply to ciphertexts generated in GCM mode as well.

Suppose a byte string $data$, whose length is an integer multiple $d \cdot bs$ of the block-size of the block cipher (Enc, Dec). Let us write $data = (data^{(1)}, \dots, data^{(d)})$ to denote individual chunks of $data$ of size bs . These chunks are processed as follows.

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

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

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

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

for $i = 2, \dots, d$.

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

See Figure 3 for an illustration of this scheme. The decryption procedure inverts this process in the obvious way.

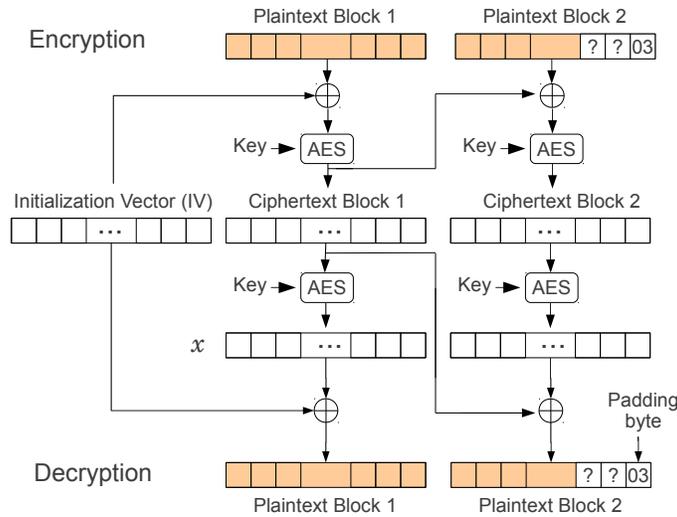


Fig. 3: Illustration of the CBC mode of operation with the XML Encryption padding scheme.

3.3 Web Services Frameworks

The rising popularity of Web Services in the recent years led to an emergence of many Web Services frameworks [11, 24, 27, 28]. A very popular example is the widely-used Apache Axis2 framework. We will execute the bulk of our experimental analyses on Axis2, therefore we describe this framework in more detail.

Remark 1. 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 have verified for SAP, for instance). Moreover, as we describe in Section 5.4 in detail, it turns out that exploiting certain additional framework-specific side-channels may even lead to dramatically more efficient attacks.

Apache Axis2 is a Java-based open source framework for deploying Web Services servers and clients. The framework includes several modules implementing various Web Service specifications, such as Apache Rampart. This module enables to utilize XML Encryption. When receiving a SOAP message containing encrypted data, Axis2 locates c_{key} and c_{data} in the XML document structure. In order to decrypt c_{key} , Axis2 performs the PKCS#1-validity checks described in Section 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_{data} , which yields the payload data $data$. Finally, $data$ 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_{key}, c_{data}) , and we modify the key encapsulation part c_{key} (this is necessary to mount Bleichenbacher’s attack). Then we obtain a modified ciphertext (c'_{key}, c_{data}) . If we send this ciphertext to the Web Service, then we will receive a **security processing failed** error message, since either processing of c'_{key} or parsing of the payload $data$ contained in c_{data} will fail (except for a negligibly small probability). Thus, we are not able to distinguish whether c'_{key} is a valid or an invalid ciphertext. This seems to thwart Bleichenbacher’s attack on the first sigh. However, in the next section, we will describe techniques for exploiting side-channels allowing us to determine the validity of c'_{key} .

4 Attacks

Imagine an attacker who intercepts a message transferred to the Web Service server and whose goal is to decrypt c_{data} . In order to gain the session key k needed for data decryption, the attacker can apply the Bleichenbacher’s attack on c_{key} . In this section, we describe two ways to obtain a side-channel that allows to determine whether a given ciphertext is valid (PKCS#1 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. Note that the stateless SOAP message exchange allows us to send an arbitrary amount of requests.

4.1 Basic Ideas

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 if and only if c_{key} is valid. Recall that in principle it is not possible to mount Bleichenbacher’s attack, since we need to modify c_{key} in a way that decrypting and parsing c_{data} fails, and thus we receive the same **security processing failed** error message in both cases. However, since c_{data} decryption is executed if and only if c_{key} is valid, the time between sending the ciphertext and receiving the error message depends on the validity of c_{key} . 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_{data} to any bit string whose length is an multiple of the block-size of the block cipher. Thus, by increasing the length of c_{data} , the attacker can also increase the timing gap between a valid and an invalid c_{key} . The challenge is to keep c_{data} 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 $data^{(1)}$. Recall that such a ciphertext consists of an iv and a ciphertext block $C^{(1)} := \text{Enc}(k, x)$, where $x := data^{(1)} \oplus iv$. Thus, by flipping bits in iv , we can implicitly flip bits in the plaintext $data^{(1)}$. In particular, we can modify the last byte of $data^{(1)}$, which contains the number of padding bytes. The crucial observation is now, that there exists one modified iv' such that the last byte of $data^{(1)'} = x \oplus iv'$ 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_{key} .

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

4.2 Timing Attack

In this section, we describe a timing oracle \mathcal{O}_t that determines if a given c_{key} is valid. Our observation is that the analyzed Web Service only then decrypts c_{data} if c_{key} is valid. Furthermore, parsing of the clear text does not start until c_{data} was fully decrypted, i.e. filling c_{data} 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_{data} leads to measurably longer decryption times as depicted in Figure 4. 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_{data} . Note that the actual content of c_{data} is irrelevant, only the size is important for the timing delay. In our experiments we enforced Axis2 to decrypt c_{data} using AES-CBC. Note that 3DES-CBC would bring even larger timing differences because the decryption process in 3DES is less efficient than AES, which would make our attack easier.

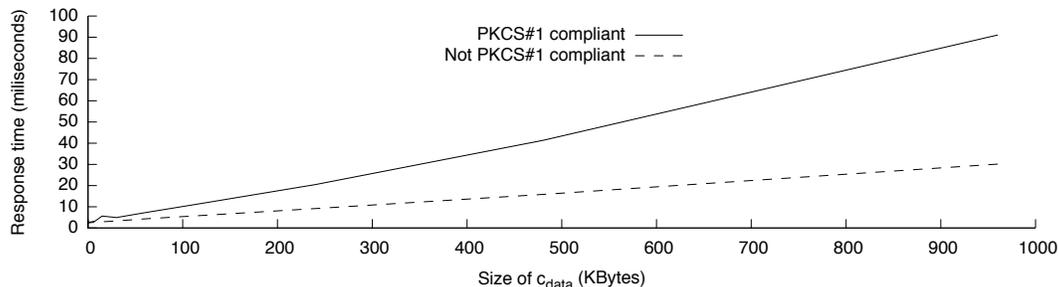


Fig. 4: Timing difference of valid c_{key} and invalid c_{key} in relation to the size of c_{data} , which was decrypted using AES-CBC

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. We propose a new algorithm which allows this. 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 $T_{c_{key}} = \langle t_1, t_2, \dots, t_i \rangle$. If any of the repeated measurements is below the boundary, the key is marked as invalid. 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_{data} .

Equation 3 formally defines the timing oracle.

$$\mathcal{O}_t(c_{key}, l) = \begin{cases} 1 & \text{if } \min(T_{c_{key}}) \geq t_{min}, \\ 0 & \text{if } \exists t \in T_{c_{key}} : t < t_{min}, \end{cases} \quad (3)$$

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.

Calibration Phase The oracle can determine if a given c_{key} 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_{key} from an invalid c_{key} . For this, we perform i requests with a valid c_{key} and record the set of timings $T_{valid} = \langle t_1, t_2, \dots, t_i \rangle$. Note that the attacker already has one valid c_{key} 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 $t'_{min} \leq \min(T_{valid})$.

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} .

Attack Phase Now that \mathcal{O}_t is calibrated, the attacker can apply the Bleichenbacher algorithm as described in Section 2.3. Figure 5 describes the procedure of \mathcal{O}_t . The Bleichenbacher algorithm calls \mathcal{O}_t and passes c_{key} as a parameter. The oracle copies c_{key} 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_{key} is indeed valid³. The oracle answers with 1 if all measurements resulted in greater response times than t_{min} .

```
def is_valid(c_key, n):
    do n times:
        start = now()
        request(c_key, l)
        end = now()

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

Fig. 5: Pseudo code sketching the validation routine of candidates of valid keys

4.3 Exploiting a Weakness of CBC

In this section we describe another attack on c_{key} , 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_{key} . Afterwards, it uses the decrypted session key k to decrypt c_{data} . 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_{key} decryption: the decrypted c_{key} was invalid

³ We used $n = 100$ in our measurements.

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

So from this error message, the attacker only then knows that c_{key} 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_{data} consisting of two randomly generated 16 bytes long blocks $c_{data} = (iv, C^{(1)})$. Then we submit the ciphertext (c_{key}, c_{data}) to the Web Service, claiming that c_{data} is generated in CBC mode. The latter is possible by simply adjusting the metadata of an XML document containing encrypted parts. 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 $data^{(1)} = iv \oplus x$. The last byte of $data^{(1)}$ is taken as a padding byte and the padding is applied. 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. As flipping a bit in iv implicitly changes the corresponding bit in the $data^{(1)}$ block, one can iteratively modify the value of the last byte in $data^{(1)}$. 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.

We can use these observations for constructing an oracle which returns 1 or 0, depending on the validity of the given c_{key} . For each tested c_{key} , the CBC-oracle \mathcal{O}_{cbc} needs to send at most 256 requests with different iv' values⁴. As shown in Equation 4, if Axis2 responds with a **security processing failed** error for a given c_{key} and all possible values of iv , then \mathcal{O}_{cbc} returns that c_{key} was invalid.

$$\mathcal{O}_{cbc}(c_{key}) = \begin{cases} 1 & \text{if } \exists iv_{16} \in \{0, 1, \dots, 255\} : Dec(c_{key}, iv) = \text{"no error"} \\ 0 & \text{if } \forall iv_{16} \in \{0, 1, \dots, 255\} : Dec(c_{key}, iv) = \text{"error"} \end{cases} \quad (4)$$

Why this attack cannot be prevented by the classical countermeasure against Bleichenbacher’s attack. The classical countermeasure against Bleichenbacher’s attack is to let the decryption algorithm return a random key k , if c_{key} is invalid, and then to proceed as if c_{key} was valid.

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_{key} is valid, then among all 256 initialization vectors chosen by the attacker there *must exist* at least one iv such that $c_{data} = (iv, C^{(1)})$ returns no error. In particular, if the attacker submits a ciphertext c_{data} 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_{key} 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 $c = (c_{key}, c_{data})$ 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_{key} is valid with probability close to 1.

5 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 5.1 for a description of this property. We did

⁴ 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 [19]. 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.

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.

In order to evaluate our attacks, we deployed a Web Service secured with XML Encryption and generated a valid SOAP message containing c_{key} in the SOAP header. This element included a symmetric key for c_{data} 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.

5.1 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 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 conformant. Moreover, in our application we will only be able to learn that a ciphertext $c = (ms)^e \bmod N$ is PKCS#1-conformant, if $ms \bmod N$ has the form

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

where 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 6 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-conformant.

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 were tested. Among these 100,000 plaintexts there were 1,543 padded plaintext that lead to *good* ciphertexts. Thus, about each 80-th ciphertext 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.

5.2 Timing-based Attack

In this section, we show the results of the empirical evaluation of \mathcal{O}_t proposed in Section 4.2. 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.

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-conformant,
 - $ms \bmod N = 00\|02\|PS\|00\|k$,
with $|k| \in \{16, 24, 32\}$,
 then set $c = c + 1$.

Fig. 6: Experimental analysis of the distribution of “good” ciphertexts.

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 [22] as the victim.

The measurement computer had 2 Intel XEON 2.4 GHz processors. Figure 7a shows the response times measured during the calibration phase with 100KB c_{data} ciphertext and a c_{key} 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

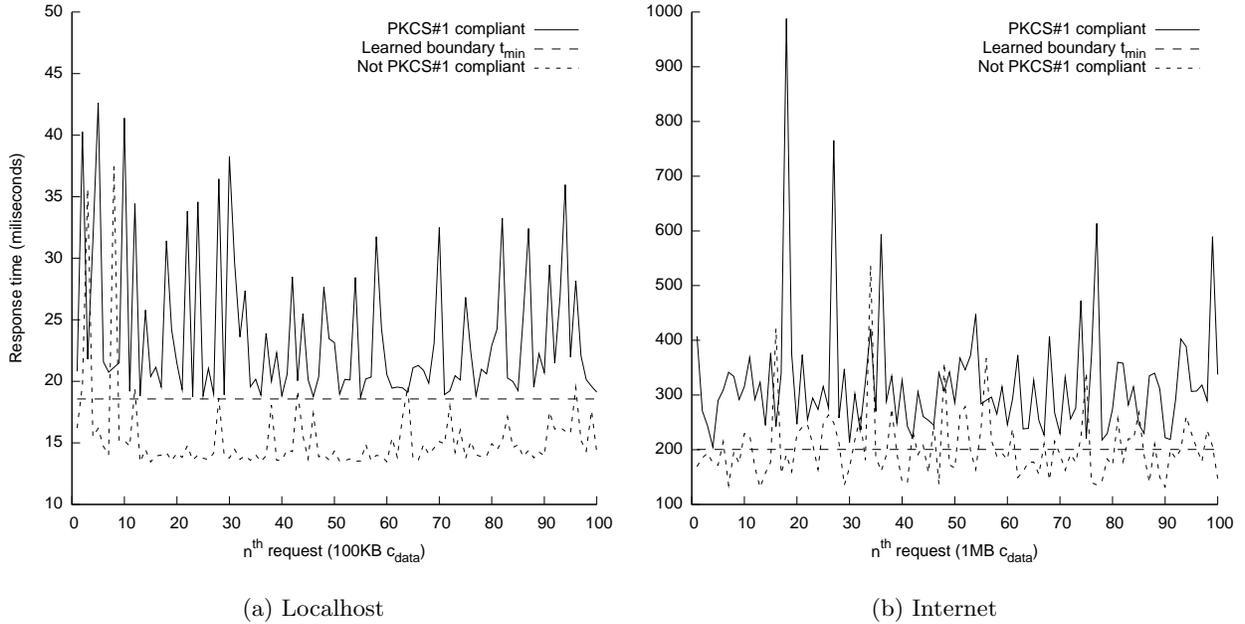


Fig. 7: Response times with valid and invalid c_{key}

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 times for confirmation. The figure suggests that only few invalid requests slipped above t_{min} leading to a repetition of the request.

As a result, c_{key} 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.

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 7b and used 1,000KB of random data as c_{data} . 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 learn c_{key} remotely across practical networks in less than one week.

5.3 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.

5.4 Exploiting JBossWS PKCS#1 Processing

All our attacks presented so far are also applicable to the XML Encryption implementation of JBossWS [11]. In addition, we discovered another side-channel in JBossWS 6.0 that allows us to mount Bleichenbacher’s attack *directly*, by adopting 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_{key} , JBossWS first decrypts c_{key} and obtains a padded plaintext $m = (m_1, \dots, m_{64})$ consisting of 64 bytes m_i , $i \in \{1, \dots, 64\}$. Then it performs the following checks:

1. Test whether $(m_1, m_2) = (0x00, 0x02)$. If true, proceed.
2. Test whether there exists $i \in \{3, \dots, 10\}$ such that $m_i = 0x00$. If false, proceed.
3. Test whether there exists $i \in \{11, \dots, 64\}$ 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, m_2) of the plaintext contained in the submitted ciphertext were equal to $(0x00, 0x02)$. This suffices to mount Bleichenbacher’s attack directly.

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.

5.5 Exploiting Additional Side-Channels in Apache Axis2

As described in Section 3.2, SOAP messages containing encrypted data typically consist out of two parts: c_{key} and c_{data} . In order to reference the c_{data} part from the c_{key} part, the **DataReference** element is used. Using **DataReference**, the message interceptor can locate the part dedicated for symmetric decryption.

By modifying the c_{key} 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 straight-forward Bleichenbacher attack. By analyzing the Axis2 framework we found out that

removing the `DataReference` elements from the c_{key} part reveals a new side-channel: When the decrypted message was *not* PKCS#1 conformant, the server responded with an obvious security error (`security processing failed`). In case of a PKCS#1 conformant message the server correctly decrypted a session key. However, as there was no `DataReference` element, the server security module skipped the c_{data} 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 also validly 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 deeper in order not to reveal details leading to cryptographic side-channels.

6 Comparison of the Timing-based and the Padding-based Attack

An attacker can choose between a timing-based and a padding-based oracle \mathcal{O} to decipher an XML Encryption message and he obviously chooses the oracle that requires the least effort to implement. We measure this effort based on the amount of Web Service requests sent per oracle query and on the amount of data transmitted. Obviously, the less requests and the less data need to be sent, the more effective the attack becomes.

By computing the amount of data transferred between the Web Service server and the attacker, we have to consider these parameters:

- $data_c$: A ciphertext consisting of c_{key} and c_{data}
- $data_o$: Data overhead coming from the SOAP XML structure and the transport data overhead (e.g. HTTP headers). In our examples, the overall overhead per request was about 2.3 KB.
- δ_{attack} : Attack coefficient, number of Web Service queries per oracle query. For the timing-based oracle, δ_{attack} depends on the network and system performance. By applying the padding-based oracle, we get $\delta_{attack} = 256$ in the worst case scenario⁵.

The attacker having an access to both oracles first needs to execute the system profiling using the calibration phase (as described in Section 4.2) to determine the amount of $data_c$ needed to send to the oracle and the oracle coefficient δ_{attack} . Afterwards, he computes the amount of data needed to be transferred to the Web Service server per oracle request:

$$data_{\mathcal{O}} = (data_c + data_o) \cdot \delta_{attack}$$

Using this equation we can compute the approximate values of transmitted data for our timing-based and padding-based oracles from the previous section. The values are given in Table 1.

	Localhost			Network		
	$data_c + data_o$	δ_{attack}	$data_{\mathcal{O}}$	$data_c + data_o$	δ_{attack}	$data_{\mathcal{O}}$
Timing-based \mathcal{O}	102.3 KB	1.2	122.76 KB	1,002.3 KB	1.24	1,242.85 KB
Padding-based \mathcal{O}	2.51 KB	256	642.56 KB	2.51 KB	256	642.56 KB

Table 1: Comparing timing-based and padding-based oracles regarding the amount of sent requests and amount of sent traffic for two different attack scenarios.

By analyzing the table it becomes clear that by application of the padding-based oracle $data_c$ stays constantly small. Thus, the attacker would use this oracle in networks with a high jitter. In different networks, where $data_c$ stays small also for the timing-based attack, the attacker would execute the attack using the timing-based oracle.

⁵ Querying the padding-based oracle with an invalid ciphertext results always in 256 Web Service queries. As most generated ciphertexts in Bleichenbacher’s attack are invalid, $\delta_{attack} \approx 256$.

Acknowledgments

We thank Daniel Bleichenbacher, Felix Freiling, Thorsten Holz, Kenny Paterson, Jörg Schwenk, and the anonymous reviewers for their helpful comments.

References

1. 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.
2. 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: 8th International Conference on Applied Cryptography and Network Security*, volume 6123 of *Lecture Notes in Computer Science*, pages 1–18. Springer, June 2010.
3. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO’98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, August 1998.
4. Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefer. 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.
5. 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/CR-xmlenc-core1-20120313>.
6. Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002. <http://www.w3.org/TR/xmlenc-core>.
7. Donald Eastlake, Joseph Reagle, David Solo, Frederick Hirsch, and Thomas Roessler. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation*, 2008.
8. Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.
9. Hugo Haas, David Booth, Eric Newcomer, Mike Champion, David Orchard, Christopher Ferris, and Francis McCabe. Web services architecture. W3C note, W3C, February 2004. <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>.
10. Tibor Jager and Juraj Somorovsky. How to break XML encryption. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11: 18th Conference on Computer and Communications Security*, pages 413–422. ACM Press, October 2011.
11. JBoss Community. JBoss WS (Web Services Framework for JBoss AS). <http://www.jboss.org/jbossws>.
12. M. Jones, E. Rescorla, and J. Hildebrand. JSON Web Encryption (JWE) – draft-jones-json-web-encryption-01, oct 2011. <http://tools.ietf.org/html/draft-jones-json-web-encryption-01>.
13. J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.
14. S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.
15. B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), March 1998. Obsoleted by RFC 2437.
16. B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437 (Informational), October 1998. Obsoleted by RFC 3447.
17. Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. Attacking RSA-based sessions in SSL/TLS. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 426–440. Springer, September 2003.
18. James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, August 2001.
19. 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/>.
20. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.
21. E. Rescorla. Preventing the Million Message Attack on Cryptographic Message Syntax. RFC 3218 (Informational), January 2002.

22. 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 Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 199–212. ACM, 2009.
23. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
24. Robert A. van Engelen. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications. <http://www.cs.fsu.edu/~engelen/soap.html>.
25. Nigel P. Smart. Errors matter: Breaking RSA-based PIN encryption with thirty ciphertext validity queries. In Josef Pieprzyk, editor, *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 15–25. Springer, March 2010.
26. Juraj Somorovsky and Jörg Schwenk. Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.
27. Thuan L. Thai and Hoang Lam. *.NET Framework Essentials (2nd Edition)*. O’ Reilly & Associates, Inc., 2002.
28. The Apache Software Foundation. Apache Axis2. <http://axis.apache.org>.
29. Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, April / May 2002.