Padding Oracle Attacks

breaking theoretical secure cryptosystems in the real world

Seminarthesis

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk

RUHR
UNIVERSITÄT
BOCHUM

RUB

Author:	Sergej Sabarnij
Field of study:	IT-Sicherheit/Informationstechnik
Matriculation number:	108007 224687
Advised through:	Juraj Somorovsky

July 27, 2011

Contents

1 Introduction

Computer security is said to rest on *confidentiality, integrity* and *availability* [Bis02]. There are lots of mechanisms which provide these aspects. For example, strong encryption algorithms like DES [FIPB], AES [FIPA], SERPENT [ABK] were introduced to resolve the confidentiality issue. On the other hand, algorithms like SHA-FAMILY, TIGER [AB96], WHIRLPOOL [RB] are supposed to guarantee the integrity. Combining them should keep the messages secret and and provide integrity. However, a poor conceived combination of cryptographic primitives can (and probably will) end in a compromised system and/or data leakage.

Even if a cryptographic method has been proven to be secure in a certain model, this model may still be broken in the real world. There are many examples of security primitives that are strong in theory but are actually not used in any real system. One part of them may provide a good or even *perfect secrecy* (like One-Time-Pad) [KL08] but have such heavy disadvantages that are overweighting all positive aspects.

Another part of primitives is not used because the model, they were designed for, does not hold anymore. This happened, for instance, to the DES, which was designed back in 1976 when the computational power was quite low.

In general, it is not trivial to determine a proper scope of a security model. Over the time it became clear that *perfect secrecy* is not sustainable. Instead, *computational security* [KL08] is considered to be the solution. This model allows an adversary to have an advantage which, however, is so small that the owner of information is agreed to take that risk. In this scenario an attacker is allowed to have certain computational resources which are not enough to get the secured system compromised.

The situation changes, if the adversary breaks this assumption and gains an access to another sort of advantage. The advantage we will be talking about in this work is a sort of a logical *side channel*. In the attack described here, the information obtained through the side channel breaks down the computational power required to compromise a system to $128 * b$ oracle calls, where b is the amount of bytes in a block cipher. This is a very significant reduction compared with the 2^k, where k is the length of the key in bits. As we can see, the complexity of the attack in not even related to the length of the key in use.

Even though, a *side channel* is mostly associated with some sort of physical characteristics like power consumption or electromagnetic radiation, it is not what we are going to discuss here. Our primary goal is to examine the responses of some encryption/decryption server and based on them to reconstruct the message. This behavior is in some sense related to the attack presented by Bleichenbacher at Crypto '98 [Ble98].

Initially, the possibility to misuse the padding property was presented at EUROCRYPT 2002 by Serge Vaudenay [Vau02]. He discovered that given an encrypted message, a decryption node will begin with checking the padding on being well formed after decryption. If the padding is fine, the server will proceed with the usual routine. If, however, the padding is corrupted, an error message will be sent back. Distinguishing between valid and invalid-messages an adversary can easily generate well formed ciphertext without having the key or to completely decrypt the communication.

After that publication many cryptographic systems were proved to have security flaws based on the padding issue. This work is supposed to explain the attack and to provide an overview of systems/protocols affected. Further, we will have a look on some examples of successful exploitation and try to summarize the countermeasures. Please note that we will neither provide a step-by-step description of the exploitations, nor attack any particular system. Rather we will try to show the wide possibilities of this security flaw and to arouse interest for this matter.

The work is organized as follows: first of all we will have a look on theoretical concepts important for understanding the attack. Next, these concepts will be combined to describe the attack itself. After that we will find out real-life systems that are affected. We will try to estimate the impact on security due to this flaw. At the end we will consider the countermeasures and summarize the work in the conclusion.

2 Foundations

This chapter provides some theoretical background about cryptographic primitives and procedures used in this work. Particularly the CBC mode of operation and padding will be described in greater details.

2.1. CBC

For the most readers CBC-mode is not new. Nonetheless, we will recall its important aspects to gain an understanding of security limits of this mode.

One of the problems in the private key cryptography is that block ciphers are designed to work only on a certain amount of bits as input. For example AES works on 128-bits states, while DES needs 64 bits.

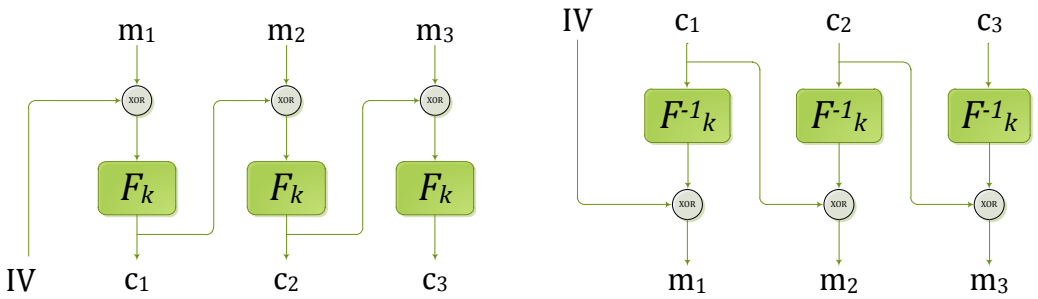
Modes of operation are supposed to solve this problem and allow to work on messages with greater length. It is clear that this solution has to provide both: security and low *ciphertext expansion* [KL08].

For simplicity we assume that the length of the message is exact a multiple of the block size. If not, *padding*, which is described in the next section, is applied.

There are a few modes of operation, the one we make use of in this work is called *CBC* (Cipher Block Chaining). Figure 2.1 provides a schematic overview of it.

In short notion, there are following steps necessary [KL08]:

- A random initial vector (IV) of block length is chosen: $IV \in_R \{0,1\}^n$.
- Each new ciphertext is generated by applying the XOR-operation to the current plaintext block and the previous ciphertext block: $C_i = Enc_K(C_{i-1} \oplus M_i)$.



(a) (b)

Figure 2.1.: Cipher Block Chaining (CBC). (a) Encryption, (b) Decryption.

- Decryption is achieved by applying XOR to the next previous ciphertext block and decryption of the current ciphertext block: $M_i = C_{i-1} \oplus Dec_K(C_i)$.

Formally we could use following algorithms 2.1 (CBC-Encryption) and 2.2 (CBC-Decryption) [Wö3] to describe the operation.

Algorithm 2.1 CBC-Encryption

Require: k-bits large key K, n-bits IV, n-bits plaintext blocks $M_1, ...M_t$
Ensure: n-bits ciphertext blocks $C_1, .., C_t$
1: $C_0 \Leftarrow IV$
2: **for** $i = 1 \rightarrow t$ **do**
3: $\quad C_i \leftarrow Enc_K(M_i \oplus C_{i-1})$
4: **end for**

Algorithm 2.2 CBC-Decryption

Require: k-bits large key K, n-bits IV, n-bits ciphertext blocks $C_1, ...C_t$
Ensure: n-bits plaintext blocks $M_1, .., M_t$
1: $C_0 \Leftarrow IV$
2: **for** $i = 1 \rightarrow t$ **do**
3: $\quad M_i \leftarrow C_{i-1} \oplus Dec_K(C_i)$
4: **end for**

We can easily prove the correctness:

$$C_{i-1} \oplus Dec_K(C_i) = C_{i-1} \oplus Dec_K(Enc_K(M_i \oplus C_{i-1})) = C_{i-1} \oplus M_i \oplus C_{i-1} = M_i$$

Often the IV($= C_0$) is a part of the ciphertext vector and is transmitted along with the other encrypted blocks: $\boldsymbol{C_0}, C_1, ..., C_t$, so it has not necessarily to be secret. Unfortunately, this behavior may leak some information. So one can clearly see if the same IV has been used multiple times just by looking on the prefix block of two ciphertexts.

After CBC-encryption all blocks are functionally dependent on each other, because each previous block is used to proceed with the current. This feature has the advantage of hiding statistical information all over the ciphertext. On the other hand, encryption can not be proceeded in parallel, which results in performance loss. Further, ciphertext blocks are not allowed to switch orders or to loss. Each block has a dedicated place and any manipulation makes decryption of some parts impossible.

For example, flip of only one bit of ciphertext, result in loss of information while decrypting. Let's say, a block C_i was transmitted with an error and got C'_i. Note that C_i and C'_i differ only in one bit. In that case, the avalanche effect makes the decryption $Dec_K(C')$ look completely random. If the next block C_{i+1} was transmitted well, the decryption M'_{i+1} will get an error on exact the place C'_i. This is called *error propagation* [Mit05].

On the other hand we can say that an error in a single block has an impact only on messages M'_i and M'_{i+1}. We call this feature *self-synchronizing* [Zda]. The same applies to the loss of a whole block.

Another positive aspect of it is that CBC-mode is efficient. It can work on an infinitive string of information within linear time and using constant memory. An exhaustive search, however, is as complex as a process of breaking the secret key.

2.2. Padding Schemes

As we have already mentioned earlier, not always the length of plaintext is a multiple of block length. To produce the needed length, a well-known technique called *padding* is applied. By padding we just add some extra information to the message in order to expand it to the proper length.

There are several methods how to pad a message. The most common are PKCS#5 [Kal00], ISO/IEC 9797-1 [ISOb], ISO/IEC 10118-1 [ISOa] and some more.

PKCS#5: The PKCS#5-standard has the name "Password-Based Cryptography Standard" and is stated in the RFC2898 [Kal00]. According to it, the padding scheme has to fill the rest of the last N words of the block with values equal to N.

In other words, we just add as much N bytes with the value N as much we need to achieve the length of the message multiple of block length. Let us have a look at an example. Suppose, the block length is 8. If so, following padding bytes are added:

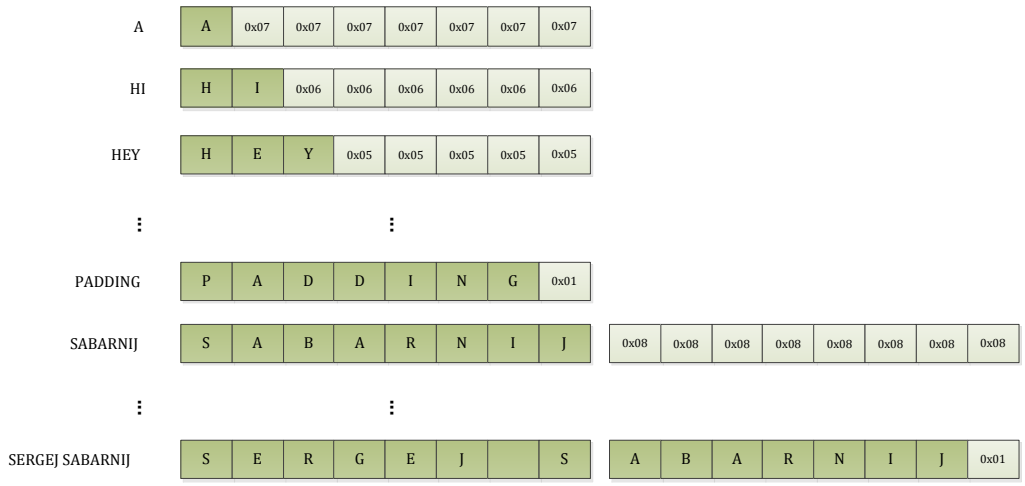


Figure 2.2.: PKCS#5 Padding Example

Please refer to the standard itself or to the page A.1 of the appendix for more information.

PKCS#5 is one of the most common padding schemes and is referenced in many RFC's. Nonetheless, there exist slightly different variants of this padding scheme. So Schneier introduces in his book [Sch96] the possibility to pad with `000...n` bytes. The standard for IP Encapsulating Security Payload (ESP), RFC2406[1] suggests to use `123...n` bytes instead of `nnn...n`.

An important thing in this context is the dealing with the padding after decryption. We assume that the server on receipt of the ciphertext decrypts it and checks if the padding is valid. If it is not, the routine classify such messages as corrupted and drops them producing an error message. On the other hand, if the padding is valid, the routine proceeds working on the message. The error messages are the crucial basis for the attack.

ISO/IEC: Not only PKCS#5 padding is used in encryption schemes. ISO/IEC also provides padding methods. There are two standards describing them: *ISO/IEC 9797-1*

[1] See page vii of Appendix

and *ISO/IEC 10118-1*, each containing three techniques. However the first two techniques overlap in both standards, so we have got effectively only four different paddings.

We will not provide any further information about them here because our primary goal lies on the PKCS#5 padding. However, an interested reader should know that the attack is not only restricted by one single method, but is applicable to any scheme where the padding can be removed unambiguously. Please refer to the standards directly to get more information.

3 Attack

The next few pages will familiarize the reader with the attack. It will make use of previous chapters and point out why this kind of security flaw is difficult to heal.

3.1. Padding Oracles by Vaudenay

Serge Vaudenay has presented the Padding Oracle Attack in the year 2002 at EUROCRYPT [Vau02]. His paper explained how the attack works and proposes first fixes. Later, a few other researches have presented their results on this topic claiming many improvements. In this chapter we will have a look at initial attack presented at EUROCRYPT'2002.

The attack presented here lies on two reasonable assumptions:

- An adversary can eavesdrop the communication and intercept the CBC-encrypted ciphertext
- An adversary has access to a *padding oracle* $\mathcal{O}$, which means he or she can distinguish between VALID and INVALID paddings.

The second point addressed here may be based on the response of the server. For example, the server may answer with following messages [Hol]:

- When the decryption of a message results in valid data with valid padding, then no error message occurs and `200 OK` or something appropriate is sent back.
- When the decryption of a message results in valid padding but corrupt data, then some custom error message is sent, but the general response may be still `200 OK`

- When neither a valid message, nor valid padding was decrypted, then we get a cryptographic exception and a response like `500 Internal Server Error`. The JSF view states return, for instance, `javax.crypto.BadPaddingExceprion: Given final block not properly padded`.

As we can see, an adversary can easily distinguish between error messages, so he or she can gain access to a padding oracle $\mathcal{O}$.

3.2. Decryption

Let b be the block length in words. For example AES has $b = 16$, DES has $b = 8$ and so on. W is the number of possible words and N is the total number of b-byte blocks to decrypt/encrypt. For example, `PADDING ORACLE ATTACK` consists of three 8-byte blocks `PADDING_||ORACLE_A||TTACK` or of two 16-byte blocks `PADDING_ORACLE_A||TTACK`. So in the first case $N = 3, b = 8$, in the second $N = 2, b = 16$.

Further we say that a block sequence $x_1, x_2, ..., x_N$ has correct padding, if the last bytes of the last block are set according to the PKCS#5 standard, i.e. `0x01, 0x02 0x02, 0x03 0x03 0x03 ...` The final decrypted message with the padding attached is the actual cleartext and is noted here as $m_1...m_b$. The intermediate step **after** the decryption but **before** XOR-ing it with the previous block C_{i-1} is noted here as $a_1...a_b$.

First of all we will introduce the *Last Word Oracle*. This oracle helps to obtain the last word of $C^{-1}(y)$ for any block y. To do so we choose a block $r = r_1, ..., r_b$ containing of random words. A forged ciphertext is generated by concatenation of r and y and sent to the oracle. If $\mathcal{O}(r||y) = \mathsf{VALID}$ then $C^{-1}(y) \oplus r$ ends with a valid padding. It is clear that the padding `0x01` has the best probability to occur, which leads to the last word of $C^{-1}(y)$ being $r_b \oplus 1$.

If, however, the oracle answered with INVALID, we compose another ciphertext $r'||y$ and try again sending it to the $\mathcal{O}$. The maximal amount of tries is equal to the number of all possible words W, so we will get a VALID-message after $W/2$ requests on average. Since W is most likely 2^b, we will have to try only $2^8 = 256$ times for DES maximally.

Less likely but still possible are paddings `0x02 0x02`, `0x03 0x03 0x03` and so on. The probabilities are $1/2^{16}, 1/2^{24}, ...$ accordingly. Therefore we have to check if the VALID-padding returned is one of these "longer" paddings. To do so we just start modifying the bytes from the top of r and observe if the oracle changes the output. If by changing the

left most byte $\mathcal{O}$ bekomes INVALID then this byte was also a part of the padding, stating that the whole block consists only of padding bytes. In this case only `0x08 0x08 ... 0x08` comes into question. By still VALID alter the next word and inspect the output again.

Algorithm 3.1 (Last Word Oracle) was proposed by Vaudenay to discover the last byte(s). It does exactly what we have described above and returns the last word of $C^{-1}(y)$.

Algorithm 3.1 Last Word Oracle

Require: Ciphertext block y, padding oracle $\mathcal{O}$
Ensure: Last word of $C^{-1}(y)$

1: $r_1...r_b \Leftarrow$ random words, $i = 0$
2: $r \leftarrow r_1...r_{b-1}(r_b \oplus i)$
3: **if** $\mathcal{O}(r||y) =$ INVALID **then**
4: $\quad i \leftarrow i + 1$
5: $\quad$ go back to the step 2
6: **end if**
7: $r_b \leftarrow r_b \oplus i$
8: **for** $n = b \rightarrow 2$ **do**
9: $\quad r \leftarrow r_1...r_{b-n}(r_{b-n+1} \oplus 1)r_{b-n+2}...r_b$
10: $\quad$ **if** $\mathcal{O}(r||y) =$ INVALID **then**
11: $\quad\quad$ stop and output $(r_{b-n+1} \oplus n)...(r_b \oplus n)$
12: $\quad$ **end if**
13: **end for**
14: output $r_b \oplus 1$

Once we have obtained the last word of a block, we can proceed with the decryption of the previous a_{b-1} word. By iterating the method above we can reconstruct the whole block $C^{-1}(y)$. Algorithm 3.2 (Block Decryption Oracle) introduces the needed steps. In the algorithm we assume to already have the last words $a_j...a_b$ of the block for some $j \leq b$. We intend to decrypt the next unknown word a_{j-1} and consecutive $a = a_1...a_b$.

The complexity is still bounded by $W/2$ trials on average for each new word. So if we need to recover the whole block of b words, we have to send about $b \cdot W/2$ requests.

Let us recall what we have got so far. We have a Last Word Oracle which returns the last word of each b-words block. Further we will bother the Block Decryption Oracle to reconstruct the whole b-words of the block. The goal is, however, to decrypt the whole message consisting of $y_1...y_N$ ciphertext blocks. But it is quite easy considering decrypting oracles we already have. The idea is to reconstruct the message iteratively

Algorithm 3.2 Block Decryption Oracle

Require: Ciphertext block y, padding oracle $\mathcal{O}$
Ensure: Decrypted word a_{j-1}
1: $r_k = a_k \oplus (b - j + 2)$ for $k = j, ..., b$
2: $r_1 ... r_{j-1} \Leftarrow$ random words, $i = 0$
3: $r \leftarrow r_1 ... r_{j-2}(r_{j-1} \oplus i) r_j ... r_b$
4: **if** $\mathcal{O}(r||y) =$ INVALID **then**
5: $\quad i \leftarrow i + 1$
6: $\quad$ go back to the previous step
7: **end if**
8: output $r_{j-1} \oplus i \oplus (b - j + 2)$

using the both algorithms. The number of steps amounts to $N \cdot b \cdot W/2$ oracle calls on average. Note that each oracle call requires two blocks: a random r-block and a ciphertext block.

3.3. Encryption

Juliano Rizzo and Thai Duong noticed that not only a decryption is possible using padding oracles, but an encryption as well. They presented their paper on Blackhat Europe 2010 [RD10b]. In this paper the encryption oracle is called CBC-R and the described method allows to encrypt any messages without knowing the secret key (by an adversary). Here we will describe the CBC-R technique.

Recall how CBC-decryption works:

$$P_i = Dec_K(C_i) \oplus C_{i-1}$$
$$C_0 = IV$$

If the attacker possesses C_{i-1} and $Dec_K(C_i)$, then he or she actually can control P_i, which is the plaintext block. Since the block C_{i-1} is not bounded to anything, an adversary can choose it arbitrarily. What adversary needs is the second part of the equation: $Dec_K(C_i)$. Fortunately for him or her, that's what decryption oracles are useful for. Shaping a query to the $\mathcal{O}$ the decryption can easily be completed. We denote such decryption in the presence of an oracle as $Dec_K^{\mathcal{O}}(C_i)$.

Since the both requirements are met, the encryption can be started. The process is following:

1. An attacker picks some random ciphertext block C_i out of the transmission he has intercepted

2. An attacker sends C_i to the oracle to obtain the plaintext: $Dec_K^{\mathcal{O}}(C_i)$

3. By direct manipulation of C_{i-1} a required P_i is forged

Let us have a look on an example. Suppose an adversary is willing to forge a particular P_x. All he need to do is to set

$$C_{i-1} = P_x \oplus Dec_K^{\mathcal{O}}(C_i).$$

Unfortunately such operation will end in a garbled block P_{i-1}, since C_{i-1} was set arbitrarily. However we can heal it by manipulating C_{i-2}:

$$C_{i-2} = P_{i-1} \oplus Dec_K^{\mathcal{O}}(C_{i-1}).$$

Now we have constructed two plaintext blocks P_{i-1} and P_i without any key. The previous block P_{i-3} is still garbled. Applying the technique from above iteratively we can generate ciphertexts for any desired plaintexts, starting from the last block. Algorithm 3.3 (Block Encryption Oracle (CBC-R)) describes the approach in pseudocode.

Algorithm 3.3 Block Encryption Oracle (CBC-R)

Require: Plaintext message P
Ensure: Encrypted blocks $C = Enc_K(P), IV$
1: Divide P into blocks of b bytes: $P_0, P_1, ..., P_{N-1}$
2: $r_1, r_2, ..., r_b \Leftarrow$ random words
3: $C_{n-1} \leftarrow r_1||r_2||...||r_b$
4: **for** $i = N - 1 \rightarrow 1$ **do**
5: $\quad C_{i-1} \leftarrow P_i \oplus Dec_K^{\mathcal{O}}(C_i)$
6: **end for**
7: $IV \leftarrow P_0 \oplus Dec_K^{\mathcal{O}}(C_0)$
8: $C = C_0||C_1||...||C_{n-1}$

3.4. The IV-Problem

So far we have left one problem untouched, the problem with the IV. If the IV of the whole message is secret, we can not obtain it. In this case the decryption of the first block is impossible. There are workarounds for different systems, but in general this problem remains unsolved. On the other hand, one single block out of N can hardly keep the secrecy of the whole message. Further, by XOR-ing the two first plaintext blocks with the same IV we can even get the decryption up to some unknown constant.

This problem remains for the CBC-R encryption as well. If an attacker can control IV, he or she can perform the complete encryption without any garbled blocks. If, however, the IV is fixed, the first block stays garbled. In general, there is no need to keep IV secret, but some schemes do so. As described in Section **Foundations** on the page 3, different approaches are used in the real systems:

- IV can be a part of the ciphertext and is therefore fully controlled by the adversary
- IV may be a fixed well known value, which is not influenced by the adversary
- Some systems keep the IV secret and fix
- Some systems keep the IV secret and change it eventually

It is clear that the first way is the most favorable by an attacker. However, there are workarounds for some cases even if the IV is not controllable. One way to overcome the trouble with the IV is to use an existing (captured) ciphertext as a prefix. It is useful if the server expects some sort of a header. By prepending the message

$$P_{valid} = Dec_K(C_{captured}||IV_{CBC-R}||P_{CBC-R})$$

we can force valid header to be accepted. Note that the middle block is still garbled. However, we can try to make it a part of some semantically nonrelevant part of a message. For example we can try to make it a comment or a label.

Another solution could be to brute force the C_0. Recall the step 3 of Algorithm 3.3. According to it C_{n-1} becomes random words. Any time when it is altered, the whole chain $C_{n-1}...C_0$ is altered as well. The plaintext, however, remains unchanged: $P_{n-1}...P_1$. Further, P_0 is composed as follows:

$$P_0 = Dec_K^{\mathcal{O}}(C_0) \oplus IV$$

Since the IV is fix and P_0 should meet some requirements, we only can change $Dec_K^{\mathcal{O}}(C_0)$. In this case we alter the chain $C_{n-1}...C_0$ so long, until P_0 eventually meets the needed requirements. Sometimes the first block has just to match some magic number, sometimes it has to be the length of the message. In any case, if the requirements are not very hard, the ciphertext can be forged efficiently.

3.5. Other Padding Schemes

As we have already mentioned in Section 2.2 Foundations on the page 5, not only the PKCS#5 padding is affected by this vulnerability. The ISO/IEC padding schemes are vulnerable as well. In their paper [PY04, YPM05] Paterson and Yau present an analysis and an attack on these paddings. Further Black and Urtubia [BU02] discuss the vulnerability of a few more padding schemes.

We are not going to explain all these attacks in this work since the mechanism is always the same: you use the padding oracle to gain a side channel to overcome the encryption protection. The attack on PKCS#5 discussed earlier should provide you the necessary background to understand the problem.

4 Applications

As the main point of this work lies not only on describing the security flaws, but rather in describing its applications as well, this chapter will pick out and discuss some real-life attacks. To do this we have analyzed many of authentic and regarded sources. To round up the discussion we also will evaluate some non academic sources posted on the Internet.

4.1. JSF view states

The forgery of JavaServer Faces was introduced by Rizzo and Duong in their paper [RD10b] in 2010. JSF is a powerful web applications-framework designed to help developers on creating user interfaces. Since HTTP-protocol is stateless, the application has to store the states somehow. The initialization parameter `javax.faces.STATE_SAVING_METHOD` controls this issue. It can be set either to `server` other to `client`. The last method is very popular by developers. By setting

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

the stored views will be handled by clients in a hidden field. The state information is sent each time to the server, so the JSF can restore the view without need to store any information on the server side.

By default the view state is protected with the encryption. The developer is, however, free to change this strategy by changing the `org.apache.myfaces.USE_ENCRYPTION` parameter:

```
<!-- To enable encryption -->
<context-param>
    <param-name>org.apache.myfaces.USE_ENCRYPTION</param-name>
    <param-value>true</param-value>
 </context-param>

<!-- To disable encryption -->
<context-param>
    <param-name>org.apache.myfaces.USE_ENCRYPTION</param-name>
    <param-value>false</param-value>
 </context-param>
```

Here is an example of using the AES encryption with the view states:

```
<context-param>
    <param-name>org.apache.myfaces.SECRET</param-name>
    <param-value>MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIz</param-value>
</context-param>
<context-param>
    <param-name>org.apache.myfaces.ALGORITHM</param-name>
    <param-value>AES</param-value>
</context-param>
<context-param>
    <param-name>org.apache.myfaces.ALGORITHM.PARAMETERS</param-name>
    <param-value>CBC/PKCS5Padding</param-value>
</context-param>
<context-param>
    <param-name>org.apache.myfaces.ALGORITHM.IV</param-name>
    <param-value>NzY1NDMyMTA3NjU0MzIxMA==</param-value>
</context-param>
```

As we can see in this example, it uses AES algorithm in CBC-mode which is vulnerable to the padding oracle attacks. Further, the IV and password are given encoded with the *Base64* method.

Let us have a look on how to exploit Java states. If the developer has not changed the configuration, the JSF framework will provide a detailed error message in case the decryption fails. So on receipt of the response

```
javax.crypto.BadPaddingException: Given final block not properly padded
```

it is a clear evidence for $Dec_K^{\mathcal{O}}(P) = \mathsf{INVALID}$.

Java EE provides the possibility to disable this kind of error messages. The authors, however, describe a workaround as follows: if an attacker is willing to decrypt some block C_i of the ciphertext $C_0||C_1||...||C_{n-1}$ then he or she just append some random block with C_i to the ciphertext. I.e. the string becomes $C_0||C_1||...||C_{n-1}||C_{random}||C_i$. Since JSF just ignore the extra blocks while decrypting the state, the server will proceed as if there was not any change of ciphertext. We can interpret it as VALID. On the other hand, if the server returns an error, it is probably because of the padding issue and we can mark the response as INVALID.

Usually, view states do not contain any sensitive information except for some cases. Therefore the impact of the decryption is not very big. However using padding oracle we can encrypt messages as well. Please recall section **Attack** on the page 3.3. According to it, an attacker can generate malicious view states with the CBC-R method. By doing so, he or she can even run a malicious code on an affected system.

Since exploiting of view states is not a part of this work, we will not describe here any details on possible attack. For further information please turn to special sources dedicated to this theme. A good start may provide David Byrnes "Multiplatform View State Tampering Vulnerabilities"[2].

4.2. Decrypting CAPTCHAs

CAPTCHA is a widely used method of preventing automated queries to web services. Usually it employs a small image with some characters on a distorted background which user has to input in a special field. Since OCR is stumped with such kinds of job, only a human can easily recognize and "authenticate" himself. Rizzo and Duong found out that some CAPTCHAs can be misused as padding oracles. A possible scenario for such a system may be the following:

[2] https://www.trustwave.com/spiderlabs/advisories/TWSL2010-001.txt

1. A random code is generated and encrypted using some key and IV:

$$ERC = Enc_{K,IV}(rand())$$

2. The ERC will be used by some CAPTCHA generator, which decrypts it and generates a distorted image. An HTML code provides an `<img>` area and a text field.

3. Since HTTP is stateless, the value of ERC has to be stored and submitted each time. If it is stored on the client side, which is widely used, a cookie or some hidden field will be used.

4. On the receipt of ERC and users response, the web application decrypts the value and compares it with the input of the user. If they match, request will be accepted. It will be denied otherwise.

As we can see, any request will be proceeded by the server, since no prior authorization take place. The receiver accepts any ERC and tries to decrypt it. If an attacker can distinguish between VALID and INVALID, the requirement for padding oracle is met.

Fortunately for him or her the distinction is very likely. If the CAPTCHA generator shows a distorted image with broken letters, the padding is probably VALID. If, however, an error message comes back or even empty response, the answer is probably INVALID.

If the padding oracle works well, the attacker is able to completely bypass the CAPTCHA protection regardless to its complexity. Note that the problem with the IV still remains. Since

$$P_0 = IV \oplus Dec_K^{\mathcal{O}}(C_0)$$

there is no easy way to decrypt the first block without it. Likely for the attacker, the content of this first block is very often equivalent to the broken code shown in the CAPTHCA-image. With other words $P_0 = image\_content$. If this assumption holds, IV can be easily determined by

$$IV = image\_content \oplus Dec_K^{\mathcal{O}}(C_0).$$

The $image\_content$ should be recognized by a human and inserted manually. At first glance the approach may seem odd, because the goal of attack was to overcome the need of user entering something. I.e. we try to fool the protection system bypass the "authorization as a human". On close examination, however, we can state that the IV is

not changed very often. This means that given IV unchanged, we only need to read the CAPTCHA-letters once and can proceed on decrypting as long as we need without any further intervention.

4.3. Distributed Cross-Site Attacks

As we already know, an attacker only need one single bit of information for performing padding oracle attacks. By default there are policies prohibiting the information to be shared between different domains (*same origin policy*). Nonetheless, there exist browser bugs allowing to misuse the events like `onError()/onLoad()`. If an attacker on `evil.com` can reference resources on `victim.com`, he can observe the responds of the server.

Suppose, JavaScript on `evil.com` can force the browser to load an image at `victim.com`. In this case the script can determine whether the image was loaded successfully or not. This observation can be interpreted as VALID or INVALID. As a consequence, the query can be distributed to different systems resulting in more powerful decryption. Moreover, by inserting the malicious JavaScript code on popular web pages, an adversary can force their visitors to perform malicious activities at their own costs. It seems to be a perfect distributed padding oracle attack.

4.4. SSL/TLS

TLS specification prescribe to use the PKCS#5 padding. To find out more please turn to the specification itself or to appendix page A.2. According to it the length of the padding may be longer than the block length. It is due to hide the actual length of the message. Nonetheless, it is still bounded to $W - 1$, where $W = 2^8 = 256$.

Optionally, the standard advises to make use of the MAC-function, which, however, does not prohibit padding oracle attacks. The problem here is that padding is attached *after* the MAC-algorithm. This means that the server *at first* checks the padding and only after that the MAC-integrity, eventually throwing a `bad_record_mac` error.

The SSLv3.0 has improved this handling and throws the same error for both exceptions. Another security feature used in SSL/TLS is that the pad padding exception ends in a fatal error and connection *must* abort. Vaudenay describes this behavior as server "explodes" and proposes a *bomb oracle* to partly overcome this mechanism. To make

the description short, a bomb oracle help to decrypt the last word of a block with the probability W^{-1}, the two last words with W^{-2} and so on.

4.5. IPSEC

IPSEC can implement the CBC-PAD and may therefore be vulnerable to padding oracle attacks. The default padding scheme is similar to specified in the ESP-standard. Turn to the standard itself or have a look on the page A.3 of Appendix for more details. The most important parts of it prescribe the padding of the length $0 - 255$ consisting of monotonically increasing sequence `0x01, 0x02, 0x03 ... 0xn`. Although the scheme differs from the PKCS#5, it is still vulnerable to our attack.

The EPS standard explicitly mentions that the padding has to be checked before proceeding with the working processes. However, the exception handling is not mentioned. Therefore there may be different behaviors for the attack. On the other hand, not proceeding with the work or even an error message leak a bit of information we need.

Optionally, the standard offers an additional authentication mechanism. If used, it protects the system from being exploited by our attack. However, it is still an optional mechanism which is not implemented everywhere.

4.6. WTLS

Wireless Transport Layer Security protocol is an instance of padding oracle. If error occurs, it sends `decryption_failed` in clear. Since this protocol is used mostly on the mobile devices, the resources to be exploited by an attacker are bounded. Therefore the attacker should better limit the number of connection tries in order to avoid fatal errors. The limitation of the query can result in slower decryption, but nonetheless it works.

Some implementations of WTLS may reduce the amount of failed connection tries to some number. In this case the attack will be limited. On the other side, this kind of limitations are non standard and are not widely used.

On the positive side stays that some devices may encapsulate the WTLS protocol in other higher level protocols. If so, an attacker has to overcome this extra layer to drive a padding oracle attack.

4.7. SSH2

The SSH2 protocol employs a different kind of padding. According to the standard, the padding field is filled with random data. The last word, however, contains the length of the padding. Therefore the attack can only recover the last word of the plaintext, which is not much. Moreover, the MAC algorithm is proposed and protect against the attack.

4.8. ASP.NET

As it has been posted by Duong and Rizzo [RD10a], the products of Microsoft are also vulnerable to padding oracle attacks. For example, SharePoint uses encrypted ASP.NET ViewState objects. If an attacker succeeds in decryption the information sent from server to the client, he or she can gain access to passwords or other confidential connection settings.

The Exchange2007 is also reported to be vulnerable, as well as Exchange2010. The last, however, may not content as much sensitive information as the the provious, so the impact is considered to be smaller.

The attack on the latest ASP.NET framework v4.0 allows an attacker to

> "steal cryptographic secret keys, forge authentication tokens and destroy the security model of every ASP.NET v4.0 application."[3]

Moreover, an attacker can also generate valid ciphertext which will be accepted by the server an will compromise the whole internal integrity if the system. The attack goes as follows:

- A special file named `web.config` stores the cryptographic keys and other sensitive information. If an attacker can gain a access to it, the keys stored in plaintext became compromised.
- A handler called `WebRessource.axd` processes the queries and return the resource to the client. It waits for a special query string which is of the form

  `WebResource.axd?d=`*`encrypted identifier`*`&t=`*`timestamp value`*

[3] [RD10a]

The parameter `d=` is crucial in the attack. If an adversary succeeds in forging it, he or she can gain access to the file system and download arbitrary files like `web.config`

- We penetrate the `WebRessource.axd` as long as it takes to forge the valid parameter `d=`. Due to different error messages, on can distinguish the padding issues. If the padding is VALID, the server throws a `404` HTTP error. It will throw `500` HTTP error otherwise.
- After an attacker has set up the decryption oracle, he uses the CBC-R technique to forge a valid `d=` parameter.

The approach is not very costly. The authors calculate the average amount of HTTP requests to 10420. After them they poses the `web.config` file and can drive further impersonating attacks. Therefore, they claim to get the server completely compromised within the shortest amount of time. The average time needed is about 30 minutes. The longest period of breaking was about 50 minutes which is still critically small. Moreover, the attack is 100% reliable and ASP.NET is widely used.

Another interesting point to stress here is that the attack is completely stealthy. Wrong submitted parameters usually do not result in a large log record. Therefore the above mentioned 10420 requests stay unnoticed. Only after a great damage has been inflicted, the attack may be detected.

After the publication Microsoft has released the patch which remedies the vulnerability. After days of work around the clock, the problem is no longer present. But the intensity of work on this issue shows how badly the problem was.

5 Conclusion

This work provided an overview of the powerful attack called *padding oracle attack*. It describes that due to one single bit of information leaking from the server, an attacker can decrypt the intercepted communication. The side channel used here is a padding oracle which returns whether the padding was VALID or not. Collecting those responses, a ciphertext can be de- or encrypted without knowing the block cipher, IV and, of course, the key.

Further we have had a look on applications affected by this error. There is a wide spectrum of applications, starting by simple CAPTCHA protection systems, to the most widely used frameworks like ASP.NET. There is no doubt that even more systems may be affected, even if no other vulnerabilities have been published yet.

Along with their papers many authors have offered some fixes to the problem. Some of them we will provide here, in the conclusion, but in general there is no easy way to heal all security issues related to padding oracles. The most successful solution was proposed by Vaudenay in his paper [Vau02]. He proposed to change the encryption strategy in order to make the communication more secure. That is, to do the following: pad the cleartext, then authenticate and encrypt it. Transmit the result. As an opposite, the decryption process should first of all decrypt and check authenticity of ciphertext and than check the padding of it. After all, remove the padding and proceed working on the message. As we can see, authentication is one of the important concepts here. If the message is not authentic, it's no use working on it at all.

On the other hand, it may not always be acceptable to introduce an extra piece of cryptography to authenticate messages. Imagine applications where transmission speed it needed, like VoIP or data transfer. Employing extra cryptography will irresistible slow

down the connection and by that the acceptance. Further, there may be just not enough resources to perform authentication. Think of embedded devices or the like.

Another possible solution described in [JSJS11] could be to avoid detailed error messages. This approach, however, is not very feasible. The developers of web applications eventually need those in order to fix their implementations. If no error at all comes back, it is very difficult to debug the application.

One can further try to detect an attack by looking for the multiple server requests. More precisely, one can try to filter the messages consisting of exactly two ciphertext blocks. However, The attack may be distributed over the time and all over the world. The requests may come from different IPs, which makes the filtering useless.

In conclusion we can state that the problem of padding oracles has been known for 9 years. Nonetheless it is still relevant to the security perspective. Many attacks exploit this vulnerability even after such a long time. The attacks on ASP.NET show that the problem is not limited to some particular but rather millions of computer systems are affected. This work attempted to show the security flaw itself and to present the wide range of its possibilities. We hope we could arise your interest to this issue.

Bibliography

[AB96] Anderson, Ross J. and Eli Biham: *Tiger: A fast new hash function.* In Gollmann, Dieter (editor): *FSE*, volume 1039 of *LNCS*, pages 89–97. Springer, 1996, ISBN 3-540-60865-6. http://dx.doi.org/10.1007/3-540-60865-6.

[ABK] Anderson, Ross, Eli Biham, and Lars Knudsen: *Serpent.* http://www.cl.cam.ac.uk/~rja14/serpent.html.

[Bis02] Bishop, Matthew A.: *Computer Security. Art and Science.* Addison-Wesley Professional, 2002.

[Ble98] Bleichenbacher, Daniel: *Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1.* In *CRYPTO'98*, pages 1–12, 1998.

[BU02] Black, John and Hector Urtubia: *Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption.* In *USENIX Security Symposium*, pages 327–338, 2002.

[DA99] Dierks, T. and C. Allen: *RFC 2246: The tls protocol version 1.0*, January 1999.

[FIPa] FIPS197: *Advanced encryption standard (aes).* http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[FIPb] FIPS46-3: *Data encryption standard (des).* http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[Gol01] Goldreich, O.: *Foundations of cryptography: Volume 1.* 2001.

[Hol] Holyfield, Brian: *Automated padding oracle attacks with padbuster.* available online on http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-with-padbuster/.

[ISOa] ISO/IEC: *Information technology – security techniques – hash-functions.*

[ISOb] ISO/IEC: *Information technology – security techniques – message authentication codes (macs).*

[JSJS11] Jager, Tibor, Juraj Somorovsky, Meiko Jensen, and Jörg Schwenk: *Character encoding pattern attacks or how to break xml encryption*, 2011.

[Kal00] Kaliski: *RFC2898*, 2000.

[KL08] Katz, Jonathan and Yehuda Lindell: *Introduction to modern cryptography.* Chapman & Hall/CRC Cryptography and Network Security. Chapman & Hall/CRC, Boca Raton, FL, 2008, ISBN 978-1-58488-551-1; 1-58488-551-3.

[Mit05] Mitchell, Chris J.: *Error oracle attacks on cbc mode: Is there a future for cbc mode encryption?* In *ISC*, pages 244–258, 2005.

[PY04] Paterson, Kenneth G. and Arnold K. L. Yau: *Padding oracle attacks on the iso cbc mode encryption standard.* In *CT-RSA*, pages 305–323, 2004.

[RB] Rijmen, Vincent and Paulo S. L. M. Barreto: *The whirlpool hash function.* http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html.

[RD10a] Rizzo, Juliano and Thai Duong: *Cryptography in the web: The case of cryptographic design flaws in asp.net.* In *2011 IEEE Symposium on Security and Privacy*, 2010.

[RD10b] Rizzo, Juliano 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. http://portal.acm.org/citation.cfm?id=1925004.1925008.

[Rot05] Rothe, Jörg: *Complexity Theory and Cryptology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005, ISBN 3540221476.

[Sch96] Schneier, Bruce: *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition.* Wiley, 2nd edition, October 1996, ISBN 0471117099. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471117099.

[Vau02] Vaudenay, Serge: *Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ....* In *EUROCRYPT'02*, pages 534–546, 2002.

[WÖ3] Wätjen, Dietmar: *Kryptographie. Grundlagen, Algorithmen, Protokolle. (Spektrum Lehrbuch).* Spektrum Akademischer Verlag, 1st edition, 2003, ISBN 3827414318.

[YPM05] Yau, Arnold K. L., Kenneth G. Paterson, and Chris J. Mitchell: *Padding oracle attacks on cbc-mode encryption with secret and random ivs.* In *FSE*, pages 299–319, 2005.

[Zda] Zdancewic, Steve: *Lecture notes: Block ciphers, hashes.* http://www2.kinneret.ac.il/mjmay/ise328/328-Lecture4-AESHashes.pdf.

A Appendix

A.1. PKCS#5

The RFC2898[4] prescribes the use of the following padding scheme:

> Concatenate M and a padding string PS to form an encoded message EM:
>
> EM = M || PS ,
>
> where the padding string PS consists of 8-(||M|| mod 8) octets each with value 8-(||M|| mod 8). The padding string PS will satisfy one of the following statements:
>
> PS = 01, if ||M|| mod 8 = 7 ;
> PS = 02 02, if ||M|| mod 8 = 6 ;
> ...
> PS = 08 08 08 08 08 08 08 08, if ||M|| mod 8 = 0.
>
> The length in octets of the encoded message will be a multiple of eight and it will be possible to recover the message M unambiguously from the encoded message. (This padding rule is taken from RFC 1423 [3].)

A.2. TLS Specification

[4] [Kal00]

According to RFC2246 [DA99], TLS uses PKCS#5 padding. Here is the part of the standard describing it.

6.2.3.2. CBC block cipher

For block ciphers (such as RC2 or DES), the encryption and MAC functions convert TLSCompressed.fragment structures to and from block TLSCiphertext.fragment structures.

```
block-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

The MAC is generated as described in Section 6.2.3.1.

padding
: Padding that is added to force the length of the plaintext to be an integral multiple of the block cipher's block length. The padding may be any length up to 255 bytes long, as long as it results in the TLSCiphertext.length being an integral multiple of the block length. Lengths longer than necessary might be desirable to frustrate attacks on a protocol based on analysis of the lengths of exchanged messages. Each uint8 in the padding data vector must be filled with the padding length value.

padding_length
: The padding length should be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length. Legal values range from zero to 255, inclusive. This length specifies the length of the padding field exclusive of the padding_length field itself.

The encrypted data length (TLSCiphertext.length) is one more than the

sum of TLSCompressed.length, CipherSpec.hash_size, and padding_length.

Example: If the block length is 8 bytes, the content length (TLSCompressed.length) is 61 bytes, and the MAC length is 20 bytes, the length before padding is 82 bytes. Thus, the padding length modulo 8 must be equal to 6 in order to make the total length an even multiple of 8 bytes (the block length). The padding length can be 6, 14, 22, and so on, through 254. If the padding length were the minimum necessary, 6, the padding would be 6 bytes, each containing the value 6. Thus, the last 8 octets of the GenericBlockCipher before block encryption would be xx 06 06 06 06 06 06 06, where xx is the last octet of the MAC.

Note: With block ciphers in CBC mode (Cipher Block Chaining) the initialization vector (IV) for the first record is generated with the other keys and secrets when the security parameters are set. The IV for subsequent records is the last ciphertext block from the previous record.

A.3. ESP

2.4 Padding (for Encryption)

Several factors require or motivate use of the Padding field.

- o If an encryption algorithm is employed that requires the plaintext to be a multiple of some number of bytes, e.g., the block size of a block cipher, the Padding field is used to fill the plaintext (consisting of the Payload Data, Pad Length and Next Header fields, as well as the Padding) to the size required by the algorithm.

- o Padding also may be required, irrespective of encryption

algorithm requirements, to ensure that the resulting ciphertext terminates on a 4-byte boundary. Specifically, the Pad Length and Next Header fields must be right aligned within a 4-byte word, as illustrated in the ESP packet format figure above, to ensure that the Authentication Data field (if present) is aligned on a 4-byte boundary.

o Padding beyond that required for the algorithm or alignment reasons cited above, may be used to conceal the actual length of the payload, in support of (partial) traffic flow confidentiality. However, inclusion of such additional padding has adverse bandwidth implications and thus its use should be undertaken with care.

The sender MAY add 0-255 bytes of padding. Inclusion of the Padding field in an ESP packet is optional, but all implementations MUST support generation and consumption of padding.

a. For the purpose of ensuring that the bits to be encrypted are a multiple of the algorithm's blocksize (first bullet above), the padding computation applies to the Payload Data exclusive of the IV, the Pad Length, and Next Header fields.

b. For the purposes of ensuring that the Authentication Data is aligned on a 4-byte boundary (second bullet above), the padding computation applies to the Payload Data inclusive of the IV, the Pad Length, and Next Header fields.

If Padding bytes are needed but the encryption algorithm does not specify the padding contents, then the following default processing MUST be used. The Padding bytes are initialized with a series of (unsigned, 1-byte) integer values. The first padding byte appended to the plaintext is numbered 1, with subsequent padding bytes making up a monotonically increasing sequence: 1, 2, 3, ... When this padding scheme is employed, the receiver SHOULD inspect the Padding

field. (This scheme was selected because of its relative simplicity, ease of implementation in hardware, and because it offers limited protection against certain forms of "cut and paste" attacks in the absence of other integrity measures, if the receiver checks the padding values upon decryption.)

Any encryption algorithm that requires Padding other than the default described above, MUST define the Padding contents (e.g., zeros or random data) and any required receiver processing of these Padding bytes in an RFC specifying how the algorithm is used with ESP. In such circumstances, the content of the Padding field will be determined by the encryption algorithm and mode selected and defined in the corresponding algorithm RFC. The relevant algorithm RFC MAY specify that a receiver MUST inspect the Padding field or that a receiver MUST inform senders of how the receiver will handle the Padding field.

2.5 Pad Length

The Pad Length field indicates the number of pad bytes immediately preceding it. The range of valid values is 0-255, where a value of zero indicates that no Padding bytes are present. The Pad Length field is mandatory.