Padding Oracle Attack on CBC Mode

*Report for *Indian Institute of Technology Bombay* - CS 409M: Introduction to Cryptography (*Autumn 2024*)

Harsh Sanjay Roniyar
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India
22b3942@iitb.ac.in

Swarup Patil
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India
22b3953@iitb.ac.in

***Abstract*—This report highlights and demonstrates a padding oracle attack on CBC (Cipher Block Chaining) mode. The attack exploits the fact that after decryption, the padding on the last block is validated. The attacker sends modified ciphertexts to the padding oracle and observes whether the padding is valid. Through repeated guesses and error analysis, the attacker can determine the plaintext without knowing the decryption key.**

***Index Terms*—cryptography, oracle, padding, cipher block chaining**

I. Introduction

A padding oracle attack in cryptography exploits the process of padding validation to decrypt ciphertext. Since variable-length plaintext messages often need padding to align with the cryptographic algorithm, the attack takes advantage of a "padding oracle" that reveals whether the padding in a message is correct. This feedback, whether explicitly provided or leaked through side-channels, enables the attacker to decrypt the message.

II. Background

The first well-known padding oracle attack was Bleichenbacher's 1998 attack on RSA with PKCS #1 v1.5 padding. The term "padding oracle" became popular after Serge Vaudenay's 2002 attack on CBC mode decryption in symmetric block ciphers. Both attacks are still relevant today. In symmetric cryptography, padding oracle attacks exploit CBC mode, where information about padding can help attackers decrypt or encrypt messages using the oracle's key, without knowing the encryption key. Vaudenay's attack is more efficient than Bleichenbacher's, and while CBC was widely used in SSL and TLS, newer timing attacks have revived the issue. TLS 1.2 introduced authenticated encryption modes to mitigate these risks.

III. The XOR (*Exclusive-OR*)

The XOR is a two-bit operator that compares corresponding bits in two binary inputs. It outputs 1 if the number of input bits that are 1 is odd and outputs 0 if it is even. The operation is mathematically equivalent to addition `mod` 2.

$$(a \oplus b = c) \Leftrightarrow (b \oplus c = a) \Leftrightarrow (c \oplus a = b)$$

Additionally, XOR is cyclic when applied to three bit strings; this means XORing the same three bit strings in any order results in the same output, and applying XOR repeatedly to the same bit strings will eventually return to the original input.

IV. Cipher Block Chaining (CBC) Mode

Cipher Block Chaining (CBC) is a mode of operation for block ciphers where data is encrypted in blocks, with each block XORed with the previous ciphertext block, using an initialization vector (IV) and a single encryption key, enabling secure encryption of large amounts of plaintext. Therefore, CBC mode enhances the security of block ciphers by introducing randomness and interdependency between ciphertext blocks.

A. Encryption in CBC Mode

In CBC, each plaintext block is XORed with the ciphertext of the previous block before being encrypted. Mathematically, this can be described as:

$$C_1 = E(P_1 \oplus IV)$$

$$C_n = E(P_n \oplus C_{n-1}) \quad \text{for } n > 1$$

where P_n is the plaintext block, C_n is the ciphertext block, $E()$ is the encryption function, and IV is the initialization vector, which is typically a random value.

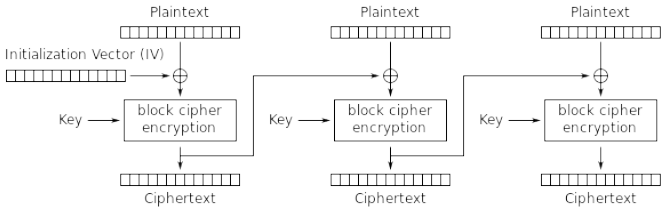

Fig. 1. Cipher Block Chaining (CBC) Mode Encryption

B. Decryption in CBC Mode

Decryption in CBC mode reverses the process. The ciphertext is decrypted, and then the result is XORed with the previous ciphertext block to recover the plaintext:

$$P_1 = D(C_1) \oplus IV$$

$$P_n = D(C_n) \oplus C_{n-1} \quad \text{for } n > 1$$

where $D()$ is the decryption function.

This chaining process ensures that even small changes in the ciphertext propagate through all subsequent blocks, enhancing security. However, this dependency also makes CBC vulnerable to Padding Oracle Attacks.

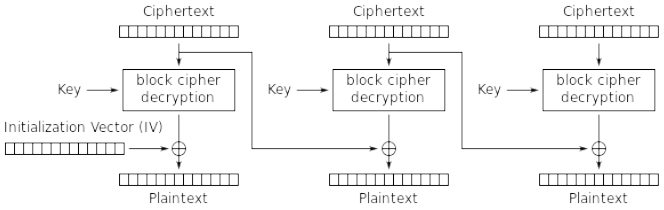

Fig. 2. Cipher Block Chaining (CBC) Mode Decryption

V. Block Ciphers and Padding

A. Block Ciphers

A block cipher is a deterministic algorithm that operates on fixed-length groups of bits, called blocks. Examples of block cipher algorithms include **AES** (128-bit blocks) and **DES** (64-bit blocks).

B. Padding Scheme

The **PKCS #7** padding scheme is a method used in cryptographic encryption to handle data that doesn't fit the exact block size of encryption algorithms like AES, which require inputs to be divided into fixed-size blocks, typically 16 bytes.

The number of padding bytes added is equal to the difference between the block size and the length of the final block. Each padding byte contains a value indicating how many bytes were added. For example, if 6 bytes are needed, each byte will have the value 06 in hexadecimal. Even when the plaintext fits the block size exactly, a full block of padding is still added, with each byte representing the block size.

When decrypting, the value of the last byte indicates how many padding bytes were used, and these bytes are then removed.This padding ensures smooth encryption and is used in secure communication protocols and encryption standards.

If a string is a multiple of the block size, an extra block of padding is added to distinguish cases like "ABC . . . O\x01" (where "\x01" is part of the string) from "ABC . . . O" with padding. Padding is crucial in identifying the end of a message and ensuring that block ciphers process complete blocks.

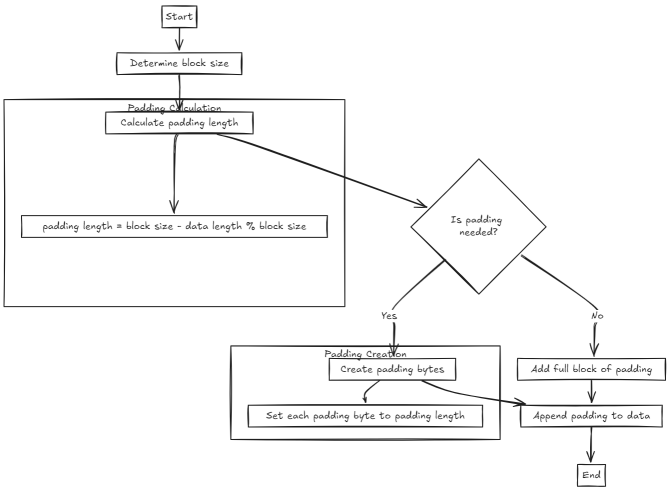

Fig. 3. Illustration of the PKCS #7 Padding Scheme

VI. The Oracle

A Padding Oracle functions as a blackbox that receives an encrypted message as input. Upon receiving the message, the oracle internally decrypts it using a predefined cryptographic key. After decryption, the oracle checks whether the decrypted message conforms to the expected padding scheme, such as PKCS #7 or another padding standard. Based on this validation, the oracle responds with a boolean outcome: it returns 'true' if the message has valid padding and 'false' if the padding is incorrect.

This seemingly simple feedback can be exploited by attackers to iteratively modify the ciphertext and extract information about the plaintext or the decryption key, making padding oracles a critical vulnerability in improperly secured systems.

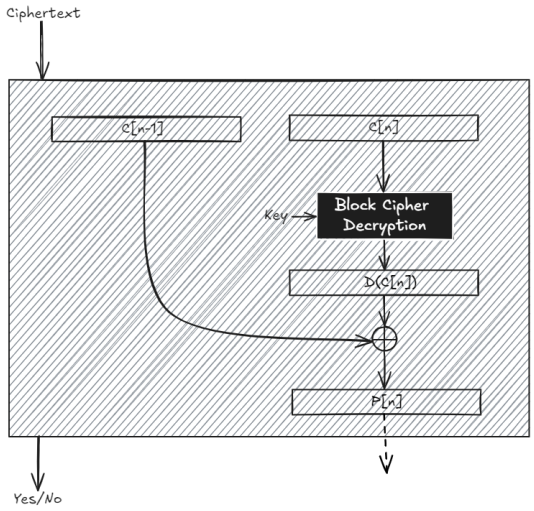

Fig. 4. The Oracle Block

VII. Attack Algorithm

In this section, we present a systematic attack algorithm that leverages vulnerabilities in the padding scheme to recover plaintext from ciphertext in the CBC mode of operation.

The attacker exploits this oracle by crafting specific ciphertext modifications to deduce the plaintext byte by byte.

The algorithm proceeds as follows:

1) The attacker intercepts the ciphertext and divides it into individual blocks. Let $C = (C_1, C_2, \ldots, C_n)$ be the intercepted ciphertext.
2) Starting from the last block, the attacker generates a custom block C' and concatenates it with C_n, sending the modified ciphertext $(C'||C_n)$ to the oracle.
3) The oracle decrypts the message and checks the padding. If the padding is valid, the attacker knows that the last byte of the plaintext has the correct padding byte (e.g., $\backslash$x01).
4) By modifying the last byte of C' and resubmitting, the attacker can iteratively guess the correct value for each byte of the plaintext.

A. Decrypting the Last Block

In this subsection, we outline the steps the attacker follows to decrypt the last block of the intercepted ciphertext, denoted as C_n. The process involves crafting a custom ciphertext block C' that, when appended to C_n, can be sent to the padding oracle to validate the padding and thereby infer the values of the plaintext bytes in C_n.

The process is as follows:

1) The attacker creates a new ciphertext block C'. Typically, a block of zeroes is used:
$$C' = 0\text{x}000\ldots0$$
2) Form a new ciphertext by concatenating C' and C_n:
$$C'||C_n$$
3) The attacker sends $C'||C_n$ to the padding oracle for decryption.

B. Understanding the Decryption

This subsection delves into the mechanics of how the padding oracle decrypts the modified ciphertext $C' \,||\, C_n$. The decryption of C' is not relevant to the attack, but the plaintext from C_n contains vital information the attacker aims to exploit. Understanding this process is key to isolating and calculating the plaintext bytes.

When the oracle decrypts $C'||C_n$, it performs the following operations:

$$P_1' = D(C') \oplus IV$$
$$P_2' = D(C_n) \oplus C'$$

where:

- P_1' is the plaintext block resulting from decrypting C'.
- P_2' is the plaintext block resulting from decrypting C_n with C'.

Since P_1' is irrelevant (garbage), the focus is on P_2':

$$P_2' = D(C_n) \oplus C'$$

Using the encryption definition $C_n = E(P_n \oplus C_{n-1})$, we substitute:

$$P_2' = D(E(P_n \oplus C_{n-1})) \oplus C' = P_n \oplus C_{n-1} \oplus C'$$

Thus:

$$P_2' = P_n \oplus C_{n-1} \oplus C'$$

The challenge we face is the presence of two unknown values: P_2' and P_n. Since a formula with two unknowns cannot be resolved directly, we can leverage the padding oracle to extract additional information. By manipulating C' and analyzing the oracle's responses, the attacker can effectively deduce the value of P_n.

C. Exploiting the Padding Oracle

The attacker aims to manipulate C' such that the padding in P_2' is valid. Specifically, the oracle reveals whether the padding is correct or not, allowing the attacker to infer information about P_n.

1) Determining the Last Byte $P_n[k]$: In this step, we focus on isolating and computing the last byte of the plaintext block P_n. By systematically varying $C'[k]$ and submitting the modified ciphertext to the oracle, the attacker can determine which byte value produces valid padding. This demonstrates the practical application of XOR properties, allowing the extraction of individual bytes from the plaintext.

To illustrate, consider the last byte, denoted as k:

$$P_2'[k] = P_n[k] \oplus C_{n-1}[k] \oplus C'[k]$$

For valid padding to be recognized, the value $P_2'[K]$ can represent valid padding bytes such as "$\backslash$x01", "$\backslash$x02", or any other permissible padding value. However, we will specifically enforce the condition that $P_2'[K]$ must equal "$\backslash$x01".

$$1 = P_n[k] \oplus C_{n-1}[k] \oplus C'[k]$$

At this point, we have identified the following variables: $P_2'[K]$, the valid padding value "$\backslash$x01"; $P_n[K]$, the unknown last byte of the plaintext; $C_{n-1}[K]$, the known last byte of the previous ciphertext block; and $C'[K]$, the controlled byte we modified for valid padding.

With these three known variables, we can reformulate the equation, leveraging the commutative property of the XOR operation to rearrange them as necessary.

Solving for $P_n[k]$, we get:

$$P_n[k] = 1 \oplus C_{n-1}[k] \oplus C'[k]$$

The execution of the attack proceeds by varying $C'[k]$ across all byte values from 0 to 255. For each value, the attacker constructs the modified ciphertext $C'||C_n$ and submits it to the oracle. The oracle's response reveals the specific value of $C'[k]$ that results in valid padding. Once the valid padding is found, the corresponding plaintext byte $P_n[k]$ can be computed using the above equation, allowing the attacker to extract the plaintext byte by byte.

2) Determining Earlier Bytes $P_n[k-i]$: We extend the attack strategy to uncover the bytes preceding the last byte of the plaintext block. By adjusting the custom ciphertext C' to enforce valid padding conditions for these earlier byte positions, the attacker can iteratively deduce the complete plaintext block. This section illustrates how the same principles applied to determine the last byte can be adapted to extract earlier bytes.

Example*: Determining $P_n[k-1]$*

First, we need to assign $C'[k]$ an appropriate value to ensure that $P'[k] = 2$. The reason for selecting the value 2 is that we aim to ascertain the second-to-last byte of P'_n. By setting the last byte to 2, we can systematically vary $C'[k-1]$ through all possible values until we no longer encounter padding errors. This successful configuration will confirm that P'_n concludes with the byte sequence "\x02\x02".

$$P'_2[k] = P_n[k] \oplus C_{n-1}[k] \oplus C'[k] = 2$$
$$2 = P_n[k] \oplus C_{n-1}[k] \oplus C'[k]$$

Solving for $P_n[k-1]$:

$$P_n[k-1] = 2 \oplus C_{n-1}[k-1] \oplus C'[k-1]$$

D. Iterative Decryption

The attacker employs an iterative process to decrypt the entire plaintext block P_n byte by byte, starting from the last byte and moving to the first. Each iteration focuses on establishing valid padding conditions and utilizing the oracle's responses to systematically reveal the values of all bytes in the block.

For each byte position i from k down to 1:

1) Set padding value to i (i.e., $\backslash \mathrm{x}i$).
2) Adjust corresponding bytes in C' to satisfy the padding condition.
3) Use the oracle to validate padding and deduce the plaintext byte $P_n[i]$.

Similarly, just as we can recover the last plaintext block P_n, we can also decrypt the preceding blocks $P_{n-1}, P_{n-2}, \ldots$. Each block can be decrypted independently, provided the ciphertext of the preceding block is available.

This brings us to the first plaintext block P_1. As previously noted, P_1 is determined by the equation:

$$P_1 = D(C_1) \oplus \mathrm{IV}$$

where:

- P_1 represents the first plaintext block.
- C_1 is the first ciphertext block.

In this case, the initialization vector (IV) acts as the previous ciphertext block for P_1, in the same way C_{n-1} does for other blocks.

After successfully decrypting all blocks of ciphertext using the padding oracle attack, we obtain the complete plaintext P. Once all the plaintext blocks are recovered, the padding added during encryption is removed by checking the last byte, which indicates the length of the padding. By stripping off the padding, we finally retrieve the original message.

VIII. Implementation

The attack algorithm was implemented in Python using AES-128 Symmetric Key Encryption. The structure of the overall code is described by the pseudocode below.

Algorithm 1: Padding Oracle Attack

```
Input: Ciphertext C, Oracle O
Output: Plaintext P
for block i ← n to 1 do
    Initialize empty plaintext block P_i;
    for byte j ← size(block) to 1 do
        for each possible byte value v ← 0 to 255 do
            Construct modified ciphertext C';
            Send C' to oracle O;
            if oracle returns True then
                Set P_i[j];
                Break loop;
return P;
```

The complete implementation can be found in the GitHub repository linked at the end of this report.

IX. Result

The padding oracle attack works within a complexity of $\mathcal{O}$(NbW) in order to decrypt the message where W is the number of possible words (typically W = 256), N is the number of blocks, and b is the block size.

After completing the implementation code, we conducted tests using custom inputs to execute the padding oracle attack. We then verified that the decrypted output matched the original input.

Attack Test Output

```
------------------------------------
Testing : b'Attack␣at␣dawn' OF LENGTH 14
The encrypted message is : b"Xi\x95RA\xb3\x97v,\
    xb9\x91\xe3;\xfa'\xd5n\xfcLT@\xb2\xa8\xf2\
    xda}\xceqM\xfff\xda"
The decrypted message (with padding) is :
    bytearray(b'Attack␣at␣dawn\x02\x02')
------------------------------------
Assertion passed!
------------------------------------
------------------------------------
Testing : b'' OF LENGTH 0
The encrypted message is : b'\xcd\xca\xbdIB\xe9\
    xde\xfd\xda\xb7#A-X\xe9!\x9c\xa6\x98f\xab\
    xb1\x07\xbe\n\x9928\x8c\xdc*\xe7'
The decrypted message (with padding) is :
    bytearray(b'\x10\x10\x10\x10\x10\x10\x10\x10
    \x10\x10\x10\x10\x10\x10\x10\x10')
------------------------------------
Assertion passed!
------------------------------------
------------------------------------
```

X. Improvement

The integration of authentication into the CBC-PAD scheme was initially deemed unnecessary, but Wagner's attack underscored the critical need for such measures in cryptographic protocols. In response, a comprehensive scheme has been proposed that simultaneously ensures authentication and confidentiality by applying both mechanisms to the padded plaintext.

The process commences with padding the cleartext, which is then authenticated and encrypted before transmission. During decryption, the first step involves verifying the authenticity of the received data, followed by padding validation to ensure integrity. Once these checks are complete, the cleartext can be accurately extracted.

It is important to differentiate between cleartext—the original message—and plaintext, the data input into the encryption algorithm. While Krawczyk's authenticate-then-encrypt method has shown promise, it remains susceptible to side-channel attacks, indicating a need for further scrutiny. Although formal proofs of the proposed scheme's security are lacking, it is expected to provide a robust defense against various threats, thereby enhancing the CBC-PAD framework's overall security.

Acknowledgment

We express our sincere gratitude to Prof. Sruthi Sekar for the opportunity to explore the topic of Padding Oracle Attack on CBC Mode through the Chalk and Talk initiative in CS 409M. This experience allowed us to deepen our understanding of this attack methodology, enhancing our exploration of cryptographic vulnerabilities. Additionally, collaborating on the presentation enabled us to effectively convey our findings and insights.

References

[1] Our code repository of attack implementation
[2] Vaudenay, S. (2002). Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS....
[3] Padding oracle attacks: in depth by SkullSecurity
[4] Katz and Lindell Book(2nd Edition) - Section 3.7.2
[5] Wikipedia - Padding Oracle Attack
[6] CS2107 Padding Oracle Attack