Analysis of the Functionality, Risks and Counter-Measures of Current Padding Attacks and the Implementation of an Attack in the Open-Source Program CrypTool 2

Bachelor Thesis
Frankfurt School of Finance and Management

Submitted to
Prof. Dr. Peter Roßbach
and
Prof. Bernhard Esslinger

Supervised by
Dr. Martin Franz

by
Alexander Colin Jüttner

Flörsheim, October 2012
(small updates in Oct 2020 for publishing)

Table of Contents

Table of Figures

Table of Tables

All figures illustrated in this thesis have been created by the author.

1 Motivation

Padding is used in cryptographic systems to increase the length of a message to the multiple of a given block size. Since the padding does not contain any sensitive information, considering its security side effects is often neglected. However, a lack of security of the padding bytes can cause vulnerabilities which negate the security measures of the whole system. Over the last years, several attacks that exploit systems with poorly integrated padding have been developed and successfully deployed against real systems. These attacks are called padding attacks.

The aim of this thesis is to explain the functionality of some current padding attacks. Additionally, possible counter measures and their effectiveness are presented. Some applications to modern systems, protocols and programs are described as well, in order to stress the threat originating from these attacks. Within the scope of this thesis, a plugin for the open-source software CrypTool 2 has been implemented. This plugin visualizes a padding oracle attack for educational purposes. The development and architecture of this plugin are also explained in this thesis. An experiment which shows the efficiency of the attack was performed with the plugin. The execution and the results of the experiment are described in the final chapter of this thesis.

2 Preliminaries

2.1 Cryptographic Building Blocks Used in this Thesis

Ciphers

Ciphers are cryptographic algorithms that are used for encryption and decryption. Encryption describes the process of transforming a message into a ciphertext. The main goal of encryption is to provide privacy. Messages transmitted over potentially insecure channels can be encrypted in order to prevent unauthorized parties from reading the messages.

Keys are used in modern ciphers to modify the transformation process. Without keys, messages could be properly encrypted and decrypted by unauthorized parties as soon as the transformation process is revealed. For all common cryptosystems choosing strong keys is essential to ensure security. Especially in computer networks it is easier to manage separate keys for different communication parties than different ciphers.[1]

Ciphers can be categorized by the way they use keys. Asymmetric ciphers use a key pair consisting of a public and a private key to encrypt and decrypt data, while symmetric ciphers use the same (private) key for encryption and decryption. In this thesis the focus will rest on symmetric ciphers.

Symmetric ciphers can further be categorized into stream and block ciphers. Block ciphers divide a plaintext message into strings of a fixed length (called blocks) and encrypt/decrypt one block at a time. Block ciphers can be run in different modes of operation. The National Institute of Standards and Technology (NIST)[2] describes five modes of operation: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR). ECB is the simplest mode and encrypts each plaintext block independently with the same key.[3]

Stream ciphers usually encrypt/decrypt on bit level by computing an XOR of each bit of a plaintext stream with the corresponding bit of a keystream.

Plaintext	0	0	0	1	1	1	0	1	1	0	1	0	1	1	0	0	0	1	…
Keystream	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1	0	1	1	…
Cipherstream	1	0	0	1	0	1	0	0	1	1	0	0	0	0	1	0	1	0	…

Figure 1: Creation of a ciphertext with a stream cipher on bit level

[1] Menezes/van Oorschot/Vanstone (1997), p. 12
[2] http://www.nist.gov/index.html
[3] Dworkin (2001)

Cipher block chaining

Cipher Block Chaining (CBC) is a mode of operation for block ciphers. One major flaw of ECB mode is that two identical plaintext blocks encrypt to identical ciphertext blocks. In some cases, this can leak information about the underlying plaintext.

CBC does not leak this kind of information: In CBC mode encryption, the XOR of a plaintext block and the ciphertext of the preliminary plaintext block is encrypted (see Figure 2). This 'chaining' causes ciphertext blocks to depend on all preceding ciphertext blocks.[4]

The probability of two identical plaintext blocks resulting in identical ciphertext blocks is therefore very low.

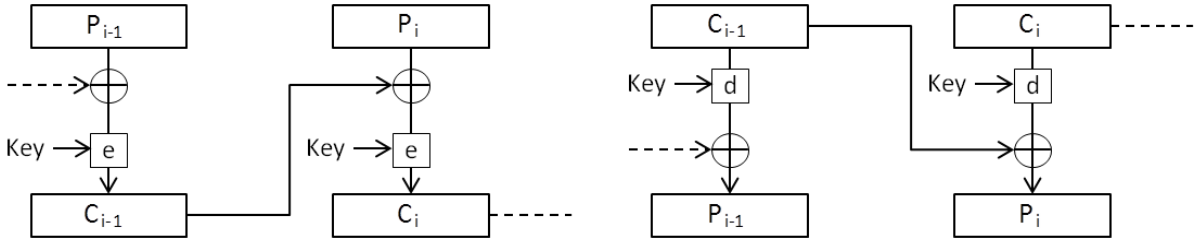

Figure 2: CBC mode encryption (left) and decryption (right)

When the first plaintext block is encrypted, no preceding ciphertext blocks exist yet. Therefore, an XOR of the first plaintext block and a special data block called initialization vector (IV) is computed. The IV consists of arbitrary values and causes two messages with identical plaintext to encrypt to different ciphertext. IVs can be sent publicly when using CBC mode, although it is recommended to use a secret IV. Attacks described in the remainder of this thesis manipulate the decryption of a ciphertext. This manipulated decryption will result in the generation of a different plaintext. Secret IVs prevent attackers to influence the generation of the first plaintext block. [5]

Padding

Padding describes practices to expand the length of messages. Applying padding to data can pursue two different goals:

1. Length-Hiding: In some cases, the length of an encrypted message can reveal information about the type or even the content of the message. In order to obscure the length of the actual message, padding can be attached to the message before encryption.[6]
2. Formatting: Some cryptographic algorithms require a certain text length. Especially block ciphers require the message length to be a multiple of the block length.

[4] Menezes/van Oorschot/Vanstone (1997), p. 230
[5] Dworkin (2001), p. 8
[6] Tezcan/Vaudenay (2011), p. 1

Although many different padding methods exist, most of them share the requirement to be 'unambiguous'.[7] This specification assures that the message data can be identified and the padding data can be completely removed without the risk of deleting other parts of the message. If the length of the actual message is known to the receiver (via another channel or from another context), also an ambiguous padding scheme can be used without any further concern.

The vulnerabilities described in this thesis apply to most unambiguous padding schemes, although the focus will rest on the padding scheme described in the Cryptographic Message Syntax (CMS).[8] CMS describes the IETF's standard syntax for cryptographically protected data.[9] In this padding scheme, the padding is appended after the message with as many bytes as required, and each byte has as value the total padding length. As an example it is assumed that the message 61 62 63 shall be padded to 8 bytes. The PKCS#7 scheme then goes: The number of required padding bytes (> 0) is encoded with binary encoding in one byte. This byte is used one or several times to pad the message. Example: 61 62 63 05 05 05 05 05. The padding length does not necessarily have to be smaller than the block length. Blocks completely consisting of padding bytes are possible, although the maximum padding length is 256.[9]

Padding Oracle

In this document, a padding oracle is a function which checks if a padding is valid or not and publishes the result of the check. When the padding oracle receives a ciphertext message, the message is first decrypted in CBC mode under a given key and then the padding is checked. After checking the padding, the padding oracle returns either 0 (bad padding) or 1 (valid padding). A padding oracle is a "black box" for attackers, since the actual transformation process, including interim results, is secret. An attacker can only see the output for a given input.

Padding oracles are commonly used in secured communication channels, although they do not have to be specifically defined as such. Vaudenay stated that when an entity receives an encrypted message, it normally decrypts it and then tries to remove the no longer required padding. If the padding cannot be identified and therefore not be removed, an error occurs. In

[7] Dworkin (2001), p. 17

[8] Other padding schemes, for example zero padding (appends (if required) zero bytes until the message is padded to a multiple of the block length. This padding method can only be inverted unambiguously if the message does not end in zero bytes. Example: 61 62 63 00 00 00 00 00),
ANSI X.923 padding (appends zero bytes, followed by a byte containing the number of message bytes in the final block in binary encoding. In a different variant the last byte contains the number of padding bytes. Example: 61 62 63 00 00 00 00 04) and 01 00 padding (appends the byte 01 and fills the rest of the block with zero bytes. Example: 61 62 63 01 00 00 00 00) exist as well, but are not discussed in the remainder of this thesis.

[9] Housley (2009), p. 28

this case, aborting any further processing can reveal padding information as well as returning an error message.[10] So the attacker misuses the decrypting entity's normal reaction by flooding it with malicious messages.
Rizzo and Duong describe different methods to find padding oracles. However, this is beyond the scope of this thesis. Further information can be found in Rizzo/Duong (2010).

Message Authentication Code

Message Authentication Codes (MACs) are used to provide data integrity and authentication between two communication parties. MAC algorithms use a message and a key to produce a fixed-size output.[11] When a party sends a message, the MAC is appended to it. The receiving party then reproduces the MAC and compares it with the transmitted one. If the MAC does not match, the integrity of the message cannot be verified and the message should be directly discarded. Authentication schemes intend to prevent attackers forging valid MACs without any knowledge about the key. In this case, attackers are unable to create valid own messages.

Transport Layer Security

Transport Layer Security (TLS) is one of the most widely used security protocols on the internet. The first version, TLS 1.0, is based on the protocol 'Secured Sockets Layer' (SSL) and has been defined in RFC2246[12] in January 1999. The current version of TLS is 1.2, which has been defined in RFC5246[13] in August 2008. Although TLS 1.2 provides stronger implementations of security mechanisms, many applications only support older versions of the protocol. The protocol is also backward compatible which means that outdated versions of the protocol may be used even if one communication party supports the newest version.[13]
TLS consists of two sub-protocols: The TLS Handshake Protocol and the TLS Record Protocol. The Handshake Protocol authenticates the communicating parties and negotiates the keys and ciphers ought to be used. The Record Protocol aims to provide privacy and integrity for the transmitted messages. As depicted in Figure 3, a MAC-then-Encode-then-Encrypt (MEE) construction is used to achieve this security goal. When sending a message under MEE, TLS first creates and adds a keyed MAC to the message. If necessary, padding data is added in the encoding step. Using block ciphers makes padding become necessary, because TLS always uses block ciphers in CBC mode. Finally, the message is encrypted with the cipher that has been selected in the handshake.
The TLS decryption of the ciphertext works similarly: After decryption, the padding is removed and then the MAC is validated and removed. Since errors can occur while removing

[10] Vaudenay (2002), p. 1
[11] Menezes/van Oorschot/Vanstone (1997), p. 321
[12] Dierks/Allen (1999)
[13] Dierks/Rescorla (2008)

the padding as well as while validating the MAC, TLS distinguishes between both padding and MAC errors.

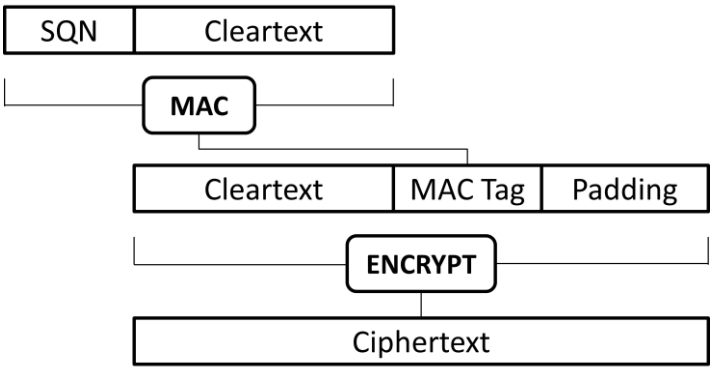

Figure 3: Ciphertext creation in TLS. SQN is a sequence number added to every message sent in TLS

In the remainder of this thesis it will always be assumed that TLS is used with a block cipher in CBC mode.

2.2 Notation Used Throughout in this Thesis

Terminology

Communication parties consist of a client and a server. Clients send messages containing secret information to a server, while servers process received messages. The secret information is defined as cleartext and needs to be protected by cryptographic means. Optional elements, such as MACs or padding data, can be added to the cleartext. The concatenation of the cleartext and other elements is denoted as plaintext. Plaintexts are encrypted in CBC mode prior to sending it.

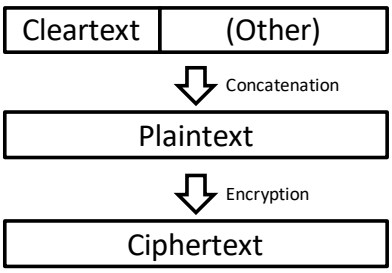

Figure 4: Types of texts used throughout this thesis

Throughout this thesis, two types of encryption/decryption exist. **Encryption** expresses the transformation of a plaintext using a cipher and a key. **CBC mode encryption** expresses the transformation of a plaintext into a ciphertext, including any 'plain' encryptions and XOR computations. This also applies to decryption / CBC mode decryption, as illustrated in Figure 5.

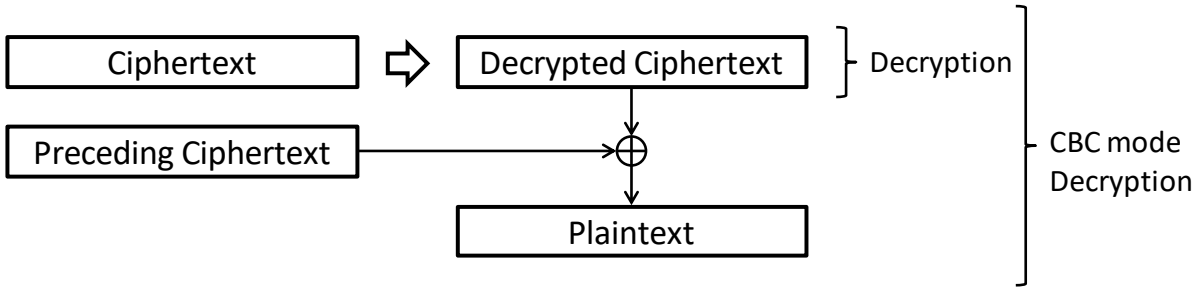

Figure 5: Type of decryptions used throughout this thesis

In all attacks described in this thesis, it is assumed that a padding oracle exists in the used server.

Variables and Symbols

P_i : plaintext block at position i; the first block is denoted with the index number 1

C_i : ciphertext block at position i, CBC mode encryption of block P_i

D_i : decryption of ciphertext block C_i

d_n : byte at position n of decrypted block D_i

$\overline{C_{i-1}}$: ciphertext block C_{i-1}, which has been modified during the attack

$\overline{c_n}$: byte at position n of modified ciphertext block $\overline{C_{i-1}}$

P_i' : plaintext block computed by the padding oracle when receiving a modified message

b : block length in bytes

l : padding length in bytes

g : guessed byte value (used in Chapter 3.2.2)

$\oplus$: bitwise XOR operation

A roof line was added on top of the variable when this value was modified by the attacker, and a superscripted "," was added for values computed by the padding oracle.

The implemented CrypTool 2 plugin uses a simplified notation. Single bytes are not denoted and the block indices are not subscripted. For example, the second plaintext block is denoted as 'P2' in the plugin, instead of P_2'.

3 Current Padding Attacks

This chapter describes current padding attacks. Chapter 3.1 focuses on how the mechanisms work in theory, analyzes the risk originating from them and presents possible counter measures. Chapter 3.2 presents some examples, where one of the mechanisms, the Padding Oracle Attack (POA), has been applied to real protocols and systems.

3.1 General Mechanisms

Two mechanisms are analyzed in this chapter. The Padding Oracle Attack (Chapter 3.1.1) can be used to decrypt an encrypted message without knowledge about the key, while the Message Distinguishing Attack (Chapter 3.1.2) enables attacker to distinguish between two messages.

3.1.1 Padding Oracle Attack

The attack described in this chapter is both a side channel and a man-in-the-middle attack that has been discovered by Serge Vaudenay in 2002.[14] It exploits information revealed by a padding oracle. Padding oracles reveal the validity of a message's and this was for a long time considered as insignificant in terms of information leakage. This chapter illustrates how this validity information can be misused to allow the attacker to decrypt ciphertext blocks.

3.1.1.1 Overview

Setup

A client tries to send a message to the server. This message was encrypted in CBC mode and consists of several ciphertext blocks. Only the client and the server know the cryptographic key that was used to encrypt the message. Before the message reaches the server, it is intercepted by an attacker. This attacker is able to modify the ciphertext blocks and send messages to the server. Upon receipt of a message, the server decrypts it in CBC mode and checks the padding afterwards. The result of the padding validation is then returned to the attacker. The attacker uses these server responses to decrypt the message. Although the attack described below, targets only one block at a time, any ciphertext block of the message can be targeted.

[14] Vaudenay (2002)

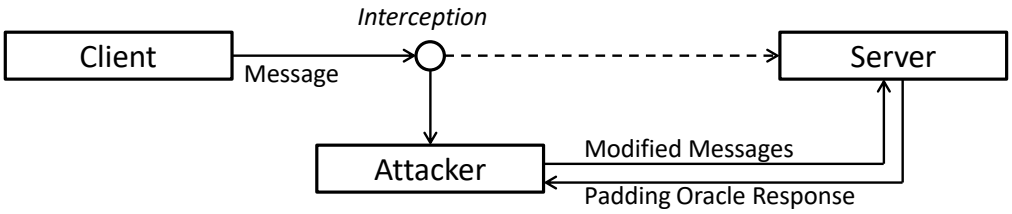

Figure 6: Attack setup

The attack consists of three phases:

- In the first phase, the attacker tries to find a message with a valid padding after decryption. If the last ciphertext block is targeted, a valid padding already exists and the first phase can be skipped.
- Different possibilities for a valid padding exist, so in the second phase the actual padding length is determined.
- In the last phase, the decryption of the ciphertext block is calculated.

Strategy

The client sends an encrypted message to the server. The attacker intercepts this message before it reaches its destination. In order to gain knowledge about the underlying plaintext, the attacker modifies the intercepted message and forwards it to the server. The server provides a padding oracle to validate any received message. The server responses are intercepted as well, since the attacker needs them for the decryption. The goal of the attacker is to ascertain the underlying plaintext P_i of a ciphertext block C_i. In CBC mode decryption, a plaintext block P_i is generated with the decrypted ciphertext block D_i and the preceding ciphertext block C_{i-1}. The formula to decrypt P_i is:

$$C_{i-1} \quad \oplus \quad D_i \quad = \quad P_i$$

An attacker who was able to intercept two ciphertext blocks C_i and C_{i-1}, is able to ascertain the underlying plaintext as soon as D_i is known. The attacker therefore tries to gain knowledge about D_i. The general idea behind the attack is to replace C_{i-1} in the above equation with a modified ciphertext block $\overline{C_{i-1}}$. This will automatically result in a different plaintext block P_i'. If the attacker knows which plaintext is generated when using $\overline{C_{i-1}}$, D_i can be computed easily:

$$\overline{C_{i-1}} \quad \oplus \quad P_i' \quad = \quad D_i$$

The next chapter describes how a padding oracle can be used to gain the necessary information about the generated plaintext P_i'.

3.1.1.2 Functionality

Phase 1: Find Valid Padding

In the first phase, a ciphertext message that results in a plaintext with a valid padding needs to be found. The original ciphertext block C_{i-1} is therefore replaced with a modified block $\overline{C_{i-1}}$. If C_i is the last ciphertext block of the message, the original C_{i-1} already generates a valid padding. The value of $\overline{C_{i-1}}$ is therefore initially set to the value of C_{i-1}. If C_i is not the last block, $\overline{C_{i-1}}$ has to be changed until a valid padding is found. Following CBC mode, the padding oracle decrypts C_i and then generates the plaintext by computing the XOR of $\overline{C_{i-1}}$ and the decrypted block D_i. Afterwards, the padding oracle checks if the padding is correct. As long as the padding oracle returns 0 (invalid padding), the last byte of $\overline{C_{i-1}}$ is changed and the message $\overline{C_{i-1}}, C_i$ is resent to the padding oracle. When the padding oracle returns 1, the padding is valid. Therefore, $\overline{C_{i-1}} \oplus D_i$ must end with '01' or '02 02' or '03 03 03', …, or 'FF…FF'.[15]

Phase 2: Find Padding Length

At this point, the message $\overline{C_{i-1}}, C_i$ results in a valid padding. The next step is to determine the length of the padding by finding the first padding byte. The padding bytes are always at the end of a block. When the position of the first padding byte is known, the length of the padding is therefore known as well. Determining the padding length can be done by using the padding oracle again.

This time, the first byte of $\overline{C_{i-1}}$ is changed. The message $\overline{C_{i-1}}, C_i$ is again sent to the padding oracle. If the padding remains valid, the first byte does not influence the padding. In this case, the second byte of $\overline{C_{i-1}}$ is changed. This process is repeated until the padding oracle returns 0. As soon as 0 is returned, a previously valid padding byte must have been changed. The changed byte and all subsequent bytes of the block therefore must be padding bytes. The amount of existing padding bytes is defined as the padding length l.

Phase 3: Block Decryption

In phase 3, the ciphertext block is decrypted. The padding length l had been determined in the previous phase. Since the value of the padding bytes equals the padding length, the last l ciphertext byte(s) can already be decrypted:

$$\overline{c_b} \oplus d_b = l$$

$$\Leftrightarrow \quad \overline{c_b} \oplus l = d_b$$

The other bytes cannot be decrypted yet, because only the value of the padding bytes is known. In order to decrypt the hindmost unknown byte d_{b-l}, the generated padding has to be

[15] Based on the padding scheme defined in Chapter 2.1 - Padding

increased. This can be achieved by modifying the last l bytes of $\overline{C_{i-1}}$, so the last l bytes of P_i' have the value $l+1$:

$$l+1 \quad \oplus \quad d_i \quad = \quad \overline{c_i} \qquad \text{| for } i = b-l+1,...,b$$

The new plaintext P_i' will only be valid if p'_{b-l} also equals $l+1$. Similar to phase 1, $\overline{C_{i-1}}$ has to be changed at position $b-l$ until the padding turns valid. As soon as the padding is valid, the value of d_{b-l} can be calculated as well. This process is repeated until the whole block is decrypted.

Example:
The last l = 2 bytes have already been decrypted. The plaintext generated by the padding oracle is therefore:

$$\overline{C_{i-1}} \quad \oplus \quad D_i = \texttt{[...] ?? 02 02}$$

The padding is now increased to l = 3 by setting the last l bytes of $\overline{C_{i-1}} = D_i \oplus l$. The generated plaintext is therefore:

$$\overline{C_{i-1}} \quad \oplus \quad D_i = \texttt{[...] ?? 03 03}$$

Then $\overline{c_{b-2}}$ is changed until the padding oracle returns 1 (=> $\overline{c_{b-2}} \oplus d_{b-2} = 03$).

3.1.1.3 Requirements

The attack only works if a padding oracle exists and if an attacker is able to identify padding errors. If an attacker is unable to gain information about the validity of the padding or is unable to distinguish a padding error from other errors, e.g. from invalid MACs, the attack is not feasible. The existence of a padding oracle represents the side-channel characteristic of the attack.
Another requirement is that a block cipher encryption mode is used where the attacker can change the value of a specific byte in the plaintext. Modifying an intercepted message is characteristic for man-in-the-middle attacks. Many block cipher modes of operation like CBC, CFB and OFB meet this requirement. Theoretically, the attack is also applicable when using stream ciphers, since changing a ciphertext byte c_n directly results in a different plaintext at position n. This also applies to CFB and especially to OFB, which uses a block cipher to create a keystream. In practice, padding is normally not used with stream ciphers, because it is unnecessary. Stream ciphers do not require the input to be a certain length and other methods to hide the message length are available. In CBC mode, a decrypted plaintext byte p_n can be changed by altering the byte at the same position of the preceding ciphertext block, as explained above.

3.1.1.4 Risk Assessment

The attack described above poses many risks for any private communication over the internet. The most obvious risk is the loss of confidentiality, since unauthorized parties are able to read private messages. Thus, the main objective of protocols such as SSL/TLS cannot be guaranteed anymore.

Another problem is that the threat caused by this attack is underestimated by many software developers. Although the attack and working solutions have been known for a long time, many applications are still vulnerable. Counter-measures have either not or only poorly been implemented. Stronger implementations of the attack have been therefore developed over time. Some of these implementations are described in the remainder of this thesis, together with some practical applications.

3.1.1.5 Possible counter measures

Arbitrary-Tail Padding (ABYT-Pad)

A very promising fix is the ABYT-Pad method.[16] The padding scheme fulfills the requirement of unambiguousness[17], although no invalid padding exists:

A message is padded by choosing an arbitrary byte value, which is distinct from the last byte of the original message. Bytes with the previously chosen value are then appended to the message, until the intended length is reached. The padding can be removed by deleting all matching trailing bytes. In order to keep the original message untouched, at least one padding byte has to be added. The padding is always correct, because there is always at least one byte which can be removed and the padding scheme does not expect a certain amount of padding bytes. Since padding should not be used to validate data integrity, there is no necessity that the values of the padding bytes depend on the overall padding length. An equivalent padding on bit level is also possible. The padding bits are simply set to the opposite value of the last bit of the message. As every padding is considered correct, an attacker is not able to gain any information necessary for the attack. Example: The message 61 62 63 shall be padded to 8 bytes. By adding the arbitrarily selected value 78 to the message, the message fulfills the required message length: 61 62 63 78 78 78 78 78.

Authenticated Encryption

As already mentioned, encryption is used to provide confidentiality only. Authenticated Encryption on the other hand also provides integrity. Authenticated Encryption can be achieved either by using special schemes or creating a composition of a standard encryption

[16] Black/Urtubia (2002), p. 7

[17] Further information in Chapter 2.1 - Padding

scheme and an authentication scheme.[18] By adding authentication mechanisms, the involved parties are able to notice manipulations in the received message which renders the attack unfeasible. This of course implies that the authentication check must be performed before any decoding, such as removing the padding. Optimally, the padding is included in the MAC generation. Since MAC algorithms produce a fixed-length output, the padding can be added to the message before computing the MAC. If every component of the message is included in the MAC computation, not bit can be changed without being detected. For example, a 24 bit cleartext 61 62 63 shall be protected with keyed MD5 hash and AES in CBC mode. MD5 produces a 128 bit hash value, so 13 padding bytes (104 bits) are required to fill all blocks. The message is first padded, then the MAC is calculated and finally the plaintext, including padding and MAC, is encrypted. If the cleartext, MAC or padding is changed, the MAC will not match anymore. Therefore, malicious modifications can be identified directly.

Electronic Cipher Block Mode

Although ECB mode lacks some security features of other operation modes, it is not vulnerable to this attack. In order to systematically forge a new padding, single bytes of the generated plaintext have to be changed. However, changing a byte of a ciphertext block leads to the manipulation of the whole plaintext **block**. Although this also applies to CBC mode, one difference remains: By changing a single byte of a ciphertext block, only one byte of the succeeding block is changed as well. In ECB mode, the ciphertext blocks are not connected at all and single bytes cannot be changed. It is therefore not possible to manipulate a single byte of the generated plaintext.

[18] Bellare/Namprempre (2007), p. 3

3.1.2 Message Distinguishing Attack

This attack was introduced by Kenneth Paterson, Thomas Ristenpart and Thomas Shrimpton in 2011 and is also a side channel and man-in-the-middle attack.[19] Instead of being able to decrypt messages, this attack only enables unauthorized parties to distinguish between two messages of different length.

3.1.2.1 Overview

Setup

A client tries to send a message to the server via TLS. The plaintext message consists of three blocks. The first block is the publicly sent IV, the second block contains the transmitted cleartext, the complete MAC and at least one padding byte and the last block only contains padding bytes. Two possible values of different length exist for the cleartext.

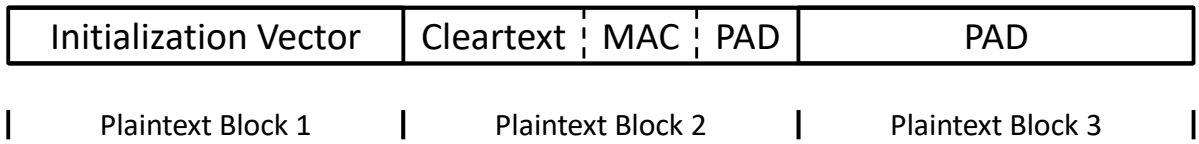

Figure 7: Structure of the transmitted plaintext message

The plaintext is encrypted in CBC mode with a key known only to the client and the server. Before the message reaches the server, the message is intercepted by an attacker (similarly to Figure 6). The attacker knows the possible cleartext values and the length of the MAC. By modifying the message and forwarding it to the server, the attacker tries to distinguish between both possible values.

Strategy

Due to the additional padding block, a so called decryption collision is possible. A decryption collision occurs when different ciphertext messages decrypt to the same cleartext. The idea behind this attack is to cause such a decryption collision by removing the padding block and adjusting the remaining padding bytes to form a valid padding. Since the encryption was performed in CBC mode, the attacker can adjust the remaining padding bytes by modifying the first ciphertext block.

[19] Paterson/Ristenpart/Shrimpton (2011)

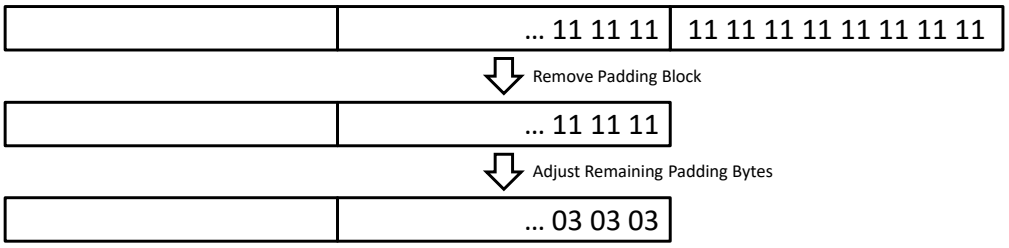

Figure 8: Creation of a decryption collision by modifying the padding

The amount of padding bytes depends on the length of the cleartext. At this point, the attacker does not know the length of the cleartext for sure, so the actual amount of padding bytes is not known either. The strategy is to change all bytes, which could belong to the padding. This will cause the server to react differently, depending on the underlying plaintext. The attacker can then interpret the server's reaction to distinguish between the cleartexts. In the case of the shorter cleartext, only the padding bytes will be changed. If the longer cleartext was transmitted, not only padding bytes will be changed but also at least one byte belonging to the MAC. This results in an error. The reaction of the server can therefore be used to determine which cleartext was sent.

3.1.2.2 Functionality

After intercepting the message, the attacker removes the last ciphertext block directly. This block can be removed without any problems, because it only contains padding bytes and does not influence the decryption of the other blocks. The value of the padding bytes is equivalent to the amount of all padding bytes. After removing the padding block, the remaining padding bytes will therefore have a wrong value. In order to generate a valid padding, the values of the padding bytes have to be reduced by the block length. Since the encryption was performed in CBC mode, this can be achieved by changing the corresponding bytes of the IV (the first block of the message).

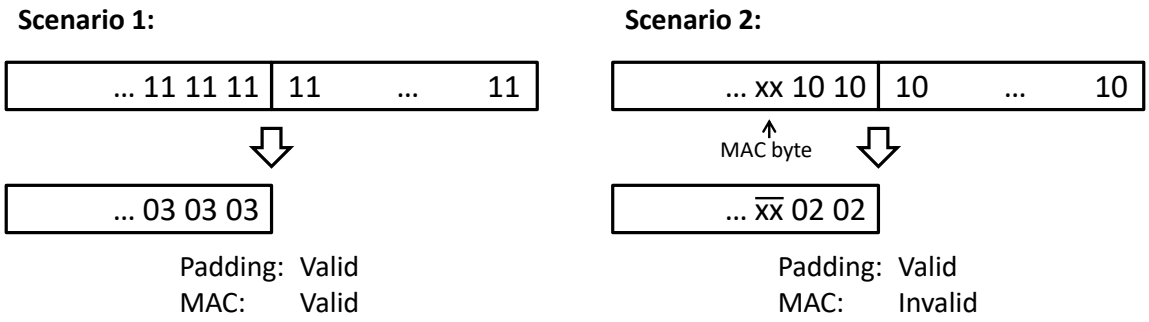

Figure 9: The two possible outcomes when removing the padding block and changing the last 3 bytes of the truncated message

The amount of padding bytes is not definitely known, so all bytes that might belong to the padding are changed. For example, if the amount of padding bytes is either 2 or 3, the last 3 bytes are changed. If the cleartext of the message is the shorter one, the message will be perfectly valid: Only padding bytes have been changed and the cleartext as well as the MAC are untouched. If the longer cleartext had been sent, the padding is valid too. The MAC on the other hand will not be valid, because at least one byte belonging to the MAC was changed.
In conclusion, an error will only occur if the longer cleartext had been transmitted. Therefore, both messages are distinguishable.

3.1.2.3 Requirements

General Requirements

The first requirement is that the possible cleartexts must have different lengths. If all cleartexts have the same length, this attack cannot be used to distinguish between them. Additionally, the attacker needs to know the lengths of the MAC and the cleartexts. Without knowing anything about the lengths, it is not possible to know which bytes need to be changed. In order to change specific bytes, a block cipher in CBC (-like) mode needs to be used. If these requirements are met, the attack is applicable when using TLS 1.0.

TLS 1.1 and 1.2

In TLS 1.1 and 1.2, attackers are not able to distinguish between MAC and padding errors. In order to distinguish between two messages, altering one message must result in an error, while altering the other message must be completely valid. The generation of a valid plaintext, although the ciphertext was changed, is only possible with a decryption collision. Therefore, the attack only works with TLS 1.1 and higher if the plaintext contains an additional padding block. Additionally, cleartext, MAC and at least one padding byte must be in the first plaintext block. In order to cause a collision decryption, the padding block(s) are removed and the remaining padding bytes are changed. These remaining padding bytes are changed by altering the preceding ciphertext or IV block. If the padding bytes are in the first plaintext block, the IV has to be changed. Changing the IV does not affect further decryption at all. However, if the padding bytes are not in the first plaintext block, a ciphertext with underlying plaintext needs to be changed. The plaintext that is generated from the altered ciphertext block most likely causes a MAC error. This error will always occur, independently of the underlying cleartext. It is therefore not possible to distinguish between the cleartexts. This requirement can be only achieved when the MAC length is shorter than the block length. AES is a recommended block cipher[20] with the longest block length (128 bit).[21] The TLS 1.2

[20] Dierks/Rescorla (2008), p. 22

specification mentions HMAC as mechanism for message authentication.[20] HMAC uses cryptographic hash functions to create the MAC.[22] The resulting MAC length therefore depends on the used hash function. Commonly used hash functions, such as MD5, SHA1 or SHA256, have an output longer than 128 bit. However, it is possible to use so called truncated MACs. Truncated MACs are 80 bit long and can be used to save bandwidth.[23] The attack is therefore applicable if a truncated MAC is used with AES and the cleartext lengths are shorter than 40 bit, leaving at least 8 bit for the padding.

Another requirement for the attack to work when using TLS 1.1 or 1.2 is that the IV is sent publicly. Without controlling the IV, influencing the generation of the first plaintext block to forge a valid padding is not possible.

3.1.2.4 Risk Assessment

This attack does not bear as many risks as Vaudenay's attack. Since the attacker needs to know the possible cleartexts, the attack is seldom applicable. If, on the other hand, all requirements are met, the attack can cause serious problems. Only one server request is necessary to allow an attacker to distinguish between the cleartexts. It is therefore irrelevant that TLS terminates the connection as soon as an error occurs. Although the attack has been only tested with TLS, applications to other protocols or systems are imaginable.

3.1.2.5 Possible counter measures

Properly implemented authenticated encryption can prevent the applicability of the attack. The attack requires the padding to be changed. If the server can detect manipulations to any part of the message, including the padding, and refuses any further processing, the attack is not feasible.

---

[21] NIST (pbl.) (2001), p. 7
[22] Krawczyk/Bellare/Canetti (1997)
[23] Eastlake 3rd (2011), p. 13

3.2 Attack Applications

This chapter presents three applications of Vaudenay's Padding Oracle Attack (see Chapter 3.1.1) to systems and protocols. Chapter 3.2.1 describes the decryption of messages sent via TLS. As explained in that chapter, the attack is not very feasible against TLS. Therefore, chapter 3.2.2 describes how the attack turns feasible, when TLS is used to protect IMAP messages. The last chapter, 3.2.3, describes how the Padding Oracle Attack can be modified to properly encrypt messages.

3.2.1 Decrypting Messages sent via TLS

This chapter describes the applicability of Vaudenay's Padding Oracle Attack (see Chapter 3.1.1) to decrypt messages transmitted via the TLS protocol. It is always assumed, that TLS 1.0 is used. To give an overview of the current situation, the TLS versions currently used in practice are described too.

3.2.1.1 Applicability

Padding Oracle Existence

The first requirement for Vaudenay's attack to be applicable is the existence of a padding oracle. Since the TLS protocol also makes use of the MEE construction[24] when an encrypted message is received, the existence of such an oracle is possible. As defined in Chapter 2.1 - Padding Oracle, Padding Oracles publish the validity of the padding. Although TLS always checks the padding, this does not necessarily mean that attackers are able to gain information about the padding validity. After a message is decrypted, first the padding is removed and then the integrity (MAC) is checked. If an error occurs while removing the padding data or while checking the MAC, an error message is returned by the protocol. As described in phase 3 of Vaudenay's attack (Chapter 3.1.1), an attacker changes bytes in order to forge a new and known padding for the generated plaintext. Over the course of the attack, bytes belonging to the MAC tag or the cleartext will be replaced with forged padding bytes.

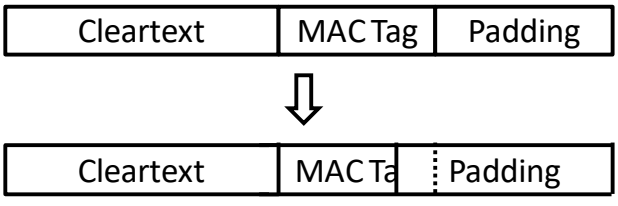

Figure 10: Overwriting bytes belonging to the MAC over the course of the attack

[24] Described in Chapter 2.1 - Transport Layer Security

Although a manipulated ciphertext may result in a valid padding, the rest of the message will be altered. This will eventually lead to an error when validating the MAC. Therefore, an attacker will most likely receive an error, every time a request is sent. If an attacker is able to distinguish between a MAC and a padding error, a padding oracle exists and can be used for the attack.

Authenticated Encryption

TLS attempts to achieve an authenticated encryption by adding a MAC before encryption. However, the integrity verification is flawed: Before checking the authenticity, the padding is removed. If the padding is not valid, it cannot be removed and an error message is sent back to the client. The server therefore sends a response before the integrity of the message is checked. In order to ensure "real" integrity of the message, the MAC has to be computed over the cleartext and the padding. Since MAC algorithms produce an output of fixed length, the amount of required padding bytes can be calculated prior to computing the MAC tag. Additionally, the MAC has to be checked before the padding. This leads to a rejection of the complete message even if only the padding was changed.

Error Distinguishability

In order to conceal the error type, TLS also encrypts error messages. This means an attacker cannot directly distinguish between padding and MAC errors. In TLS 1.0, an error message which does not reveal the type of error is returned and no further processing is performed as soon as any error occurs.[25] Computing and checking MACs require a noticeable processing time, because cryptographic operations are expensive and need to be performed on the complete message. This leads to the conclusion that the response time of a server differs, depending on the type of error occurred. Canvel et al. demonstrated that the time difference between the server responses can be interpreted in a way which makes it possible to distinguish between padding and MAC errors. In order to make the results more reliant, the aforementioned discrepancy can be increased. This is done by adding random blocks at the beginning of the message, because the longer the message is, the longer it takes to calculate the MAC tag.[26]

This vulnerability has been removed in TLS 1.1 by forcing the server to check the MAC, even if a padding error occurred. Additionally, the error message for both errors is the same. Therefore, the errors are not distinguishable and the attack is not applicable to TLS version 1.1 or higher.[27]

[25] Dierks/Allen (1999), p. 26
[26] Canvel et al (2003), p. 4
[27] Dierks/Rescorla (2006), p. 23

Throughout this thesis version 1.0 will be regarded when analyzing TLS.

Fatal Alerts

Both padding and MAC errors count as fatal errors. When a fatal error occurs, the connection is immediately closed.[28] This means, that if during the attack a padding or MAC error occurs, the connection is lost and no further tickets will be processed by the server. Creating a new connection also includes a TLS handshake, so the cryptographic keys are renegotiated. The attacker can therefore not continue the decryption of any formerly intercepted messages.

This means, that if a message is intercepted and the attacker has no information about the underlying plaintext, it is only possible to guess. A single byte can have 256 (2^8) possible values. Guessing a single byte correctly on the first try therefore has a probability of 2^{-8} (almost 0.4%). Although it is possible to guess multiple bytes at the same time, the probability to guess correctly decreases exponentially. An attacker is only able to make one guess with one connection, because even if the guess is correct, a MAC error arises and the connection is terminated.

In conclusion, the attack is not very feasible when trying to decrypt messages, sent via TLS, where no further information about the underlying plaintext is available. Still, the next chapter describes how the existing vulnerability can be effectively used to decrypt sensitive information.

3.2.1.2 TLS versions used in practice

The TLS version an application supports is defined by the used SSL engine. Some applications have an SSL engine included, while others use engines provided by the operating system. The table below shows which TLS versions are supported by different client-side applications based on the operating system and SSL engine used.

[28] Dierks/Rescorla (2008), p. 30

Engine	OS	Application	TLS 1.0	TLS 1.1	TLS 1.2
NSS[29]	All	Firefox 15.0.1/ Thunderbird 12.0.1	Yes	No	No
NSS	All	Chrome 21[30]	Yes	No	No
SCHANNEL[29]	XP/2000/Vista/2008	IE7/IE8/Safari	Yes	No	No
SCHANNEL	7/2008R2	Safari 5	Yes	No	No
SCHANNEL	7/2008R2	IE8/IE9	Yes[31]	Yes	Yes
Opera 10	All	Opera 10	Yes	Yes	Yes
Safari 5	OSX	Safari 5	Yes[32]	No	No

Table 1: TLS versions used in client-side applications as per 15.09.2012

Table 1 illustrates that TLS 1.1 and 1.2 are hardly supported by modern browser applications. Even if an application that supports newer versions of TLS is used, a connection via TLS 1.0 may be established. This is caused by the backwards compatibility of TLS: If newer versions of the protocol are not supported by all communication parties, an older version is used. Although Table 2 demonstrates that the support of newer TLS versions is slightly more common on the server-side, the backwards compatibility still raises many issues.

Engine	Webserver	TLS 1.0	TLS 1.1	TLS 1.2
IIS6	Windows 2003	Yes	No	No
IIS7	Windows 2008	Yes	Yes	No
IIS7.5	Windows 2008R2	Yes	Yes	Yes
mod_ssl	Apache HTTP Server	Yes	No	No
mod_gnutls	Apache HTTP Server	Yes	Yes	Yes
JSSE	Tomcat	Yes	No	No
NSS	Apache/Redhat/Sun Java Enterprise	Yes	Yes	Yes

Table 2: TLS versions used on the server side as per 15.09.2012

The vulnerability against padding oracle attacks has been known since 2002. TLS 1.1, which is immune against this attack, has been released in 2006. Nonetheless, most applications still do not support the newest standard. Although including TLS 1.1 support in applications is a good start, it is still not enough to completely secure a TLS connection against the attack described above. Since TLS 1.0 is vulnerable against this and other attacks, clients and servers should support TLS 1.1 or 1.2 and refuse any connection with older versions or at least inform the user about the security risks.

[29] Zoller (2011)

[30] Google tried to implement TLS 1.1 in version 21, but this caused too many issues so it was removed: http://code.google.com/p/chromium/issues/detail?id=142172 (Extracted 15.09.2012).

[31] TLS 1.1 and 1.2 are supported, but TLS 1.0 is set as default.

[32] Apple states TLS is supported, but does not specify the version: http://www.apple.com/safari/features.html (Extracted 15.09.2012).

3.2.2 Decrypting IMAP Messages under TLS

The Instant Message Access Protocol (IMAP) is a protocol which allows users to access electronic messages on a server. Its current version is 4rev1 and has been defined in RFC3501[33] in March 2003. Most modern mail servers support IMAP, so e-mail clients (e.g. Microsoft Outlook, IBM Lotus Notes or Mozilla Thunderbird) can access these servers via IMAP. IMAP allows clients to retrieve messages, permanently remove messages, alter mailboxes and folders, etc. IMAP can also be used with TLS to provide more security concerning privacy and integrity.[34]

The attack described below is an extension of the attack on the TLS protocol, described in Chapter 3.2.1. It has been introduced in 2003 by Brice Canvel, Alain Hiltgen, Serge Vaudenay and Martin Vuagnoux.[35]

3.2.2.1 Client Server Communication in IMAP

The communication between an IMAP client and server is text based. If a network connection is established, the server will send an initial greeting. Further interaction is initiated by the client with so called client commands. These commands consist of a tag, the command name and optional arguments. The tag is used as identifier and can be freely defined by the client. Upon receiving a command, the server uses the same tag to allocate the response to the initial request. Most commands are only available when the client is authenticated, so the first client command usually is “login” with the username and password as arguments.

```
a001 login myname test1234
```

Figure 11: Sample login command. The username is "myname" and the password is "test1234"

IMAP sends all messages unencrypted. SSL/TLS is used commonly with IMAP to ensure privacy.

3.2.2.2 The Attack

Strategy

The following attack does not have the goal to decrypt transmitted e-mails, but the login information. If an attacker would be able to decrypt the login information, the security of the whole mailbox would be breached.

[33] Crispin (2003)
[34] Newman (1999)
[35] Canvel et al. (2003)

Considering a block length of 8 byte, the first plaintext block contains only the sequence number and some bytes of the login command.[36] The second block on the other hand already contains some bytes of the username (see Figure 12). Therefore, the second block is targeted first.

Block	1								2								3								4		
Login Command	a	0	0	1		l	o	g	i	n		m	y	n	a	m	e		t	e	s	t	1	2	3	4	…

Figure 12: Fragmentation of the sample login command into plaintext blocks

Following Vaudenay's attack scheme, the attacker starts by altering the hindmost byte c_8 of the first ciphertext block until a valid padding is found. In contrast to the original attack, searching for a valid padding is not performed via brute force. Instead, the attacker selects a variable g as guess for the value of the last plaintext byte. This guess is used with the original ciphertext to set the modified ciphertext byte $\overline{c_8}$:

$$c_8 \quad \oplus \quad g \quad \oplus 01 \; = \quad \overline{c_8}$$

The '01' represents the intended padding. If the guess is correct, all values except the intended padding will be offset. Currently, the last byte is targeted. If the generated plaintext ends with '01', the padding will be valid. Over the course of the attack, the intended padding will change in order to forge a longer padding, similarly to phase 3 of Vaudenay's attack (Chapter 3.1.1).

Upon receipt of the modified message, the server will decrypt it in CBC mode:

$$\begin{aligned}
& \overline{c_8} \quad \oplus \; d_8 && = \quad p'_8 && \mid \overline{c_8} = c_8 \oplus g \oplus 01 \\
\Leftrightarrow \quad & c_8 \quad \oplus \; d_8 \; \oplus \; g \; \oplus 01 && = \quad p'_8 && \mid c_8 \oplus d_8 = p_8 \\
\Leftrightarrow \quad & p_8 \quad\quad\; \oplus \; g \; \oplus 01 && = \quad p'_8 &&
\end{aligned}$$

If the guess was right, g equals p_8, so $p_8 \oplus g$ equals '00'. The generated plaintext byte is therefore '01' and the server responds that the padding is valid. It can be assumed, that the padding length is only one byte long, because it is unlikely that the username or password contains characters that would form a valid padding.[37] If the padding is invalid, the attacker must wait for the client to initiate a new connection, change the guess, recalculate $\overline{c_8}$ and resend the request to the padding oracle. After the hindmost byte was decrypted, the attack can be continued as described in phase 3 of Vaudenay's attack.

[36] Assuming a tag length smaller than 7 bytes

[37] In order to rule out special cases or when using different padding schemes the padding length can be checked anyway, although this is not included in this thesis.

Optimization

Instead of guessing each byte via brute force, the attack can be optimized by using dictionaries. Passwords and usernames often contain "real" words or names. Word lists with possible words or character sequences used in passwords[38] and their probability of occurrence can be easily found on the Internet. These word lists can be used to create a search tree. First of all, a word list needs to be transformed in a way to represent possible plaintext blocks. Each node level of the tree then represents a possible byte value at the position of the plaintext block. The first node level therefore represents the possible values for the last byte of the plaintext block. Furthermore, each value at a certain position (node) has a certain probability of occurrence. The probability that the plaintext block ends with a certain character sequence can be calculated by computing the sum of the probabilities of every possible plaintext block that ends with this sequence. The calculation of these conditional probabilities statistically reduces the amount of requests sent to the padding oracle. During the search, the value of the guessed g will be based on probabilities rather than a straight forward brute-force approach.

3.2.2.3 Applicability

As shown in chapter 3.2.2.2, the attack is feasible in real life. Although TLS looses the connection when a fatal error occurs, cipher blocks can still be decrypted when the plaintext remains the same. This does not only apply to IMAP, but to every message that is sent via TLS. As long as the underlying plaintext is constant, it can be retrieved. The only remaining exception where the cleartext cannot be completely retrieved is when one plaintext block contains a cleartext component as well as parts of the MAC: The MAC is generated based on the message and the key. If the key or the message changes, the generated MAC will be different as well. In the case of IMAP, the command tag can be chosen by the client. Microsoft Outlook for example selects 4 random alphanumeric characters as tag.[39] Therefore, attackers cannot assume that 2 messages have the same MAC and the underlying plaintext of that block is not constant.

Waiting for the client to initiate new connections may seem very exhaustive, but in reality most clients establish enough connections in order to make the attack feasible. Examples:

- Many clients check for new messages every few minutes. Instead of maintaining a connection, the client often logs in again to the server for every check.
- Depending on the client, multiple connections may also be established if folders exist in the mailbox or when messages are sent.

[38] In this paragraph, password stands for password as well as username or other information that may be targeted.
[39] Canvel et al (2003), p. 14

- Microsoft Outlook 2007 and Mozilla Thunderbird 12 perform a new login every time a folder is selected. These are the default settings as of September 2012.

Compared to a brute-force approach, where attackers try to guess a valid username or password directly, the attack described in chapter 3.2.2.2 is a greater security threat. The first reason for this is that when performing a classical brute-force approach, the username and password have to be completely correct in order to succeed. The padding oracle attack on the other hand provides information on byte level: It is possible to gain information about single bytes, which reduces the average amount of tries significantly. The second reason is that the IMAP standard[40] states that failed login attempts should be limited or delayed. If an error occurs in the TLS protocol, IMAP will not count the request as a failed login attempt. Thus, in comparison more attacks are possible before malicious activities are detected.[41]

[40] Crispin (2003), p. 93
[41] Assuming that TLS does not block clients when too many errors occur or allow more tries than IMAP.

3.2.3 Encrypting Messages with CBC-R

CBC-R is a technique introduced by Juliano Rizzo and Thai Duong in 2010 and is an extension of Vaudenay's Padding Oracle Attack.[42] The Padding Oracle Attack enables attackers to **de**crypt messages without knowing the key. CBC-R on the other hand allows attackers to **en**crypt messages without knowing the key. In contrast to Chapters 3.2.1 and 3.2.2, the MAC is not specifically incorporated. Properly implemented authenticated encryption discards messages with an invalid MAC, even if it was correctly encrypted. This chapter only covers message encryption, rather than forging completely valid messages, including the MAC.

3.2.3.1 The Attack

Strategy

An attacker intercepted a message and successfully used a padding oracle to decrypt it. During the decryption of a block C_i, the preceding block C_{i-1} was modified in order to forge a plaintext P'_i with a valid padding:

$$\overline{C_{i-1}} \quad \oplus \quad D_i \quad = \quad P'_i$$

Instead of generating a plaintext with a valid padding, P'_i can be set to anything the attacker wishes. This includes executable code, server commands, SQL statements, etc. If D_i is known, P'_i can be generated by simply setting $\overline{C_{i-1}}$ to:

$$P'_i \quad \oplus \quad D_i \quad = \quad \overline{C_{i-1}}$$

When the server receives this modified message, it will decrypt it and generate the malicious plaintext block P'_i. Of course, changing $\overline{C_{i-1}}$ will subsequently lead to a different plaintext block P'_{i-1} which will most likely consist of incoherent, arbitrary bytes. In order to generate a reasonable P'_{i-1}, C_{i-2} needs to be changed as well. If the intercepted message does not have a block C_{i-2}, the attacker can simply create it.

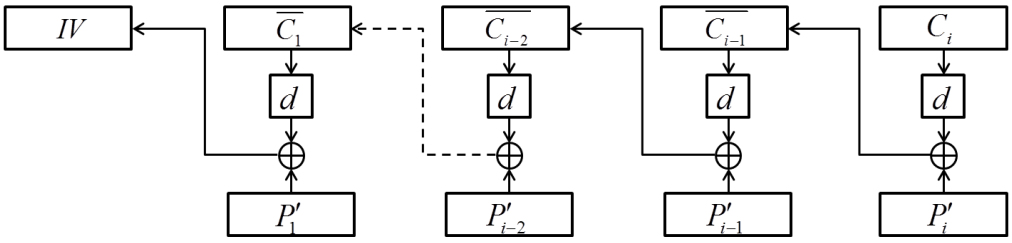

Figure 13: Forging a plaintext with CBC-R

This can be continued to create a message containing complex malicious code. If the attacker is able to set the IV, the message can be created completely. If not, the first plaintext will be garbled. Some systems require a message to start with a certain value or magic number. Java

[42] Rizzo/Duong (2010)

serialized object streams for example, require streams to begin with the constant hexadecimal value 'aced'.[43] Messages which do not begin with the expected value are normally directly rejected.

Workarounds for uncontrolled IVs

Controlling the IV means that the IV can be modified by an attacker. This is usually possible when IVs are sent publicly with the message. If an attacker does not control the IV and uses CBC-R, the first plaintext block will consist of incoherent, arbitrary bytes. Some systems require the message to begin with a certain set of values, e.g. constant identifiers or the length of the message. This set of values will be subsequently referred to as "header". Rizzo and Duong found two possible workarounds that can make the attack applicable.

The first possible workaround is to reuse a valid header. The intercepted message is very likely to have a valid format. The first plaintext block P_1 therefore must begin with the required header. This block is generated by computing the XOR of the IV and the first ciphertext block C_1. The decryption of the first block is not influenced by other ciphertext blocks in any way. In order to forge a ciphertext message with a valid header, C_1 can be simply put at the beginning of the message. By adding C_1 to the message, it will decrypt to a plaintext with a valid header and malicious code. The resulting plaintext will consist of a valid first plaintext block P_1, a garbled block P_2' and one or more valid malicious plaintext blocks. While some servers may accept the valid header, the garbled block P_2' may still lead to the rejection of the message. However, if it is possible to place the garbled block as part of a string, the server will most likely accept the message.[44] The placement can be achieved by analyzing the intercepted message. The message had already been decrypted by using the padding oracle, so the underlying plaintext is known. If the message contains free text elements like comments or labels, these can be used to hide the arbitrary block. Instead of just reusing the first ciphertext block, all blocks up to the opening of the free text field are reused. Following this block, the ciphertext block which decrypts to a garbled plaintext is placed. The free text field can then be closed in the next block, directly before the malicious code begins.

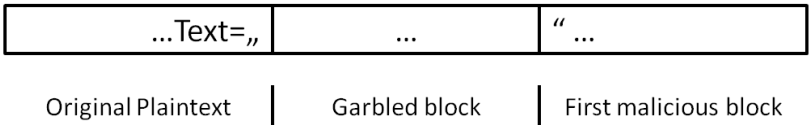

Figure 14: Hiding the garbled block in a string

The second possible workaround is brute forcing C_1. Reusing a valid header as described above does not always work. Sometimes, the header is not fixed, but depends on different

[43] Oracle (2012)
[44] Rizzo/Duong (2010), p. 6

parameters, such as the message length. Therefore, it may not be possible to find a header that matches the malicious message. In this case, an attacker can try to find a valid C_1 with brute-force. The remaining problem when brute-forcing C_1 directly is the existence of the garbled block. The underlying plaintext of C_1 is unknown, so the begin of free text fields cannot be identified. Systematically hiding the garbled block is therefore not possible. A different strategy is to find a valid C_1 by changing the last ciphertext block. Although the attack described above uses a ciphertext block of the intercepted message as basis, this is not required. Any ciphertext block can be used. Changing the initial ciphertext block will require a recalculation of the other ciphertext blocks, but the underlying plaintext will stay the same. The only block that will result in a different plaintext is the first ciphertext block. Therefore, a valid plaintext P_1 can be brute-forced by changing the last ciphertext block. Additionally, the created message will not have a garbled block. The remaining problem with this technique is the increased processing time. All ciphertext blocks need to be recalculated in every iteration. This also involves many padding oracle requests. Especially for longer messages, this technique is very inefficient.[45]

3.2.3.2 Applicability

The attack is applicable, if a decryption oracle exists. Although Vaudenay's Padding Oracle Attack was used here to decrypt the message, any decryption oracle does suffice. Depending on the system, an attacker might also have to be able to modify the IV or hide the garbled block. Another security flaw which makes CBC-R applicable to many systems is using encryption to achieve authenticity. This thesis demonstrates various examples, where keys are not necessary to modify encrypted messages. Vaudenay's attack, for example, enables attackers to decrypt messages without knowledge about the key. Assuming that correctly encrypted messages must come from a trustworthy source is therefore wrong.

Rizzo and Duong were able to use CBC-R to create malicious view states.[45] View states are often used in websites to save form information. When a user submits a form to a server it is normally rejected if mandatory fields are blank or other problems occur. By using a view state, the entered information can be restored upon rejection and the user does not have to fill out the whole form again. View states are included in frameworks such as Microsoft's ASP.NET or Oracles JavaServer Faces (JSF). Since the view states are usually stored on the client, these can be accessed easily.[46] Attackers can modify these view states, so when loaded, malicious code is executed by the server. Depending on the implementation of the view states, attackers could gain access to critical methods of the application, enable cross-site scripting,

[45] Rizzo/Duong (2010) p. 6

[46] In this scenario, a MITM attack is not necessary: Attackers can directly communicate with the server

download private files and much more.[45] Additional security measures, such as authentication, should therefore be implemented when using view states. The JSF implementation Apache MyFaces, for example, advises that states should be encrypted to prevent modifications to the view state.[47] Since JSF is vulnerable to padding oracle attacks and CBC-R, this specification does not prevent the creation of malicious view states.

CBC-R is also usable against ASP.NET applications. ASP.NET provides several padding oracles and also relies on encryption to ensure authenticity. Rizzo and Duong were able to successfully implement CBC-R attacks to access critical components.[48] It was even possible to send valid commands to the server. One of these commands induced the download of the "web.config" file. By default, this file contains all cryptographic keys in plaintext. If an attacker is able to get this file, the security of the whole application is breached.

[47] Geiler/Silvert (2009)
[48] Rizzo/Duong (2011)

4 Padding Oracle Attack Plugin

Chapter 4.1 describes the placement of the plugin in the context of this thesis and into CrypTool 2. Chapter 4.2 explains how the plugin was developed and Chapter 4.3 illustrates the architecture of the finished plugin. The plugin was used to conduct an experiment, which evaluates the efficiency of the attack. The expected results, execution and analysis of this experiment are explicitly explained in Chapter 4.4. On average, the attacks performed during the experiment required between 585 and 1469 requests, with a mean value of 1048 requests.[49]

4.1 Plugin Placement

Goal

The practical part of the thesis describes the implementation of a CrypTool 2 plugin. This plugin visualizes Vaudenay's Padding Oracle Attack[50] in order for easier understanding.
The plugin performs the complete attack and also explains it step by step. The current version only supports 64 bit ciphers, such as DES and Triple-DES.[51] Users are able to modify different components of the attack, for example the cryptographic key, the cipher and the block cipher mode of operation, to see the effects on the attack.

CrypTool 2

CrypTool 2 is an open-source cryptography e-learning platform.[52] It demonstrates cryptographic concepts and techniques for educational purposes. CrypTool 2 implements several algorithms, such as cryptographic, mathematical, or data processing functions. These functions are provided as plugins. Plugins are programmed in C# and since CrypTool 2 is open source, developers can easily create new plugins or improve existing ones. The CrypTool 2 homepage provides a "Plugin Developer Manual", which explains how to setup the development environment and how to create a new plugin. Interfaces enable communication between different plugins. Via interfaces, plugins receive input data and send output data. Most plugins only perform a single transformation step. By connecting plugins with each other, a sequence of several transformation steps can be established. These sequences are referred to as workflows and can be modeled with a graphical user interface (GUI). The GUI allows visual programming, so users can create workflows without having to know any programming language. More than 100 workflows are already available in

[49] For further information refer to Chapter 4.4.
[50] As explained in Chapter 3.1.1.
[51] Right after the bachelor thesis the author added to CrypTool 2 a POA template for attacking AES (128 bit).
[52] www.cryptool.org

CrypTool 2 by default. These workflows are called templates and delivered as cwm file. They can be used to visualize complex cryptographic systems that exist in practice.

4.2 Plugin Development Approach

The Padding Oracle Attack plugin performs the main actions of the attack. Chapter 4.2.1 describes how the plugin is embedded in the template. Chapter 4.2.2 describes how the plugin was created and which problems occurred during the creation.

4.2.1 Creating the Template

Three communication parties, the client, the server and the attacker, are involved in the attack. The complete communication flow of the attack is visualized; this allows users to understand the role of every party. Instead of implementing every component in one single plugin, the attack was designed as a template, consisting of several plugins. Although the main actions are performed solely by the Padding Oracle Attack plugin, other plugins are used to provide client and server functionality, such as data input, encryption and decryption and the padding check. This allows the attack to be more resource efficient and flexible. One reason for this is that many algorithms, for example ciphers, already exist as plugins in CrypTool 2. Reproducing the available functionality in order to compress the whole attack in a single plugin would have therefore not been resource-efficient. Additionally, the attack can easily be modified without changing any plugin code. By using different plugins to design the workflow, users can replace plugins or change the settings to modify the attack. For example, users can choose the used cipher by replacing the encryption/decryption plugin.

In order to create the template, the required functionality of each communication party is analyzed. The client has to be able to enter a message, which is then encrypted in CBC mode. The server has to be able to decrypt received messages and check the padding. The attacker has to be able to modify the intercepted message, to send requests to the server and to react to any received server response. Constructing the decisions and actions of the attacker with different plugins is too complex and is therefore implemented as one new plugin. Since CrypTool 2 provides data input and encryption plugins, the client and server functionality can almost completely be reproduced with existing plugins. However, the padding oracle functionality of the server does not exist and therefore has to be implemented as well. The remainder of this thesis only describes the creation of the padding oracle attack plugin.

4.2.2 Creating the Padding Oracle Attack Plugin

Overview

The plugin is developed iteratively. Each iteration consists of the phases analysis, design, implementation and testing. The creation of the padding oracle attack plugin can be categorized in 3 major iterations, which will be described roughly in the following.

Functional

First of all, the required interfaces and the functionality of the plugin are analyzed. The main goal of the plugin is to help users understand the attack. Therefore, the attack has to be divided into small steps. Complex algorithms are not necessary, since most steps only require computing an XOR and updating the output. However, the program flow is very complex, because the plugin has to perform different actions depending on the user input, response from the padding oracle and the overall progress of the attack. The actions are triggered by events. Two types of events exist: click events and input-changed events. The click event allows users to decide when the next step is performed, and the input-changed event allows the plugin to automatically react as soon as the padding oracle sends a response. Event listeners invoke a specified method when an event occurs. As already mentioned, actions are performed depending on several factors. Since the event listener always invokes the same method, states are used to decide which actions are performed. After modeling the program flow, the padding oracle attack plugin can be programmed.

Informational

At this point, the plugin is able to perform the whole attack, although the information provided by the plugin is very scarce. In order for users to understand the attack, more information, such as explanations, interim results and instructions, have to be included. By explaining the attack to different people, the necessary information is determined. The questions, comments and reactions from the test subjects are analyzed to define which information is required during certain stages of the attack. One of the resulting insights is that users can understand the attack process easier, if not only the transformation, but also the input and output data is shown constantly. The relations among used data blocks, especially how the XOR computations are performed, have to be displayed too. Some actions, such as incrementing the forged padding in phase 3, are too complicated and have to be divided into smaller sub steps. Since the steps are defined by the states, additional states had to be included. In order to make the modifications to the preceding ciphertext block more comprehensible, the ciphertext block is no longer modified directly. Instead, the ciphertext block remains constant and a data block named ‘overlay’ represents the changes. The

modified ciphertext block is therefore received by computing an XOR of the original ciphertext block and the overlay.

Graphical

The goal of this iteration is to create a dashboard for the plugin which allows users to review all relevant data, receive information concerning the current actions, and control the progress of the attack. Standard CrypTool 2 plugins only provide a limited selection of UI elements. Due to the complexity of the attack, this type of interface is not suitable. However, developers are able to add a presentation view to plugins. Presentation views are based on the Windows Presentation Foundation[53] (WPF) and allow developers to add content such as images, text fields, buttons and other elements to the plugin. The presentation view consists of the user interface defined with the Extensible Application Markup Language (XAML) and code behind, to control the interface. In the preceding iteration, the necessary information had been distinguished. Based on those results, the design of the interface is modeled. The user has to be able to understand the data flow during the attack. The GUI is therefore divided into an input, attack logic and output section. For consistency reasons, only the text based information, but not the structure of the dashboard are changed during the progress. The user is able to control the progress of the attack with buttons.

Problems

The biggest problems during the implementation of the **functionality** were caused by a bug in the CrypTool 2 core program. This bug always added the click event listener twice. This caused actions, for example sending a server request, to be performed twice too. The bigger problem resulting from this was a so called call overflow. When the output data of a CrypTool 2 plugin changes, this has to be processed by all connected plugins before the output can be changed again. Since the output was directly updated twice, the other plugins were not able to continue processing. The problem was solved with a workaround: By removing the event listener in a ‘try…catch’ statement, an existing event listener is always removed before a new one is added. Therefore, only one event listener exists at a time and actions are not performed multiple times. Another problem was that plugins only start processing, when **all** input interfaces receive data. Logically, the padding oracle should only send a response if a request from the Padding Oracle Attack Plugin had been received. However, the Padding Oracle Attack Plugin was unable to send a request, because it did not receive an input from the padding oracle. Since both plugins required an input from each other in order to be able to send a message, the process could not be executed. A workaround for this problem was to let

[53] http://msdn.microsoft.com/en-us/library/ms754130.aspx, extracted 05.10.2012

one plugin initially send an output, without having any input. Since it is easier to let the forge a Boolean message, than to create a complete ciphertext message, it was decided that the Padding Oracle should initially send a message. The sources of the problems were very hard to find, because both problems caused all plugins to stop processing. The problem sources were therefore only detectable by analyzing the plugin behavior, instead of analyzing the code. Additionally, all problems occurred at the same time, so even if a solution for one problem was found, the other problems still prevented the plugins from working.

Although some complication occurred during the implementation of the **GUI**, no major problems resulted from these. One complication was that CrypTool 2 automatically scales elements like images and buttons, while it leaves text elements unchanged. The scale and the position of text based information were therefore always wrong. By setting the presentation to a fixed size, the automatic scaling was deactivated. Another minor complication was that the main plugin methods are in a different thread than the presentation view. The presentation view can therefore not be directly changed by these methods. This problem was solved by using the Dispatcher-Class, a gateway which manages a thread's work queue.[54]

4.3 Plugin Architecture

Plugin Interface

The plugin provides three interfaces to enable communication with other plugins. The first interface receives the encrypted message from the client as input. Since changing the message during the attack is neither necessary nor useful, the ciphertext interface is only accessed at the beginning. The other input interface is connected to the server and receives the result of the padding check. An output interface which is also connected to the server exists too. The plugin uses this interface to send the modified messages to the server.

Workflow

The attack consists of 3 different phases, and the actions performed by the plugin depend on the currently active phase. The actions themselves are triggered by events. Two different kinds of events exist: Click events and input-changed events. Click events occur when the user clicks on a button while input-changed events occur when an input from the padding oracle is received. In order to perform the appropriate action regarding the currently active phase, states are used. When an event occurs, the current state defines which actions are performed.

[54] http://msdn.microsoft.com/de-de/magazine/cc163328.aspx, extracted 05.10.2012

User Interface

The user interface is divided into three major parts: input, attack logic and output.

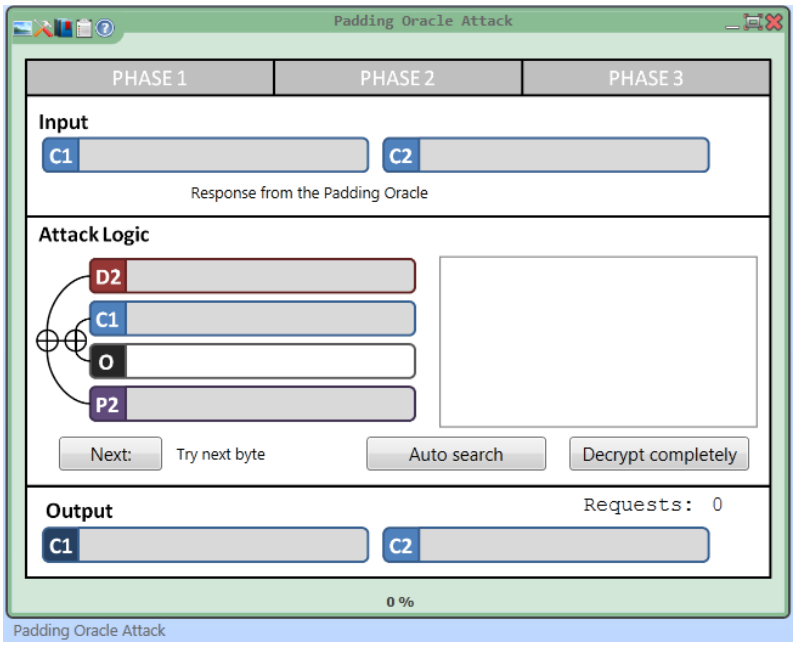

Figure 15: The GUI of the Padding Oracle Attack plugin before execution

The input part displays the intercepted ciphertext blocks and the response from the padding oracle. Only two ciphertext blocks are used during the attack. The second block contains an underlying plaintext, which the attacker tries to ascertain. The first block is manipulated to influence the generation of the plaintext. While the ciphertext blocks remain constant during the attack, the response from the padding oracle is variable.

The attack logic displays the data blocks relevant for the attack. This includes the decrypted second ciphertext block (D2), the first ciphertext block (C1), an overlay (O) and the plaintext (P2). The decrypted block and the plaintext block are only completely known at the end of the attack. The overlay displays how the first ciphertext block is modified. During the attack, the plaintext block displays the result of computing an XOR of the decrypted block, ciphertext block and the overlay. Therefore, it reflects which padding is currently generated or targeted. Since only one byte is targeted during a step, a pointer indicates which byte is currently changed. By including explanations of every block as tooltips, the interface itself is less confusing and users are still able to easily access all relevant explanations. The attack logic also contains a text field which informs the user about interim results and further actions. The actions are triggered by buttons. The button labeled ‘Next’ performs the next step only. Occasionally, brute forcing is necessary. Another button, ‘Auto Search’, was included, so users do not have to click through the complete brute forcing sequence. By clicking this button, the currently targeted byte is automatically changed until the wanted value is found. In

phase 3, up to 7 values have to be brute forced. Therefore, users are able to decrypt the whole message with the button ‘Decrypt Completely’.

The output section displays the first ciphertext block with its current modifications and the second ciphertext block. A counter which reflects the amount of server requests is also included.

Figure 16 illustrates the Padding Oracle Attack plugin during execution.

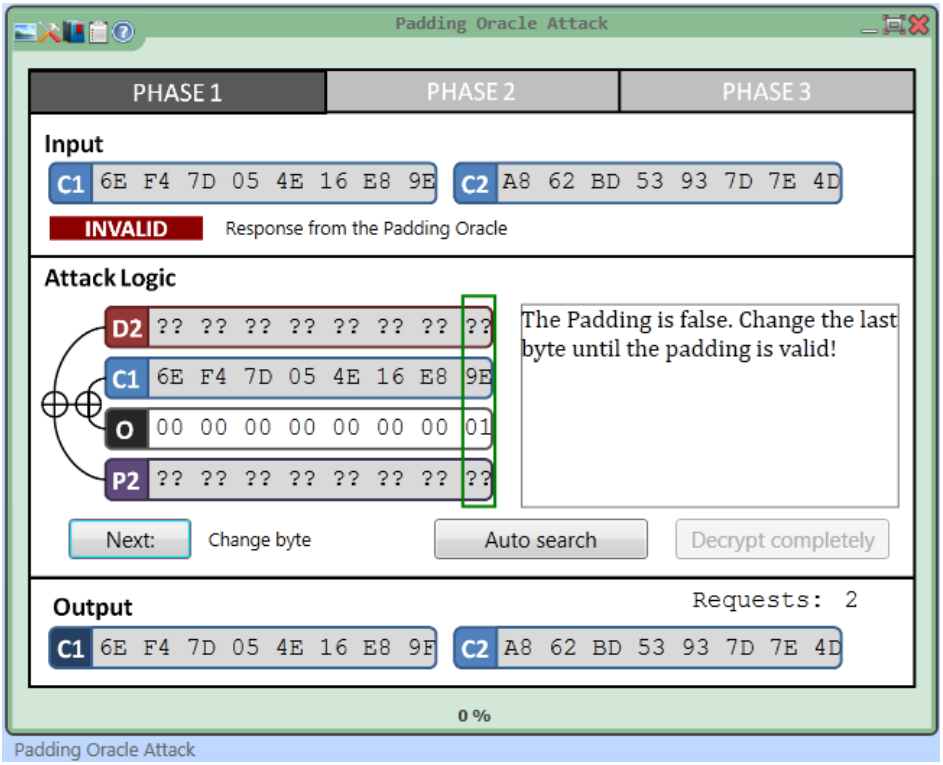

Figure 16: The Padding Oracle Attack plugin during execution

4.4 Experiment: Attack Efficiency

Goal

The goal of the experiment is to evaluate the efficiency of the attack, and how many padding oracle requests are required on average to decrypt a ciphertext block. The attack was therefore performed 100 times with randomly chosen plaintext blocks. As illustrated at the end of this chapter, the amount of necessary request per attack varied.

Setup

Every attack decrypts a ciphertext message provided by the client. The message consists of two ciphertext blocks, the first one being the encrypted IV and the second one being the cleartext encrypted in CBC mode. 256 possible values exist for each byte. The cleartext bytes were generated with the Microsoft Excel 2007 standard random function. Since the cleartext bytes are created randomly, the probability of each byte is equal. Although the IV influences the encryption and the amount of necessary requests, it is statistically irrelevant. During the attack, the first ciphertext block is modified to forge a plaintext with valid padding. The padding oracle generates the plaintext by computing an XOR of the decrypted ciphertext block and the encrypted IV. Since the decrypted ciphertext block is not known to the attacker, the bytes of the first ciphertext block are changed arbitrarily with brute force. Generating a specific plaintext byte therefore requires 128 tries on average.

Expectation

The expected amount of requests can be calculated for each phase. Phase 1 is completed when a valid padding is found. As long as the padding remains invalid, the last byte is changed. The easiest valid padding is when the plaintext ends with '01'. This padding does not depend on any other values, so after 256 requests, a valid padding is definitely found. In some cases, two possible values that result in a valid padding exist. For example, if the 6th and 7th plaintext bytes have the value '03', the padding is valid if the last byte is either '01' or '03'. Let X be the event of having a valid padding. The probability, that two values are able to form a valid padding is:

$$P(X) = \sum_{n=1}^{7} \frac{1}{256}^{n} = \frac{1}{255}$$

The probabilities to find a valid padding after k requests are listed in Table 3:

# of Requests	1	2	3	…	k
Probability	$\frac{2}{256}$	$\frac{254}{256} \times \frac{2}{255}$	$\frac{254}{256} \times \frac{253}{255} \times \frac{2}{254}$		$\frac{(256-k) \times 2}{256 \times 255}$

Table 3: Probability mass function of the amount of requests required to find a valid padding

The expected amount of requests necessary to find a valid padding is therefore:

$$E(X) = \sum_{k=1}^{256} \frac{(256-k)\times 2\times k}{256\times 255} \approx 85.67$$

If only one value, '01', is able to form a valid padding, the expected amount of requests is:

$$E(X) = \frac{1}{256}\sum_{n=1}^{256} n = 128.5$$

On average, 128 requests are therefore necessary to complete phase 1:

$$E(X) = \frac{1}{255}\times 85.67 + \frac{254}{255}\times 128.5 \approx 128$$

In phase 2, the actual padding has to be determined. Let A be the event of having the padding '01'. The probability of A is 256^{-1}. Since X always occurs when A occurs ($P(X \mid A) = 1$), the probability that the valid padding is produced by event A can be calculated with the Bayes' theorem:

$$P(A \mid X) = \frac{P(X \mid A)\times P(A)}{P(X)} = \frac{P(A)}{P(X)} = \frac{256^{-1}}{255^{-1}} \approx 0{,}9961$$

The probability that the padding equals '01' is therefore 99.61%. Since the plugin changes the bytes from left to right, phase 2 requires 7 requests on average. If the padding is '01', changing the 7th byte will still result in a valid padding. By starting the search at the 7th byte, the amount of requests can be reduced to a single try. Although this optimization is known, it was not implemented in the plugin. Most users understand the attack better, if the search starts at the beginning, and not at the 7th byte. During the experiment, the optimized version of phase 2 was not used either.

In phase 3, each byte is brute forced separately and then decrypted. The decryption does not require any padding oracle requests, but the brute forcing does. In contrast to phase 1, the targeted byte must have one specific value to produce a valid padding. On average, 128.5 requests are therefore necessary to decrypt one byte:

$$E(X) = \frac{1}{256}\sum_{n=1}^{256} n = 128.5$$

Since the amount of bytes to decrypt depends on the initial padding length, this has to be included in the overall calculation of phase 3. The expected amount of requests decreases linearly with the amount of initial padding bytes. Each of those expected values is then multiplied with the probability that the corresponding padding length occurs. The sum of these values results in the mean amount of necessary requests to complete phase 3:

$$\sum_{n=1}^{8} \frac{256^{-n}}{255^{-1}} \times (8-n) \times E(X) \approx 899$$

In conclusion, phase 1 should require about 128, phase 2 7, and phase 3 899 requests. On average, 1034 requests are therefore required for the whole attack.

Results of the Experiment

The experiment was performed with 100 random cleartext blocks. The attacks required between 585 and 1469 requests to decrypt a message. On average, 1048 requests with a standard deviation of 199 were necessary per attack. Each attack required around 1.35% more requests than expected. The chart below illustrates the probability density for specific request ranges.

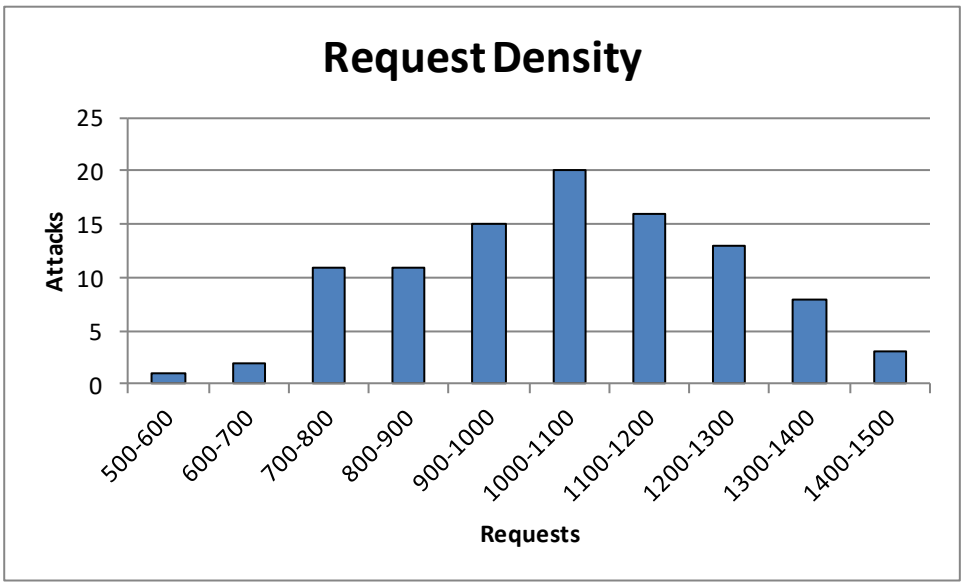

Figure 17: Density of request ranges during the attack

The request density resembles a normal distribution, having the typical bell curve alike form with a maximum around the expected value and decreasing values at both ends.

5 Conclusion

This thesis presented several padding attacks which are applicable to commonly used systems and programs. Some of these, for example Canvel et al.'s attack to gain IMAP login information (Chapter 3.2.2) or an extended version of Duong and Rizzo's CBC-R, which can be used to access cryptographic keys of ASP.NET applications[55], severely threaten security mechanisms and should therefore be prevented. The counter measures described in this thesis can prevent most padding attacks, although they might be vulnerable against others.

Vaudenay's Padding Oracle Attack (POA) was one of the most important padding attacks published in the last 10 years. This POA was successfully implemented within the CrypTool 2 framework for educational purposes.

[55] Rizzo/Duong (2011)

6 References

Bellare, M./Namprempre, C. (2007): Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.): Advances in Cryptology - ASIACRYPT 2000, Lecture Notes in Computer Science Vol. 1976, pp. 531-545, Kyoto, Japan, Springer

Black, J./Urtubia, H. (2002): Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In: Boneh, D. (ed.): Proceedings of the 11th Usenix UNIX Security Symposium, San Francisco, California, USA, pp. 327-338, USENIX Association Berkeley, CA, USA

Canvel, B./Hiltgen, A.P./Vaudenay, S./Vuagnoux, M. (2003): Password interception in a SSL/TLS channel. In: Boneh, D. (ed.): CRYPTO 2003, Lecture Notes in Computer Science Vol. 2729, pp. 583-599, Springer

Crispin, M. (2003): Internet Message Access Protocol -Version 4rev1, RFC3501 IETF standard tracks

Dierks, T./Allen, C. (1999): The TLS Protocol - Version 1.0, RFC2246 IETF standard tracks

Dierks, T./Rescorla E. (2006): The Transport Layer Security (TLS) Protocol - Version 1.1, RFC4346 IETF standard tracks

Dierks, T./Rescorla E. (2008): The Transport Layer Security (TLS) Protocol - Version 1.2, RFC5246 IETF standard tracks

Dworkin, M. (2001): Recommendation for Block Cipher Modes of Operation. US Department of Commerce, NIST Special Publication 800-38A

Eastlake 3rd, D. (2011): Transport Layer Security (TLS) Extensions: Extension Definitions, RFC6066 IETF standard tracks

Geiler, M./Silvert, S. (2009): Class StateManager. In Apache Myfaces JSF Core-1.2 API 1.2.13-SNAPSHOT API

Housley, R. (2009): Cryptographic Message Syntax (CMS), RFC5652 IETF standard tracks

Krawczyk, H./Bellare, M./Canetti,R. (1997): HMAC: Keyed-Hashing for Message Authentication, RFC2104 IETF

Menezes, A./van Oorschot, P./Vanstone, S. (1997): Handbook of Applied Cryptography, CRC Press, Inc.

Newman, C. (1999): Using TLS with IMAP, POP3 and ACAP, RFC2595 IETF standard tracks

NIST (pbl.) (2001): Announcing the ADVANCED ENCRYPTION STANDARD (AES), Federal Information Processing Standards Publication 197

Oracle (2012): Object Serialization Stream Protocol. http://docs.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html [extracted 21.09.2012]

Paterson, K./Ristenpart, T./Shrimpton, T. (2011): Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol. In: Lee, D.H./Wang, X. (eds.): Asiacrypt 2011, Lecture Notes in Computer Science Vol. 7073, pp. 372-389, Springer

Paterson/Yau (2004): Padding Oracle Attacks on the Iso CBC Mode Encryption Standard. In: Okamoto, T. (ed.), Proc. CT-RSA04, Lecture Notes in Computer Science Vol. 2964, pp. 305-323, Springer

Ristic, I./Kandek, W. (2012): SSL and Browsers: The Pillars of Broken Security. RSA Conference 2012

Rizzo, J./Duong, T. (2010): Practical Padding Oracle Attacks. WOOT'10 Proceedings of the 4th USENIX conference on Offensive Technologies

Rizzo, J./Duong, T. (2011): Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 481-489

Tezcan, C./Vaudenay, S. (2011): On Hiding a Plaintext Length by Preencryption. In: Lopez, J./Tsudik, G.: Applied Cryptography and Network Security, Lecture Notes in Computer Science Vol. 6715, pp. 345-358, Springer Berlin Heidelberg

Vaudenay (2002): Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS…. In: Advances in Cryptology EUROCRYPT'02, Amsterdam, Netherland, Lecture Notes in Computer Science Vol. 2332, pp. 534-545, Springer

Zoller, T. (2011): TLS/SSL hardening and compatibility Report 2011.