CNS Lab 03 – CBC Padding Oracle on PKCS#7

Leonardo Tamiano

November 3, 2022

1 CBC Padding Oracle

The attack we will consider in the following notes is the *CBC padding oracle attack*. First published in 2002 by Serge Vaudenay [4], the CBC padding vulnerability has had a big impact on TLS and other cryptographic implementations. Even after a lot of attention was devoted to fix it, AlFardan and Paterson showed that the vulnerability was still exploitable by performing a new version of the attack named Lucky 13 [1]. After almost two years, in October 2014, another version of the CBC padding oracle was discovered by a Google Security Team in SSLv3. The attack was called the POODLE attack [3] and nowadays SSLv3 is considered cryptographically broken because of the existence of such attack.

The CBC padding oracle attack exploits all those cryptographic implementations that use block ciphers in CBC mode and that offer no integrity mechanisms to protect the padding. In these cases, if the attacker is able to perform a *choosen ciphertext attack* (CCA) in a way that leaks information regarding the correctness of the padding in the decrypted message, then the attacker is also able to decrypt and encrypt arbitrary data with a clever exploitation of the CBC construction.

This attack has plagued many TLS implementations, because in the early design of SSL, the precusor of TLS, it was decided to implement a *MAC-THEN-ENCRYPT* scheme to protect the integrity and confidentiality of the messages sent in the TLS session. The consequence of this decision is that when confidentiality is granted through the usage of block ciphers, such as CBC-AES, given that padding is added only after the MAC has been computed, there is no integrity protection for the padding. Ultimately, this enables an attacker to change the padding portion of the message during transport without being recognized.

In 2014 a new TLS extension named *ENCRYPT-THEN-MAC* was intro-

duced with RFC 7366 [2]. This extension changes the order of operations so that the padding is included in MAC computation in order to also preserve the padding's integrity during transport. With this new scheme the server is immediately able to recognize if an external attacker has changed the original padding and act accordingly.

We will now proceed by discussing the vulnerability with mathematical detail. This will allow us to solve a challenge which includes a AES-CBC construction vulnerable to a PKCS#7 padding oracle.

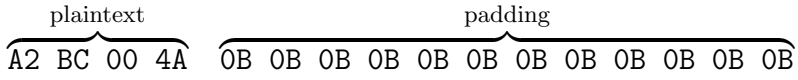
1.1 The Problems of Mac-then-Encrypt

To understand the problem of *MAC-THEN-ENCRYPT* we have to briefly review how encryption and integrity are guaranteed in TLS when block ciphers are used. When using block ciphers in TLS the following steps are taken:

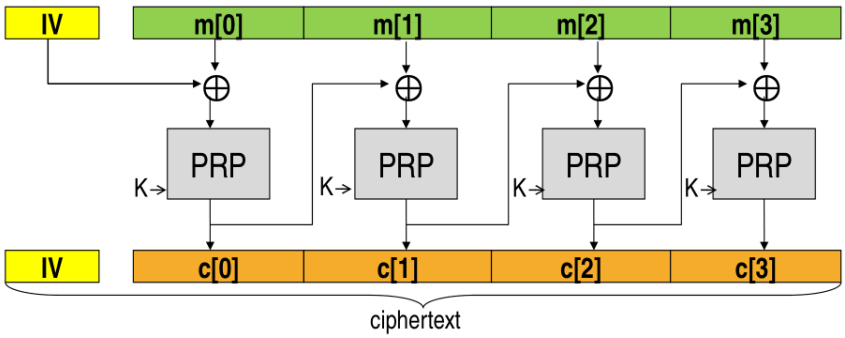
- First, a MAC is computed on the following aggregated data:
  - The *sequence number* of the TCP packet which will encapsulate the TLS record packet.
  - The *TLS header* of the TLS record packet, which in particular contains the *type*, *version* and *length* field of the *TLSPlaintext* structure shown previously.
  - The plaintext data to be sent within the TLS record packet.
- Then, an optional padding is added to make sure that the data to be encrypted, which is the combination of plaintext data and MAC, is a multiple of the block size.
- An IV is generated of the same length as the block size to be used, and CBC mode is used along with the specific block cipher agreed in the handshake phase to encrypt plaintext, MAC, and padding.
- At this point the final packet, which consists of the header information, the IV used and the ciphertext, can be sent to the other endpoint.

This construction is technically known as a *MAC-THEN-ENCRYPT* construction, because first the MAC is computed on the plaintext data, and then both the plaintext data and the MAC are encrypted to form the ciphertext.

The actual padding used in TLS for block ciphers is the *PKCS #7* padding scheme, which works by appending n bytes each of value n to the message until enough bytes for an entire block are obtained. To make things clearer, consider the case of **AES**, which has a block size of 16 bytes, and suppose we want to encrypt the message $m =$ `A2 BC 00 4A`. Then, when using this padding scheme, we would need to add $16 - 4 = 12$ bytes, each of which will have the value `0B`. This means that the final plaintext will be



Consider now the typical CBC construction already shown in the previous challenge.



and suppose we are able to find an oracle on the server that takes in input a ciphertext, and gives as output a binary outcome, depending on whether the ciphertext, when decrypted, transforms into a plaintext with a correct *PKCS #7* padding or not.

More specifically, let P be the plaintext associated with the ciphertext C. We can assume to have a function $O(C)$ such that

$$O(C) = \begin{cases} 1 & , \quad P \text{ is correctly padded according to PKCS \# 7} \\ 0 & , \quad \text{otherwise} \end{cases}$$

We will also denote with P^j the j th byte of a given plaintext block. The same applies for cipher blocks, so that C^j will be the j th byte of the cipher block.

Given the situation just described, an attacker is able to execute an efficient attack which allows for the decryption and encryption of all blocks starting from the second one onwards. Let C_0, C_1, C_2 be the various blocks, and let IV be the initialization vector used to boostrap the CBC construction. Our objective now is to describe how an attacker is able to decrypt C_1. To this end, consider only the first two blocks and the IV

$$C := IV \quad , \quad C_0 \quad , \quad C_1$$

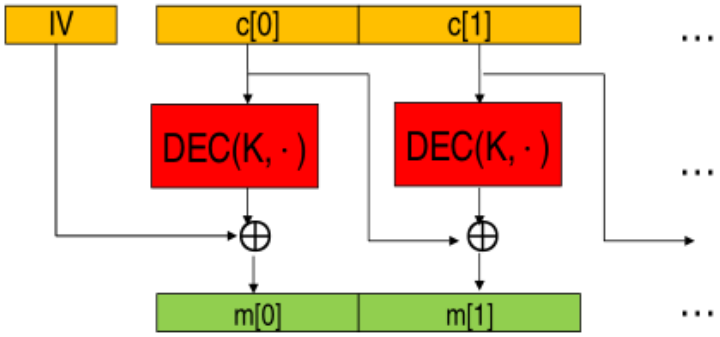
Let M_1 be the plaintext corresponding to C_1. To understand if the last byte of M_1 is a given value, say A, the attacker constructs a new $C_0^{'}$, which is equal to C_0 for all but the last byte, and the last byte of $C_0^{'}$ is obtained by XORing the last byte of C_0 with $A \oplus 01$. Specifically, if n is the byte length of the block cipher, we have

$$C_0^{'} := C_0^1 \ , \ C_0^2 \ , \ C_0^3 \ , \ \ldots C_0^{n-1} \ , \ C_0^n \oplus A \oplus 1$$

With this new block the attacker is able to construct the following new ciphertext

$$C^{'} := IV \quad , \quad C_0^{'} \quad , \quad C_1$$

Suppose now the attacker sends this ciphertext to the oracle, and the oracle replies with $O(C^{'}) = 1$. What can the attacker infer about the guess made on the last byte of M_1? If $O(C^{'}) = 1$, then, by definition, the padding of the plaintext $M^{'}$ is correct. Notice also that by following the decryption process of the CBC construction



the last byte of $M^{'}$, which is equivalent of saying the last byte of $M_1^{'}$ is obtained by XORing together the last byte of $C_0^{'}$ with the last byte obtained by applying the decryption procedure to the cipertext C_1. In other words,

$$(M_1^{'})^n = (C_0^{'})^n \oplus (\text{DEC}(K, C_n))^n$$

Given that the padding is correct, it must mean that this last byte is exactly 1

$$\begin{aligned}(M_1^{'})^n &= (C_0^{'})^n \oplus (\text{DEC}(K, C_n))^n \\ &= C_0^n \oplus A \oplus 1 \oplus (\text{DEC}(K, C_n))^n \\ &= 1\end{aligned}$$

For this to be the case we need that

$$(\text{DEC}(K, C_n))^n = C_0^n \oplus A$$

Which means that the last byte of the original plaintext M_1 is exactly A. So, to recap, we have shown that if the oracle replies to us with the value 1, then we know that our guess is correct.

What if instead the oracle replies to us with the value 0? Then the padding is not correct, which means that the guess we made on the last byte of M_1 was not correct.

The idea of the attack then is to perform an iterative attack, trying all 256 values for the last byte of M_1, until we find the correct byte and get a correct padding message from the oracle. After discovering the first byte, say it is the byte C, we can then discover the second. This time the new ciphertext block $C_0^{'}$ will be constructed as follows for a new guess A

$$C_0^{'} := C_0^1 \ , \ C_0^2 \ , \ C_0^3 \ , \ \dots \ , \ C_0^{n-1} \oplus A \oplus 2 \ , \ C_0^n \oplus C \oplus 2$$

Notice how this time the last two bytes of the original C_0 change. By iterating this attack, we can break k bytes with $256 \cdot k$ requests.
We are now ready to face the third and final challenge.

1.2 Challenge # 3: Yet Another Oracle

This challenge is made up of two python scripts: a private script, *secret.py*, that contains the a secret flag, and a TCP server *server.py*, that implements a CBC padding oracle using the padding scheme *PKCS # 7*. While the *secret.py* file should not be disclosed, as it contains the secret flag, the code for the server *server.py* should be made public. The objective of the challenge is to write another script, called *solution.py*, which implements a CBC padding oracle attack and which allows us to decrypt the flag value in order to solve the challenge.

```
[leo@archlinux challenge_3] ls -lha
totale 16K
drwxr-xr-x 2 leo users 4,0K 26 mag 22.06 .
drwxr-xr-x 7 leo users 4,0K 26 mag 22.06 ..
-rw-r--r-- 1 leo users   14 26 mag 21.19 secret.py
-rw-r--r-- 1 leo users 2,2K 26 mag 21.18 server.py
```

In terms of dependencies, once again the only dependency is the *pcryptodome* library, which can be installed with *pip* as follows

```
pip install pycryptodome==3.14.1
```

As we have done for the previous challenges, before reading the code let us understand how the challenge works in with a black-box approach. Starting the *server.py* script we see the following output

```
[leo@archlinux challenge_3] python3 server.py
[INFO] - Start of challenge: Yet Another Oracle
[INFO] - Listening on 4444...
```

By connecting with a TCP client such as *nc* to the given port of the *localhost* interface, we get

```
Hi, I've been told to show you this.
═══════════════════════════════════════════════

ENCRYPTED_FLAG WITH CBC-AES: eB93iBsbwI6bGwZXpjF+
    SeolzIvMx4bL7Erkl0qsNao=

>
```

As we can see, we see the encrypted flag, and we are told that it was encrypted with the AES block cipher in CBC mode. We also see the start of a prompt. By playing with it a bit we see the following behavior

```
Hi, I've been told to show you this.
═══════════════════════════════════════════════

ENCRYPTED_FLAG WITH CBC-AES: eB93iBsbwI6bGwZXpjF+
    SeolzIvMx4bL7Erkl0qsNao=

> a
NOPE
> hello
NOPE
> 09jsd8H0CdmrtxXpSjk9lwQ/UNPgxiCwMZyWY1dZ0tc=
OK!
>
```

Notice how this behavior is very similar to the first challenge, where we had on oracle on the PCKS#1 v1.5 padding of RSA encrypted messages. If we send random bytes, we get a "NOPE", but if we send exactly the

original encrypted message, we get an "OK!" back. This suggests that the server implements some kind of checks, and if those checks are passed, then we get the positive outcome "OK!", otherwise we get the negative outcome "NOPE". Given the binary output, looks like we have an oracle.

To understand if our deduction was correct, we can now view the code for the challenge.

As usual, the *server.py* script starts off by importing various libraries as well as the *FLAG* variable from the *secret.py* file, which is not available to us.

```
#!/usr/bin/env python3

import binascii
import socketserver
import signal
import time
from hashlib import md5
from base64 import b64decode
from base64 import b64encode

from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

from secret import FLAG
```

After that we see the definition of a series of global variables

```
IV = None
KEY = None
CIPHER = None
ENCRYPTED_FLAG = None
```

If we jump at the end of the script, we see that when invoked directly from the command line the script executes the *challenge_3_main()* function

```
if __name__ == "__main__":
    challenge_3_main()
```

The function generates a random *IV* and *KEY* and then instantiates a new cipher using the block cipher AES in CBC mode. After that the flag is encrypted and a TCP server is started on port 4444.

```
def challenge_3_main():
    global IV, KEY, CIPHER, ENCRYPTED_FLAG

    IV = get_random_bytes(AES.block_size)
    KEY = get_random_bytes(AES.block_size)
    CIPHER = AES.new(KEY, AES.MODE_CBC, IV)

    data_to_encrypt = b"A" * 16 + FLAG
    ENCRYPTED_FLAG = b64encode(CIPHER.encrypt(pad(data_to_encrypt,
        AES.block_size)))

    print("[INFO] - Start of challenge n. 3: Yet Another Oracle")
    print("[INFO] - Listening on 4444...")

    socketserver.TCPServer.allow_reuse_address = True
    server = ReusableTCPServer(("0.0.0.0", 4444), incoming)
    server.serve_forever()
```

Notice how before encryptign the flag, 16 bytes were appended before the bytes of the flag itself. This is done because, as we explained previously, when executing a CBC padding oracle the first block cannot be attacked whenever the attacker is not able to see the IV used, and only the subsequent blocks are vulnerable.

As always, the TCP implementation is standard and it uses the *ReusableTCPServer* class

```
    class incoming(socketserver.BaseRequestHandler):
        def handle(self):
                signal.alarm(300)
                req = self.request
                while True:
                        challenge(req)

class ReusableTCPServer(socketserver.ForkingMixIn, socketserver.
    TCPServer):
        pass
```

The most important function is the *challenge* function, which is executed as soon as the user connects to the server of the challenge. The first thing

the function does is send in the TCP socket the initial welcoming message with the encrypted flag

```
def challenge(req):
    global IV, KEY, CIPHER, ENCRYPTED_FLAG

    req.sendall(b'Hi, I\'ve been told to show you this.\n' +\
                b'==========\n\n' +\
                b'ENCRYPTED_FLAG WITH CBC-AES: ' +\
                ENCRYPTED_FLAG + b'\n\n> ')
    time.sleep(0.2)
```

Then the server starst to listen for the user data, and whenever the user sends something, the function *oracle()* is called, and a response is prepared according to its output: if the function returns *True*, then the server sends back "OK!" to the client, otherwise it sends "NOPE".

```
    while True:
        try:
            client_payload = req.recv(4096)
            if len(client_payload) > 0:
                oracle_output = oracle(client_payload)
                if oracle_output == True:
                    req.sendall(b'OK!\n> ')
                else:
                    req.sendall(b'NOPE\n> ')
        except Exception as e:
            print(e)
            exit()
```

To finish off, the function *oracle* is the function that, as the name suggest, implements the actual CBC padding oracle.

```
def oracle(payload):
    try:
        payload = b64decode(payload)
        CIPHER = AES.new(KEY, AES.MODE_CBC, IV)
        decrypted_payload = CIPHER.decrypt(payload)
        unpadded_payload = unpad(decrypted_payload, AES.block_size)
        return True
    except ValueError as e:
        return False
```

The payload of the client is base64 decoded, and then its decrypted using the same *IV* and *KEY* used for encrypting the flag. Notice how the function *unpad* is used to remove the *PKCS #7* padding. If the function fails because the plaintext is not correctly padded, a *ValueError* exception is thrown, and the oracle returns *False*. Otherwise, if everything goes according to plan and if the plaintext is correctly padded, then the oracle returns *True*.

As we had previously deducted, we can now be certain that the server implements a CBC padding oracle with *PKCS #7* padding. This leads us inevitably to the solution, which will be written in another python script named *solution.py*.

The solution starts off by importing various libraries, many of which were also imported by the server.

```
#!/usr/bin/env python3

import binascii
import socketserver
import signal
import time
import socket
import string
from hashlib import md5
from base64 import b64decode
from base64 import b64encode

from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
```

Then we define a series of global variables

```
HOST = "127.0.0.1"
PORT = 4444
SOCK = None
ENCRYPTED_MSG = None
BLOCK_SIZE = 16
```

Notice in particular the *BLOCK_SIZE* variable, which contains the number of bytes in each block of the block ciphers that we want to attack. Given

that the server uses the AES block cipher, and that AES mainly works with block of 16 bytes, we set the variable also to 16. The *SOCK* variable instead is used to interact with the server.

Continuing, we find the *query_oracle()* function, which implements the main abstraction used to interact with the server.

```
def query_oracle(payload):
    global SOCK

    encoded_payload = b64encode(payload)
    SOCK.send(encoded_payload)
    oracle_reply = SOCK.recv(4096)

    if b"OK" in oracle_reply:
        return True
    else:
        return False
```

As we can see, the function is used to send arbitrary payloads. It assumes to receive bytes in input, and so, before sending them, it encodes them in base64. After the payload is sent, the function waits for the server reply, and depending on the reply it either returns *True*, when the server replies with "OK", or it returns *False*, when the server replies with "NOPE".

Then we find the *decrypt_block()* function. This function takes three inputs: the encrypted text to decrypt, an index n which tells which block of the encrypted text we want to decrypt, where the first block is indexed by 1, the second by 2 and so on. The third and last argument of the function is an optional argument called *block_size*, which is used to specify the size of each block. Given that the attacker cannot decrypt the first block, the function quits whenever the n argument is less than or equal to 1.

```
def decrypt_block(encrypted_text, n, block_size=8):
    """ Decrypts the n-th block of the encrypted_text """
    assert n > 1, "Cannot decrypt first block as we don't know the
        IV"
```

The first thing that the function does is check whether or not the block we want to decrypt is the last one or not.

```
last_block = True if int(len(encrypted_text) / block_size) == n
    else False
```

This check is done because the last block is the only block that is properly padded by default, and if not properly handled it could cause a false positive answer when guessing the last byte with the values of `0x00` or `0x01`, since then the effect of the XOR operation would effectively cancel itself, the encrypted payload would be the same as the original one, and therefore the server would reply to us with a false positive "OK".

Continuing, we save in the *saved_bytes* variable the block that sits right before the one we want to decrypt, and we initialize another variable *guess_so_far* that will hold pieces of the plaintext obtained when decrypting the block. As the decryption continues, the *guess_so_far* will expand until the whole block is completely decrypted.

```
saved_bytes = bytes(encrypted_text[block_size * (n-2):
    block_size * (n-1)])
guess_so_far = [0] * block_size
```

At this point we have the main loop of the attack, which in a way is very similar to the nested loop implemented for the BEAST attack in the previous challenge. The outer loop goes over all bytes of the block we want to decrypt. For each byte we have have an inner loop that goes over all the 256 possibilities for that byte.

```
for i in range(0, block_size):
    msg = list(encrypted_text[:block_size * n])
    found = False
    for c in range(0, 256):
        if last_block and i == 0 and c == 1:
            continue
```

The first we do in the inner loop is to check if we are in the last block, so as to avoid false positive that could ruin the entire decryption process.

Then we compute the new correct value for the padding, we insert our guess in the current guess for the block, and we prepare the message to send to the server depending on how far we are in the decryption of the current block

```
for j in range(0, i + 1):
global_index = block_size * (n - 1) - j - 1
relative_index = block_size - j - 1
msg[global_index] = saved_bytes[relative_index] ^ guess_so_far[
    relative_index] ^ padding
```

After the message is prepared, we send it to the server, and depending on the answer we set the variable *found* to true and we break out of the loop after printing to the screen the byte that was found.

```
if r:
    found = True
    if chr(c) in string.printable:
        print(f"[{n}]: byte {block_size - i} is: {c}, {chr(c)}")
    else:
        print(f"[{n}]: byte {block_size - i} is: {c}, non-
            printable byte")
    break
```

Having exited from the inner loop, we finish to handle the special case when the block to decrypt is the last one.

```
if not found and last_block and i == 0:
    guess_so_far[block_size - 1 - i] = 1
```

At the end of the outer-loop, after all the characters have been found, we return the decrypted block

```
return guess_so_far
```

To finish off the solution, the only function that needs to be discussed is the *challenge_3_solution_main()* function, which puts all the pieces together.

The function starts by connecting to the server TCP socket and extracting the encrypted flag.

```
def challenge_3_solution_main():
    global HOST, PORT, SOCK, ENCRYPTED_MSG, BLOCK_SIZE

    SOCK = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    SOCK.settimeout(300)
    SOCK.connect((HOST, PORT))

    server_data = SOCK.recv(4096)
    start_delimiter = b"\n\nENCRYPTED_FLAG WITH CBC-AES: "
    end_delimiter = b"\n\n> "
    i = server_data.find(start_delimiter) + len(start_delimiter)
    j = server_data.find(end_delimiter)

    ENCRYPTED_MSG = server_data[i:j]
```

Then the number of total blocks of the encrypted flag are computed, and, starting from the second block onwards, each block is decrypted using the *decrypt_block* function.

```
    encrypted_msg_bytes = b64decode(ENCRYPTED_MSG)
    num_blocks = int(len(encrypted_msg_bytes) / BLOCK_SIZE)
    plaintext = []
    for i in range(2, num_blocks + 1):
        plaintext += decrypt_block(encrypted_msg_bytes, i,
            block_size=BLOCK_SIZE)
```

Once we have all the plaintext, we can simply print it and close the socket

```
    plaintext = "".join(map(lambda x : chr(x), plaintext))
    print(f"Plaintext obtained is: {plaintext}")

    SOCK.close()
```

And this concludes the solution.

```
if __name__ == "__main__":
    challenge_3_solution_main()
```

By executing the solution we get the following output

```
[leo@archlinux challenge_3] python3 solution.py
[2]: byte 16 is: 84, T
[2]: byte 15 is: 80, P
[2]: byte 14 is: 49, 1
[2]: byte 13 is: 82, R
[2]: byte 12 is: 67, C
[2]: byte 11 is: 78, N
[2]: byte 10 is: 51, 3
[2]: byte 9 is: 45, -
[2]: byte 8 is: 78, N
[2]: byte 7 is: 51, 3
[2]: byte 6 is: 72, H
[2]: byte 5 is: 84, T
[2]: byte 4 is: 45, -
[2]: byte 3 is: 67, C
[2]: byte 2 is: 52, 4
[2]: byte 1 is: 77, M
[3]: byte 16 is: 3, non-printable byte
[3]: byte 15 is: 3, non-printable byte
[3]: byte 14 is: 3, non-printable byte
[3]: byte 13 is: 83, S
[3]: byte 12 is: 85, U
[3]: byte 11 is: 48, 0
[3]: byte 10 is: 82, R
[3]: byte 9 is: 51, 3
[3]: byte 8 is: 71, G
[3]: byte 7 is: 78, N
[3]: byte 6 is: 52, 4
[3]: byte 5 is: 68, D
[3]: byte 4 is: 45, -
[3]: byte 3 is: 83, S
[3]: byte 2 is: 49, 1
[3]: byte 1 is: 45, -
Plaintext obtained is: M4C-TH3N-3NCR1PT-1S-D4NG3R0US
```

As we can see, the final flag is

"M4C-TH3N-3NCR1PT-1S-D4NG3R0US"

2 Bibliography

[1] Nadhem J. Al Fardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

[2] Peter Gutmann. *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7366. Sept. 2014. DOI: 10.17487/RFC7366. URL: https://www.rfc-editor.org/info/rfc7366.

[3] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE Bites: Exploiting The SSL 3.0 Fallback". In: 2014.

[4] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT '02. Berlin, Heidelberg: Springer-Verlag, 2002, 534–546. ISBN: 3540435530.