ICR - labo 001

Date: April 10, 2016

Author: Joel Gugger

Email: joel.gugger@master.hes-so.ch

Task 1

Test suite for the server side

Make sure that all `.py` have execution privilege and python3 is installed on the machine. Run `chmod +x test_server.py && chmod +x server.py` to add privileges.

To run the server test suite:

1. start the server `./server.py`
2. run the test suite `./test_server.py`

Test case:

1. Legal message
2. Wrong timestamp
3. Wrong padding
4. Wrong MAC

Output:

```
Result expected : OK
[ANSWER IS: OK]

Result expected : WRONG TSP
[ANSWER IS: WRONG TSP]

Result expected : PADDING ERROR
[ANSWER IS: PADDING ERROR]

Result expected : MAC ERROR
[ANSWER IS: MAC ERROR]
```

Test suite for the client side

Make sure that all `.py` and bash script have execution privilege and python3 is installed on the machine. Run `chmod +x test_client.py && chmod +x client.py` and `chmod +x run_client_test.sh` to add privileges.

The `test_client.py` file is a modified version of the regular server. This version send responses containing volunteers errors to test the reaction of the client. A bash script `run_client_test.sh` is used to make all client's calls.

To run the server test suite:

1. start the server `./test_client.py`
2. run the test suite `./run_client_test.sh`

Test case:

1. Correct response message
2. Incorrect response length
3. Invalid response code
4. Invalid response timestamp
5. Incorrect response signature

Output:

```
[CORRECT RESPONSE EXPECTED]
[ANSWER IS: OK]

[INCORRECT LENGTH EXPECTED]
[ANSWER WITH INVALID LENGTH]

[INCORRECT CODE EXPECTED]
[ANSWER WITH INVALID CODE]

[INCORRECT TIMESTAMP EXPECTED]
[ANSWER WITH INVALID TIMESTAMP][TIMESTAMP = -1]

[INCORRECT MAC EXPECTED]
[ANSWER WITH WRONG MAC]
```

Questions 1

#1

Which strategy is using SSH? SSH use Encrypt-and-MAC

Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm

Which strategy is using TLS? TLS use MAC-then-Encrypt strategie

TLS Protocol Version 1.2

Which one of the above three strategies is it recommended to use in practice?

> ...theoretically "good" way is to apply the MAC on the encrypted data. This is called "encrypt-then-MAC". See this question on crypto.SE. As a summary, when you apply the MAC on the encrypted data, then whatever the MAC does cannot reveal anything on the plaintext data, and, similarly, since you verify the MAC before decrypting, then this will protect you against many chosen ciphertext attacks.

Source: Combining MAC and Encryption

#2

Instead of using a random IV, the CBC mode implements a nonce-based approach. What can you tell about its security?

First, the "nonce IV" must be impredictable. If we use the current timestamp, it's broken. Second, it must not be the same twice nonce. With this method, we use only 8 bits, we have much less opportunity and arrive more quickly when the nonce will be repeated.

Difference between a nonce and IV

Task 2

#1

Write a three-paragraphs summary of the history of padding oracle attacks.

The original attack was published in 2002 by Serge Vaudenay. In 2010 the attack was applied to several web frameworks, including JavaServer Faces, Ruby on Rails and ASP.NET. In 2012 it was shown to be effective against some hardened security devices. A new variant, the Lucky Thirteen attack, published in 2013, used a timing side-channel to re-open the vulnerability even in implementations that had previously been fixed. As of early 2014, the attack is no longer considered a threat in real-life operation.

#2

Explain how a malicious client can turn the server into a decryption oracle able to decrypt any observed encrypted packet (without the malicious client knowing the encryption key).

In the padding oracle attack, the client uses the server responses to determine if the padding of the cipher text sent is right. In this way, by changing the bits of the first "custom" block (C1) used by the CBC mode, we can search and find to have the value right value for the clear text block (P2'). WWith this value we can gradually determined the intermediate state (I2) of the

searched block (C2).

The intermediate state in deciphering is after the secret key, but before the XOR with the previous block. Knowing the intermediate state (I2) and the cipher text of the previous block (C1), we can retreave the clear text (P2).

Source of inspiration: [The Padding Oracle Attack - why crypto is terrifying]

#3

Write a program in the language of your choice simulating this attack.

The implementation of this attack have been maded in python, by modify a little the server side. The check for the timestamp and HMAC have been disabled to facilities the work. To run the attack:

1. start the server for testing padding oracle attack `./server_oracle.py`
2. run the malicious client `./malicious_client.py`

Scenario

An hacker have sniffed a cipher text providing in a other client:

```
\x00\x00\x00\x00\x00\x00\x00\x00\x0c\x8a\r=&\xf6z\xea\xea\xa1i\xb7Y\xfan\xb3E\xc
fH<\xccwQ\xc7b\xed\xe3b3\xa2\x9c~\xda\xea\x06\x04\x97s\xd5\x92\xe8.\xe6\xd3c\x12
\xb64\xa2J\xdd\xa3\xee\x9fBq\xd8\x15Hl\x11\xc8\xa2^-\xd6k\x05\xf3\xa7\t\xb6\x90\
xdd`\xc8j\x80\xb6\x0eG\x08\x83{J#[\xb9\xdc\xdcc\xcf\xb0J\xac>\xa4W\xdf\xff\x87\x
c8\x8f3\xf6\xa3\x0c
```

this cipher text is croped in multiple blocks (header, first block and second block). The malicious client tried to deciphering the second block of 16 bytes.

Result expected:

```
$ ./malicious_client.py
1
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00e')
2
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xc1e')
3
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1b\xc1e')
4
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x95\x1b\xc1e')
5
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00)\x95\x1b\xc1e')
6
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x97)\x95\x1b\xc1e')
7
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x0c\x97)\x95\x1b\xc1e')
8
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\xc5\x0c\x97)\x95\x1b\xc1e')
9
bytearray(b'\x00\x00\x00\x00\x00\x00\xca\xc5\x0c\x97)\x95\x1b\xc1e')
10
bytearray(b'\x00\x00\x00\x00\x00\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
11
bytearray(b'\x00\x00\x00\x00\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
12
bytearray(b'\x00\x00\x00\x97\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
13
bytearray(b'\x00\x00\x00U\x97\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
14
bytearray(b'\x00\x00NU\x97\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
15
bytearray(b'\x00hNU\x97\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
16
```

```
bytearray(b'\xe7hNU\x97\x1d\x8f\xca\xc5\x0c\x97)\x95\x1b\xc1e')
bytearray(b'message de pour ')
```

The number represent the progression 1..16. The bytearray store the value of I2, the intermediate state of de deciphering block C2. When we have the complete bytearray we can compute:

`P2 = C1 ^ I2`

in python:

```
i = 0
while i < 16:
    result[i] = cipher0[i] ^ i2[i]
    i += 1
```

Deciphering message:

> message de pour

Question 2

What is the average complexity of your attack in terms of queries to the padding oracle for decrypting a n-byte message?

Normaly, we can't change or know the IV as it should be. This has the effect of not being able to deciphering the first block in certain case. For each restant blocks, they are 256 possibilites for eache byte. So I think, maximum `( n - 16 ) * 256`

Task 3

#1

Propose a modification of the protocol that makes it resistant to padding oracle attacks, while keeping as much as possible the same functionalities.

> Namely, a "padding oracle" leaks some information about secret data through how it reacts to maliciously crafted invalid input. A good protocol will first validate the input data through a MAC before considering decryption and its corollary, padding processing. That's what the "encrypt-then-MAC" construction is about.

Source: [How to protect against “padding oracle attacks.”]

In the implementation I have desactived the timestamp and HMAC checks, but the code is working the same. If we can catch the padding error before other error, we can determine if the padding is ok, even if an error is throw.

To fix against this attack we can check HMAC before checking padding.

#2

before:

```
def decrypt_check_data (data):
    global TIMESTAMP

    # First, we check that the data length looks like to be correct
    # 0x00 + TTTTTTTT + I + MMMMMMMMMM = 20 bytes
    if (len(data) - 19) % 16 != 0:
        return (None, None)
    else:
        ## Data parsing

        # Checking that first byte is 0x00
        if data[0] != 0:
            return (None, None)
        r_timestamp  = data[1:9]
        r_ciphertext = data[9:-10]
        r_tag        = data[-10:]
```

```
        _

        # Checking that the timestamp is acceptable
        t = struct.unpack('>q', r_timestamp)
        if (t[0] > TIMESTAMP):
            # We accept and update it
            print ("[ACCEPTED TIMESTAMP = %s]" % str(t[0]))
            set_timestamp (t[0])
        else:
            return (None, prepare_error_message("TSP"))

        IV = b'\x00' * 8 + r_timestamp
        r_cleartext = AES.new (KEY_E, AES.MODE_CBC, IV).decrypt (r_ciphertext)

        # Checking padding
        count = r_cleartext[-1]
        if count == 0 or count > 16:
            return (None, prepare_error_message("PAD"))
        else:
            for i in range (1, count+1):
                if r_cleartext[-i] != count:
                    return (None, prepare_error_message("PAD"))

            # Checking authentication tag
            if (HMAC.new(KEY_A, r_cleartext, SHA256).digest()[:10] != r_tag):
                # INVALID MAC
                return (None, prepare_error_message ("MAC"))
            else:
                return (r_cleartext[:-count], prepare_OK_message())
```

after fix:

```
def decrypt_check_data (data):
    global TIMESTAMP

    # First, we check that the data length looks like to be correct
    # 0x00 + TTTTTTTT + I + MMMMMMMMM = 20 bytes
    if (len(data) - 19) % 16 != 0:
        return (None, None)
    else:
        ## Data parsing

        # Checking that first byte is 0x00
        if data[0] != 0:
            return (None, None)
        r_timestamp  = data[1:9]
        r_ciphertext = data[9:-10]
        r_tag        = data[-10:]

        # Checking that the timestamp is acceptable
        t = struct.unpack('>q', r_timestamp)
        if (t[0] > TIMESTAMP):
            # We accept and update it
            print ("[ACCEPTED TIMESTAMP = %s]" % str(t[0]))
            set_timestamp (t[0])
        else:
            return (None, prepare_error_message("TSP"))

        IV = b'\x00' * 8 + r_timestamp
        r_cleartext = AES.new (KEY_E, AES.MODE_CBC, IV).decrypt (r_ciphertext)

        # Checking authentication tag
        if (HMAC.new(KEY_A, r_cleartext, SHA256).digest()[:10] != r_tag):
```

```
            # INVALID MAC
            return (None, prepare_error_message ("MAC"))
        else:
            # Checking padding
            count = r_cleartext[-1]
            if count == 0 or count > 16:
                return (None, prepare_error_message("PAD"))
            else:
                for i in range (1, count+1):
                    if r_cleartext[-i] != count:
                        return (None, prepare_error_message("PAD"))
            return (r_cleartext[:-count], prepare_OK_message())
```

Question 3

#1

What is the security role of the timestamp in the protocol?

It allows you to assuer that the initialization vector is always different.

#2

Can you think a way for a malicious engineer to trigger a denial-of-service attack on the device?

We can crash the server in many different ways:

1. Set the first byte to different that `0x00`
2. Make an overflow on the timestamp (i.e set to `\xff\xff\xff\xff\xff\xff\xff\xff`) with `./client.py 9223372036854775807 ""`
3. Send a message that have a size of 15 bytes
4. ...

Task 4

List all remaining security vulnerabilities you have found in this protocol.

1. Check the padding before the MAC (padding oracle attack)
2. Truncate the HMAC to 10 left digits (collisions)
3. Timestamp owerflow (DoS)
4. Predictible IV
5. IV based only on 8 bytes