Assignment: CBC-MAC

PV079, Autumn 2022 | Deadline Nov 01 2022, 23:59 (CET)

Introduction

The topic of this assignment are the Message Authentication Codes (MAC) driven by AES cipher in CBC mode (CBC-MAC). Your goal will be to exploit misconfigured CBC-MAC instances. By doing so, you will learn the difficulties of using CBC-MAC. The assignment is divided into three tasks, for which you can gain up to 10 points.

Task 1 (1 point)

In this task, the initialization vector (IV) for AES is always set to 16 zero bytes. Someone decided that it might be a great idea to use CBC-MAC as a hash function, making the underlying key k public. You are to explore one of the reasons why this is actually a bad idea (ignore other reasons now, like the insufficient block length of AES). Given a hash h_1 of a message m_1 *that fits one block*, you should submit m_1 as a solution to this assignment (unpad all zero bytes from the end), conducting a pre-image attack. The hash was obtained as

$$h_1 = \text{CBC-MAC}_k(m_1).$$

Task 2 (3 points)

In this task, the IV for AES is always set to 16 zero bytes. You are given two messages m_2, m_3 and their corresponding MACs (under unknown key k):

$$t_2 = \text{CBC-MAC}_k(m_2),$$
$$t_3 = \text{CBC-MAC}_k(m_3).$$

Further denote $m_4 = m_2||m_3$, where $||$ is concatenation. You are guaranteed that $16 \mid \text{length}(m_2)$. Your goal is to perform MAC forgery. In exact, submit $m_5 \neq m_4$ such that $t_5 = t_4$, where

$$t_4 = \text{CBC-MAC}_k(m_4),$$
$$t_5 = \text{CBC-MAC}_k(m_5).$$

In other words, you should create a message m_5 different from m_4 such that they have an identical MAC.

Task 3 (6 points)

Background story

Imagine a scenario when one of the tutors wrote his own server to communicate with a fictitious bank of MUNI. That bank establishes a static AES symmetric key that is used for all communication with the given customer. Tired of the bank's interface, the tutor extracted this key and wrote a server to redirect encrypted transaction commands to the bank. The tutor's server handles both the encryption of the commands (using the key shared between the tutor and the bank) and also their redirection to the bank. As the name suggests, the transaction commands are meant to realize financial transactions between the students/employees of MUNI. Your goal will be to send carefully crafted HTTP requests to tutor's server, such that they will lead to a transaction execution at the bank. You only need to communicate with the server of the tutor; no need to get involved directly with the bank.

Transaction commands

The transaction commands are strings and have the following structure:

/trans?to=<uco>&from=<uco>&amount=<number>&id=<16 hex encoded bytes>

For example, the following constitutes a valid command

/trans?to=344&from=410390&amount=10000&id=8befaba643c56ab578acc55085b73690

The command above, when executed at the bank, will transfer 10000 CZK from someone with UCO 410390 to someone with UCO 344.

Command encryption

The commands for the bank (and for the endpoint `/send` of pv079.fi.muni.cz) are encrypted using the following scheme. Denote a transaction command as pt. The resulting ciphertext ct is constructed as

$$\text{ct} = \text{nonce}||\text{AES-CTR}_{k,\text{nonce}}(\text{pt})||\text{IV}||\text{CBC-MAC}_{k,\text{IV}}(\text{pt}),$$

where nonce is a random 16-byte nonce sequence used for encryption with AES in counter (CTR) mode; IV is a random 16-byte long initialization vector used for CBC-MAC construction, k is the key shared between the bank and the customer; and $||$ is concatenation. Before executing this command, the bank will first validate it:

1. Given the UCO of the sender (don't worry about how the bank learns the UCO of the sender, this is handled by the tutor's server), the bank will decrypt the command with k,
2. the bank will then check the authenticity of the plaintext by looking at the CBC-MAC value,
3. finally, if the plaintext constitutes a valid transaction command, it will be executed.

What you have

You have obtained the following:

1. A working copy of the tutor's server that provides `/encrypt` and `/send` endpoints, available at pv079.fi.muni.cz.
2. a single ciphertext (ct) of a valid transaction command that the tutor never sent to the bank.

What do /encrypt and /send do?

/encrypt: takes two hexadecimally encoded parameters, `msg, iv`. It will then encrypt the `msg` parameter using the tutor's key k and the `iv` parameter according to the encryption scheme for the transaction commands outlined above. The ciphertext will be returned as a response, hexadecimally encoded. The following example shows how you can call this endpoint from Python:

```python
import requests

enc_response = requests.get(
    "https://pv079.fi.muni.cz/encrypt",
    # NOTE the following values are used only for the purpose of the example
    params={"msg":"aa", "iv":"00"},
)
print(enc_response.json())
```

/send: takes one hexadecimally encoded parameter, `encrypted_command`. The tutor's server will take this command and do the same command validation as the bank would do. If the parameter constitutes a valid encrypted command, the server will pass it to the bank that will execute the transaction. The following example shows how you can call this endpoint from Python:

```python
import requests

send_response =  requests.post(
    "https://pv079.fi.muni.cz/send",
    # NOTE the following values are used only for the purpose of the example
    data={"encrypted_command":"aa"},
)
print(send_response.json())
```

Full documentation of the endpoints is at the end of this document.

What is your goal

Your goal is to decrypt the intercepted ciphertext ct, and change the recipient in the underlying transaction to your UCO. You shall then send the modified command ct′ using the `/send` endpoint. You submit both the recovered plaintext and the ciphertext of the modified command. E.g., if you manage to obtain the plaintext of the intercepted command

$$\text{pt} = \text{/trans?to=344\&from=410390\&amount=10000\&id=8befaba643c56ab578acc55085b73690}$$

then you should submit **this** plaintext, and also a **ciphertext** ct′ of

$$\text{pt}' = \text{/trans?to=XXXXXX\&from=410390\&amount=10000\&id=8befaba643c56ab578acc55085b73690.}$$

where XXXXXX stands for your UCO. You can earn

- 3 points for submitting the plaintext of the encrypted command ct,
- 1 point for submitting a valid ciphertext of pt′ under key k,
- 2 additional points if $\text{CBC-MAC}_{k,\text{IV}}(\text{pt}) = \text{CBC-MAC}_{k,\text{IV}'}(\text{pt}')$, for IV' in your pt′. In other words, we ask you to submit a command with the *same MAC that the original command has*.

Submission format

Pack all the source code that you used into one ZIP file together with a file `results_XXXXXX.json` where XXXXXX stands for your UCO. This JSON file shall contain:

- *Task 1:* Description (< 500 characters) of how you solved the task, and m_1 (unpad all zero bytes from the end).
- *Task 2:* Description (< 500 characters) of how you solved the task, and m_5.
- *Task 3:* Description (< 500 characters) of how you solved the task, the plaintext pt, and the modified encrypted command ct′.

The file `results_XXXXXX.json` **must be** of the following format:

```
{
    "task_one": {
        "description": "I solved this task by...",
        "m1": "two words"
    },
    "task_two": {
        "description": "I solved this task by...",
        "m5": "0123456789ABCDEF..."
    },
    "task_three": {
        "description": "I solved this task by...",
        "plaintext": "/trans?from=000000...",
        "modified_encrypted_command": "0123456789ABCDEF..."
    }
}
```

For correct submission, just replace the values with your solution. In particular, do not remove or add json keys. Each value should be one of the following:

- A hexadecimal string[1] (`m5`, `modified_encrypted_command`)
- A human readable string (descriptions, `m1`, `plaintext`)
- If you do not wish to submit a value, or the whole task (e.g. you skipped the given task) use `null` instead e.g. `"m1": null`, or `"task_one": null`

For your convenience, we have prepared a short script `validate_format.py` that will check that your submission has the correct format. You can run it with

```
python validate_format.py your_zip.zip
```

Make sure to check your submission as any deviations from the stated format will be heavily **penalized**.

[1] Hexadecimal string should be here represented without the 0x prefix and spaces.

Notes

- All CBC-MAC messages in this assignment are padded with zero-byte padding. This means that no padding is added when message length is a multiple of block length; otherwise, necessary number of zero bytes is added to the message. AES-CTR messages are not paddded. All AES instances use 128-bit long keys.
- The underlying keys are different for each task. Only the key for Task 1 is known to you.
- The choice of a programming language is up to you. However, if marking your solution will require a deep investigation or even modification of your code, we support only the following: Python, C/C++, and Rust.
- You can check that you adhere to the referential implementation of CBC-MAC, AES-CTR, and padding functions from `referential_encryption_implementation.py`.
- The assignment contains *no* tasks that require non-trivial computing power.
- There is no need to reverse-engineer the server, to exploit it, or to overly load it. Several calls to the endpoints are fully sufficient. Any excessive load of the server may lead to IP blocking.
- All the tasks are to be solved for your personal values that were generated individually. You can find the values for your UCO in IS notebook `HW03_inputs`.
- Beware, the requirements for JSON submission are slightly different from those in Assignment 2
- As always, feel free to ask any question in the discussion forum for this assignment.

Good luck!

PV079 Team

Task 2 Server API documentation

`/encrypt` endpoint is documented here, `/send` endpoint on the next page.

get **/encrypt?{msg, iv}**

Encrypt the `msg` parameter with AES-CTR and attach CBC-MAC (computed with `iv` parameter).

Parameter

msg	Hexadecimally encoded sequence of bytes to encrypt.
iv	Optional. Hexadecimally encoded sequence of bytes to use as IV for CBC-MAC.

Response application/x-www-form-urlencoded

200 ok

```
{"result": ciphertext (hex-encoded)}
```

400 Error: Bad request

```
{"Error": "The 'msg' GET parameter was not present, but is required.}
{"Error":" The 'msg' GET parameter was not recognized as HEX encoded bytes."}
{"Error":"The 'iv' GET parameter was not recognized as HEX encoded bytes."}
{"Error": "The byte-length of 'iv' GET parameter is not 16."}, 400
```

405 Error: Method not allowed

```
{"Error": "Use the GET HTTP method"}
```

post **/send?{encrypted_command}**

Send the `encrypted_command` parameter to the bank.

Parameter

encrypted_command Hexadecimally encoded transaction command to send to bank.

Response application/x-www-form-urlencoded

200 ok

```
{"result": "The command was accepted by the bank."}
```

400 Error: Bad request (Possible reasons below)

```
{"Error": "The command was not accepted by the bank."}
{"Error": "The 'encrypted_command' POST parameter was not present, but is required."}
{"Error":"The 'encrypted_command' POST parameter was not recognized as HEX encoded bytes
    ."}
{"Error": "The 'encrypted_command' POST parameter is too short to constitue a valid
    ciphertext."}
```

405 Error: Method not allowed

```
{"Error": "Use the POST HTTP method"}
```

500 Error: Internal server error

```
{"Error": "There was an unknown error. Please, contact the author of the assignment if
    it prevails."}
```