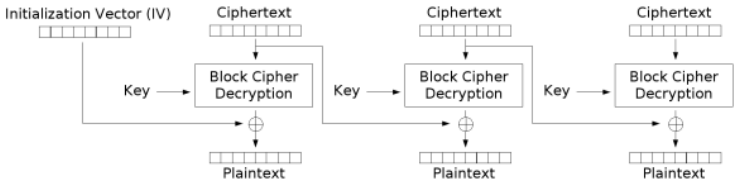
# ICR 2014-2015

# Practical Work #1

# *MAC-and-Encrypt and Padding Oracles*



Cipher Block Chaining (CBC) mode decryption

## Rules of the Game

This practical work can be performed by *groups of two*, or *individually*. A written report must be delivered, that will contain your answers to the questions, all the code you wrote (that must be commented), an introduction, a conclusion, and if necessary, screenshots. The report can be written in French, German, or English, and it must be delivered as a PDF file named

`lab01_report_Lastname(s).pdf`.

The code must be delivered as a ZIP file named

`lab01_code_Lastname(s).zip`.

The two files must be sent by electronic mail to

`pascal.junod@heig-vd.ch`

before

**Monday November 10th 2014, 18h00 CET.**

Being up to one day late will cost you one grade point, from one day to two days will cost two grade points, etc. Please do not forget to cite all your sources in a clear and precise manner!

# 1 Preliminaries

As a cryptography engineer, you are working on the security of the command interface of an industrial device that drives and monitors a nuclear reactor. This device is supposed to activate or deactivate the use of potentially very dangerous nuclear material and to give back some highly confidential information such as the current internal level of radiation, temperature and pressure of the nuclear reactor, for instance.

In the following, one assumes that the client used by the engineer responsible to send commands to and get answers from the device share two 256-bit symmetric keys with it: one, $k_c$, is used for encrypting the communications and one, $k_a$, is used to authenticate them. The communication protocol involves command packets (i.e., flowing from the client to the device), whose first byte is fixed to 0x00 and having the following format:

|0|TTTTTTTT|I|C...C|P...P|MMMMMMMMMM|

where TTTTTTTT denotes an 8-byte timestamp, I a 1-byte command ID, C...C a variable-length command payload, P...P padding bytes, MMMMMMMMMM a 10-byte MAC and | the concatenation operator. Answer packets (i.e., flowing from the device to the client) have a similar format, however, they begin with a byte equal to 0xFF, instead of 0x00.

The timestamp bytes are interpreted as a monotonly increasing counter, which means that both the client and the device must store the last observed timestamp value and accept new messages only if the timestamp is strictly larger than any previously observed timestamp. Otherwise, the command or answer packets must be ignored and destroyed.

Each packet is encrypted using AES-256 in CBC mode under key $k_c$. The 128-bit IV is built as 8 bytes set to 0x00 concatenated with the timestamp: IV = 00000000TTTTTTTT. For all types of packets, the bytes |I|C...C| are padded with sufficiently many additional bytes in order to get a multiple of 16 bytes. If one byte is missing, then one byte equal to 0x01 is appended. If two bytes are missing, then two bytes equal to 0x02 are appended. If 15 bytes are missing, then 15 bytes equal to 0x0F are appended. If no byte is missing, then 16 bytes equal to 0x10 are appended. In this manner, the padding can be removed in an unambiguous manner.

The packet payload data are authenticated using HMAC-SHA256 under key $k_a$. More precisely, a HMAC-SHA256 value, truncated to the 10 most significant (leftmost) bytes, is computed on the bytes |I|C...C| before encryption, and the MAC value is appended at the end of the CBC-encrypted packet (formed by the MMMMMMMMMM bytes).

Upon reception of an encrypted and authenticated packet, one proceeds as follows: first, the packet is decrypted. Then, the padding is removed. If this operation fails for some reason, an encrypted and authenticated error packet having the following format is sent back:

|E|TTTTTTTT|0...0|MMMMMMMMMM|

where E is a byte set to the value 0x0A and 0...0 is a 16-byte string of zeroes. If the padding removal operation was successful, the HMAC-SHA256 of the received and decrypted data is computed and compared to the received value of MMMMMMMMMM. If this operation fails, an encrypted and authenticated error packet having the following format is sent back:

|E|TTTTTTTT|0...0|MMMMMMMMMM|

where E is a byte set this time to the value 0x0B and 0...0 is a 16-byte string of zeroes. If the MAC check operation is successful, then the command or the answer is accepted and processed.

## 2 Programming a Protocol Simulator

The first of your tasks consists in writing a client, simulating the engineer, able to encrypt and authenticate arbitrary commands and a server able to receive, decrypt and check the authenticity of the received packets, and, if necessary, send error packets. In the following, one assumes that both the client and the server send random values of $|\texttt{I}|\texttt{C}\ldots\texttt{C}|$ with a varying length of 8 to 40 bytes.

**Task 1.**

1. Write a client/server simulator of this protocol in the programming language of your choice.
2. Write a test suite showing that your simulator works as specified by the protocol, including in case of error.

Given a message $M$, one can think of three distinct strategies to encrypt and authenticate data:

- The first one is called `MAC-and-Encrypt` and computes $\texttt{Enc}(M)|\texttt{MAC}(M)$.
- The second one is called `MAC-then-Encrypt` and computes $\texttt{Enc}(M|\texttt{MAC}(M))$.
- Finally, the third one is called `Encrypt-then-MAC` and computes $\texttt{Enc}(M)|\texttt{MAC}(\texttt{Enc}(M))$.

Here, $|$ denotes the concatenation operator.

**Question 1.**

1. Which one of the above three strategies is it recommended to use in practice? Which strategy is using TLS? SSH?
2. What is the security role of the timestamp in the protocol?
3. Can you think a way for a malicious engineer to trigger a denial-of-service attack on the device?
4. Instead of using a random IV, the CBC mode implements a nonce-based approach. What can you tell about its security?
5. Do you have other remarks concerning the overall security level of this protocol?

## 3 Transforming the Server in a Decrypting Oracle

This protocol is unfortunately vulnerable to a *padding oracle attack*.

**Task 2.**

1. Write a three-paragraphs summary of the history of padding oracle attacks.
2. Explain how a malicious client can turn the server into a decryption oracle able to decrypt any encrypted packet (without the client knowing the encryption key).
3. Write a program in the language of your choice running this attack.

**Question 2.**

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

## 4 Fixing the Protocol and your Simulator

**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.
2. Modify the implementation of your server in order to implement the tweak you propose and to make it as resistant as possible to attacks exploiting padding oracles.