Guião 3

Cifra Autenticada de Ficheiro

Pretende-se melhorar a funcionalidade no programa de cifra do ficheiro para garantir simultaneamente

- a confidencialidade dos dados e
- a integridade da informação.

Numa primeira abordagem, vamos estabelecer essas garantias combinando as técnicas criptográficas já estudadas: *cifra simétrica* e *MAC*. A
questão que surge é como combinar essas primitivas, sendo que é concebível considerar diferentes possibilidades:

- **encrypt and MAC**: onde tanto cifra como o MAC são aplicados sobre o texto limpo;
- **encrypt then MAC**: onde o texto limpo passa originalmente pela cifra, e o MAC é calculado já sobre o criptograma;
- **MAC then encrypt**: onde é primeiro calculado o MAC sobre o texto limpo, e só depois é cifrado (texto limpo e *tag* de autenticação).

Na realidade, a garantias de segurança obtidas diferem consideravelmente -- a alternativa **encrypt-then-MAC** é a que oferece melhores garantias, sendo que a alternativa **encrypt and MAC** é normalmente considerada "insegura".

> **Note**
>
> Para mais detalhes sobre as diferentes alternativas, sugerem-se os seguintes apontadores:
>
> - https://en.wikipedia.org/wiki/Authenticated_encryption#Approaches_to_authenticated_encryption
> - Secção "Message Authentication Codes" de https://raw.githubusercontent.com/crypto101/crypto101.github.io/master/Crypto101.pdf e, em particular atente à sub-secção "Authenticate-and-encrypt"
> - https://crypto.stackexchange.com/questions/202/should-we-mac-then-encrypt-or-encrypt-then-mac

PROG: `pbenc_aes_ctr_hmac.py`

Adapte o programa `pbenc_chacha20.py` realizado no guião anterior por forma utilizar a cifra AES no modo CTR e adicionar um MAC segundo a estratégia **encrypt-then-MAC**. Sugere-se a utilização do HMAC, definido sobre a função de hash `SHA256`.

Note que irá precisar de uma nova chave simétrica para o MAC (em criptografia, nunca se devem reutilizar chaves criptográficas para fins distintos). Isso pode ser facilmente ultrapassado solicitando "mais bytes" à KDF utilizada para derivar a chave.

Utilização de cifra autenticada

Actualmente, a generalidade das bibliotecas criptográficas já oferecem a funcionalidade de **Cifra Autenticada**, que combina as garantias de confidencialidade e integridade, em modos de funcionamento próprios e/ou em combinações pré-definidas.

PROG: `pbenc_chacha20_poly1305.py` e `pbenc_aes_gcm.py`

Adapte a solução realizada no ponto anterior para fazer uso de cifras autenticadas (ChaCha20Poly1305, e AES-GCM).

QUESTÃO: Q1

Qual o impacto de executar o programa `chacha20_int_attck.py` sobre um criptograma produzido por `pbenc_chacha20_poly1305.py` ? Justifique.

(EXTRA) Ataque ao CBC-MAC

Recorde o referido na aula teórica reference ao modo CBC:

- (...) o último bloco de criptograma do modo CBC pode ser utilizado como um MAC (CBC-MAC).
- (...) esse método só é seguro para mensagens de comprimento fixo

Considere **atentamente** a descrição do CBC-MAC e a questão da vulnerabilidade com mensagens de comprimento variável em: https://en.wikipedia.org/wiki/CBC-MAC

Imagine o seguinte cenário: num contexto onde existe apenas necessidade de integridade/autenticidade (sem confidencialidade), um adversário constatou que o CBC-MAC estaria a ser utilizado em mensagens de comprimento variável. Entretanto teve acesso a 2 pares de (mensagem, tag): `(m1,t1)` e `(m2,t2)` .

O adversário pretende agora atacar o MAC, ou seja, **forjar um tag válido para uma mensagem nova construída por ele**. Em concreto, ele irá produzir um nova mensagem `m3` , diferente de `m2` , mas que será validada pelo *tag* `t2` .

PROG: `cbc-mac-attck.py`

Altera o programa apresentado abaixo, ajustando a criação de `m3` (ou seja, removendo a linha `m3 = bytearray(len(m1)+len(m2))` e implementando o código necessário de tal forma que `r5` seja `True`). Se executar o programa tal como está deverá observar a impressão de `False` no terminal. Submeta o ficheiro alterado com o nome `cbc-mac-attck.py` .

```
import sys, os, base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding


def cbcmac_auth(m_bytes, k):
    # CBC mode requires padding
    padder = padding.PKCS7(128).padder()
    padded_m = padder.update(m_bytes) + padder.finalize()
    iv = bytearray(16) # zero-filled IV
    cipher = Cipher(algorithms.AES(k), modes.CBC(iv))
    encryptor = cipher.encryptor()
    ct = encryptor.update(padded_m) + encryptor.finalize()
    tag = ct[-16:] # last block of ciphertext
    return tag

def cbcmac_verify(tag, m_bytes, k):
    padder = padding.PKCS7(128).padder()
    padded_m = padder.update(m_bytes) + padder.finalize()
    iv = bytearray(16)
    cipher = Cipher(algorithms.AES(k), modes.CBC(iv))
```

```
    encryptor = cipher.encryptor()
    ct = encryptor.update(padded_m) + encryptor.finalize()
    newtag = ct[-16:]
    return tag == newtag

def cbcmac_lengthextension_example(m1, m2):
    key = os.urandom(32)
    tag1 = cbcmac_auth(m1,key)
    tag2 = cbcmac_auth(m2,key)
    #print(m1, end=" ; tag : ")
    #print(base64.b64encode(tag1))
    #print(m2, end= " ; tag : ")
    #print(base64.b64encode(tag2))
    # check if tag1 verifies with m1, m2 / tag2 with m1, m2 :
    r1 = cbcmac_verify(tag1, m1, key)
    r2 = cbcmac_verify(tag1, m2, key)
    r3 = cbcmac_verify(tag2, m1, key)
    r4 = cbcmac_verify(tag2, m2, key)
    #print("tag1 + m1: " + str(r1))
    #print("tag1 + m2: " + str(r2))
    #print("tag2 + m1: " + str(r3))
    #print("tag2 + m2: " + str(r4))
    # create a m3 (based on m1 and m2 that verifies with tag2)
    m3 = bytes(len(m1)+len(m2))
    r5 = cbcmac_verify(tag2, m3, key)
    #print("tag2 + m3: " + str(r5))
    return r5

def main(args = sys.argv):
    if len(args) != 3:
        print("Utilização: python3 cbc-mac.py <msgs1> <msg2>")
    else:
        print(cbcmac_lengthextension_example(args[1].encode('utf-8'), args[2].encode('utf-8')))

if __name__ == '__main__':
    main()
```

Tip

Durante o desenvolvimento, podem descomentar os `#print(...)` do código para obterem mais *feedback* da estratégia do ataque. Mas não esqueça de comentar ou apagar essas linhas na versão submetida.

Tip

Utilize uma mensagem `m2` com mais de 16 byte.

QUESTÃO: Q2

Qual o motivo da sugestão de usar `m2` com mais de 16 byte? Será possível contornar essa limitação?