Trabalho Prático 1 - Grupo 15

João Gonçalves - pg46535

Sara Queirós - pg47661

Exercício 1

1. Use o "package" Cryptography para
   A. Implementar uma AEAD com "Tweakable Block Ciphers" conforme está descrito na última secção do texto +Capítulo 1: Primitivas Criptográficas Básicas. A cifra por blocos primitiva, usada para gerar a "tweakable block cipher", é o AES-256 ou o ChaCha20.
   B. Use esta construção para construir um canal privado de informação assíncrona com acordo de chaves feito com "X448 key exchange" e "Ed448 Signing&Verification" para autenticação dos agentes. Deve incluir uma fase de confirmação da chave acordada.

Autenticação dos agentes - Ed448 Signing&Verification

Com o esquema de assinatura da curva elíptica de Edwards, é possível utilizar um algoritmo que instancie os parâmetros necessários à autenticação dos agentes envolvidos. Assim, o **receiver**:

In [ ]:
```
#Inicialmente é gerada a chave privada segundo este algoritmo
def generate_Ed448_private_key(self):
    return Ed448PrivateKey.generate()

#A partir da privada gera-se a pública
def generate_Ed448_public_key(self):
    return self.Ed448_private_key.public_key()

#Com uma mensagem de assinatura definida pelo receiver, cria-se a codificaçã
# gerando a assinatura
def generate_Ed448_signature(self):
    return self.Ed448_private_key.sign(self.signing_message)
```

Dada um assinatura introduzida pelo utilizador, o emitter cria a partir dela, a sua assinatura com a chave privada. Para o **receiver** verificar a mensagem, ele recebe a chave pública e a assinatura produzida e deve ser capaz de confirmar se corresponde assinatura gerada que recebeu corresponde à que foi introduzida pelo utilizador:

In [ ]:
```
#Verificar se as assinaturas batem certo entre si
def verify_Ed448_signature(self, signature, public_key):
        try:
            public_key.verify(signature, self.signing_message)
        except: #InvalidSignature:
            raiseExceptions("Autenticação dos agentes falhou!")
```

X448 key exchange

Após a verificação de assinaturas, inicia-se o processo de troca de chaves entre as duas partes. Para isso, geram-se as chaves privadas e públicas. Com a partilha da chave pública entre emitter e receiver, gera-se a partilhada, através de uma KDF, neste caso com o SHA256.

In [ ]:
```
def generate_X448_private_key(self):
    # Generate a private key for use in the exchange.
    return X448PrivateKey.generate()

def generate_X448_public_key(self):
    return self.X448_private_key.public_key()

def generate_X448_shared_key(self, X448_emitter_public_key):
    key = self.X448_private_key.exchange(X448_emitter_public_key)
    self.X448_shared_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(key)
```

Feito isto, é necessário verificar o acordo de chaves. Para tal, o emitter produz um ciphertext da key que o receiver deve ser capaz de decifrar e verificar:

In [ ]:
```
#emitter
def key_to_confirm(self):
        nonce = os.urandom(16)
        #Cifra a chave partilhada com ChaCha20
        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        encryptor = cipher.encryptor()
        ciphered = encryptor.update(self.X448_shared_key)
        #Envia o nonce e chave cifrada
        ciphered = nonce + ciphered
        return ciphered

#receiver
def confirm_key(self, cpht):
        #16 bytes reservados para o nonce
        nonce = cpht[0:16]
        #o restante do texto cifrado corresponde à key
        key = cpht[16:]
        #Utilização do Chacha20 para decifrar a mensagem
        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        decryptor = cipher.decryptor()
        d_key = decryptor.update(key)
        #Se corresponder à chave partilhada :
        if d_key == self.X448_shared_key:
            print("\nChaves acordadas com sucesso!\n")
        else:
            raiseExceptions("Erro na verificacao das chaves acordadas")
```

“Tweakable Block Ciphers”

Realizado todo o acordo e verificação de assinaturas e chaves, é necessário cifrar a mensagem que se quer enviar. Para isso utilizamos Tweakable Block Ciphers, em que, segundo o Capítulo 1, se deve gerar um tweak para cada bloco. Assim o **emitter** possui a seguinte função para cifrar:

In [ ]:
```
#função que cria a mensagem de autenticação com o SHA256
def create_authentication(self, message):
        h = hmac.HMAC(self.X448_shared_key, hashes.SHA256(), backend=default
        h.update(message)
        self.mac = h.finalize()

#Função que gera os tweaks
def generate_tweak(self, contador, tag):
    #Um tweak é constituído por 8 bytes de nonce + 7 de contador + 1 de tag
    #Tal como diz no capítulo 1
    nonce = os.urandom(8) #Utiliza-se um nonce para dar aleatorieadade à pri
    return nonce + contador.to_bytes(7,byteorder = 'big') + tag.to_bytes(1,b



def encodeAndSend(self):
    #Guardar o tamanho da mensagem
    size_msg = len(self.message)
    # Add padding à msg
    padder = padding.PKCS7(64).padder()
    padded = padder.update(self.message) + padder.finalize()
    cipher_text = b''
    contador = 0
    #Dividir a mensagem em blocos de 16
    for i in range(0,len(padded),16):
        p=padded[i:i+16]
        #Se corresponder ao último bloco
        if (i+16+1 > len(padded)):
            #Ultimo bloco com tag 1
            tweak = self.generate_tweak(size_msg,1)
            cipher_text += tweak
            middle = b''
            for index, byte in enumerate(p):
                #aplicar a máscara XOR aos blocos . Esta mascara é compostas
                mascara = self.X448_shared_key + tweak
                middle += bytes([byte ^ mascara[0:16][0]])
            cipher_text += middle
        #Se não for o último bloco
        else:
            #Blocos intermédios com tag 0
            tweak = self.generate_tweak(contador,0)
            #O bloco é cifrado com AES256, num modo de utilização de tweaks
            cipher = Cipher(algorithms.AES(self.X448_shared_key), mode=modes
            encryptor = cipher.encryptor()
            ct = encryptor.update(p)
            cipher_text += tweak + ct
        contador += 1
    #a mensagem final cifrada é composta por tweak(16)+bloco(16)
    print("size:", len(cipher_text))

    #Adicionalmente é enviada uma secção de autenticação para verificação an
    self.create_authentication(cipher_text)
    final_ciphered = self.mac + cipher_text
    return final_ciphered
```

Inversamente ao que o emitter fez, o **receiver** tem de verificar a autenticidade da

mensagem e decifrá-la da mesma forma:

In [ ]:

```
    #verifica se a assinatura que vem no texto cifrado corresponde ao que el
    # de acordo com a chave acordada com os tweaks
    def verify_authenticate_message(self, mac_signature, ciphertext):
        h = hmac.HMAC(self.X448_shared_key, hashes.SHA256(), backend=default
        h.update(ciphertext)
        h.verify(mac_signature)

    #recebe um tweak e decompõe, de forma a extrair a posição do bloco e o ú
    def degenerate_tweak(self, tweak):
        #8 bytes nonce + 7 bytes do numero do bloco + 1 byte tag final
        nonce = tweak[0:8]
        contador = int.from_bytes(tweak[8:15], byteorder = 'big')
        tag_final = tweak[15]
        return nonce, contador, tag_final


    def receiveAndDecode(self, ctt):
        #primeiros 32 bytes são de autenticação
        mac = ctt[0:32]
        ct = ctt[32:]
        try:
            #Verificar se a mensagem mac enviada corresponde ao esperado
            self.verify_authenticate_message(mac, ct)
        except:
            raiseExceptions("Autenticação com falhas!")
            return
        #Se correponder, temos de a decifrar da mesma forma que foi cifrada:

        plaintext = b''
        f = b''
        print("size of the received:" , len(ct))

        #no total: bloco + tweak corresponde a corresponde a 32 bytes.
        tweak = ct[0:16]
        block = ct[16:32]
        i = 1
        _, contador, tag_final = self.degenerate_tweak(tweak)
        #Se não for o último bloco:
        while(tag_final!=1):
            #decifrar com o algoritmo AES256 e o respetivo tweak
            cipher = Cipher(algorithms.AES(self.X448_shared_key), mode=modes
            decryptor = cipher.decryptor()
            f = decryptor.update(block)
            plaintext += f
            #obtem o proximo tweak e o proximo bloco
            tweak = ct[i*32:i*32 +16]
            block = ct[i*32 +16:(i+1)*32]
            #desconstroi o proximo tweak
            _, contador, tag_final = self.degenerate_tweak(tweak)
            i+= 1
        #Se for o ultimo bloco
        if (tag_final == 1):
            c =b''
            for index, byte in enumerate(block):
                #aplicar as máscaras XOR aos blocos para decifrar
                mascara = self.X448_shared_key + tweak
                c += bytes([byte ^ mascara[0:16][0]])
            plaintext += c

        #realiza o unpadding
        unpadder = padding.PKCS7(64).unpadder()
```

```
        unpadded_message = unpadder.update(plaintext) + unpadder.finalize()

        #Uma vez que o último bloco possui o tamanho da mensagem cifrada, ba
        # se correspondem os valores e não houve perdas de blocos da mensage
        if (len(unpadded_message.decode("utf-8")) == contador):
            print("Tweak de autenticação validado!")
            return unpadded_message.decode("utf-8")
        else: raiseExceptions("Tweak de autenticação inválido")
```

Invocação da AEAD

De forma a que a utilização destas funções faça sentido, utilizamos uma classe main que as invoca e interliga da seguinte forma:

In [ ]:
```
assinatura = input("Introduz a assinatura a utilizar:")
mensagem = input("Introduz a mensagem a cifrar:")
emitter = emitter(mensagem, assinatura)
receiver = receiver(assinatura)

#Autenticacao dos agentes
receiver.verify_Ed448_signature(emitter.signature, emitter.Ed448_public_key)

#Setup do Key exchange (X448)
emitter.generate_X448_shared_key(receiver.X448_public_key)
receiver.generate_X448_shared_key(emitter.X448_public_key)

# Verificar se as chaves foram bem acordadas
key_ciphertext = emitter.key_to_confirm()
receiver.confirm_key(key_ciphertext)


#Envio da mensagem cifrada e a respetiva decifragem dela
ciphertext = emitter.encodeAndSend()
plaintext = receiver.receiveAndDecode(ciphertext)
print("Mensagem Decifrada: \n" , plaintext)
```