tp1-ex1

March 7, 2023

0.1 Trabalho Prático 1

Grupo 13, constituído por:
– Rui Monteiro, PG50739 – Rodrigo Rodrigues, PG50726

Use a package **Criptography** para

1. Criar um comunicação privada assíncrona entre um agente *Emitter* e um agente *Receiver* que cubra os seguintes aspectos:
    1. Autenticação do criptograma e dos metadados (associated data). Usar uma cifra simétrica num modo **HMAC** que seja seguro contra ataques aos "nounces" .
    2. Os "nounces" são gerados por um gerador pseudo aleatório (PRG) construído por um função de hash em modo XOF.
    3. O par de chaves **cipher_key**, **mac_key** , para cifra e autenticação, é acordado entre agentes usando o protocolo **ECDH** com autenticação dos agentes usando assinaturas **ECDSA**.

0.2 Imports

O código importa vários módulos do pacote cryptography para lidar com funções de criptografia e hash

[6]:
```
import secrets
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.hmac import HMAC
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric import utils, padding
```

0.2.1 PrivateKey

A classe PrivateKey é usada para gerar um par de chaves pública-privada, e é possível obter as chaves em formato de bytes para uso posterior. Também é possível assinar uma mensagem com a chave privada e verificar a assinatura com a chave pública.

[7]:
```
class PrivateKey:
    def __init__(self, curve):
        self._private_key = ec.generate_private_key(curve)
        self._public_key = self._private_key.public_key()
```

```
    def get_private_key_bytes(self):
        return self._private_key.private_bytes(
            encoding=serialization.Encoding.DER,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )

    def get_public_key_bytes(self):
        return self._public_key.public_bytes(
            encoding=serialization.Encoding.DER,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )

    def sign(self, message):
        signature = self._private_key.sign(
            message,
            ec.ECDSA(hashes.SHA256())
        )
        return signature


    def verify(self, signature, message):
        self._public_key.verify(
            signature,
            message,
            ec.ECDSA(hashes.SHA256())
        )
```

0.2.2 Protocolo Elliptic-curve Diffie-Hellman (ECDH)

A classe ECDH é usada para realizar o acordo de chave, gerando um par de chaves pública-privada e trocando a chave pública com a outra parte. Em seguida, o código usa o algoritmo HKDF para derivar as chaves de criptografia e MAC a partir da chave compartilhada.

[8]:
```
class ECDH:
    def __init__(self, curve):
        self._private_key = ec.generate_private_key(curve)
        self._public_key = self._private_key.public_key()
        self._shared_key = None

    def get_public_key_bytes(self):
        return self._public_key.public_bytes(
            encoding=serialization.Encoding.DER,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
```

```
    def compute_shared_key(self, peer_public_key_bytes):
        peer_public_key = serialization.
↪load_der_public_key(peer_public_key_bytes)
        self._shared_key = self._private_key.exchange(ec.ECDH(),␣
↪peer_public_key)

    def derive_cipher_and_mac_keys(self, info):
        if self._shared_key is None:
            raise Exception("Shared key has not been computed yet")
        cipher_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=info,
        ).derive(self._shared_key)

        mac_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b"mac key",
        ).derive(self._shared_key)
        return cipher_key, mac_key
```

0.2.3 Comunicação

A classe Communication é usada para implementar a comunicação segura entre as duas partes. Ela usa a classe **ECDH** para realizar o acordo de chave, e cifra e autentica as mensagens usando as chaves de criptografia e MAC derivadas.

[4]:
```
class Communication:
    def __init__(self, curve=ec.SECP384R1()):
        self.curve = curve
        self.emitter = ECDH(self.curve)
        self.receiver = ECDH(self.curve)
        self.emitter_private_key = PrivateKey(self.curve)
        self.receiver_private_key = PrivateKey(self.curve)

    def authenticate(self, message, signature, public_key_bytes):
        public_key = serialization.load_der_public_key(public_key_bytes)
        public_key.verify(signature, message, ec.ECDSA(hashes.SHA256()))

    def send_message(self, message):
        # generate nonce
        nonce = secrets.token_bytes(16)

        # compute shared key
```

```
        self.emitter.compute_shared_key(self.receiver.get_public_key_bytes())
        info = b"ciphersuite"
        cipher_key, mac_key = self.emitter.derive_cipher_and_mac_keys(info)

        # encrypt message
        cipher = Cipher(algorithms.AES(cipher_key), modes.CTR(nonce))
        encryptor = cipher.encryptor()
        encrypted_message = encryptor.update(message) + encryptor.finalize()

        # authenticate ciphertext and metadata
        hmac = HMAC(mac_key, hashes.SHA256())
        hmac.update(nonce + encrypted_message)
        tag = hmac.finalize()

        # sign nonce and ciphertext
        signature = self.emitter_private_key.sign(nonce + encrypted_message)

        # return encrypted message, tag, signature, and public key

        return encrypted_message, tag, signature, self.emitter_private_key.
↪get_public_key_bytes()


    def receive_message(self, encrypted_message, tag, signature,␣
↪public_key_bytes):
        # load public key and check if it matches the sender's public key
        sender_public_key = serialization.load_der_public_key(public_key_bytes)
        if sender_public_key != self.emitter_private_key._public_key:
            raise Exception("Invalid sender public key")

        # authenticate signature
        self.receiver_private_key.verify(signature, encrypted_message + tag)

        # compute shared key
        self.receiver.compute_shared_key(public_key_bytes)
        info = b"ciphersuite"
        cipher_key, mac_key = self.receiver.derive_cipher_and_mac_keys(info)

        # authenticate ciphertext and metadata
        hmac = HMAC(mac_key, hashes.SHA256())
        hmac.update(tag + encrypted_message)
        hmac.verify(tag)

        # decrypt message
        nonce = encrypted_message[:16]
        cipher = Cipher(algorithms.AES(cipher_key), modes.CTR(nonce))
        decryptor = cipher.decryptor()
```

```
        message = decryptor.update(encrypted_message[16:]) + decryptor.
↪finalize()

        return message
```

0.2.4 Definição da função principal `main`

Cria dois objetos de comunicação (*comm_emitter* e *comm_receiver*) usando a classe *Communication*.

Autentica as chaves públicas do emissor e do receptor, garantindo que elas sejam legítimas e pertençam às partes corretas.

O emissor envia uma mensagem cifrada para o receptor usando a chave pública do receptor.

O receptor verifica a assinatura da mensagem e a decifra usando sua própria chave privada.

[ ]:
```
def main():
    # create communication objects
    comm_emitter = Communication()
    comm_receiver = Communication()

    # exchange public keys and authenticate them
    comm_receiver.authenticate(
        comm_emitter.emitter.get_public_key_bytes(),
        comm_emitter.emitter_private_key.sign(
            comm_emitter.emitter.get_public_key_bytes()
        ),
        comm_emitter.emitter_private_key.get_public_key_bytes()
    )

    comm_emitter.authenticate(
        comm_receiver.emitter.get_public_key_bytes(),
        comm_receiver.emitter_private_key.sign(
            comm_receiver.emitter.get_public_key_bytes()
        ),
        comm_receiver.emitter_private_key.get_public_key_bytes()
    )

    # send a message from emitter to receiver
    message = b"Hello, world!"
    encrypted_message, tag, signature, public_key_bytes = comm_emitter.
↪send_message(message)

    # verify signature on message and decrypt it
    comm_receiver.authenticate(
        public_key_bytes,
        signature,
```

```
        comm_emitter.emitter_private_key.get_public_key_bytes()
    )
    decrypted_message = comm_receiver.receive_message(encrypted_message, tag,␣
 ↪signature, public_key_bytes)

    # output the decrypted message
    print(decrypted_message.decode("utf-8"))


main()
```