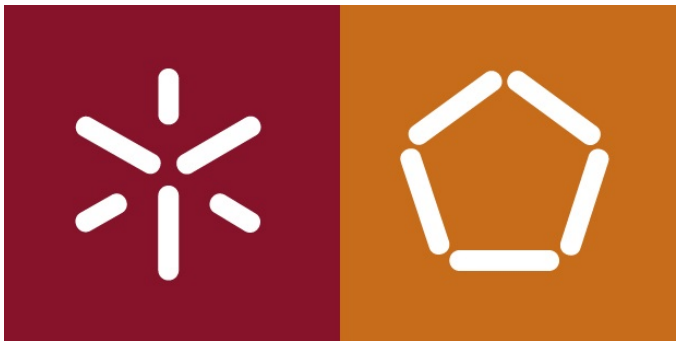
Universidade do Minho

Mestrado Integrado em Engenharia Informática -
Criptografia e Segurança da Informação



Estruturas Criptográficas

Relatório do Trabalho Prático 1- Grupo 9

Sessão síncrona de comunicação segura entre dois agentes

Carla Cruz
A80564

Adriana Meireles
A82582

March 16, 2020

Exercicio1

March 16, 2020

1 TP1: Sessão síncrona de comunicação segura entre dois agentes

1.1 Definição do problema

Neste trabalho prático pretende-se que seja realizada a construção de uma sessão síncrona de comunicação segura entre dois agentes (Emitter e Receiver) recorrendo à cifra simétrica *AES*,usando autenticação de cada criptograma com HMAC e um modo seguro contra ataques aos vectores de iniciação (iv's).

A primeira parte do trabalho consiste no uso do protocolo de acordo de chaves Diffie-Hellman com verificação da chave, e autenticação dos agentes através do esquema de assinaturas DSA.

Na segunda parte do trabalho, utilizam-se curvas elípticas, isto é, substitui-se o protocolo de acordo de chaves - o DH pelo *ECDH* e o DSA pelo *ECDSA*.

Por fim fizemos a limpeza da informção para que não seja possível a recuperação de informação sensível que permita recuperar a chave de sessão acordada.

1.2 Metodologia da solução

Primeiramente foi necessário realizar todos os import's que seriam necessários para a execução do programa e isto faz com que haja uma maior perceção caso nos falte algum.

O ficheiro Auxs contém todas as funções já previamente utilizadas no TP0, bem como o ficheiro BiConn. Desta forma conseguimos uma maior organização também pois separamos nestes ficheiros funções que partimos do princípio que já estão corretas e que iremos voltar a utilizar.

Dado que os parâmetros do Diffie-Hellman são partilhados por ambos os agentes, estes são declarados como variável global. Uma vez que o BiConn corre em modo automático, este não funciona se ambos os agente utilizarem o stdin, então é necessário que as passwords das chaves privadas dos agentes também sejam declaradas como variável global.

[206]:
```
from Auxs import cypher, decypher, hashs, kdf, mac
from BiConn import BiConn
from cryptography.exceptions import InvalidKey, InvalidSignature
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh, dsa
from cryptography.hazmat.primitives.ciphers.algorithms import AES
from cryptography.hazmat.primitives.ciphers import Cipher, modes
from cryptography.hazmat.primitives import hashes, hmac, serialization
from datetime import date
```

```
import ctypes, io, os, sys

# Geram-se os parâmetros para o Diffie-Hellman
parameters_dh = dh.generate_parameters(generator=2, key_size=1024,
                                       backend=default_backend())

password_Emitter = b'1234'
password_Receiver = b'4321'

default_algorithm = hashes.SHA256
```

1.2.1 Geração das chaves DSA

Em primeira instância, é necessário gerar as chaves DSA tanto públicas como privadas para os agentes de comunicação. É então na classe DSA que realizamos a geração destas chaves que por fim são armazenadas nas respetivas variáveis: `private_key` e `public_key`.

[207]:
```
class DSA(object):
    """
    Armazena o par de chaves DSA
    """
    def __init__(self):
        """
        Gera o par de chaves DSA
        """
        self.private_key = dsa.generate_private_key(
                            key_size=2048,
                            backend=  default_backend())

        self.public_key = self.private_key.public_key()
```

1.2.2 Armazenamento das chaves DSA em ficheiros

Numa instância da classe DSA criamos as chaves públicas e privadas quer para o Emitter quer para o Receiver sendo estas depois serializadas e armazenadas em ficheiros distintos no formato PEM. Após guardarem a informação pretendida, pressupondo que ambos os agentes obtiveram a chave pública de com quem pretendem comunicar através de um canal seguro, são armazenados na diretoria em que nos encontramos.

[208]:
```
def Gen_Key(priv_file="privKey.pem",pub_file="pubKey.pem", password=None):
    """
    Gera as chaves DSA publica e privada e armazena-as em ficheiro
    Argumentos:
        priv_file -- string com caminho para ficheiro que armazenará a chave␣
 ↪privada
        pub_file -- string com caminho para ficheiro que armazenará a chave␣
 ↪publica
```

```
        password -- string com a password da chave privada
    """
    dsa_object = DSA()

    pk_pem = dsa_object.private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.BestAvailableEncryption(password))

    pub_pem = dsa_object.public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    fd = open(priv_file,"wb"); fd.write(pk_pem); fd.close()

    fd = open(pub_file,"wb"); fd.write(pub_pem); fd.close()

#Gera o par de chaves do Emitter
Gen_Key("privadaEmitter.pem","publicaEmitter.pem",password_Emitter)
#Gera o par de chaves do Receiver
Gen_Key("privadaReceiver.pem","publicaReceiver.pem",password_Receiver)
```

1.3 Protocolo Diffie-Hellman com verificação da chave

A implementação da função responsável pelo acordo de chaves Diffie-Hellman foi feita de modo a ser utilizada de forma genérica por qualquer agente.

Cálculo e verificação da chave de sessão Primeiramente, é gerada a chave pública a partir dos parâmetros e é enviada para o outro agente. Seguidamente,é recebida a chave pública do outro agente e calculada a chave de sessão Diffie Hellman. Como esta chave não possui o tamanho indicado para o AES é aplicada uma função de hash sobre a mesma de modo a ficar com o tamanho correspondente à chave do AES. Posteriormente, o agente calcula o HMAC da chave e envia ao outro agente,verificando de seguida o HMAC enviado pelo outro agente. Desta forma, procede-se à verificação da chave de sessão.

Autenticação dos agentes Com a chave DSA privada, o agente assina a sua chave pública do DH concatenada com a chave pública DH do outro agente.Posteriormente, cifra a assinatura com a chave de sessão calculada anteriormente e envia ao outro agente. Após receber a assinatura cifrada do outro agente, decifra-a usando a chave de sessão. Por último, verifica a assinatura usando a chave DSA pública do outro agente.

[209]:
```
def dh(conn, dsa_private_key=None, dsa_peer_public_key=None):
    """
    Estabelece o protocolo Diffie-Helldman com autenticação dos agentes através␣
 ↪do
    esquema de assinaturas DSA.
    Argumentos:
```

```
        conn -- conexão entre os agentes
        dsa_private_key -- chave DSA privada para produzir assinatura
        dsa_peer_public_key -- chave DSA publica para verificar a assinatura do␣
↪outro agente
    Valor de retorno: chave de sessão Diffie-Hellman
    """
    # Em cada sessão é gerada uma chave publica e privada, enviando-se a␣
↪primeira ao outro agente
    privKey = parameters_dh.generate_private_key()
    pubKey = privKey.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)
    conn.send(pubKey)

    # Recebe a chave pública do outro agente e calcula a shared_key
    peer_pub_key_bytes = conn.recv()
    peer_pub_key = serialization.load_pem_public_key(
            peer_pub_key_bytes,
            backend=default_backend())

    sk = privKey.exchange(peer_pub_key)

    shared_key = hashs(sk)
    hmac_key = mac(shared_key,shared_key)

    conn.send(hmac_key)
    hmac_peer = conn.recv()

    mac(shared_key,shared_key,hmac_peer)

    # Se as chaves DSA foram passadas como parâmetro autenticam-se os agentes
    if dsa_private_key is not None and dsa_peer_public_key is not None:
        # O agente assina a shared_key com a sua chave privada DSA, de seguida␣
↪cifra esta
        # assinutura com a shared_key da sessão e envia para o outro agente␣
↪juntamente com o nonce
        signature = dsa_private_key.sign(pubKey + peer_pub_key_bytes, hashes.
↪SHA256())
        nonce, cypherSignature = cypher(shared_key,signature)
        conn.send((nonce,cypherSignature))

        # De seguida recebe este par do outro agente de modo a verificar a␣
↪shared_key
        (peer_nonce, peer_cypher_signature) = conn.recv()
```

```
        # Primeiro decifra o cyphertext retirando a assinatura enviada pelo␣
↪outro agente e de
        # seguida verifica a assinatura usando a chave pública DSA do outro␣
↪agente
        peer_signature = decypher(shared_key,peer_cypher_signature,peer_nonce)
        dsa_peer_public_key.verify(peer_signature,peer_pub_key_bytes + pubKey,␣
↪hashes.SHA256())

    return shared_key

    #Eliminar dados
    privKey = None
    pubKey = None
    peer_pub_key_bytes = None
    peer_pub_key = None
    shared_key = None
    hmac_peer
```

1.3.1 Cifra AES em modo CTR

O trabalho pede a utilização da cifra simétrica por blocos, o **AES**,num modo seguro contra ataques aos vectores de inicialização.

Para evitar estes ataques são utilizados vectores repetidos que permitem aos criptogramas de 2 mensagens com um prefixo comum preservem essa propriedade. O modo CBC, está sujeito a ataques quando o envio do vetor de inicialização é feito em plaintext, o que permite ao intruso alterar os bits do primeiro bloco caso altere os respetivos bits do vetor.

Desta forma, o modo de operação escolhido foi o CTR,em que o vector de inicialização é gerado aleatoriamente e concatenado com um contador.

1.3.2 Sessão síncrona de comunicação

Numa comunicação síncrona, cada bloco de informação é transmitido e recebido num instante de tempo bem definido e conhecido pelo transmissor e receptor, ou seja, estes têm que estar sincronizados. Deste modo, à medida que a mensagem vai sendo enviada pelo `Emitter` tem que ser recebida pelo `Receiver` numa determinada ordem : o primeiro bloco a ser enviado, tem que ser o primeiro a ser recebido e assim sucessivamente.

Tamanho dos blocos Para uma determinada mensagem,o `Emitter` divide a mesma em blocos de 256 bytes que vão sendo lidos sucessivamente, cifrados e enviados ao `Receiver`. De modo a diminuir o overhead no envio de cada pacote de comunicação, o tamanho de cada bloco lido foi escolhido de forma a que se possam enviar vários blocos da cifra (32 bytes) no mesmo pacote.

Autenticação de cada criptograma Na autenticação de cada criptograma gera-se um `HMAC` inicial com os metadados da comunicação e envia-se um `finalize` da cópia desse mac. Conforme os blocos forem sendo cifrados, atualiza-se o `HMAC` através do método `mac.update(ciphertext)` e envia-se um `finalize` da cópia desse mac juntamente com cada criptograma. À medida que cada

mac é recebido, o `Receiver` verifica-o e interrompe a comunicação caso algum mac se encontre errado.Por último, o `Emitter` gera um `finalize` do mac que foi acumulando toda a mensagem enviada e encaminha para o `Receiver` que o verifica e, de seguida,possa terminar a comunicação.

[210]:
```
message_size = 2**10

def Emitter(conn):
    """
    Agente que envia uma mensagem.
    """
    # Lê a chave DSA pública do Receiver para o autenticar
    with open("publicaReceiver.pem", "rb") as fd:
        dsa_peer_public_key = serialization.load_pem_public_key(
            fd.read(),
            backend=default_backend())
    fd.close()

    # O BiConn não funciona se ambos os processos utilizarem o stdin
    # password = bytes(getpass.getpass('password '),'utf-8')

    # Lê a sua chave DSA privada para se autenticar
    with open("privadaEmitter.pem", "rb") as fd:
        dsa_private_key = serialization.load_pem_private_key(
            fd.read(),
            password=password_Emitter,
            backend=default_backend())
    fd.close()

    # Estabelece a chave de sessão
    key = dh(conn,dsa_private_key,dsa_peer_public_key)

    # Cria um input stream com a mensagem a enviar
    inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

    # nonce para inicialização do modo CRT
    nonce  = os.urandom(16)

    # Dados associados
    dadosAssociados = bytes(str(date.today()),'utf-8')

    # geração da chave e do contexto de cifra
    cipher = Cipher(AES(key), modes.CTR(nonce),
                        backend=default_backend()).encryptor()

    # Gerar um mac e inicializa-lo com os dados associados
    mac = hmac.HMAC(key,default_algorithm(),default_backend())
    mac.update(dadosAssociados)
```

```
    # comunicação e operação de cifra
    conn.send((nonce,mac.copy().finalize(),dadosAssociados))
    # define um buffer para onde vão ser lidos, sucessivamente, os vários␣
↪blocos do input
    buffer = bytearray(256)

    # lê, cifra e envia sucessivos blocos do input
    try:
        while inputs.readinto(buffer):
            ciphertext = cipher.update(bytes(buffer))
            mac.update(ciphertext)
            conn.send((ciphertext, mac.copy().finalize()))

        conn.send((cipher.finalize(), mac.finalize()))    # envia a finalização
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    inputs.close()          # fecha a 'input stream'
    conn.close()

    #Eliminar dados
    key = None
```

[211]:
```
def Receiver(conn):
    """
    Agente que recebe a mensagem
    """
    # Lê a chave DSA pública do Emitter para autenticar o outro agente
    with open("publicaEmitter.pem", "rb") as fd:
        dsa_peer_public_key = serialization.load_pem_public_key(
            fd.read(),
            backend=default_backend())
    fd.close()

    # O BiConn não funciona se ambos os processos utilizarem o stdin
    #password = bytes(getpass.getpass('password '),'utf-8')

    # Lê a sua chave DSA privada para se autenticar
    with open("privadaReceiver.pem", "rb") as fd:
        dsa_private_key = serialization.load_pem_private_key(
            fd.read(),
            password=password_Receiver,
            backend=default_backend())
    fd.close()

    # Estabelece a chave de sessão
```

```
    key = dh(conn,dsa_private_key,dsa_peer_public_key)

    # Inicializa um output stream para receber o texto decifrado
    outputs = io.BytesIO()

    # Recuperar a informação de nonce e salt
    nonce,tag1,rec = conn.recv()

    # geração da chave e do contexto de cifra
    cipher = Cipher(AES(key), modes.CTR(nonce),
                            backend=default_backend()).decryptor()

    # Gera um mac e inicializa-lo com os dados associados
    mac = hmac.HMAC(key,default_algorithm(),default_backend())
    mac.update(rec)

    # Verifica se os dados associados não foram corrompidos
    mac.copy().verify(tag1)

    # operar a cifra: ler da conecção um bloco, autenticá-lo, decifrá-lo e␣
↪escrever o
    # resultado no 'stream' de output
    try:
        while True:
            try:
                ciphertext, tag = conn.recv()
                mac.update(ciphertext)
                if tag != mac.copy().finalize():
                    raise InvalidSignature("erro no bloco intermédio")
                if len(ciphertext) == 0:
                    raise EOFError
                outputs.write(cipher.update(ciphertext))


            except EOFError:
                if tag != mac.finalize():
                    raise InvalidSignature("erro na finalização")
                outputs.write(cipher.finalize())
                break
            except InvalidSignature as err:
                raise Exception("autenticação do ciphertext ou metadados: {}".
↪format(err))
        print(outputs.getvalue())     # verificar o resultado

    except Exception as err:
        print("Erro no receptor: {0}".format(err))
```

```
    outputs.close()     # fechar 'stream' de output
    conn.close()        # fechar a conecção

    #Eliminar dados
    key = None
```

[212]:
```
BiConn(Emitter,Receiver,timeout=30).auto()
```

```
b'111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111'
```

[ ]:

Exercicio2

March 16, 2020

1 Exercício 2: Sessão síncrona de comunicação segura entre dois agentes usando Curvas Elípticas

Neste segundo exercício, criamos uma versão do esquema do primeiro exercício recorrendo ao uso de Curvas Elípticas.

Assim, em vez do protocolo de acordo de chaves o Diffie–Hellman substituímos pelo Elliptic-curve Diffie–Hellman e o Digital Signature Algorithm pelo Elliptic Curve Digital Signature Algorithm. Desta forma, foram necessárias poucas alterações à variante que não usa curvas elípticas. Posto isto, as operações em que foram realizadas mais alterações foram as de inicialização das estruturas. Constatamos que a fase de inicialiação do algoritmo é mais rápida no caso das curvas elípticas.

[11]:
```
from Auxs import cypher, decypher, hashs, mac
from BiConn import BiConn
from cryptography.exceptions import InvalidKey, InvalidSignature
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric.ec import ECDH
from cryptography.hazmat.primitives.ciphers.algorithms import AES
from cryptography.hazmat.primitives.ciphers import Cipher, modes
from cryptography.hazmat.primitives import hashes, hmac, serialization
from datetime import date
import io, os

password_Emitter = b'1234'
password_Receiver = b'4321'

default_algorithm = hashes.SHA256
```

[12]:
```
class ECDSA(object):
    """
    Armazena o par de chaves ECDSA
    """
    def __init__(self):
        """
        Gera o par de chaves ECDSA
        """
        self.private_key = ec.generate_private_key(
```

```
                                ec.SECP384R1,default_backend())

        self.public_key = self.private_key.public_key()
```

[13]:
```
def Gen_Key(priv_file="pk.pem",pub_file="pub.pem", password=None):
    """
    Gera as chaves ECDSA publica e privada e armazena-as em ficheiro
    Argumentos:
        pk_file -- string com caminho para ficheiro que armazenará a chave␣
↪privada
        pub_file -- string com caminho para ficheiro que armazenará a chave␣
↪publica
        password -- string com a password da chave privada
    """
    ECDSA_object = ECDSA()

    pk_pem = ECDSA_object.private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.BestAvailableEncryption(password))

    pub_pem = ECDSA_object.public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    fd = open(priv_file,"wb"); fd.write(pk_pem); fd.close()

    fd = open(pub_file,"wb"); fd.write(pub_pem); fd.close()

#Gera o par de chaves do Emitter
Gen_Key("privadaEmitter.pem","publicaEmitter.pem",password_Emitter)
#Gera o par de chaves do Receiver
Gen_Key("privadaReceiver.pem","publicaReceiver.pem",password_Receiver)
```

[14]:
```
def ecdh(conn, ecdsa_private_key=None, ecdsa_peer_public_key=None):
    """
    Estabelece o protocolo ECDH com autenticação dos agentes através do esquema␣
↪de assinaturas
    ECDSA.
    Argumentos:
        conn -- conexão entre os agentes
        dsa_private_key -- chave DSA privada para produzir assinatura
        dsa_peer_public_key -- chave DSA publica para verificar a assinatura do␣
↪outro agente
    Valor de retorno: chave de sessão Diffie-Hellman
    """
```

```
    # Em cada sessão é gerada uma chave publica e privada, enviando-se a␣
↪primeira ao outro agente
    privKey = ec.generate_private_key(ec.SECP384R1,default_backend())
    pubKey = privKey.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)
    conn.send(pubKey)

    # Recebe a chave pública do outro agente e calcula a shared_key
    peer_pub_key_bytes = conn.recv()
    peer_pub_key = serialization.load_pem_public_key(
            peer_pub_key_bytes,
            backend=default_backend())

    sk = privKey.exchange(ec.ECDH(),peer_pub_key)
    shared_key = hashs(sk)

    hmac_key = mac(shared_key,shared_key)

    conn.send(hmac_key)
    hmac_peer = conn.recv()

    mac(shared_key,shared_key,hmac_peer)

    # Se as chaves ECDSA foram passadas como parâmetro autenticam-se os agentes
    if ecdsa_private_key is not None and ecdsa_peer_public_key is not None:
        # O agente assina a shared_key com a sua chave privada ECDSA, de␣
↪seguida cifra esta
        # assinutura com a shared_key da sessão e envia para o outro agente␣
↪juntamente com o nonce
        signature = ecdsa_private_key.sign(pubKey + peer_pub_key_bytes, ec.
↪ECDSA(hashes.SHA256()))
        nonce, cypher_signature = cypher(shared_key,signature)
        conn.send((nonce,cypher_signature))

        # De seguida recebe este par do outro agente de modo a verificar a␣
↪shared_key
        (peer_nonce, peer_cypher_signature) = conn.recv()

        # Primeiro decifra o cyphertext retirando a assinatura enviada pelo␣
↪outro agente e de
        # seguida verifica a assinatura usando a chave pública ECDSA do outro␣
↪agente
        peer_signature = decypher(shared_key,peer_cypher_signature,peer_nonce)
        ecdsa_peer_public_key.verify(peer_signature,peer_pub_key_bytes +␣
↪pubKey, ec.ECDSA(hashes.SHA256()))
```

```
    return shared_key

    #Eliminar dados
    privKey = None
    pubKey = None
    peer_pub_key_bytes = None
    peer_pub_key = None
    shared_key = None
    hmac_peer = None
```

[15]:
```
message_size = 2**10

def Emitter(conn):
    """
    Agente que envia uma mensagem.
    """
    # Lê a chave ECDSA pública do Receiver para o autenticar
    with open("publicaReceiver.pem", "rb") as fd:
        ecdsa_peer_public_key = serialization.load_pem_public_key(
            fd.read(),
            backend=default_backend())
    fd.close()

    # O BiConn não funciona se ambos os processos utilizarem o stdin
    # password = bytes(getpass.getpass('password '),'utf-8')

    # Lê a sua chave ECDSA privada para se autenticar
    with open("privadaEmitter.pem", "rb") as fd:
        ecdsa_private_key = serialization.load_pem_private_key(
            fd.read(),
            password=password_Emitter,
            backend=default_backend())
    fd.close()

    # Estabelece a chave de sessão
    key = ecdh(conn,ecdsa_private_key,ecdsa_peer_public_key)

    # Cria um input stream com a mensagem a enviar
    inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

    # nonce para inicialização do modo CRT
    nonce  = os.urandom(16)

    # Dados associados
    dadosAssociados = bytes(str(date.today()),'utf-8')
```

```
    # geração da chave e do contexto de cifra
    cipher = Cipher(AES(key), modes.CTR(nonce),
                        backend=default_backend()).encryptor()

    # Gerar um mac e inicializa-lo com os dados associados
    mac = hmac.HMAC(key,default_algorithm(),default_backend())
    mac.update(dadosAssociados)

    # comunicação e operação de cifra
    conn.send((nonce,mac.copy().finalize(),dadosAssociados))
    # define um buffer para onde vão ser lidos, sucessivamente, os vários␣
↪blocos do input
    buffer = bytearray(32)

    # lê, cifra e envia sucessivos blocos do input
    try:
        while inputs.readinto(buffer):
            ciphertext = cipher.update(bytes(buffer))
            mac.update(ciphertext)
            conn.send((ciphertext, mac.copy().finalize()))

        conn.send((cipher.finalize(), mac.finalize()))    # envia a finalização
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    inputs.close()          # fecha a 'input stream'
    conn.close()            # fecha a conecção

    #Eliminar dados
    key = None
```

[16]:
```
def Receiver(conn):
    """
    Agente que recebe a mensagem
    """
    # Lê a chave ECDSA pública do Emitter para autenticar o outro agente
    with open("publicaEmitter.pem", "rb") as fd:
        ecdsa_peer_public_key = serialization.load_pem_public_key(
            fd.read(),
            backend=default_backend())
    fd.close()

    # O BiConn não funciona se ambos os processos utilizarem o stdin
    #password = bytes(getpass.getpass('password '),'utf-8')

    # Lê a sua chave ECDSA privada para se autenticar
    with open("privadaReceiver.pem", "rb") as fd:
```

```
        ecdsa_private_key = serialization.load_pem_private_key(
            fd.read(),
            password=password_Receiver,
            backend=default_backend())
    fd.close()

    # Estabelece a chave de sessão
    key = ecdh(conn,ecdsa_private_key,ecdsa_peer_public_key)

    # Inicializa um output stream para receber o texto decifrado
    outputs = io.BytesIO()

    # Recuperar a informação de nonce e salt
    nonce,tag1,rec = conn.recv()

    # geração da chave e do contexto de cifra
    cipher = Cipher(AES(key), modes.CTR(nonce),
                            backend=default_backend()).encryptor()

    # Gera um mac e inicializa-lo com os dados associados
    mac = hmac.HMAC(key,default_algorithm(),default_backend())
    mac.update(rec)

    # Verifica se os dados associados não foram corrompidos
    mac.copy().verify(tag1)

    # operar a cifra: ler da conecção um bloco, autenticá-lo, decifrá-lo e␣
↪escrever o
    # resultado no 'stream' de output
    try:
        while True:
            try:
                ciphertext, tag = conn.recv()
                mac.update(ciphertext)
                if tag != mac.copy().finalize():
                    raise InvalidSignature("erro no bloco intermédio")
                if len(ciphertext) == 0:
                    raise EOFError
                outputs.write(cipher.update(ciphertext))


            except EOFError:
                if tag != mac.finalize():
                    raise InvalidSignature("erro na finalização")
                outputs.write(cipher.finalize())
                break
            except InvalidSignature as err:
```

```python
                raise Exception("autenticação do ciphertext ou metadados: {}".
↪format(err))
        print(outputs.getvalue())     # verificar o resultado

    except Exception as err:
        print("Erro no receptor: {0}".format(err))

    outputs.close()    # fechar 'stream' de output
    conn.close()       # fechar a conecção
    #Eliminar dados
    key = None
```

[17]:
```python
BiConn(Emitter, Receiver).auto()
```

```
b'111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111'
```

[ ]: