t1.0

April 5, 2021

1 T1

A85272	Jorge Mota
A83840	Maria Silva

2 1

Para este exercicio temos 2 entidades que comunicam entre si, reaproveitando o modelo de threads do TP anterior e adaptando-o para realizar uma derivação de segredos num canal inseguro assinando as mensagens.

Alguns imports relevantes para este exercicio…

[1]:
```
from threading import Thread
from queue import Queue
import os

from cryptography.hazmat.primitives import hashes, hmac
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes


# Diffie-Hellman & DSA
from cryptography.hazmat.primitives.asymmetric import dh, dsa, ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import Encoding,␣
↪ParameterFormat, PublicFormat
```

Como este exercicio tem duas versões em que se tem a derivação de chaves e a autenticação dos agentes, defenimos estas variáveis para trocar quais os algoritmos que se quer usar

[2]:
```
DH_AND_DSA = 0
ECDH_AND_ECDSA = 1
# THIS VARIABLE DEFINES WHICH METHOD WILL BE
# USED FOR KEY EXCHAGE
key_exchange_method = ECDH_AND_ECDSA
```

Para completar o progresso do TP anterior, criamos 2 classes, uma para ajudar no processo de comunicação entre as threads uma classe que apenas especifica um canal bidirecional de comunicação e uma classe para reaproveitar código de logs das threads

[1]:
```
class Channel:
    def __init__(self):
        self.e2r = Queue(5)
        self.r2e = Queue(5)

class Logger:
    def __init__(self,name):
        self.thread_name = name

    def sent(self,msg):
        print("[{}] Sent > {}".format(self.thread_name,msg))

    def received(self,msg):
        print("[{}] Received > {}".format(self.thread_name,msg))

    def log(self,msg):
        print("[{}] > {}".format(self.thread_name,msg))
```

Abaixo encontra se a defenição das funções para cifrar e decifrar as mensagens a qual é feita também a autenticação.

Como era requerido que a cifra AES usasse um modo *seguro contra ataques aos vectores de iniciação* foi escolhido o modo CTR.

Quanto à autenticação, é feita com HMAC como requerido, e usando um esquema de *Encrypt & MAC*

> **Nota:** 1. a função de cifragem retorna o resultado como um tuplo com o nounce para o modo counter e ainda o codigo de autenticação gerado 2. a função de decifragem vai lançar uma exceção (a mesma que o a biblioteca `cryptography` lança para o verify) quando a verificação falha e o código de autenticação está mal

[3]:
```
def encrypt(message,key):
    # Encrypt-and-MAC method
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    mac = h.finalize()

    # AES CTR mode
    nounce = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CTR(nounce))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(message) + encryptor.finalize()

    return (nounce,ciphertext,mac)
```

```python
def decrypt(nounce_ciphertext_mac,key):
    nounce = nounce_ciphertext_mac[0]
    ciphertext = nounce_ciphertext_mac[1]
    mac = nounce_ciphertext_mac[2]

    cipher = Cipher(algorithms.AES(key), modes.CTR(nounce))
    decryptor = cipher.decryptor()
    message = decryptor.update(ciphertext) + decryptor.finalize()

    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    h.verify(mac)

    return message
```

Para implementar a derivação de chaves fornecemos 4 funções auxiliares à execução normal das threads, 2 para cada agente, para o emitter: - `emitter_key_exchange` - `emitter_ec_key_exchange` para o receiver: - `receiver_key_exchange` - `receiver_ec_key_exchange`

`*_key_exchange` 'erelativo à troca de chaves por DH e `*_ec_key_exchange` é usando curvas elípticas

[3]:
```python
# Diffie-Hellman Key Exchange
def emitter_key_exchange(channel,logger):
    global emitter_private_key
    global receiver_public_key
    # Emiter generates some DH parameters and sends to Receiver
    parameters = dh.generate_parameters(generator=2, key_size=1024)
    logger.log("Generated parameters")

    signature = emitter_private_key.sign(
        parameters.parameter_bytes(Encoding.DER,ParameterFormat.PKCS3),
        hashes.SHA256()
    )
    channel.e2r.put((parameters,signature))
    logger.sent("DH parameters")

    # Entities generate both private and public keys
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    logger.log("Generated private and public key")

    # Send public_key to other peer
    signature = emitter_private_key.sign(
        public_key.public_bytes(Encoding.DER,PublicFormat.SubjectPublicKeyInfo),
        hashes.SHA256()
    )
```

```python
    channel.e2r.put((public_key,signature))

    # Receive other peer's public_key
    peer_public_key, signature = channel.r2e.get()
    try:
        receiver_public_key.verify(
            signature,
            peer_public_key.public_bytes(Encoding.DER,PublicFormat.
↪SubjectPublicKeyInfo),
            hashes.SHA256()
        )
    except InvalidSignature:
        logger.log("Invalid signature for Peer Public Key")

    # Derive shared key
    shared_key = private_key.exchange(peer_public_key)
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)

    return key
```

[4]:
```python
# Elliptic Curve Key Exchange
def emitter_ec_key_exchange(channel,logger):
    global emitter_private_key
    global receiver_public_key

    # Entities generate both private and public key
    private_key = ec.generate_private_key(ec.SECP384R1())
    public_key = private_key.public_key()
    logger.log("Generated private and public key")

    # Send public_key to other peer
    signature = emitter_private_key.sign(
        public_key.public_bytes(Encoding.DER,PublicFormat.SubjectPublicKeyInfo),
        ec.ECDSA(hashes.SHA256())
    )
    channel.e2r.put((public_key,signature))

    # Receive other peer's public_key
    peer_public_key, signature = channel.r2e.get()
    try:
        receiver_public_key.verify(
            signature,
```

```
            peer_public_key.public_bytes(Encoding.DER,PublicFormat.
↪SubjectPublicKeyInfo),
            ec.ECDSA(hashes.SHA256())
        )
    except InvalidSignature:
        logger.log("Invalid signature for Peer Public Key")

    # Derive shared key
    shared_key = private_key.exchange(ec.ECDH(),peer_public_key)
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)

    return key
```

Para demonstração da comunicação entre os dois agentes, o *Emitter* faz a derivação de segredo com o *Receiver* e de seguida ainda lhe envia uma mensagem a dizer "Hello World!" cifrada devidamente com a chave partilhada.

[5]:
```
# Emitter thread function
def emitter(channel):
    logger = Logger("Emitter")
    logger.log("Logger initialized")

    if key_exchange_method == DH_AND_DSA:
        key = emitter_key_exchange(channel,logger)
    if key_exchange_method == ECDH_AND_ECDSA:
        key = emitter_ec_key_exchange(channel,logger)
    logger.log(key)

    msg = b"Hello World!"
    nounce_ct = encrypt(msg,key)
    logger.log("Encrypted: {}".format(nounce_ct))
    channel.e2r.put(nounce_ct)
```

[6]:
```
# Diffie-Hellman Key Exchange
def receiver_key_exchange(channel,logger):
    global receiver_private_key
    global emitter_public_key

    # Get Emitter DH parameters
    parameters, signature = channel.e2r.get()
    try:
        emitter_public_key.verify(
```

```
            signature,
            parameters.parameter_bytes(Encoding.DER,ParameterFormat.PKCS3),
            hashes.SHA256()
        )
    except InvalidSignature:
        logger.log("Invalid signature for DH parameters")
    logger.received("DH parameters")

    # Entities generate both private and public key
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    logger.log("Generated private and public key")

    # Send public_key to other peer
    signature = receiver_private_key.sign(
        public_key.public_bytes(Encoding.DER,PublicFormat.SubjectPublicKeyInfo),
        hashes.SHA256()
    )
    channel.r2e.put((public_key,signature))

    # Receive other peer's public_key
    peer_public_key, signature = channel.e2r.get()
    try:
        emitter_public_key.verify(
            signature,
            peer_public_key.public_bytes(Encoding.DER,PublicFormat.
↪SubjectPublicKeyInfo),
            hashes.SHA256()
        )
    except InvalidSignature:
        logger.log("Invalid signature for Peer Public Key")

    # Derive shared key
    shared_key = private_key.exchange(peer_public_key)
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)

    return key
```

[7]:
```
# Elliptic Curve Key Exchange
def receiver_ec_key_exchange(channel,logger):
    global receiver_private_key
    global emitter_public_key
```

```python
    # Entities generate both private and public key
    private_key = ec.generate_private_key(ec.SECP384R1())
    public_key = private_key.public_key()
    logger.log("Generated private and public key")

    # Send public_key to other peer
    signature = receiver_private_key.sign(
        public_key.public_bytes(Encoding.DER,PublicFormat.SubjectPublicKeyInfo),
        ec.ECDSA(hashes.SHA256())
    )
    channel.r2e.put((public_key,signature))

    # Receive other peer's public_key
    peer_public_key, signature = channel.e2r.get()
    try:
        emitter_public_key.verify(
            signature,
            peer_public_key.public_bytes(Encoding.DER,PublicFormat.
↪SubjectPublicKeyInfo),
            ec.ECDSA(hashes.SHA256())
        )
    except InvalidSignature:
        logger.log("Invalid signature for Peer Public Key")

    # Derive shared key
    shared_key = private_key.exchange(ec.ECDH(),peer_public_key)
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)

    return key
```

[8]:
```python
# Receiver thread function
def receiver(channel):
    logger = Logger("Receiver")
    logger.log("Logger initialized")

    if key_exchange_method == DH_AND_DSA:
        key = receiver_key_exchange(channel,logger)
    elif key_exchange_method == ECDH_AND_ECDSA:
        key = receiver_ec_key_exchange(channel,logger)

    logger.log(key)
```

```
    nounce_ct = channel.e2r.get()
    try:
        message = decrypt(nounce_ct,key)
        logger.log("Decrypted: {}".format(message))
    except InvalidSignature:
        logger.log("Invalid MAC")
```

Para testar todo este código temos a instanciação das threads e a geração das chaves dos algoritmos que se pretende usar

[ ]:
```
if key_exchange_method == DH_AND_DSA:
    emitter_private_key = dsa.generate_private_key(key_size=1024)
    receiver_private_key = dsa.generate_private_key(key_size=1024)
elif key_exchange_method == ECDH_AND_ECDSA:
    emitter_private_key = ec.generate_private_key(ec.SECP384R1())
    receiver_private_key = ec.generate_private_key(ec.SECP384R1()) #

emitter_public_key = emitter_private_key.public_key()
receiver_public_key = receiver_private_key.public_key()

channel = Channel()

# Create emitter and receiver threads
e = Thread(target=emitter,args=(channel,))
r = Thread(target=receiver,args=(channel,))

# Start both threads
e.start()
r.start()

# Wait for them to finish to exit program
e.join()
r.join()
print("[INFO] > Finished program execution")
```