TP00_P1

March 14, 2021

Solução Para o Trabalho Prático 0

Problema 01:

1. Conforme solicitado foi implementada uma comunicação entre o Agente Emitter e o Agente Receiver baseada em sockets;
2. Usamos o AES no GCM para cifrar e autenticar o conteúdo das nossas mensagens;
3. Para a derivação de chave usamos uma password que os Agentes tem de inserir, e fornecêmo-la ao algoritmo PBKDF2HMAC;
4. Para autenticar a chave usada para a comunicação usamos um HMAC.

Observação: Considerando que não queremos sobrecarregar este Notebook, iremos omitir os aspectos de comunicação e de input mas se for necessário consultá-los sempre é possível ver o código e a sua documentação.

Funcionamento dos Agentes:

- Imports do que é necessário para correr os Agentes, salientando o uso da package cryptography e de getpass para ler a password inserida no terminal.

[1]:
```
import getpass
import socket
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
```

Começando pelas funções em comum entre o Emitter e o Receiver nós temos as seguintes:

- pbkdf2Hmac;
- mac;

A primeira, como o nome indica, está relacionada com o processo de extender a password para uma chave de 256 bits. Esta função usa um salt gerado aleatóriamente pela os.urandom(16) para inicializar o objecto do PBKDF que é usado posteriormente para derivar a chave que nós queremos usando a password.

[2]:
```
def pbkdf2Hmac(self, salt):
    kdf = PBKDF2HMAC(
```

```
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend())
    return kdf
```

Esta função usa um segredo e uma mensagem convencionalmente para criar um código de autenticação para a nossa mensagem usando a chave como segredo. Neste caso o propósito do MAC é diferente, nós usamos a password inserida pelo Agente como um segredo e a chave como se fosse a mensagem com o propósito de gerar um código para autenticar a chave que foi usada para cifrar a mensagem. Como não foi especificado, a nossa escolha foi usar o HMAC com o SHA256 para obter o código de autenticação.

[3]:
```
def mac(self, secret, msg):
    h = hmac.HMAC(secret, hashes.SHA256(), default_backend())
    h.update(msg)
    return h.finalize()
```

Agente Emitter:

Passando para a função exclusiva do Agente Emitter temos: - encrypt

A encrypt usa a chave que foi gerada e um nonce aleatóriamente gerado para cifrar a mensagem inserida usando o AESGCM. Esta função devolve o nonce e o criptograma para uso posterior. Uma nota de importância, não é preciso fazer mais nada para depois autenticarmos a mensagem no receiver porque o criptograma devolvido AESGCM tem uma tag de 16 bytes anexada no fim.

[4]:
```
def encrypt(self, key, msg):
    nonce = os.urandom(12)
    aes_gcm = AESGCM(key)
    ct = aes_gcm.encrypt(nonce, msg, None)
    return nonce, ct
```

Agente Receiver:

Agora para o caso do Receiver temos: - validate_key - decrypt

A validate_key pega na chave que foi gerada no Receiver, na password inserida no Receiver e no key_mac recebido na mensagem vinda do Emitter. Se o key_mac coincidir com o mac da chave do Receiver então a chave é válida.

[5]:
```
def validate_key(self, key, password, key_mac):
    our_key_mac = self.mac(key, password)
    return our_key_mac == key_mac
```

A decrypt pega na chave, no nonce recebido e no criptograma (que vem com a sua própria tag) e usa o AESGCM para decifrar o criptograma. Se este processo funcionar é porque não houve problemas e temos o nosso texto limpo. Se houveram problemas é porque a tag não consegue validar a mensagem.

[6]:
```python
def decrypt(self, key, nonce, ct_and_tag):
    aesgcm = AESGCM(key)
    try:
        limpo = aesgcm.decrypt(nonce, ct_and_tag, None)
        return limpo
    except:
        return 'Autenticacao Falhou'
```

Execução do Agente Emitter:

Agora vamos ver o corpo de execução do Agente Emitter e dizer passo a passo o que foi feito que não foi coberto acima. Após estabelecer a comunicação, inicia-se o fluxo:

1. Primeiro pedimos ao Agente para inserir uma password. Depois geramos um salt seguro para usos criptográficos para ser usado na derivação da chave.
2. Depois derivamos a chave usando a password depois de inicializar o PBKDF com o salt acima mencionado. Como temos a chave podemos calcular o MAC dela para ser usado posteriormente.
3. Agora pedimos ao Agente para inserir uma mensagem para ser enviada para o Receiver.
4. Quando já temos a mensagem tratamos de a cifrar usando a função encrypt.
5. Finalmente, juntamos o salt para a PBKDF, o MAC da chave, o nonce para o AESGCM e o criptograma e enviamos tudo para o Receiver.

Algo que devemos salientar é que o nonce que usamos tem de ser enviado para o Receiver junto com a mensagem, uma alternativa para isto seria usando um counter em ambos os lados. O nonce é enviado às claras e não há problema, desde que o mesmo nonce nunca seja repetido para a mesma chave, mas tendo o uso de uma password e a geração aleatória do nonce nós não nos preocupamos em verificar se o mesmo nonce é gerado duas vezes, tendo em conta o teor deste exercício.

[ ]:
```python
password = self.requestPassWord()
salt = os.urandom(16)
key = self.pbkdf2Hmac(salt).derive(password)
key_mac = self.mac(key, password)
msg = self.ask_message()

try:
    nonce, ciphertext_and_tag = self.encrypt(key, msg)
    bundle = salt + key_mac + nonce + ciphertext_and_tag
    self.conn.send(bundle)
    print('Mensagem Enviada')
except:
    print("Erro na Comunicacao\n")
    self.finish()
```

Execução do Agente Receiver:

Agora vamos analizar o corpo de execução de cada ligação no Agente Receiver.

1. Primeiro pedimos a password ao Agente.
2. Depois recebemos a mensagem vinda do Emitter e se não houver um problema tratamos de a processar. Os primeiros 16 bytes são para o salt, os 32 a seguir são para o MAC da chave, os 12 depois desses são para o nonce do AESGCM e o resto é para o criptograma.
3. Usando a nossa password e o salt que recebemos nós calculamos a chave para decifrar este criptograma. Mas antes de o decifrar temos de autenticar a chave com a nossa password e o salt recebido, garantindo que não é possível forjar esta autenticação ao mudar o salt do criptograma a menos que se saiba a password.
4. Depois da validação com sucesso nós deciframos o criptograma, que mesmo assim pode não ter uma tag válida e o Receiver é avisado dessa situação.

[ ]:
```
password = self.requestPassWord()
msg = adrr.recv(lim)
if not msg:
    print('Nenhuma Mensagem Recebida!\n')
    break
else:
    salt = msg[0:16]
    key_mac = msg[16:48]
    nonce = msg[48:60]
    ciphertext = msg[60:]

    key = self.pbkdf2Hmac(salt).derive(password)

    try:
        if self.validate_key(key, password, key_mac):
            plain_text = self.decrypt(key, nonce, ciphertext)
            print('Mensagem Recebida:\n', plain_text.decode())
        else:
            print('Falha de Autenticacao')
        except:
            print('Comunicacao Corrompida...')
            self.finish(self.conn)
```