TP00_P1

March 14, 2021

Solução Para o Trabalho Prático 00

Problema 01:

1. Conforme solicitado foi implementado uma comunicação entre o Agente Emitter e o Agente Receiver, e tal comunicação foi 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. E, 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 imput mas se for necessário consultá-los sempre é possível ver o código e a sua documentação.

Funcionamento do Code:

- 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]:
```
""" Imports necessários para a execução. """

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]:
```
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 para o Agente inserir uma password. Depois geramos um salt seguro para usos c

2. Depois derivamos a chave usando a password depois de inicializar o PBKDF com o salt acima men

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.

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 na forma de um counter.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

5. Finalmente, juntamos o salt para a PBKDF, o MAC da chave, o nonce para o AESGCM e o criptogr

[ ]:
```
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 recevemos a mensagem vinda do emitter, e se não houver um problema tratamos de a proce

3. Usando a nossa password e o salt que recebemos nós calculamos a chave para decifrar este crip

4. Depois da validação com sucesso nós deciframos o criptograma, que mesmo assim pode não ter um
```

[ ]:
```
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)
```

Problema 02:

1. Usamos uma XOF e um PRG para fazer um pad que consiste em 2^N palavras de 64 bits.
2. Como pedido usamos a password como seed para o PRG.
3. Para cifrar ou decifrar fazemos um XOR como no caso do One Time Pad.

Observação: Além do que foi mencionado para o problema 1, neste problema não há autenticação.

Funcionamento do Code:

- Imports do que é necessário para correr os Agentes, salientando o uso da package cryptography para o SHAKE.

[ ]:
```
import getpass
import socket
import math
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
```

O Receiver e o Emitter usam as mesmas funções para a comunicação entre eles.

- A primeira é a shake que é responsável por criar uma string longa com 2^N * 8 (bytes) de tamanho para servir de pad.

[11]:
```
def shake(self, size, password):
    digest = hashes.Hash(hashes.SHAKE256(size), default_backend())
    digest.update(password)
    return digest.finalize()
```

Tendo o pad maior esta função separa o pad num array de palavras.

[12]:
```
def split(self, pad):
    n = 8
    len_pad = len(pad)
    x = [pad[i:i + n] for i in range(0, len_pad, n)]
    return x
```

A função xor faz XOR do pad com a mensagem. É de salientar a variável word que conta em que palavra é que vamos para nos certificarmos que não repetimos a mesma palavra duas vezes.

[13]:
```
def xor(self, pad, message):
    size = len(message)
    xored = bytearray(size)
    word = self.counter
    position = 0
    for i in range(size):
        xored[i] = pad[word][position] ^ message[i]
        position += 1
        if position == 8:
            word += 1
            position = 0
    return xored
```

Execução do Agente Emitter:

1. Inicialmente perguntamos qual é o valor desejado para o N, este valor tem de coincidir em ambos os agentes.
2. Depois disso calculamos o número total de palavras a gerar com a password que é pedida logo a seguir. Tendo N e a password usamos a shake para gerar o pad e a split para dividir o pad em palavras.
3. Agora, para cada mensagem inserida verificamos se ela é maior que o pad e a nossa decisão é

que embora fosse possível criar tantos pads quanto necessários para acomodar a mensagem nós decidimos que seria melhor impor um limite máximo.

4. A seguir verificamos se o tamanho da mensagem faz com que seja necessário criar um pad novo, e se for, o resto por utilizar deste pad é descartado e geramos um pad novo.

É de salientar o uso da math.ceil nestes casos que faz com que não usemos as mesmas palavras duas vezes, por exemplo se usarmos 32 bits de uma palavra não vamos voltar a usar essa palavra, mas também não vamos usar o resto dos 32 bits.

5. Depois disto é só fazer o XOR do pad com a mensagem e enviá-la, atualizando o counter tendo em conta o tamanho da mensagem.

[ ]:
```
n = int(input("N?\n"))
lim = pow(2, n)
self.establish_connection()
password = self.requestPassWord()
pad = self.shake(lim * 8, password)
pad_words = self.split(pad)
while True:
    msg = self.ask_message()
    msg_size = len(msg)
    if msg_size < int(lim*8):
        try:
            if (self.counter + math.ceil(msg_size / 8)) > lim:
                password = self.requestPassWord()
                pad = self.shake(lim * 8, password)
                pad_words = self.split(pad)
                self.counter = 0
            ciphertext = self.xor(pad_words, msg)
            self.counter += math.ceil(msg_size / 8)
            if len(ciphertext) > 0:
                self.conn.send(ciphertext)
                print('Mensagem Enviada')
            else:
                self.finish()
        except:
            print("Erro na Comunicacao\n")
    else:
        print("Mensagem demasiado grande\n")
        self.finish()
```

Execução do Agente Receiver:

O Receiver funciona exatamente como o Emitter, mas em vez de se inserir uma mensagem nós recebêmo-la pela socket.

[ ]:
```
Self.establish_connection()
i = 0
while i < 1:
```

```
    n = int(input("N?\n"))
    lim = pow(2, n)
    adrr, emitter = self.conn.accept()
    print("Ligado a: ", emitter)
    i += 1
    password = self.requestPassWord()
    pad = self.shake(lim * 8, password)
    pad_words = self.split(pad)
    while True:
        msg = adrr.recv(lim*8)
        msg_size = len(msg)
        if (self.counter + math.ceil(msg_size/8)) > lim:
            password = self.requestPassWord()
            pad = self.shake(lim * 8, password)
            pad_words = self.split(pad)
            self.counter = 0
        if not msg:
            print('Nenhuma Mensagem Recebida!\n')
            break
        else:
            cleantext = self.xor(pad_words, msg)
            self.counter += math.ceil(msg_size/8)
            print(cleantext.decode())
    self.finish(self.conn)
```

Alínea 2.C :

Não achamos que seja necessário fazer um grande número de medições nem outros tipos de estudos à lá Performance Engineer, mas sim podemos olhar de um ponto de vista mais removido para os dois algoritmos com que estamos a trabalhar. O gargalo do segundo algoritmo é gerar o pad, enquanto que o gargalo do primeiro pode ser o facto de que por mensagem temos de gerar uma chave, o código de autenticação para ela e cifrar o criptograma para o caso do Emitter ou então no caso do Receiver que tem o trabalho adicional de autenticar a chave que foi recebida. Tendo isto podemos imaginar facilmente que num cenário em que temos um grande número de mensagens trocadas e um N grande, a situação em que o segundo algoritmo passa pelo gargalo é rara o suficiente para o segundo algoritmo ser mais rápido que o segundo. A diferença é clara o suficiente para se poder demonstrar que o segundo algoritmo é mais rápido on demand, algo que pode ser feito facilmente na apresentação.

[ ]: