

# Trabalho Prático 1: Construindo um Sistema de Mensagens Seguras em Python

Disciplina: Segurança da Informação - Cursos Redes de Computadores

Prof. Michel Sales

**Objetivo:** Este exercício desafia você a construir uma aplicação de troca de mensagens segura em **Python** usando **sockets**, incorporando princípios criptográficos fundamentais para garantir confidencialidade, integridade e autenticidade.

**Cenário:** Você desenvolverá dois programas: um `servidor.py` e um `cliente.py`. O cliente enviará uma mensagem ao servidor, e o servidor a receberá e validará. Toda a comunicação das mensagens reais deverá ser protegida.

## Requisitos de Segurança

---

1. **Confidencialidade:** A mensagem deve ser ilegível para um bisbilhoteiro.
  - **Mecanismo:** Criptografia simétrica usando **AES** (Advanced Encryption Standard) no modo CBC (Cipher Block Chaining).
2. **Integridade:** A mensagem não deve ser modificada em trânsito sem ser detectada.
  - **Mecanismo:** Uso de um código de autenticação de mensagem baseado em hash (**HMAC**).
3. **Autenticidade:** A origem da mensagem deve ser verificável (garantindo que a mensagem veio do cliente legítimo).
  - **Mecanismo:** O **HMAC** também servirá para autenticação da origem, pois apenas as partes que compartilham a chave do HMAC podem gerar e verificar o HMAC corretamente.

## Requisitos de Geração de Chaves

---

1. **Chave de Troca DH:** Para estabelecer uma chave compartilhada entre cliente e servidor, utilize o algoritmo **Diffie-Hellman (DH)**. Esta chave DH será a “semente” para a derivação das chaves finais.

2. **Handshake com Assinatura Digital (ECDSA)**: Para aumentar a segurança do protocolo de troca de chaves DH, esta extensão propõe o uso de **assinaturas digitais com Elliptic Curve Digital Signature Algorithm (ECDSA)** durante o handshake. O objetivo é garantir a **autenticidade das chaves públicas DH** trocadas entre cliente e servidor, prevenindo ataques do tipo *man-in-the-middle*. Para tanto:
  - Cada parte (cliente e servidor) possui um par de chaves ECDSA (privada e pública).
  - As chaves públicas estão publicadas em <https://github.com/USER.keys>.
3. **Derivação de Chaves para AES e HMAC**: Após a troca DH, a chave secreta compartilhada pelo DH não deve ser usada diretamente como chave de criptografia ou HMAC. Em vez disso, **utilize o algoritmo PBKDF2 (Password-Based Key Derivation Function 2)** para derivar duas chaves separadas e seguras a partir da chave DH:
  - Uma chave para a criptografia AES (**Key\_AES**).
  - Uma chave para o HMAC (**Key\_HMAC**).
  - **Salt**: O PBKDF2 requer um *salt*. Você pode gerar um *salt* aleatório e trocá-lo durante o handshake inicial (ou defini-lo como um valor fixo, embora menos seguro para aplicações reais).
  - **Iterações**: Escolha um número de iterações adequado para o PBKDF2 (ex: 100.000 ou mais).

## Estrutura da Mensagem (Após o Handshake DH/PBKDF2)

— A mensagem enviada pelo cliente ao servidor deve ter a seguinte estrutura (ou similar):

[HMAC\_TAG] + [IV\_AES] + [MENSAGEM\_CRIPTOGRAFADA]

- **HMAC\_TAG**: A resultado do cálculo HMAC sobre a IV\_AES e a MENSAGEM\_CRIPTOGRAFADA.
- **IV\_AES**: O Vetor de Inicialização (IV) usado na criptografia AES. Este IV deve ser único para cada mensagem e transmitido em texto claro (não precisa ser secreto, mas deve ser imprevisível).
- **MENSAGEM\_CRIPTOGRAFADA**: A mensagem original criptografada com AES usando Key\_AES.

## Passos para Implementação

### 1. Configuração Inicial:

- **Definir os parâmetros públicos para o Diffie-Hellman** (um número primo grande  $p$  e um gerador  $g$ ). Estes **podem ser hardcoded** no seu programa para simplificação do exercício, mas em uma aplicação real seriam negociados ou viriam de uma fonte confiável.

## 2. Handshake Diffie-Hellman:

- **Cliente:**

- Gera par DH:  $a, A = g^a \mod p$
- Assina  $A + \text{username\_cliente}$  com sua chave privada ECDSA:  $\text{sig\_A} = \text{ECDSA\_sign}(A + \text{username\_cliente})$
- Envia para o servidor:  $A, \text{sig\_A}, \text{username\_cliente}$

- **Servidor:**

- Recebe  $A, \text{sig\_A}, \text{username\_cliente}$
- Baixa a chave pública do cliente de [https://github.com/username\\_cliente.keys](https://github.com/username_cliente.keys)
- Verifica  $\text{sig\_A}$  com a chave pública
- Gera par DH:  $b, B = g^b \mod p$
- Assina  $B + \text{username\_servidor}$ :  $\text{sig\_B} = \text{ECDSA\_sign}(B + \text{username\_servidor})$
- Envia para o cliente:  $B, \text{sig\_B}, \text{username\_servidor}$
- Calcula a chave secreta compartilhada  $S_{\text{servidor}} = A^b \pmod p$ .

- **Cliente:**

- Recebe  $B, \text{sig\_B}, \text{username\_servidor}$
- Baixa a chave pública do servidor de [https://github.com/username\\_servidor.keys](https://github.com/username_servidor.keys)
- Verifica  $\text{sig\_B}$  com a chave pública
- Calcula a chave secreta compartilhada  $S_{\text{cliente}} = B^a \pmod p$ .

- **Verificação:** Ambas as partes devem ter chegado ao mesmo valor  $S$ .

## 3. Derivação de Chaves (PBKDF2):

- Ambas as partes usarão o valor  $S$  (do DH) como "password" para o PBKDF2.
- Derivar  $\text{Key\_AES}$  e  $\text{Key\_HMAC}$  usando um *salt* (o mesmo para ambos os lados) e um número de iterações.
  - Exemplo:  $\text{Key\_AES} = \text{PBKDF2}(S, \text{salt}, \text{iterations}, \text{length\_AES\_key})$
  - Exemplo:  $\text{Key\_HMAC} = \text{PBKDF2}(S, \text{salt}, \text{iterations}, \text{length\_HMAC\_key})$

## 4. Troca de Mensagens Seguras:

- **Cliente (Envio):**

- Gera uma mensagem em texto claro.
- Gera um IV aleatório para AES.
- Criptografa a mensagem com  $\text{Key\_AES}$  e o IV gerado.
- Calcula o HMAC da  $\text{IV\_AES} + \text{MENSAGEM\_CRIPTOGRAFADA}$  usando  $\text{Key\_HMAC}$ .
- Concatena  $\text{HMAC\_TAG}$ ,  $\text{IV\_AES}$  e  $\text{MENSAGEM\_CRIPTOGRAFADA}$  e envia como um único pacote via socket.

- **Servidor (Recebimento):**

- Recebe o pacote da mensagem.
- Separa o  $\text{HMAC\_TAG}$ ,  $\text{IV\_AES}$  e  $\text{MENSAGEM\_CRIPTOGRAFADA}$ .
- Calcula o HMAC esperado da  $\text{IV\_AES} + \text{MENSAGEM\_CRIPTOGRAFADA}$  usando sua  $\text{Key\_HMAC}$ .

- **Verifica Integridade e Autenticidade:** Compara o HMAC\_TAG recebido com o HMAC esperado. Se não forem idênticos (em tempo constante, para evitar ataques de temporização), rejeita a mensagem.
- Se o HMAC for válido, descriptografa a MENSAGEM\_CRIPTOGRAFADA usando Key\_AES e IV\_AES.
- Exibe a mensagem em texto claro.

## Avaliação

---

Seu programa será avaliado com base na:

- Correta implementação dos protocolos DH, PBKDF2, AES e HMAC.
- Garantia da confidencialidade, integridade e autenticidade da mensagem.
- Capacidade do cliente de enviar e do servidor de receber e validar a mensagem.
- Organização e clareza do código.

**Boa sorte!**