PMR3412 - Redes Industriais

Prof. Dr. Agesinaldo Matos Silva Jr.
Prof. Dr. André Kubagawa Sato
Prof. Dr. Edson Kenji Ueda
Prof. Dr. Marcos de Sales Guerra Tsuzuki

Entrega: Criptografia na Internet

Escola Politécnica da Universidade de São Paulo

2º semestre de 2020

Sumário

1 Exercício 1 - Senhas e Login

A autenticação em sites na Internet geralmente utiliza o esquema de usuário/senha. Essas informações serão enviadas pelo cliente, através de uma requisição do *browser*, e devem ser checadas pelo servidor, que geralmente armazena as informações em um banco de dados. No entanto, caso o banco de dados seja comprometido, o que ocorre com frequência, não desejamos que as senhas sejam completamente vazadas. Sendo assim, uma forma de contornar este problema é armazenar apenas o *hash* da senha no banco de dados. Esta primeira atividade consiste no desenvolvimento de um servidor Flask, escrito em Python, que gerencia senhas e realiza o login/logout do usuário no sistema.

São pedidas quatro versões do programa: 1) API RESTful de usuários; 2) Hashing de senhas e 3) Login com Cookie e 4) Login com Cookie criptografado. Em todos os casos o programa completo deve estar contido em um único arquivo `py`. Desenvolva cada um em arquivos Python separados; sendo assim, devem ser elaborados quatro programas, cada um com um arquivo.

1.1 API RESTful de usuários

Nesta versão inicial, é pedido para gerar uma API REST para gerenciar os usuários do sistema. O servidor deve ser desenvolvido utilizando o framework Flask, cuja documentação pode ser encontrada em <https://flask.palletsprojects.com/en/1.1.x/>. Inicialmente é feita uma introdução ao Flask e o Postman; caso já esteja familiarizado com eles, pode pular para a seção 1.1.6.

1.1.1 Introdução ao Flask

O Flask é um *microframework* voltado para o desenvolvimento de aplicações Web. Nas nossas aplicações, utilizaremos o Flask para simular o servidor de gerenciamento de usuário e senhas. Para instalá-lo, utilize o seu método favorito de instalação de bibliotecas em Python, ou confira em <https://flask.palletsprojects.com/en/1.1.x/installation/>.

Para testar a instalação do Flask, vamos gerar uma aplicação inicial. Crie um arquivo `server.py` com o conteúdo:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello, World!'
```

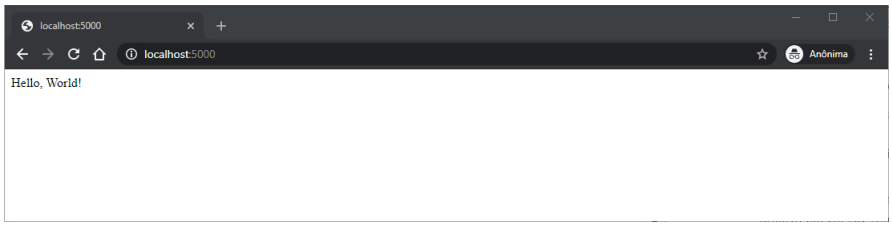
Para executá-lo, abra uma janela de terminal no mesmo diretório do arquivo `server.py` e digite:

```
$ env:FLASK_APP = "server.py"
$ python -m flask run
```

No Linux ou macOS, substitua o primeiro comando por `export FLASK_APP=hello.py`. No prompt do Windows, utilize `set FLASK_APP=hello.py`

Nas listagens deste documento, é inserido o símbolo `$` no início da linha para indicar instruções que devem ser executadas em um terminal (prompt de comando, Powershell, bash, terminal, etc...).

Agora, abra uma aba do seu *browser* e digite o endereço <http://localhost:5000/>. A seguinte tela deve aparecer:



1.1.2 Criando rotas no Flask

A aplicação do exemplo anterior criava apenas uma rota para o caminho `'/'`, com o método GET. Para criar uma nova rota, devemos utilizar o decorador `route` com outro caminho:

```
1 @app.route('/page')
2 def page():
3     return 'This is a Page!'
```

que pode ser acessada no endereço <http://localhost:5000/page>.

Também podemos adicionar seções variáveis a uma URL com a sintaxe `<converter: variable_name>` (onde `converter` pode ser `string`, `int`, `float`, `path` ou `uuid`). Veja o exemplo a seguir:

```
1 @app.route('/hello/<username>')
2 def say_hello(username):
3     return f'Hello, {username}!'
4
5 @app.route('/post/<int:post_id>')
6 def show_post(post_id):
7     return f'Post, {post_id + 1}!'
```

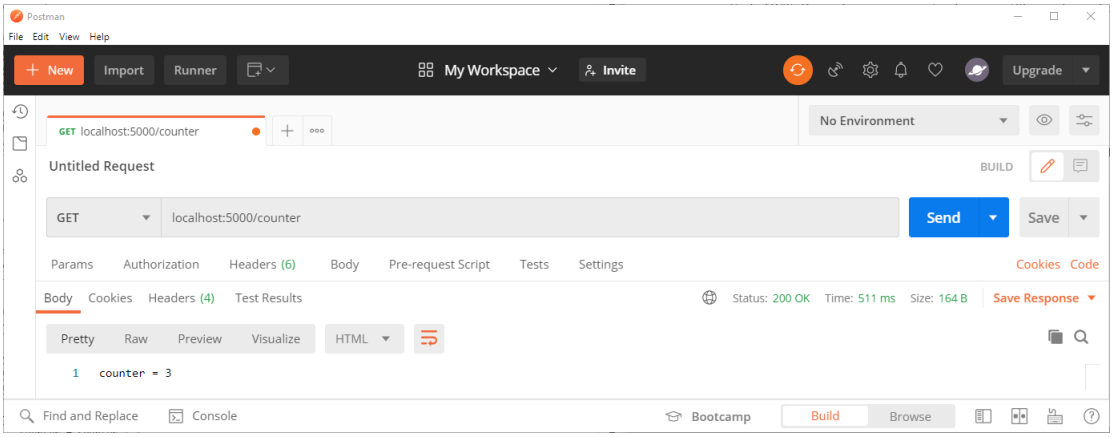
Teste acessando <http://localhost:5000/hello/me> e <http://localhost:5000/post/1>.

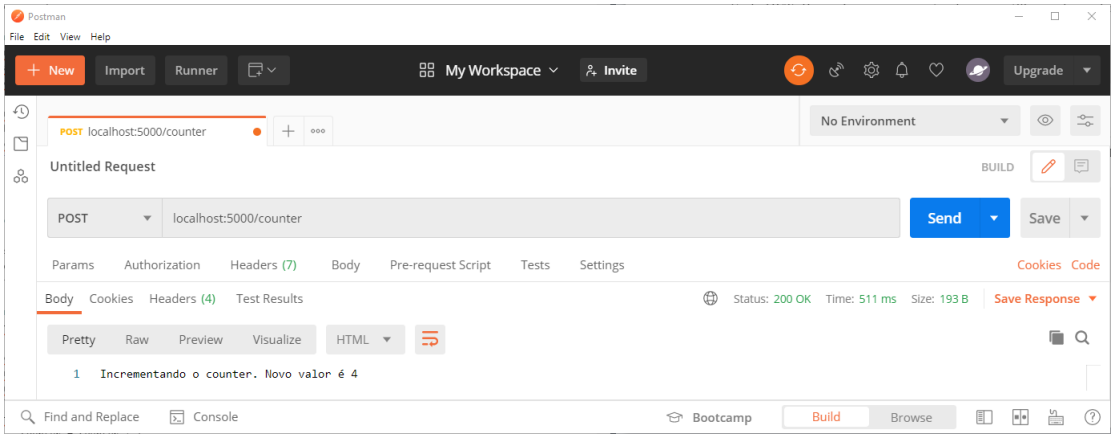
1.1.3 Métodos HTTP e Postman

É possível configurar quais métodos HTTP são aceitos pela rota e processar a requisição de acordo com o método. Para isso, veja o exemplo:

```
1  from flask import request
2
3  counter = 0
4  @app.route('/counter', methods=['GET', 'POST'])
5  def process_counter():
6      global counter
7      if request.method == 'POST':
8          counter = counter + 1
9          return f'Incrementando o counter. Novo valor e {counter}'
10     else:
11         return f'counter = {counter}'
```

Como o *browser* só realiza requisições GET quando utilizado diretamente, vamos adotar um outro software para testar as nossas requisições/respostas: o Postman. Faça o download do Postman em <https://www.postman.com/downloads/> e instale no seu sistema operacional. Com o servidor Flask em execução, vamos testar fazer requisições GET e POST em <http://localhost:5000/counter>. As imagens a seguir mostram as respostas obtidas:





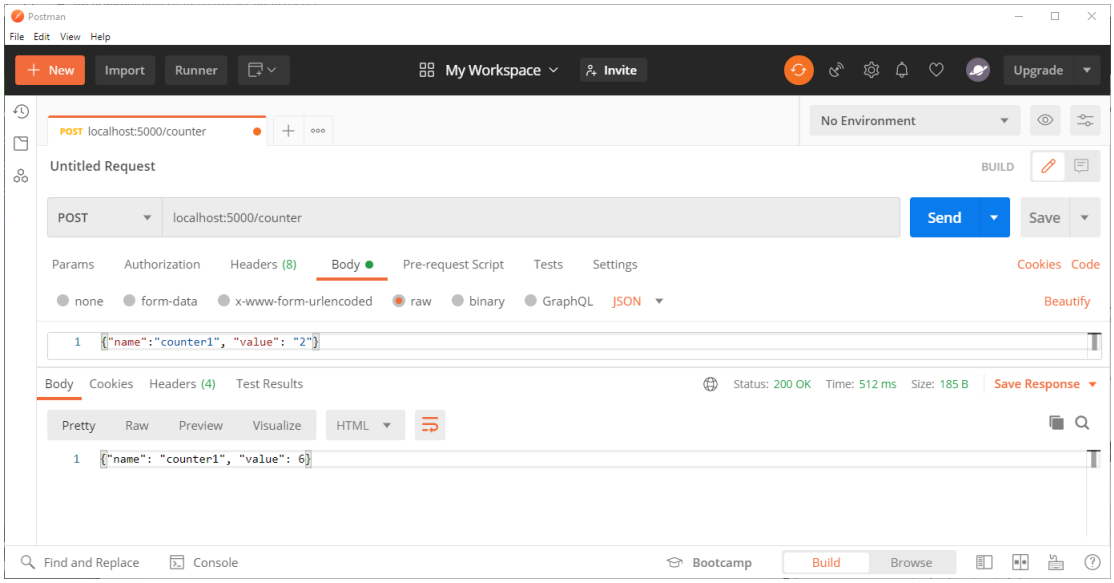
1.1.4 Acessando dados JSON em requisições POST

No Postman, é possível configurar o corpo (*Body*) da requisição POST para enviar dados. Selecionando a opção Raw/JSON, é possível inserir dados em formato JSON, que são geralmente utilizados em APIs Restful. No Flask, é possível receber e enviar JSON em requisições/respostas, como mostra o código abaixo:

```
1  counters = [
2      {"name": "counter1", "value": 0},
3      {"name": "counter2", "value": 0},
4      {"name": "counter3", "value": 0},
5  ]
6
7  @app.route('/counter', methods=['POST'])
8  def process_counter():
9      global counters
10     if request.method == 'POST':
11         name = request.json["name"]
12         value = request.json["value"]
13         counter = next((item for item in counters if item["name"] == name), None)
14         counter["value"] += int(value)
15         return counter
```

Como pode ser observado, os dados recebidos são armazenados no dicionário `request.json`. Já para enviar uma resposta em JSON, basta retornar o dicionário.

Ao fazer uma requisição POST em <http://localhost:5000/counter> com o Postman, temos o seguinte resultado:



1.1.5 Códigos de resposta

Para definir o código de resposta no Flask, basta retornar uma tupla de modo que o segundo valor é o código de resposta. Por exemplo, no código anterior, podemos retornar 404 caso o contador não é encontrado. Para tal, podemos escrever:

```
13    if request.method == 'POST':
14        name, value = itemgetter('name', 'value')(request.json)
15        counter = next((item for item in counters if item["name"] == name), None)
16        if not counter:
17            return 'Error: counter not found', 404
18        ...
```

Quando omitido, é enviado o código de status 200 (OK).

1.1.6 Enunciado da primeira parte do primeiro exercício

A primeira parte consiste em criar uma API REST para realizar as quatro operações básicas de banco de dados: CRUD (criar, recuperar, atualizar e deletar). Seguem os requisitos funcionais do programa:

- Para evitar utilizar banco de dados, os dados de usuário devem ser armazenados em uma variável global. Esta deve ser uma lista de dicionários.
- A lista de usuários deve ser inicializada com pelo menos uma entrada, que será nosso usuário padrão. A senha deste usuário deve ser o NUSP do aluno.
- Cada entrada na lista de usuários deve ser um dicionário com pelo menos os seguintes campos: id, nome de usuário, e-mail e senha.
- O id deve ser único para cada usuário. O primeiro usuário criado deve possuir `id=1` e cada vez que for criado um usuário o id deve ser incrementado. Não se deve

decrementar o contador quando usuários são deletados. Assim, por exemplo, ao criar 4 usuários, em seguida deletar um e depois criar um novo, o seu id deverá ser 5.

- As rotas devem responder a requisições de coleções e de itens individuais de acordo com a tabela:

Método HTTP	CRUD	Rota coleção (/users)	Rota específica (/users/{id})
POST	Criar	201 (Created) + usuário	405 (Method Not Allowed)
GET	Recuperar	200 (OK) + lista de usuários	200 (OK) + usuário ou 404 (Not Found)
PUT	Atualizar	405 (Method Not Allowed)	200 (OK) + usuário ou 404 (Not Found)
DELETE	Deletar	405 (Method Not Allowed)	200 (OK) ou 404 (Not Found)

- As requisições do tipo POST e PUT devem incluir no corpo os dados de criação/atualização de usuários no formato JSON.
- As repostas de usuário e lista de usuários também devem estar no formato JSON.
- Por fim, criar uma nova rota "POST /login", que deve receber um par de valores usuário/senha em JSON. O servidor deve checar se a senha bate e responder com `{"login": "true"}` ou `{"login": "false"}` em caso de sucesso e falha, respectivamente.
- Não é necessário criar nenhuma interface gráfica (HTML, CSS, JS). As requisições podem ser realizadas apenas utilizando o Postman.

No relatório apresente:

1. Listagem do código em Python.
2. Printscreen da tela do Postman mostrando a requisição em cada rota suportada especificada na tabela. Ou seja, pelo menos 5 printscreens ("POST /users", "GET /users", "GET /users/{id}", "PUT /users/{id}", "DELETE /users/{id}").
3. Dois printscreens da tela do Postman mostrando a checagem da senha do usuário padrão na rota rota "POST /login", no caso de sucesso e falha.

1.2 Hashing de senhas

Na primeira parte, estamos armazenando as senhas diretamente no nosso "banco de dados". Dessa forma, caso alguma pessoa tenha acesso a esses dados, ela pode facilmente se autenticar no nosso sistema. A principal estratégia para contornar esta fragilidade é fazer o hashing da senha e armazenar apenas o seu *digest*. Para implementar esta função, vamos utilizar o algoritmo Scrypt da biblioteca `cryptography` do Python. Inicialmente é feita uma pequena demonstração da biblioteca e depois é apresentado o enunciado na seção 1.2.3

1.2.1 Derivando o digest da senha

Para gerar o digest (aka *hash* ou fingerprint), vamos utilizar o método `derive` da classe `Scrypt`. A sua utilização é bastante direta, o único cuidado que deve ser tomado é adotar os parâmetros corretos do algoritmo. O livro texto[1] possui um exemplo com os parâmetros recomendados na listagem 2-7:

```
1 import os
2 from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
3 from cryptography.hazmat.backends import default_backend
4
5 salt = os.urandom(16)
6 kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1, backend=default_backend())
7
8 digest = kdf.derive (b"my great password")
```

1.2.2 Verificação da senha

Para verificar se a senha está correta existe o método `verify` da classe `Scrypt`. O código a seguir é apresentado na listagem 2-8 do livro texto:

```
1 kdf = Scrypt(salt =salt, length =32, n=2**14, r=8, p=1, backend=default_backend())
2 kdf.verify(b"my great password", key)
3 print("Success! (Exception if mismatch)")
```

1.2.3 **Enunciado da segunda parte do primeiro exercício**

A segunda parte consiste em implementar o hashing para armazenar e verificar a senha no "banco de dados" da nossa aplicação. Lembre-se que não estamos utilizando um banco de dados real, apenas estamos armazenando em um lista na memória. Seguem os requisitos funcionais do programa:

- Trocar o campo senha do dicionário de usuário para um nome contendo o sufixo digest. Por exemplo: senha_digest.
- Quando for criado ou atualizado um usuário, ao invés de armazenar a senha diretamente, concatene o sal e o digest (codificados em Base64) e armazene no campo apropriado. Para tal, utilize a função base64.b64encode(valor).
- Adapte a entrada *hardcoded* do usuário padrão para o novo formato de armazenamento de senha. A senha deve continuar sendo o NUSP.
- Altere a rota "POST /login" para funcionar com o novo esquema de armazenamento de senhas.

[1] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

No relatório apresente:

1. Listagem do código em Python.
2. Printscreen que mostre como ficou o campo senha_digest. Para tal, provisoriamente imprima o campo senha_digest na tela (com o comando print) toda vez que um usuário é criado ou alterado. Crie um usuário com a senha "123456".
3. Prinstscreen mostrando o login bem sucedido com a senha "123456".
4. Imagine que os dados da sua aplicação foram vazados. Explique por que utilizar uma senha tão simples não é seguro mesmo com a implementação do hashing de senha.

1.3 Login com Cookie

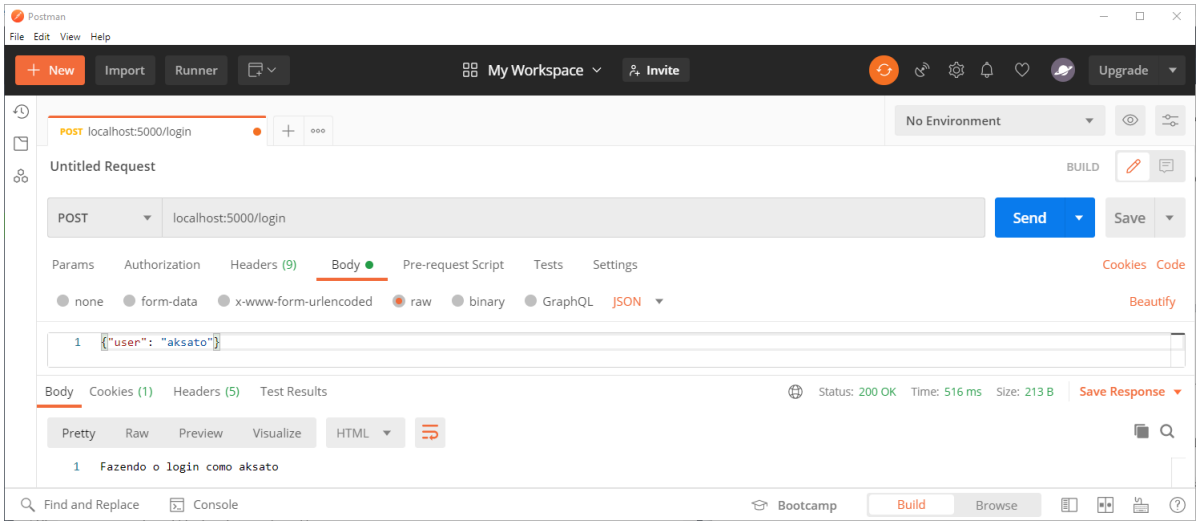
Na nossa aplicação, o login apenas verifica se a senha está correta. Se acessarmos alguma outra rota, não é possível determinar se já fizemos o login ou não. Isso ocorre porque o protocolo HTTP é *stateless*, ou seja, não guarda informações entre requisições diferentes. No entanto, podemos adicionar informações específicas em todas as requisições vindas de um mesmo browser utilizando cookies. Inicialmente é feita uma breve explicação de como trabalhar com cookies no Flask e no Postman; caso já esteja familiarizado, pode pular para a seção 1.3.2.

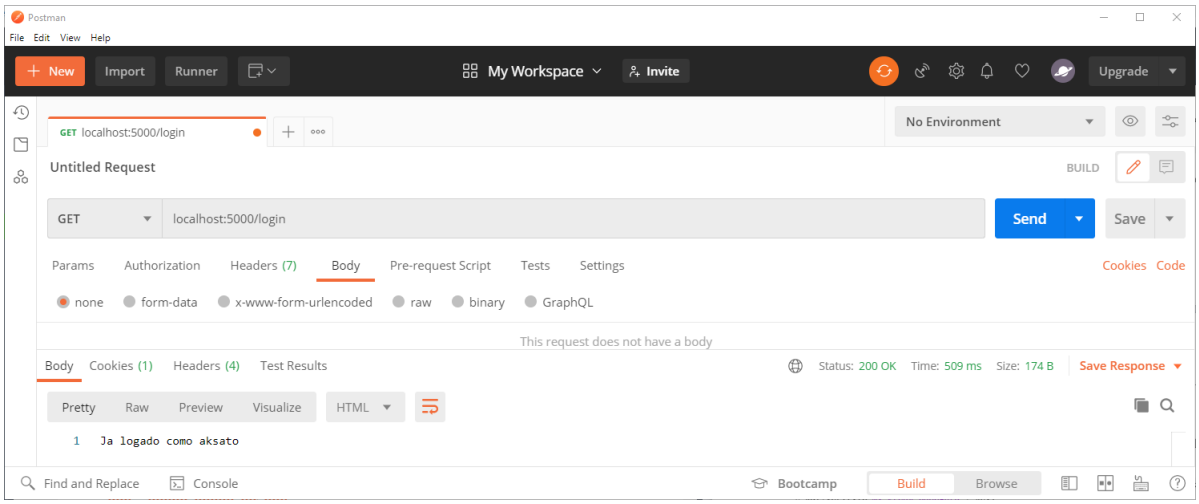
1.3.1 Armazenando e lendo cookies com o Flask

No Flask, podemos armazenar um cookie ao configurar a resposta de uma requisição. Para tal, vamos basear no exemplo da documentação do Flask em <https://flask.palletsprojects.com/en/1.1.x/quickstart/#cookies>. O código abaixo cria duas rotas, uma delas armazena um cookie com o nome do usuário e a outra faz a leitura do cookie para saber se o usuário já está definido:

```
1  from flask import make_response
2
3  @app.route('/login', methods=['GET', 'POST'])
4  def fake_login():
5      if request.method == 'POST':
6          resp = make_response(f'Fazendo o login como {request.json["user"]}')
7          resp.set_cookie('user', request.json["user"])
8          return resp
9      else:
10         user = request.cookies.get('user')
11         if user:
12             return f'Ja logado como {user}'
13         else:
14             return 'Nao logado ainda'
```

O Postman armazena automaticamente os cookies; sendo assim, ao acessar o "POST /login" com os dados formatados corretamente e, em seguida, acessar a rota "GET /login", a resposta deve indicar que o usuário já está logado:





De qualquer forma, é possível manualmente editar os cookies no Postman ao clicar no link "Cookies", localizado logo abaixo do botão "Send". Para apagar o cookie, é necessário utilizar um pequeno "truque": sobrescrever o cookie antigo e definir a expiração imediata como no código:

```
1 @app.route('/delete-cookie')
2 def delete_cookie():
3     res = make_response("Cookie Removed")
4     res.set_cookie('user', 'qualquer_coisa', max_age=0)
5     return res
```

1.3.2 Enunciado da terceira parte do primeiro exercício

A terceira parte consiste em implementar cookies para manter sessões de usuários logados. Seguem os requisitos funcionais do programa:

- A rota "POST /login" agora deve, após verificar que o usuário/senha estão corretos, enviar um cookie com o id do usuário logado.
- Criar uma rota "GET /welcome" que deve responder com uma mensagem personalizada contendo o nome do usuário caso esteja logado. Caso contrário, exibir mensagem informando que é necessário logar antes de acessar o conteúdo desta página.
- Criar uma rota "DELETE /logout" para realizar o logout e apagar o cookie.
- Não utilizar o objeto sessions do Flask, limite-se ao método `set_cookie` do objeto de resposta e `cookies.get` do objeto de requisição.

No relatório apresente:

1. Listagem do código em Python.
2. Printscreens do Postman mostrando a sequência de requisições/respostas em "POST /login" → "GET /welcome" → "DELETE /logout" → "GET /welcome". No total serão 4 printscreens no mínimo.
3. Descreva e insira printscreens do Postman mostrando como você acessaria a rota "GET /welcome" como um usuário logado (sem realizar o login antes). Ou seja, demonstre que é possível enxergar o conteúdo de "GET /welcome" de um usuário sem precisar da senha dele.

1.4 Login com Cookie criptografado

Para resolver a falha de segurança introduzida pelo cookie/sessão, podemos encriptar o valor do cookie. Como tanto a encriptação como a decodificação vão ocorrer no servidor, podemos utilizar um esquema de criptografia simétrica sem necessidade de autenticação. Para implementar esta função, vamos utilizar o algoritmo AES-CTR da biblioteca `cryptography` do Python. Inicialmente é feita uma pequena demonstração da biblioteca e depois é apresentado o enunciado na seção 1.4.2

1.4.1 Encriptação e decodificação com o AES-CTR

O uso do AES-CTR é bastante simples, só é necessário fornecer dois parâmetros: a chave e o nonce, também conhecido como IV ou initialization vector. Ambos tem tamanho fixo e podem ser gerados aleatoriamente, sem nenhuma restrição, utilizando algoritmos randômicos criptograficamente seguros (para o nosso contexto pelo menos). Para verificar sua utilização, vamos verificar a listagem 3-9 do livro texto[2]:

[2] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os

class EncryptionManager:
    def __init__(self):
        key = os.urandom(32)
        nonce = os.urandom(16)
        aes_context = Cipher(algorithms.AES(key), modes.CTR(nonce), backend=default_backend())
        self.encryptor = aes_context.encryptor()
        self.decryptor = aes_context.decryptor()

    def updateEncryptor(self, plaintext):
        return self.encryptor.update(plaintext)

    def finalizeEncryptor(self):
        return self.encryptor.finalize()

    def updateDecryptor(self, ciphertext):
        return self.decryptor.update(ciphertext)

    def finalizeDecryptor(self):
        return self.decryptor.finalize()

# Auto generate key/IV for encryption
manager = EncryptionManager()

plaintexts = [
    b"SHORT",
    b"MEDIUM MEDIUM MEDIUM",
    b"LONG LONG LONG LONG LONG LONG"
]

ciphertexts = []

for m in plaintexts:
    ciphertexts.append(manager.updateEncryptor(m))
ciphertexts.append(manager.finalizeEncryptor())

for c in ciphertexts:
    print("Recovered", manager.updateDecryptor(c))
print("Recovered", manager.finalizeDecryptor())
```

1.4.2 Enunciado da quarta parte do primeiro exercício

A terceira parte consiste em encriptar as informações veiculadas através de cookies para corrigir as falhas de segurança verificadas anteriormente. Seguem os requisitos funcionais do programa:

- Ao iniciar o servidor, deve ser configurado uma chave e um nonce para o algoritmo AES-CTR. Estes podem ser gerados em um programa a parte e seus valores podem ser inseridos diretamente (hardcoded) no programa.

- Do mesmo modo, inicialize também uma variável `counter` com uma string aleatória. Esta pode ser gerada com o comando: `python -c "import os; import base64; print( base64.b64encode( os.urandom(16) ) )"`.
- Agora a rota "POST /login" deve, após verificar que o usuário/senha estão corretos, enviar um cookie encriptado. O valor a ser encriptado é a variável `counter` concatenada com o id do usuário logado (em formato string). Por exemplo, ser o counter é `"/K54fcvZpgtA0gI/+QFtfQ=="` e o id é 3, o texto simples a ser encriptado é `"/K54fcvZpgtA0gI/+QFtfQ==3"`. Essa encriptação deve ser feita utilizando o algoritmo AES-CTR com a chave/nonce pré-configurados.
- Na rota "GET /welcome", é necessário descriptografar a informação do cookie utilizando o AES-CTR com a chave/nonce pré-configurados.
- Não utilizar o objeto `sessions` do Flask, limite-se ao método `set_cookie` do objeto de resposta e `cookies.get` do objeto de requisição.

No relatório apresente:

1. Listagem do código em Python.
2. Prinscreen mostrando o texto cifrado armazenado no cookie na resposta da rota "POST /login". Para tal, provisoriamente imprima o seu valor com a instrução `print` do Python.
3. Prinscreen mostrando o texto cifrado lido do cookie na requisição da rota "GET /welcome". Para tal, provisoriamente imprima o seu valor com a instrução `print` do Python.
4. Printscreens do Postman mostrando a sequência de requisições/respostas em "POST /login" → "GET /welcome" → "DELETE /logout" → "GET /welcome". No total serão 4 printscreens no mínimo.
5. Simule a encriptação de valores de id sem a concatenação do `counter`. Ou seja, teste a listagem da seção 1.4.1 com a lista de textos simples `plaintexts = [b"1", b"2", b"3", b"4", b"5"]`. Discorra sobre a vulnerabilidade de utilizar o id sem concatenar o `counter`.

2 Exercício 2 - Aplicativo de troca de mensagens

Hoje em dia, até mesmo na comunicação pessoal com aplicativos de mensagens existe a preocupação com a segurança. Sendo assim, é possível implementar criptografia para garantir confidencialidade, integridade e autenticação nestas comunicações. O segundo exercícios consiste basicamente em criar uma aplicação que permite realizar a comunicação entre duas pessoas com criptografia simétrica e assimétrica.

São pedidas três versões do programa: 1) Comunicação apenas com criptografia simétrica; 2) Troca de Chaves com RSA; 3) Autenticação com certificados. Em todos os casos deverão ser criados dois programas: o cliente e o servidor. Desenvolva cada um em arquivos Python separados; sendo assim, devem ser elaborados pelo menos dois arquivos para cada versão.

2.1 Comunicação apenas com criptografia simétrica

No exercício 1 foi utilizada a criptografia simétrica para encriptar o valor do cookie de sessão. Sendo assim, é sugerido revisar o código da sessão 1.4.1 para elaborar esta primeira parte do segundo exercício.

Para garantir a integridade, é empregado o HMAC. O livro texto[1] fornece o seguinte exemplo para a implementação do HMAC na biblioteca crytphraphy do Python:

```
1 from cryptography.hazmat.backends import default_backend
2 from cryptography.hazmat.primitives import hashes, hmac
3
4 key = b"CorrectHorseBatteryStaple"
5 h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
6 h.update(b"hello world")
7 h.finalize().hex()
```

onde `key` é a chave MAC e `"hello world"` é a mensagem a ser autenticada.

Além do algoritmo AES-CTR, também será utilizada a biblioteca `sockets` do Python para realizar a comunicação na rede. Não será considerada a possibilidade de múltiplas conexões no servidor, por isso, a revisão do exemplo básico da documentação, que pode ser acessado em <https://docs.python.org/3/library/socket.html#example> já é suficiente para desenvolver os programas desta parte.

[1] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

2.1.1 Enunciado da primeira parte do segundo exercício

A primeira parte consiste em desenvolver uma aplicação cliente-servidor que realiza troca de mensagens com criptografia simétrica (AES-CTR).

Seguem os requisitos funcionais do programa:

- A chave AES, o IV e a chave MAC devem ser geradas em um programa separado e salvos em um ou mais arquivos. Apenas um conjunto é necessário para a comunicação.
- Cada ponta da comunicação (usuário do aplicativo de troca de mensagens) deve executar o servidor, que deve permanecer em execução (em um loop infinito). Ao inicializar, o servidor deve carregar o conjunto de chaves simétricas (AES + IV + MAC).
- Quando o usuário deseja enviar uma mensagem, este executa a aplicação cliente, fornecendo dois argumentos: o endereço/porta do destinatário e a mensagem em texto simples.
- O programa do cliente, ao iniciar, também deve carregar as chaves a partir de um arquivo padrão. Depois disso, a chave AES-CTR e o IV são utilizados para encriptar a mensagem.
- Ainda no cliente, o HMAC é gerado a partir do texto cifrado e concatenado com o mesmo antes de ser enviado.
- No servidor, uma vez recebida a mensagem, ela deve ser separada em texto cifrado + HMAC.
- A mensagem deve ser então descriptografada e o HMAC deve ser calculado e verificado. Se os HMACs coincidirem, o servidor deve imprimir a mensagem recebida.

No relatório apresente:

1. Listagem do código em Python do servidor e do cliente.
2. Prinscreen mostrando o funcionamento do programa. As imagens devem mostrar pelo menos um envio de mensagem do cliente para o servidor.
3. Printscreen mostrando o texto cifrado + HMAC enviado pelo cliente e recebido pelo servidor. Para tal, provisoriamente imprimir na tela com o comando `print` os dados.
4. Responda a pergunta: no esquema proposto, é realizado primeiramente a encriptação e depois gerado o HMAC. Quais outras opções existem? Qual é a mais recomendada?

2.2 Troca de Chaves com RSA

Na prática, em comunicações em rede não é possível garantir que ambos os usuários da comunicação consigam obter as chaves simétricas de modo seguro. Para tal, de alguma forma eles precisariam se encontrar para gerar a chave, o que é inviável no caso de aplicativos de troca de mensagens pela Internet. Por este motivo, é adotada a criptografia assimétrica para realizar a troca de chaves de sessão com segurança. Isto funciona pois, na criptografia assimétrica, ambas as partes podem iniciar uma comunicação sem nunca se encontrarem fisicamente. Inicialmente é feito um breve comentário da troca de chaves com RSA em Python e depois o enunciado é apresentado na seção 2.2.3

2.2.1 Troca de chaves RSA com a biblioteca `cryptography` do Python

Iremos adotar uma forma bastante simplificada de comunicação com troca de chaves RSA. Basicamente, para cada comunicação, serão geradas chaves simétricas, que serão encriptadas e enviadas juntas com a mensagem, assinatura e o HMAC. Com isso, quando a transmissão completa é recebida pelo servidor, ele será capaz de decodificá-la apenas utilizando a chave pública do cliente.

Considere o envio de uma mensagem do cliente para o servidor. No nosso protocolo, a transmissão é realizada com um *stream* de bytes resultante da concatenação das seguintes informações:

- A chave AES de encriptação, o IV e o HMAC (encriptado com a chave pública do servidor).
- A assinatura do cliente sobre o *hash* da chave AES de encriptação, o IV e o HMAC.
- Mensagem encriptada com a chave AES+IV.
- HMAC sobre toda a transmissão (utilizando a chave MAC).

O livro texto[2] apresenta a implementação deste algoritmo na listagem 6-1:

```
1  import os
2  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
3  from cryptography.hazmat.primitives import hashes, hmac
4  from cryptography.hazmat.backends import default_backend
5  from cryptography.hazmat.primitives.asymmetric import padding, rsa
6
7  # WARNING: This code is NOT secure. DO NOT USE!
8  class TransmissionManager:
9      def __init__(self, send_private_key, recv_public_key):
10         self.send_private_key = send_private_key
11         self.recv_public_key = recv_public_key
```

[2] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

```
12          self.ekey = os.urandom(32)
13          self.mkey = os.urandom(32)
14          self.iv = os.urandom(16)
15          self.encryptor = Cipher( algorithms.AES(self.ekey),
16              modes.CTR(self.iv), backend=default_backend()).encryptor()
17          self.mac = hmac.HMAC( self.mkey, hashes.SHA256(), backend=default_backend())
18
19      def initialize(self):
20          data = self.ekey + self.iv + self.mkey
21          h = hashes.Hash(hashes.SHA256(), backend=default_backend())
22          h.update(data)
23          data_digest = h.finalize()
24          signature = self.send_private_key.sign(data_digest,
25              padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
26              salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
27          ciphertext = self.recv_public_key.encrypt(data,
28              padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
29              algorithm=hashes.SHA256(), label=None))
30          ciphertext += signature
31          self.mac.update(ciphertext)
32          return ciphertext
33
34      def update(self, plaintext):
35          ciphertext = self.encryptor.update(plaintext)
36          self.mac.update(ciphertext)
37          return ciphertext
38
39      def finalize(self):
40          return self.mac.finalize()
```

Gaste um tempo analisando e tentando compreender o código, este contém os elementos principais para a parte dois.

2.2.2 Gerando as chaves RSA com o OpenSSL

O OpenSSL é utilizado pela biblioteca `cryptography` do Python e, portanto, já deve estar instalado caso você tenha desenvolvido as outras etapas até aqui. Caso contrário, é possível obtê-lo com o gerenciador de pacotes do seu sistema operacional (`sudo apt-get install openssl`, `brew install openssl` ou `choco install openssl`). Os binários do Windows também podem ser encontrados em <https://slproweb.com/products/Win32OpenSSL.html>.

Com OpenSSL, para criar uma chave privada RSA, basta executar:

```
$ openssl genpkey -algorithm RSA -out my_key.pem -pkeyopt rsa_keygen_bits:2048
```

Para obter a chave pública RSA a partir da privada, podemos executar:

```
$ openssl rsa -in my_key.pem -out my_key_pub.pem -pubout
```

No Python, para carregar a chave privada e gerar a chave pública, podemos escrever (trecho retirado da listagem 4-4 do livro texto):

```
with open("my_key.pem", "rb") as private_key_file_object:
    private_key = serialization.load_pem_private_key( private_key_file_object.read(),
        backend = default_backend(), password = None)
    public_key = private_key.public_key()
```

Por fim, para carregar apenas a chave pública, podemos usar

```
with open("my_key_pub.pem", "rb") as public_key_file_object:
    public_key = serialization.load_pem_public_key(public_key_file_object.read(),
        backend=default_backend())
```

OBS: outro modo de gerar chaves RSA e salvá-las em disco é utilizar a biblioteca `cryptography` em um novo programa. A listagem 4-4 do livro texto[3] pode ser utilizada sem alterações para este fim.

2.2.3 Enunciado da segunda parte do segundo exercício

A segunda parte consiste em aplicar a troca de chaves RSA para a comunicação no nosso aplicativo de mensagens.

Seguem os requisitos funcionais do programa:

- Devem ser gerados dois pares de chaves RSA (privada + pública): um para o cliente e outro para o servidor.
- O servidor deve carregar o seu par de chaves RSA e a chave pública do cliente quando for executado. Analogamente, o cliente deve carregar o seu par de chaves RSA e a chave pública do servidor quando executado.
- Utilizar o protocolo de transmissão descrito na seção 2.2.1.
- De resto, a operação do usuário deve ser similar ao da parte um.

No relatório apresente:

1. Listagem do código em Python do servidor e do cliente.
2. Prinscreen mostrando o funcionamento do programa. As imagens devem mostrar pelo menos um envio de mensagem do cliente para o servidor.
3. Printscreen mostrando a mensagem completa enviada pelo cliente e recebido pelo servidor. Para tal, provisoriamente imprimir na tela com o comando `print` os dados.

[3] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

2.3 Autenticação com certificados

Nossa aplicação já está quase completamente funcional, mas tem um detalhe: para ela funcionar, ambas as pontas da comunicação precisam ter acesso à chave pública do outro. Até aí, como a chave é publica mesmo, você pode achar razoável que a chave pública possa ser enviada junto com a mensagem. No entanto, se isto for feito desta maneira, quem garante que a chave pública pertence a pessoa com quem creio estar comunicando? É aí que entram os certificados, emitidos por um ente confiável, que garante a identidade do proprietário da chave pública. Nesta parte, deve ser desenvolvido um novo software para gerar e assinar certificados. Por fim, o cliente do aplicativo deve ser adaptado para enviar o certificado e o servidor deve conferir sua autenticidade. Inicialmente é feito um breve comentário sobre geração de certificados, que nada mais é que um conjunto de informações (dentre elas a chave pública do requerente) com a assinatura do emissor. O enunciado é então apresentado na seção 2.3.2

2.3.1 Gerando e assinando certificados digitais

O nosso certificado é um dicionário em Python que contém três chaves:

- Nome do requerente
- Nome do emissor
- Chave pública do requerente

E mais a assinatura feita pelo emissor.

Para gerar o certificado, podemos serializar o dicionário em JSON e convertê-lo para bytes; em seguida concatenamos com a assinatura. Este arquivo pode então ser salvo em um arquivo. O código da listagem 5-6 do livro texto[4] contém uma função que realiza essa operação:

```
1  import json
2
3  ISSUER_NAME = "fake_cert_authority1"
4
5  def create_fake_certificate(pem_public_key, subject, issuer_private_key):
6      certificate_data = {}
7      certificate_data["subject"] = subject
8      certificate_data["issuer"] = ISSUER_NAME
9      certificate_data["public_key"] = pem_public_key.decode('utf-8')
10     raw_bytes = json.dumps(certificate_data).encode('utf-8')
11     signature = issuer_private_key.sign(raw_bytes,
12         padding.PSS(mgf=padding.MGF1(hashes.SHA256()),salt_length=padding.PSS.MAX_LENGTH),
13         hashes.SHA256())
14     return raw_bytes + signature
```

[4] "Practical Cryptography in Python: Learning Correct Cryptography by Example" de Seth James Nielson e Christopher K. Monson.

No servidor, para verificar a assinatura e recuperar os dados do certificado transmitido, podemos escrever (retirado da listagem 5-7 do livro texto):

```
def validate_certificate(certificate_bytes, issuer_public_key):
    raw_cert_bytes, signature = certificate_bytes[:-256], certificate_bytes [-256:]
    issuer_public_key.verify(signature,raw_cert_bytes,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),salt_length=padding.PSS.MAX_LENGTH),
        hashes.SHA256())
    cert_data = json.loads(raw_cert_bytes.decode('utf-8'))
    cert_data["public_key"] = cert_data["public_key"].encode('utf-8')
    return cert_data
```

2.3.2 Enunciado da terceira parte do segundo exercício

A terceira e última parte consiste em criar e validar certificados para estabelecer a identidade do cliente na nossa aplicação.

Seguem os requisitos para esta parte:

- O par de chaves RSA deve ser gerado para o emissor, que também deve receber um nome. Tanto a chave pública do emissor como a sua identidade é conhecida pelo servidor e cliente (devem ser carregados na inicialização da aplicação ou podem estar *hardcoded*)
- Um certificado deve ser gerado para o cliente, que ser nomeado. Esse certificado deve ser assinado pelo emissor.
- Ao inicializar o cliente, é necessário carregar o certificado.
- No cliente, envie o certificado junto com a mensagem a cada transmissão.
- No servidor, o certificado deve ser carregado primeiro e a assinatura deve ser verificada com a chave pública do emissor. Depois disso a chave pública pode ser utilizada.
- Na hora de imprimir a mensagem decodificada, exibir também o nome do cliente (obtido no campo subject do certificado).
- De resto, a operação do usuário deve ser similar ao da parte um.

No relatório apresente:

1. Listagem do código em Python do servidor, do cliente e do programa que gera o certificado.
2. Listagem do certificado gerado para o cliente.
3. Listagem da chave pública do emissor.

4. Prinscreen mostrando o funcionamento do programa. As imagens devem mostrar pelo menos um envio de mensagem do cliente para o servidor (no servidor deve aparecer o nome do cliente que enviou a mensagem).

5. Printscreen mostrando a mensagem completa enviada pelo cliente e recebida pelo servidor. Para tal, provisoriamente imprimir na tela com o comando `print` os dados.