



Segurança do Código-Fonte: Uso incorreto de criptografia

Segurança de Aplicações

Gustavo Zilles

Jhonnyffer Hannyel Ferro da Silva

Lucas Souza Dias

Ricardo dos Santos Alves

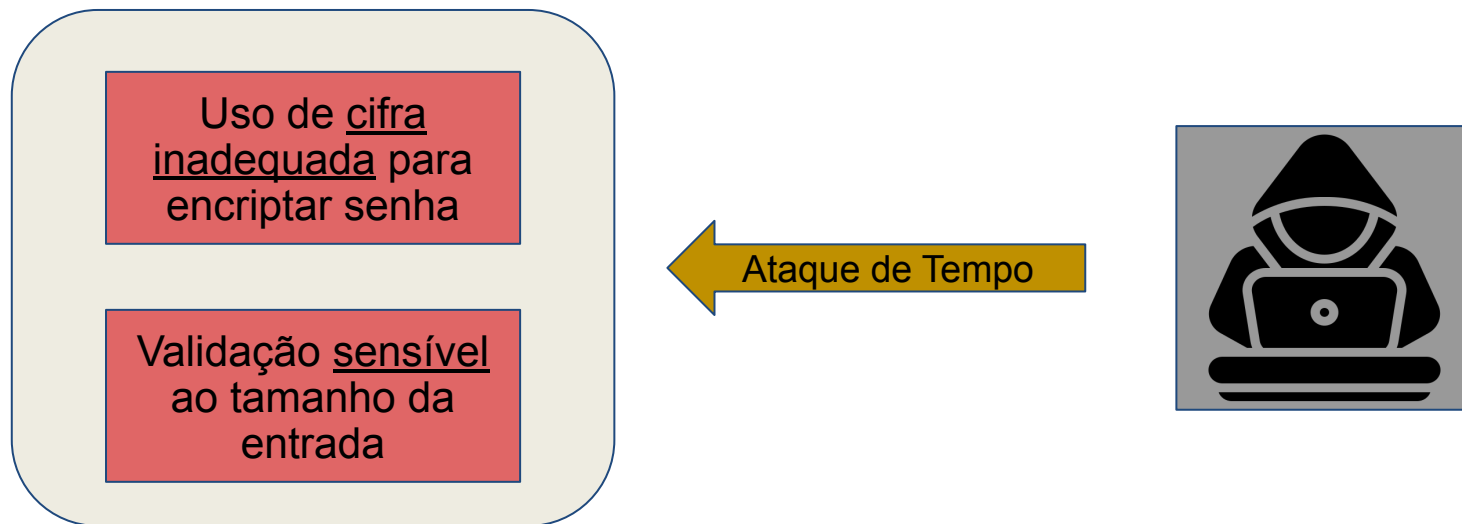
Criptografia

Criptografia é a técnica de transformar dados legíveis em dados ininteligíveis, na tentativa de tornar a informação acessível apenas para agentes autorizados.

- Não uso de criptografia (como usar HTTP ao invés de HTTPS);
- Uso de chaves fracas/aleatoriedade fraca;
- Uso de hash/função fraca: em alguns casos, quanto maior o custo computacional, melhor.

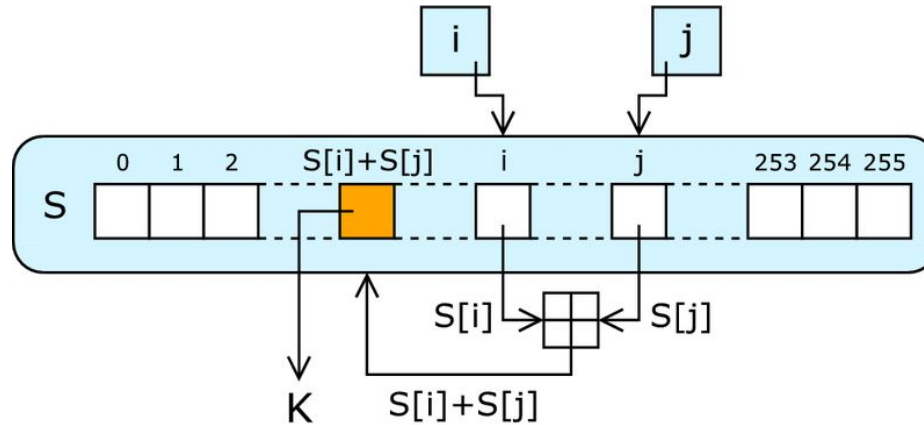
CWE-208: Observable Timing Discrepancy
CWE-328: Use of Weak Hash
CWE-353: Missing Support for Integrity Check

Exemplo 1: Cipher Misuse + Timing Attack



Exemplo 1: Cifra de fluxo

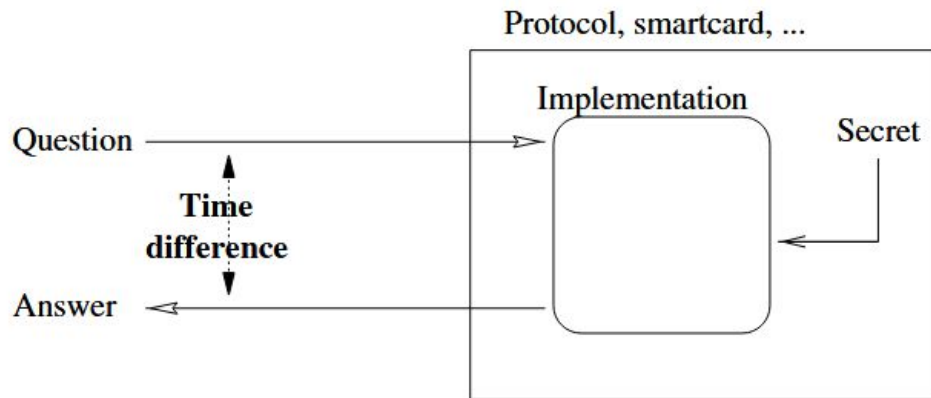
- Cifras de fluxo (*Stream ciphers*) não são adequadas para dados sensíveis.
- São muito suscetíveis a *bit flipping*.
- Tamanho da saída é diretamente proporcional ao tamanho da entrada.



- <https://cryptii.com/pipes/rc4-encryption>

Exemplo 1: Timing Attack

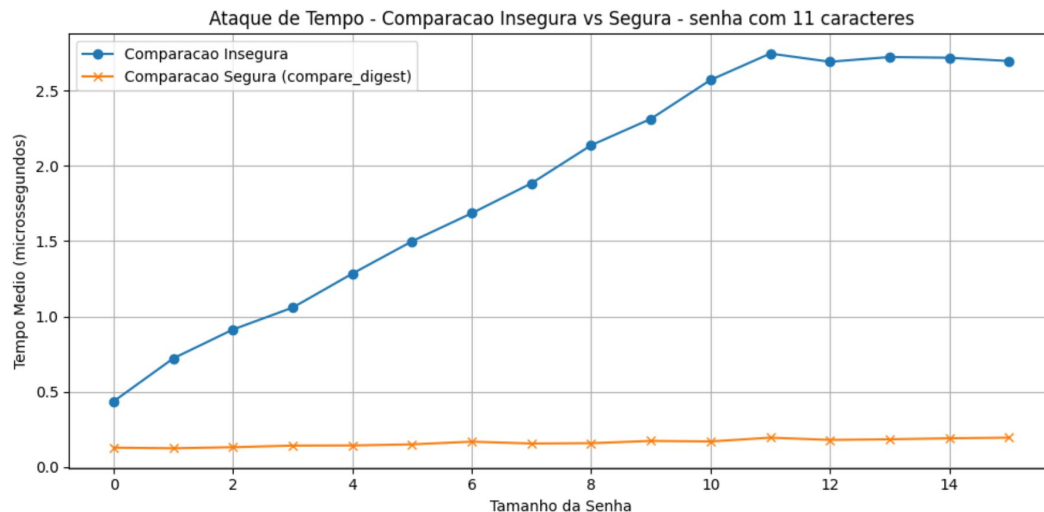
- Fraqueza definida pela **CWE-208: Observable Timing Discrepancy**.
- A variação no tempo de resposta de uma operação pode revelar informações relevantes de segurança.
- A variação no tempo de validação de uma senha pode expor caracteres válidos.



Exemplo 1: Timing Attack




```
# Comparacao insegura
def comparar_inseguro(a, b):
    for x, y in zip(a, b):
        if x != y:
            return False
    return len(a) == len(b)

# Comparacao segura
def comparar_seguro(a, b):
    return hmac.compare_digest(a, b)
```



Exemplo 2: Armazenamento de Senhas

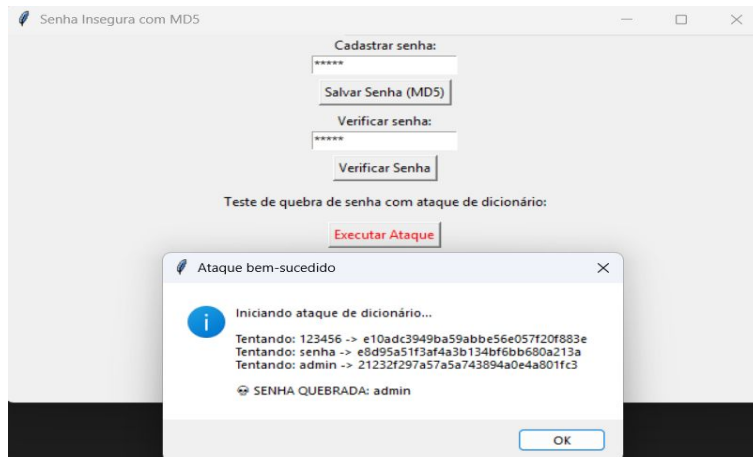
- Senhas são a primeira linha de defesa no controle de acesso, portanto nunca devem ser armazenadas em texto plano, de forma reversível ou usando hash fraco.
- A criptografia, quando bem aplicada, garante que mesmo que dados sejam vazados, as senhas não possam ser recuperadas diretamente.
- Erro comum: Codificação \neq Hash

Conceito	O que é	É seguro para senha?
Codificação	Transformação para outro formato legível (Ex: Base64, Hex)	 Não. É reversível.
Criptografia	Protege dados com chave (Ex: AES, RSA)	 Depende, exige gestão de chaves. Não é o ideal para senhas.
Hash	Função unidirecional, irreversível, própria para verificar dados (Ex: bcrypt, SHA, Argon2)	 Sim, com hash seguro e salt.

Exemplo 2: Código Vulnerável

- Hash MD5: Uma função hash fraca, vulnerável a ataques de colisão e Rainbow tables.

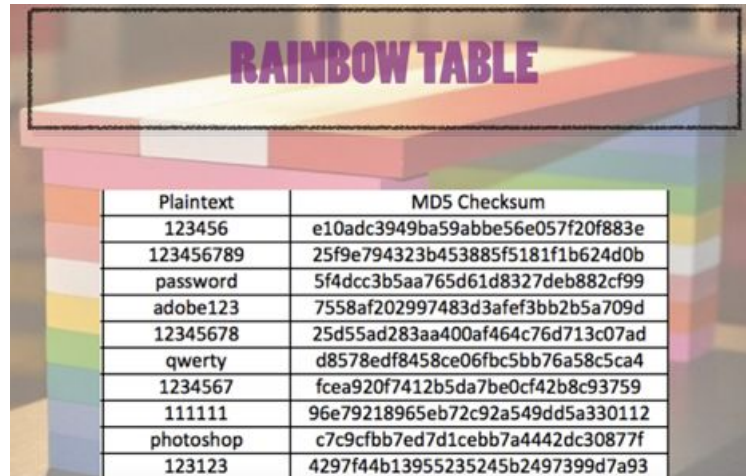
```
12 ]
13
14 # Função para gerar hash MD5
15 def armazenar_md5(senha: str) -> str:
16     return hashlib.md5(senha.encode()).hexdigest()
17
18 # Verificar se o hash corresponde
19 def verificar_md5(senha: str, hash_armazenado: str) -> bool:
20     return armazenar_md5(senha) == hash_armazenado
21
22 # Salvar a senha (hash MD5)
23 def salvar():
24     global hash_inseguro
25     senha = entry_senha.get()
26     if not senha:
27         messagebox.showwarning("Erro", "Digite uma senha.")
28         return
29     hash_inseguro = armazenar_md5(senha)
30     messagebox.showinfo("Salvo", f"Senha armazenada em MD5 (inseguro).\nHash: {hash_inseguro}")
31
32 # Verificar se a senha está correta
```



```
14 # Função para gerar hash MD5
15 def armazenar_md5(senha: str) -> str:
16     return hashlib.md5(senha.encode()).hexdigest()
17
```


Exemplo 2: Por que é vulnerável?

- Hash sem salt: Se duas pessoas usarem a mesma senha, o hash será exatamente igual.
- Hash rápido: O MD5 foi projetado para velocidade, o que é péssimo para senhas (facilita brute-force).
- Obsoleto e quebrado: Existem bilhões de hashes MD5 prontos na internet (rainbow tables).

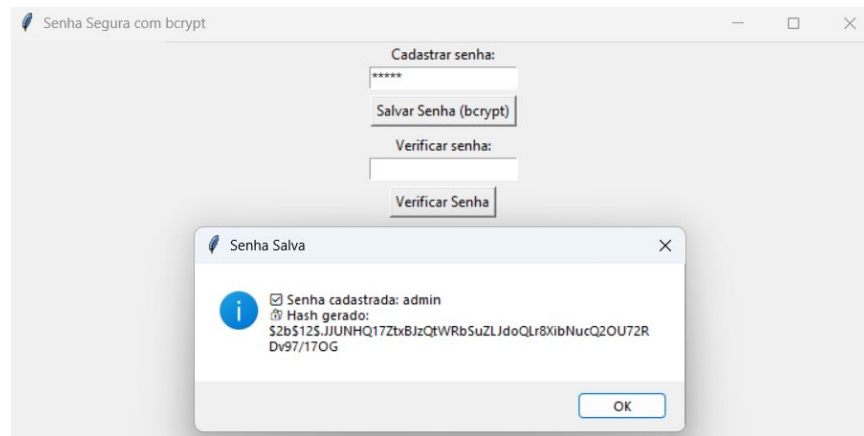
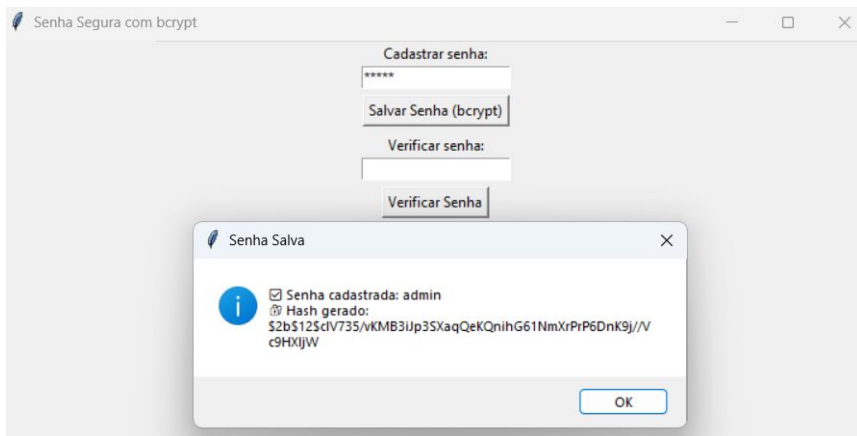
A graphic of a rainbow table, which is a data structure used for cracking password hashes. It is depicted as a multi-colored, 3D rectangular block with the words "RAINBOW TABLE" in bold, purple, capital letters on its top surface. Below the graphic is a table showing a mapping of plaintext passwords to their MD5 checksums.

Plaintext	MD5 Checksum
123456	e10adc3949ba59abbe56e057f20f883e
123456789	25f9e794323b453885f5181f1b624d0b
password	5f4dcc3b5aa765d61d8327deb882cf99
adobe123	7558af202997483d3afef3bb2b5a709d
12345678	25d55ad283aa400af464c76d713c07ad
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567	fcea920f7412b5da7be0cf42b8c93759
111111	96e79218965eb72c92a549dd5a330112
photoshop	c7c9cfbb7ed7d1cebb7a4442dc30877f
123123	4297f44b13955235245b2497399d7a93

Exemplo 2: Código Corrigido (bcrypt)

- Hash: Um algoritmo projetado para senhas.

```
9 def armazenar_bcrypt(senha: str) -> str:
10     salt = bcrypt.gensalt()
11     return bcrypt.hashpw(senha.encode(), salt).decode()
12
13 def verificar_bcrypt(senha: str, hash_armazenado: str) -> bool:
14     return bcrypt.checkpw(senha.encode(), hash_armazenado.encode())
15
```



- Senha testada: admin = hashes diferentes.

Exemplo 2: Por que este código é seguro?

- Uso de salt automaticamente: `bcrypt.gensalt()` gera um salt único para cada senha.
- Mesmo senha → hashes diferentes: Por conta do salt, mesmo que dois usuários usem a mesma senha, os hashes nunca serão iguais.
- Hash lento (intencional): `bcrypt` foi projetado para ser computacionalmente custoso, dificultando ataques de força bruta e dicionário.
- Resistente a rainbow tables: Rainbow tables não funcionam porque cada hash é único devido ao salt.

`$2y$10$6z7GKa9kpDN7KC3ICW1Hi.f d0/to7Y/x36WUKNP0IndHdkdR9Ae3K`

The diagram illustrates the structure of the bcrypt hash string `$2y$10$6z7GKa9kpDN7KC3ICW1Hi.f d0/to7Y/x36WUKNP0IndHdkdR9Ae3K`. It is divided into three main sections by vertical lines:
1. **Algorithm** (red line): The first two characters, `$2y`.
2. **Algorithm options (eg cost)** (blue line): The next two characters, `$10`.
3. **Salt** (green line): The next 22 characters, `$6z7GKa9kpDN7KC3ICW1Hi.`.
The final 22 characters, `f d0/to7Y/x36WUKNP0IndHdkdR9Ae3K`, are labeled as the **Hashed password** (orange line).

Exemplo 3: Falha na verificação de integridade

- Criptografar dados com AES sem verificar a integridade pode ser perigoso
- Alguém pode alterar o ciphertext (bit flipping) e a aplicação pode aceitar o resultado
- O AES-CBC descriptografa sem saber que os dados foram modificados
- Solução: usar HMAC ou um modo autenticado como AES-GCM

CWE-353: Missing Support for Integrity Check

CWE-302: Authentication Bypass by Assumed-Immutable Data

Exemplo 3: Exemplo de código

```
# Criptografia sem autenticar (inseguro)
def criptografar_sem_hmac(msg, chave, iv):
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    return cipher.encrypt(pad(msg, AES.block_size))

# Criptografia com HMAC (seguro)
def criptografar_com_hmac(msg, chave, iv, chave_hmac):
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(msg, AES.block_size))
    mac = hmac.new(chave_hmac, ciphertext, hashlib.sha256).digest()
    return ciphertext + mac
```

```
# Verifica integridade e descriptografa (inseguro)
def descriptografar_sem_hmac(dados, chave, iv, chave_hmac):
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(bytes(dados)), AES.block_size)

# Verifica integridade e descriptografa (seguro)
def descriptografar_com_hmac(dados, chave, iv, chave_hmac):
    ciphertext = dados[:-32]
    mac_recebido = dados[-32:]
    mac_calculado = hmac.new(chave_hmac, ciphertext, hashlib.sha256).digest()
    if not hmac.compare_digest(mac_recebido, mac_calculado):
        return b"ERRO: dados modificados"
    cipher = AES.new(chave, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size)
```

Exemplo 3: Exemplo de código

```
def main():
    msg = b"mensagem secreta"
    chave = get_random_bytes(32)
    iv = get_random_bytes(16)
    chave_hmac = get_random_bytes(32)

    # Criptografia insegura
    cripto_inseguro = criptografar_sem_hmac(msg, chave, iv)
    alterado_inseguro = bytearray(cripto_inseguro)
    alterado_inseguro[0] ^= 0x01 # altera um byte

    # Criptografia segura
    cripto_seguro = criptografar_com_hmac(msg, chave, iv, chave_hmac)
    alterado_seguro = bytearray(cripto_seguro)
    alterado_seguro[0] ^= 0x01 # altera um byte
```

Exemplo 3: Exemplo de código

```
try:
    resultado_inseguro = descriptografar_sem_hmac(alterado_inseguro, chave, iv, chave_hmac)
except ValueError:
    resultado_inseguro = b"ERRO: dados corrompidos"

resultado_seguro = descriptografar_com_hmac(bytes(alterado_seguro), chave, iv, chave_hmac)

print("Resultado sem HMAC :", resultado_inseguro.decode(errors="ignore"))
print("Resultado com HMAC :", resultado_seguro.decode(errors="ignore"))
```

```
Resultado sem HMAC : ERRO: dados corrompidos
Resultado com HMAC : ERRO: dados modificados
```

Exemplo 4: Uso de hash MD5 para senhas/tokens

- Problema: MD5 é vulnerável a colisões e ataques de dicionário.
- Exploração: A senha pode ser descoberta com um ataque simples se for comum.
- Correção: Usar bcrypt com salt e fator de custo.

CWE-328 – Use of Weak Hash

Exemplo 4: Uso de hash MD5 para senhas/tokens

```
import hashlib
from flask import Flask, request, jsonify
app = Flask(__name__)

# Função para carregar credenciais do arquivo senha.txt
def carregar_usuarios():
    usuarios = {}
    with open("senha.txt", "r") as f:
        for linha in f:
            partes = linha.strip().split(":")
            if len(partes) == 2:
                usuarios[partes[0]] = partes[1] # username -> senha em texto puro
    return usuarios
usuarios = carregar_usuarios()

@app.route("/login", methods=["POST"])
def login():
    data = request.json
    username = data.get("username")
    password = data.get("password")

    if username in usuarios and usuarios[username] == password: # Comparação direta
        token = hashlib.md5(password.encode()).hexdigest() # Gerando token MD5
        return jsonify({"token": token}), 200
    return jsonify({"error": "Credenciais inválidas"}), 401

@app.route("/dados", methods=["GET"])
def dados():
    token = request.headers.get("Authorization")
    for user, senha in usuarios.items():
        if token == hashlib.md5(senha.encode()).hexdigest():
            return jsonify({"message": f"Acesso concedido! Usuario: {user}"}), 200
    return jsonify({"error": "Acesso negado"}), 403

if __name__ == "__main__":
    app.run(debug=True)
```

- Implementação API vulnerável
- Efetua login enviando usuário e senha
- API consulta arquivo senha.txt
- Retorna Token MD5 da senha
- Utiliza este Token MD5 para acessar dados protegidos

Exemplo 4: Uso de hash MD5 para senhas/tokens

Porque é vulnerável

- Como API aceita qualquer token MD5 válido um atacante pode reutilizar esse token sem precisar da senha original.
- Atacante pode gerar manualmente o hash e utilizar.
- Atacante pode quebrar o hash utilizando hascat para descobrir a senha original.
 - `hashcat -m 0 -a 3 <TOKEN_MD5> ?a?a?a?a?a?`
- MD5 é rápido a senha pode ser quebrada facilmente.
- MD5 têm vulnerabilidades conhecidas de colisão, onde dois valores diferentes podem gerar o mesmo hash.

Exemplo 4: Uso de hash MD5 para senhas/tokens

```
import bcrypt
import jwt
import datetime
from flask import Flask, request, jsonify

app = Flask(__name__)
SECRET_KEY = "chave_super_secreta"
# Função para carregar credenciais do arquivo senha.txt (convertendo para bcrypt)
def carregar_usuarios():
    usuarios = {}
    with open("senha.txt", "r") as f:
        for linha in f:
            partes = linha.strip().split(":")
            if len(partes) == 2:
                senha_segura = bcrypt.hashpw(partes[1].encode(), bcrypt.gensalt()).decode()
                usuarios[partes[0]] = senha_segura # username -> senha criptografada
    return usuarios
usuarios = carregar_usuarios()

@app.route("/login", methods=["POST"])
def login():
    data = request.json
    username = data.get("username")
    password = data.get("password")

    if username in usuarios and bcrypt.checkpw(password.encode(), usuarios[username].encode()):
        token = jwt.encode({"user": username, "exp": datetime.datetime.utcnow() + datetime.timedelta(
            seconds=300)}, SECRET_KEY, algorithm="HS256")
        return jsonify({"token": token}), 200
    return jsonify({"error": "Credenciais invalidas"}), 401

@app.route("/dados", methods=["GET"])
def dados():
    token = request.headers.get("Authorization")
    try:
        decoded = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
        return jsonify({"message": f"Acesso concedido para {decoded['user']}!")), 200
    except jwt.ExpiredSignatureError:
        return jsonify({"error": "Token expirado"}), 403
    except jwt.InvalidTokenError:
        return jsonify({"error": "Token invalido"}), 403

if __name__ == "__main__":
    app.run(debug=True)
```

- Implementação API corrigida
- Efetua login enviando usuário e senha
- API consulta arquivo senha.txt se senha for correta gera um Token JWT com tempo de expiração de 5 minutos
- Recebe um Token JWT no cabeçalho Authorization
- Verifica expiração e autenticidade do Token para acessar dados protegidos

Exemplo 4: Uso de hash MD5 para senhas/tokens

Porque é mais seguro

- Armazenamento seguro de senhas com bcrypt.
- Autenticação baseada em JWT (JSON Web Token) em vez de MD5.
- Tokens assinados e com tempo de expiração para evitar reutilização.
- Senhas protegidas com bcrypt não podem ser quebradas por força bruta

Conclusão

"O que vimos até aqui não são falhas em algoritmos quebrados ou tecnologia ultrapassada — são erros de uso incorreto da criptografia moderna.
Ou seja: não basta criptografar, é preciso saber como criptografar do jeito certo."

- Exemplo 1: Comparação insegura

Mostrou que **detalhes pequenos** — como `==` em vez de `compare_digest()` — podem abrir portas para ataques sofisticados como **timing attacks**, mesmo com sistemas que parecem protegidos.

- Exemplo 2: Uso de chaves fracas/aleatoriedade fraca;

Mostrou que **a escolha errada de função de hash** compromete todo o sistema.

Mesmo que a senha seja secreta, um hash previsível pode ser quebrado com um simples dicionário.

- Exemplo 3 e 4: Uso de hash/função fraca: em alguns casos, quanto maior o custo computacional, melhor.

Mostrou que **criptografia sozinha não basta**.

Sem HMAC ou modos autenticados, dados podem ser **alterados silenciosamente** — e o sistema nem percebe.

Esses três exemplos mostram que a segurança não está só no algoritmo — está no detalhe da implementação."

Referências

- **Sotirov, Alexander, et al.** MD5 considered harmful today, creating a rogue CA certificate. 25th Annual Chaos Communication Congress. 2008. Disponível em: <<https://cir.nii.ac.jp/crid/1570572701274438656>>
- **Michael Howard, David LeBlanc, John Viega.** 2010. 24 Deadly Sins of Software Security: Programming Flaws and how to Fix Them (1 ed.). McGraw-Hill, Inc., New York, NY, USA. ISBN: 978-0-071-62675-0.
- **Dhem, JF., Koeune, F., Leroux, PA., Mestré, P., Quisquater, JJ., Willems, JL.** A Practical Implementation of the Timing Attack. In: Quisquater, JJ., Schneier, B. (eds) Smart Card Research and Applications. CARDIS 1998. Lecture Notes in Computer Science, vol 1820. Springer, Berlin, Heidelberg, 2000. p. 167–182. ISBN 978-3-540-44534-0.