Trabalho 1 REST API com Autenticação Segura e Criptografia

Leandro Beloti Kornelius - 211020900

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
Tópicos Avançados em Segurança Computacional, 2025/1

1. Introdução

Uma REST API é um tipo de interface que permite a modularização e a comunicação eficiente entre sistemas por meio de requisições HTTP. No escopo deste trabalho, foram desenvolvidas duas REST APIs: uma responsável pela autenticação de usuários e outra protegida, cujos dados só podem ser acessados por usuários devidamente autenticados.

Nesse contexto, para garantir a confidencialidade, integridade e segurança dos dados, é fundamental a adoção de mecanismos de proteção que impeçam acessos indevidos, alterações não autorizadas ou vazamento de informações sensíveis. Um dos métodos mais comuns utilizados em aplicações modernas é o uso de tokens, como o JWT (JSON Web Token). Esses tokens são gerados após um login bem-sucedido, ou seja, quando o sistema reconhece um usuário válido. O token encapsula informações essenciais, como a identidade do usuário, suas permissões no sistema e o tempo de expiração do acesso.

O token é geralmente enviado no cabeçalho das requisições HTTP, no formato Authorization: Bearer token, sendo validado a cada nova chamada para assegurar o controle de acesso. No entanto, como os tokens podem ser interceptados por terceiros em conexões inseguras, é indispensável o uso de HTTPS — que implementa SSL/TLS — para criptografar a comunicação entre cliente e servidor. Dessa forma, mesmo que os dados trafeguem pela internet, eles estarão protegidos contra interceptações e ataques do tipo man-in-the-middle.

Assim, a segurança é reforçada tanto no nível da autenticação e autorização quanto na própria transmissão de dados entre cliente e servidor, garantindo uma comunicação segura e confiável nas APIs desenvolvidas. Abaixo, iremos detalhar alguns aspectos da implementação desta lógica de autenticação e proteção dos dados durante a comunicação entre cliente-servidor.

2. Implementação

Todas as seções descritas abaixo, podem ser verificadas com mais detalhes no repositório do github.

https://github.com/LeandroKornelius/UnB-Advanced-Security/tree/main/Trabalho%201/main

Neste relatório foram descritas e mostradas apenas as etapas mais importantes para o funcionamento adequado do sistema conforme solicitado.

2.1. Banco de Dados com SQLITE

Para a camada de persistência foi escolhido o SQLITE como banco de dados por sua simplicidade e integração nativa com a linguagem do projeto, python. O banco da aplicação terá apenas uma tabela de usuários com os seguintes atributos:

- **id**: chave primária, auto-incrementada
- **email**: atributo único, de tal forma que dois usuários não possam se cadastrar no sistema com o mesmo email
- **senha**: atributo essencial para autenticar usuários

O arquivo também define as funções relacionadas ao banco de dados, cujas particularidades estão descritas a seguir:

- **Inicialização**: inicia o banco com a tabela e atributos discutidos acima.
- **Existência de um usuário no Banco de Dados**: através do email recebido, verifica se aquele email já pertence ao Banco de Dados do sistema.
- **Inserção de novo usuário com hash da senha**: através do email e senha recebidos, verifica se o usuário já pertence ao Banco de Dados e, caso não exista o cadastra no sistema com hash da senha para maior proteção. Este funcionamento pode ser visualizado na Figura 1 presente no apêndice deste relatório.
- **Obtenção dos emails cadastrados no sistema**: Na aplicação, esta será a informação sigilosa, a qual só poderá ser obtida para usuários autenticados. A função apenas retorna o email de todos usuários cadastrados no sistema.
- **Verificação de senha**: através do email e senha recebidos, esta função irá validar o acesso do usuário, ou seja verificar se a senha recebida corresponde aquela presente no Banco de Dados para o usuário com o email recebido. Este funcionamento está detalhado na Figura 2 do apêndice do relatório

2.2. API Protegida

A API protegida deve implementar duas atividades importantes:

A primeira consiste em tratar uma requisição do tipo ”POST” em que qualquer usuário, para este sistema específico, possa se cadastrar no sistema através de um email e senha. Este método da API deve lidar com aspectos como: verificação da existência de email e senha no corpo da requisição, verificação da existência de um usuário com aquele email e criação do usuário caso tudo esteja certo. Todos estes aspectos podem ser verificados no apêndice do relatório através da Figura 3.

Ademais, também é necessário implementar o aspecto protegido da API que é a rota que retorna o email de todos usuários cadastrados no sistema. Sob essa ótica, esta função deve receber um token no cabeçalho da requisição e chamar a função para validar o token a qual será detalhada em uma próxima subsessão. Ao receber a resposta da validação do token, as informações são disponibilizadas pelo sistema. Caso não haja token, o token esteja expirado e/ou inválido, o usuário não receberá as informações. Estas características podem ser notadas na Figura 4 a qual detalha os casos em que os email não são disponibilizados e a Figura 5 em que a API de Autenticação valida o usuário a obter essas informações.

2.3. API de Autenticação

A API de autenticação tem como objetivo gerar o token para que o usuário possa solicitar dados da API Protegida e, caso tenha permissão para acessar estas informações, consiga. Nesse sentido, é necessário validar se tanto o email e senha para realizar o login foram recebidas. Com isso, validar se a senha corresponde aquela guardada no banco para o email recebido. Finalmente, caso tudo acima não tenha tido problemas, gerar o token

de acordo com o algoritmo de autenticação utilizado. A Figura 2 do apêndice realiza a verificação da senha e a função de geração de token será detalhada na próxima sub-sessão do relatório.

2.4. JWT Handler

Este arquivo implementa duas funcionalidades muito importantes e que são diretamente relacionadas ao JWT token.

Como mencionado anteriormente, é necessário gerar o token a partir de um algoritmo de encriptação, das informações do usuário e com um tempo de expiração definido. É importante salientar que é recomendado ter tempo de expirações curto e uma lógica de reatualização do token para ele não expirar enquanto o usuário estiver usando o sistema, porém isto não foi implementado no sistema. O token JWT gerado pela API de autenticação utiliza os algoritmos HMAC, RSA PKCS1, ou RSA PSS e inclui um tempo para sua expiração que limita seu tempo de validade, conforme exigido pela especificação do trabalho. A geração do token está ilustrada na Figura 6 do apêndice.

Além deste funcionamento, faz-se necessário validar o token. Ou seja, verificar se o token não está ausente, expirado ou inválido/comprometido. A função de validação do token, presente na Figura 7 do apêndice, implementa essa lógica. Logo, possibilita que o token do usuário seja checado antes de disponibilizar qualquer informação.

Nota-se também que no código três informações sensíveis dos algoritmos foram inseridas diretamente no arquivo: o segredo do HMAC e as chaves públicas e privadas do RSA. Isto foi feito para simplificar o funcionamento da aplicação. Em um ambiente de produção e de segurança, a segredo usado no HMAC deveria estar em uma variável de ambiente enquanto as chaves do RSA devem estar protegidas pelos respectivos usuários. A prática adotada no código é aceitável apenas em ambientes de desenvolvimento ou aprendizado, nunca em produção.

Em breve, será feito um contraste entre os diferentes algoritmos de encriptação utilizados.

2.5. Cliente

O cliente tem como intuito simular alguém usando o sistema implementado tanto com as diferentes possibilidades de algoritmos de encriptação utilizados para realizar a autenticação:

- **HMAC**
- **RSA PKCS1**
- **RSA PSS**

Quanto para testes com diferentes cenários do token, ou seja:

- **Casos de login bem sucedidos e utilização do token para obtenção de informações sigilosas**
- **Token ausente**
- **Token inválido**
- **Token expirado**

Para isso, foram feitas as seguintes funções:

- **Função que invalida token**: Esta função particiona o token e o inverte (Figura 8)
- **Expira token**: Esta função gera um token vencido a depender do algoritmo de encriptação (Figura 9)

Por fim, este arquivo realiza todos testes necessários e pode ser visualizado através da Figura 10.

2.6. Segurança na comunicação entre requisições e respostas

Observa-se nas Figuras 11 e 12 que foi implementado o SSL na API. Tal implementação consiste em criar um contexto SSL que especifica que o servidor atuará como parte receptora de uma conexão segura em que o certificado e a chave são carregadas neste contexto. Portanto, o servidor será iniciado envolvido com este contexto sendo assim um servidor HTTPS.

Ademais, todas as requisições que envolvem envio de credenciais (e.g., login) foram realizadas sob protocolo HTTPS, garantindo que o e-mail e a senha sejam transmitidos de forma segura, sem risco de exposição mesmo que o tráfego seja interceptado.

O SSL permite que os dados tráfegados entre cliente e servidos estejam protegidos. Tornando a interceptação, visualização de informações sensíveis como token e outras mais difíceis.

3. Análises

O uso de tokens JWT permite a utilização de diferentes algoritmos de encriptação como o HMAC (método HS256 no código), RSA PSS (método no código) e RSA PKCS (método no código).

O HMAC é uma encriptação simétrica, ou seja, faz uso da mesma chave privada para gerar e verificar o token. Enquanto ambas possibilidades do RSA implementadas se tratam de encriptações assimétricas, em que há uma chave privada para gerar o token e outra chave pública para verificar o token. Ambas opções são seguras e preferíveis a depender da aplicação que a está usando. Por exemplo, é notável que em sistemas menores ou internos o JWT com HMAC se destaca por ser mais simples de implementar e desempenho superior em termos de processamento. Já em arquiteturas distribuídas e aplicações de larga escala, onde é vantajoso somente o emissor conhecer a chave privada o JWT com RSA se torna mais interessante.

Diante do exposto, observa-se como principal desvantagem do HMAC é a necessidade de compartilhar e manter segura a mesma chave entre todas as partes que assinam e verificam os tokens.

Por outro lado, o uso de RSA seja com PSS ou PKCS oferece maior segurança para validação pública e facilita o escalonamento, pois a chave privada nunca precisa ser exposta. Contudo, esse modelo apresenta desvantagens como maior custo computacional e tokens potencialmente maiores, além da complexidade na gestão das chaves.

No cenário 1 (HMAC), uma vulnerabilidade comum é o vazamento da chave secreta, que permitiria a qualquer parte gerar tokens válidos. Para mitigar, é recomendado o uso de HSMs e rotação periódica de chaves.

Já no cenário 2 (RSA), ataques podem ocorrer caso a chave privada seja exposta ou se algoritmos obsoletos forem utilizados. O uso de RSA-PSS reduz riscos devido à sua

resistência a ataques por padding. Além disso, tokens devem conter claims como ‘exp‘, ‘iat‘ e ‘nbf‘ para evitar ataques de replay.

Em ambos os casos, a verificação constante do tempo de validade e a invalidação de tokens comprometidos são medidas essenciais.

Para mitigar as desvantagens do HMAC, especialmente em ambientes distribuídos, é possível adotar estratégias como o uso de serviços centralizados de autenticação, rotação frequente de chaves e uso de HSMs (Hardware Security Modules) para proteger as chaves. Já para reduzir o impacto do custo computacional e tamanho dos tokens com RSA, pode-se utilizar algoritmos de curva elíptica (como ES256) como alternativa mais eficiente, além de adotar cache inteligente para validação de tokens e reduzir chamadas repetidas ao processo de verificação. Dessa forma, é possível aproveitar os benefícios de segurança de cada abordagem, adaptando-se às necessidades específicas da aplicação.

4. Conclusão

O desenvolvimento da REST API com autenticação segura e uso de criptografia baseada em JWT permitiu compreender como é possível garantir a autenticação e autorização de usuários por meio de tokens assinados com diferentes algoritmos criptográficos, como HMAC e RSA.

Além disso, a proteção da comunicação via SSL (HTTPS) assegura a integridade e confidencialidade das informações transmitidas entre cliente e servidor, impedindo ataques de interceptação e man-in-the-middle. A escolha de armazenar senhas com hashing e de validar tokens em cada requisição sensível reforça o compromisso com boas práticas de segurança e pode ser verificado a partir da Figura 13.

Embora o projeto tenha simplificado alguns aspectos — como o armazenamento direto das chaves criptográficas no código —, ele cumpre o objetivo proposto de aplicar os conceitos fundamentais de autenticação e criptografia, sendo uma base sólida para projetos mais complexos em ambientes reais.

Portanto, o trabalho evidenciou a relevância da segurança desde as camadas mais básicas, como o banco de dados e autenticação, até a transmissão dos dados, consolidando um sistema funcional e seguro.

Apêndice

A. Anexo I: Função que realiza inserção de novo Usuário

```python
def add_user(email, password):
    # Checks if user exists
    if user_exists(email):
        return False  # User already exists

    # Hashes the password to increase security
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
    try:
        con = sqlite3.connect(DB_PATH)
        cur = con.cursor()
        cur.execute('INSERT INTO users (email, password) VALUES (?, ?)', (email, hashed_password))
        con.commit()
        return True
    except sqlite3.IntegrityError:
        return False
    finally:
        con.close()
```

Figure 1. Função que realiza a criação de um novo usuário no banco caso não exista alguém cadastrada com o mesmo email

B. Anexo II: Função que valida equivalência da senha

```python
def check_user_password(email, password):
    user = get_user(email)
    if not user:
        return False
    stored_hash = user[1]
    return bcrypt.checkpw(password.encode('utf-8'), stored_hash)
```

Figure 2. Função que realiza a verificação da equivalência da senha recebida com a armazenada no banco para permitir login

C. Anexo III: Método que adiciona novo usuário

```
try:
    data = json.loads(body)
    email = data.get('email')
    password = data.get('password')
except:
    self.send_response(400)
    self.end_headers()
    return

if not email or not password:
    self.send_response(400)
    self.end_headers()
    self.wfile.write(b'Email and password required')
    return

if add_user(email, password):
    self.send_response(201)
    self.end_headers()
    self.wfile.write(b'User created')
else:
    self.send_response(409)
    self.end_headers()
    self.wfile.write(b'User already exists')
```

Figure 3. Função que valida se os atributos necessários para criação de usuário foram recebidos e trata possíveis erros de ausência de atributos ou usuário já pertence ao banco

D. Anexo IV: Parte que verifica se foi recebido um token

```
auth_header = self.headers.get('Authorization')
if not auth_header or not auth_header.startswith('Bearer '):
    self.send_response(401)
    self.end_headers()
    self.wfile.write(b'Token not provided')
    return

token = auth_header.split()[1]
```

Figure 4. Parte do método que valida se foi recebido um token para prosseguir com validação do mesmo

E. Anexo V: Parte que verifica possíveis erros com o token recebido

```
try:
    payload = validate_token(token, method)
except TokenMissingError:
    self.send_response(401)
    self.end_headers()
    self.wfile.write(b'Token is missing.')
    return
except TokenExpiredError:
    self.send_response(401)
    self.end_headers()
    self.wfile.write(b'Token has expired.')
    return
except TokenInvalidError:
    self.send_response(403)
    self.end_headers()
    self.wfile.write(b'Token is invalid.')
    return
except UnsupportedAlgorithmError:
    self.send_response(500)
    self.end_headers()
    self.wfile.write(b'Unsupported token algorithm.')
    return
except Exception as e:
    self.send_response(500)
    self.end_headers()
    self.wfile.write(b'Internal server error.')
    return

emails = get_users_emails()
```

Figure 5. Parte do método que valida cada possível caso do token: se está ausente, expirado, inválido e se o algoritmo do token não está previsto na aplicação

F. Anexo VI: Geração do Token

```python
def generate_token(email, method):
    payload = {
        'sub': email,
        'exp': datetime.datetime.utcnow() + datetime.timedelta(minutes=10)
    }
    if method == 'HS256':
        return jwt.encode(payload, HMAC_SECRET, algorithm=method)
    elif method == 'RS256':
        private_key = b"-----BEGIN RSA PRIVATE KEY-----\nMIIEpAIBAAKCAQEAwI
        return jwt.encode(payload, private_key, algorithm=method)
    elif method == 'PS256':
        private_key = b"-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEAuI
        return jwt.encode(payload, private_key, algorithm=method)
    else:
        raise ValueError('Unsupported token generate algorithm')
```

Figure 6. Conforme o algoritmo de autenticação recebido, realiza a encriptação do token

G. Anexo VII: Validação do Token

```python
def validate_token(token, method):
    if not token:
        raise TokenMissingError('Missing token')
    try:
        if method == 'HS256':
            return jwt.decode(token, HMAC_SECRET, algorithms=['HS256'])
        elif method == 'RS256':
            public_key = b"-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ
            return jwt.decode(token, public_key, algorithms=[method])
        elif method == 'PS256':
            public_key = "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8
            return jwt.decode(token, public_key, algorithms=[method])
        else:
            raise UnsupportedAlgorithmError('Unsupported token validation algorithm')
    except jwt.ExpiredSignatureError:
        raise TokenExpiredError('Token has expired')
    except jwt.InvalidTokenError:
        raise TokenInvalidError('Token is invalid')
```

Figure 7. Realiza decriptação do token para validar autenticação do usuário e permitir acesso à informações protegidas

H. Anexo VIII: Função que invalida token

```python
def simulate_invalid_signature(token):
    parts = token.split(".")
    corrupted = parts[2][::-1]
    return ".".join(parts[:2] + [corrupted])
```

Figure 8. Esta função invalida o token para testar caso de token inválido

I. Anexo IX: Função que gera token expirado

```python
def simulate_expired_token(email, method):
    payload = {
        'sub': email,
        'exp': int(time.time() - 61)
    }

    if method == 'HS256':
        return jwt.encode(payload, SHARED_SECRET, algorithm=method)
    elif method == 'RS256':
        private_key = b"-----BEGIN RSA PRIVATE KEY-----\nMIIEpAIBAAK
        return jwt.encode(payload, private_key, algorithm=method)
    elif method == 'PS256':
        private_key = b"-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAK
        return jwt.encode(payload, private_key, algorithm=method)
```

Figure 9. Esta função gera token expirado para testar caso de token expirado

J. Anexo X: Testes com sistema

```python
if __name__ == '__main__':
    email = 'cliente@teste.com'
    password = '123456'

    sign_up(email, password)

    for method in ['RS256', 'PS256', 'HS256']:
        print(f'\nIniciating tests with {method}...\n')

        token = login(email, password, method)

        if token:
            get_user_emails(token, method, 'Valid Token')

            get_user_emails(None, method, 'Missing Token')

            corrupted_token = simulate_invalid_signature(token)
            get_user_emails(corrupted_token, method, 'Invalid Token')

            expired_token = simulate_expired_token(email, method)
            get_user_emails(expired_token, method, 'Expired Token')

        else:
            print(f'Either user does not exist or the credentials are incorrect')
```

Figure 10. Este trecho do código teste o sistema para todos algoritmos de encriptação e todos casos de token: válido, expirado, ausente e inválido

K. Anexo XI: HTTPS na API Protegida

```python
if __name__ == '__main__':
    cert_path = os.path.join(os.path.dirname(__file__), '..', 'cert.pem')
    key_path = os.path.join(os.path.dirname(__file__), '..', 'key.pem')

    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile=cert_path, keyfile=key_path)

    server_address = ('localhost', 3000)
    httpd = HTTPServer(server_address, ProtectedHandler)
    httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

    print('🔒 Protected API running securely on https://localhost:3000')
    httpd.serve_forever()
```

Figure 11. Carregamento de contexto SSL para tornar a API Protegida em uma HTTPS

L. Anexo XII: HTTPS na API de Autenticação

```python
if __name__ == '__main__':
    cert_path = os.path.join(os.path.dirname(__file__), '..', 'cert.pem')
    key_path = os.path.join(os.path.dirname(__file__), '..', 'key.pem')

    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile=cert_path, keyfile=key_path)

    server_address = ('localhost', 3333)
    httpd = HTTPServer(server_address, AuthHandler)
    httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

    print('🔐 Auth API running securely on https://localhost:3333')
    httpd.serve_forever()
```

Figure 12. Carregamento de contexto SSL para tornar a API de Autenticação em uma HTTPS

M. Anexo XIII: Visualização do hash das senhas no banco

	id	email	password
1	1	email@gmail.com	$2b$12$0kRJJLvkPYePGccyfce3pOusNEE027Krqc048p.nwGe4QFtJkSY3i
2	2	cliente@teste.com	$2b$12$O0y.5B95ksGV.iblT6fGR.JghQfRdtty0nobulw3wW6LYadsIN.ce

Figure 13. Ao visualizar a coluna de senha dos usuários, nota-se que há apenas o hash das senhas dos usuários