



Security of Information and Organizations

2024/2025

Security and Technical report

Regent teacher

João Paulo Barraca

Practical teacher

Vitor Cunha

Paulo Bartolomeu

Team members

Rafael Kauati - 105925

Vasco Vouzela - 108130

Alírio Rego - 95088

Introduction.....	4
High-level implemented features.....	5
Subject credentials generation.....	5
Creation of organization, subjects and documents.....	6
Document management.....	9
Authentication, Re-authentication and session token.....	13
Document encryption/decryption management.....	18
Secured communication (partially implemented).....	22
Nonce control for replay attacks prevention (partially implemented).....	24
Role-Permission access control.....	26
Role-Permission management.....	28
Context of the application for security analysis.....	35
Security Evaluation (V3 - Session Management).....	35
3.1 Fundamental Session Management Security.....	35
3.1.1 Verify the application never reveals session tokens in URL parameters.....	35
3.2 Session Binding.....	37
3.2.1 Verify the application generates a new session token on user authentication.....	38
3.2.2 Verify that session tokens possess at least 64 bits of entropy.....	39
3.2.3 Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML 5 session storage.....	40
3.2.4 Verify that session tokens are generated using approved cryptographic algorithms.....	42
Lack of Centralized Key Management.....	43
3.3 Session Termination.....	44
3.3.1 Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.....	44
3.3.2 If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period.....	46
3.3.3 Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties.....	49
3.3.4 Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.....	50
3.4 Cookie-based Session Management.....	51
3.5 Token-based Session Management.....	51
3.5.1 Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.....	51
Analysis:.....	51
3.5.2 Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.....	52
Context:.....	52
3.5.3 Verify that stateless session tokens use digital signatures, encryption, and other	

<u>countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.....</u>	<u>52</u>
<u>Recommendations for L3 Compliance:.....</u>	<u>59</u>
<u>3.6 Federated Re-authentication.....</u>	<u>59</u>
<u>3.7 Defenses Against Session Management Exploits.....</u>	<u>60</u>
<u>3.7.1 Verify the application ensures a full, valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.....</u>	<u>60</u>
<u>Conclusion.....</u>	<u>61</u>

Introduction

This report will provide both a description over the features implemented and an evaluation of the technical and security of the document-sharing application developed by the team members for the two previous deliveries, analysing if the certain aspects of documented security concerns are applicable within the application context and, if applicable, evaluate qualitatively how the application fulfill the security and technical requirements of the ASVS v3 chapter's topic.

High-level implemented features

Subject credentials generation

The application provides the command *rep_subject_credentials*, that generates a key pair for the subject based on a given password, using ECC encryption algorithm, then it stores in a credentials file that will be used when creating the first subject of the organization or adding a new subject to the organization.

```
# Derive a seed from the password
salt = os.urandom(16) # Random salt for key derivation
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32, # Length of the derived seed
    salt=salt,
    iterations=100_000,
)
seed = kdf.derive(password.encode()) # Generate seed from password

# Use the seed to generate an ECC private key deterministically
private_key = ec.derive_private_key(int.from_bytes(seed, byteorder="big"), ec.SECP256R1())
public_key = private_key.public_key()
tk, 4 weeks ago • setup for del2

# Serialize keys
private_pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption(),
).decode('utf-8')

public_pem = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo,
).decode('utf-8')

# Save the keys and salt in a JSON file
credentials_data = {
    "public_key": public_pem,
    "private_key": private_pem,
    "salt": salt.hex(), # Save salt for future regeneration of keys
}

with open(credentials_file, "w") as cred_file:
    json.dump(credentials_data, cred_file, indent=4)
```

code snippet for the user credentials

Creation of organization, subjects and documents

The application provides the capability of creating organizations, add subject into these organizations (initially performed by the first subject created when the organization is created) and add documents to its organizations

When the first subject is created, or when you add a new subject into the organization, the subject's public key(inside the credentials file) is stored in the DB. This key will be used in the authentication later on

```
class OrganizationController:
    def create_organization():
        return jsonify({"error": "Encrypted payload is missing"}), 400
        encrypted_payload = binascii.unhexlify(encrypted_payload)
        decrypted_payload = decrypt_with_chacha20(chacha_key, chacha_nonce, encrypted_payload)
        payload_data = json.loads(decrypted_payload)

        # Extrair dados do payload
        org_name = payload_data.get("name")
        subject_data = payload_data.get("subject")
        public_key = payload_data.get("public_key")
        if not org_name or not subject_data or not public_key:
            return jsonify(
                {'error': f'Organization name, subject and subject public data are required {payload_data}'}
            )

        # Extrair dados do sujeito
        username = subject_data.get("username")
        full_name = subject_data.get("full_name")
        email = subject_data.get("email")

        if not username or not full_name or not email:
            return jsonify({'error': 'All subject fields are required'}), 400

        # Criar instâncias
        organization = Organization(name=org_name)
        subject = Subject(
            username=username,
            full_name=full_name,
            email=email,
            public_key=public_key
        )

        # Criar a role "Manager" e associar permissões
        manager_role = Role(name="Manager", organization=organization)
        db.session.add(manager_role)

        # Associar todas as permissões existentes à role "Manager"
        permissions = Permission.query.all() # Recupera todas as permissões da tabela 'permissions'
        for permission in permissions:
            role_permission = RolePermission(role=manager_role, permission=permission)
            db.session.add(role_permission)
```

```
# Criar a role "Manager" e associar permissões
manager_role = Role(name="Manager", organization=organization)
db.session.add(manager_role)

# Associar todas as permissões existentes à role "Manager"
permissions = Permission.query.all() # Recupera todas as permissões da tabela 'permissions'
for permission in permissions:
    role_permission = RolePermission(role=manager_role, permission=permission)
    db.session.add(role_permission)

# Adicionar o sujeito à organização
organization.subjects.append(subject)

# Associar a role "Manager" apenas ao sujeito recém-criado
subject.roles.append(manager_role)

# Salvar entidades no banco de dados
db.session.commit()

return jsonify({
    'message': 'Organization and subject created successfully, and role "Manager" created and associated'}), 201
```

code snippet for the creation of an organization and its first subject(the manager)

```

def add_subject_to_organization(session_key, nonce, username, name, email, public_key):

    # Salvar o novo nonce
    new_nonce_entry = Nonce(nonce=new_nonce, used=False)
    db.session.add(new_nonce_entry)
    db.session.commit()

    session = check_session(session_key)
    if session is None:
        return {"error": "Sessão inválida ou não encontrada", "new_nonce": new_nonce}, 404

    if not has_permission(session_key, "SUBJECT_NEW"):
        return {"error": "Subject must have SUBJECT_NEW permission to perform this operation", "new_nonce": new_nonce}, 404

    # Verificar organização associada à sessão
    organization = session.organization
    if not organization:
        return {"error": "Organização associada à sessão não encontrada.",
                "new_nonce": new_nonce}, 404
    print(f"Username: {username}, Name: {name}, Email: {email}")
    # Verificar se o username já existe na organização
    existing_subject = db.session.query(Subject).join(subject_organization).filter(
        subject_organization.c.organization_id == organization.id,
        Subject.username == username
    ).first()
    if existing_subject:
        print(f"Existing subject: {existing_subject}")
        return {"error": "Um usuário com esse username já existe nesta organização.",
                "new_nonce": new_nonce}, 400

    # Criar novo Subject e associá-lo à organização
    new_subject = Subject(
        username=username,
        full_name=name,
        email=email,
        public_key=public_key
    )

    try:
        # Associar o novo Subject à organização usando a tabela de relacionamento
        organization.subjects.append(new_subject)

        # Persistir no banco de dados
        db.session.add(new_subject)
        db.session.commit()

        return {"id": new_subject.id, "message": "Sujeito adicionado com sucesso.",
                "new_nonce": new_nonce}, 201
    except Exception as e:
        db.session.rollback()
        return {"error": f"Ocorreu um erro ao adicionar o sujeito: {str(e)}"}, 500

```

code snippet for the adding a new subject to the organization

Document management

The application provides several features to manage files within the organization scope:

- create (add/upload) a document in a organization;

```

if not has_permission(session_key, "DOC_NEW"):
    print(f"\nnew nonce : {new_nonce}")
    return {"message": "Subject must have DOC_NEW permission to perform this operation",
            "new_nonce": new_nonce}, 404

organization = session.organization
subject = session.subject

# Salva o arquivo de forma segura
filename = secure_filename(file.filename)
filepath = os.path.join(app.config['UPLOAD_FOLDER'], file_name)
file.seek(0) # Move o ponteiro de leitura do arquivo para o início antes de salvar
file.save(filepath)
print(f"Encrypting vars received: {json.loads(encryption_vars)}")

# Carrega a chave pública para a criptografia da chave do arquivo
public_key = load_ec_public_key(private_key_path)
print(f" file key antes de encriptação: {file_encryption_key}")
# Criptografa a chave do arquivo com a chave pública mestre
encrypted_file_key, ephemeral_public_key, iv, tag = encrypt_file_key_with_ec_master(
    file_encryption_key, public_key
)
print(f"Encrypted file key durante criptografia: {encrypted_file_key}")
print(f"IV gerado: {iv}")
print(f"Tag gerado: {tag}")

# Serializa a chave pública efêmera para armazenamento no banco de dados
ephemeral_public_key_serialized = ephemeral_public_key.public_bytes(
    encoding=serialization.Encoding.DER,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Cria um novo documento, incluindo o campo `encryption_vars`
new_document = Document(
    name=file_name,
    create_date=datetime.now(),
    creator=subject.username,
    file_handle=file_handle,
    acl={}, # Inicialmente sem ACL
    organization_id=organization.id,
    encrypted_file_key=encrypted_file_key, # Salva a chave de criptografia criptografada
    iv=iv, # Armazena o IV diretamente
    tag=tag, # Armazena o TAG diretamente
    ephemeral_public_key=ephemeral_public_key_serialized, # Armazena a chave pública efêmera
    encryption_vars=json.loads(encryption_vars) # Converte o JSON de string para dicionário e armazena
)

```

code snippet for the document upload

- list documents from organization;

```
@staticmethod
def get_documents_by_session_key(session_key, username=None, date_str=None, filter_type='all'):
    # Encontre a sessão com base na session_key
    session = check_session(session_key)
    if not session:
        return {"error": "Session not found"}, 404

    organization = session.organization

    # Crie a consulta para obter documentos
    query = Document.query.filter_by(organization_id=organization.id)

    if username:
        subject = Subject.query.filter_by(username=username).first()
        if subject:
            query = query.filter_by(creator=subject.username) # Use o username em vez do id

    if date_str:
        # Converta a data de string para objeto datetime
        from datetime import datetime
        date_obj = datetime.strptime(date_str, '%d-%m-%Y')

        if filter_type == 'more_recent':
            query = query.filter(Document.create_date > date_obj)
        elif filter_type == 'older':
            query = query.filter(Document.create_date < date_obj)
        elif filter_type == 'equal':
            query = query.filter(Document.create_date == date_obj)

    documents = query.all()

    return [doc.to_dict() for doc in documents] # Suponha que você tenha um método to_dict em Document
```

- get information (metadata) of the document;
 - The metadata (including the encryption metadata) is automatically stored in a .json file, ready to be used in the decryption command, it obviously will require the encrypted document

```
# Busca o documento na organização especificada
document = Document.query.filter_by(
    organization_id=organization.id, name=document_name
).first()

if not document:
    return jsonify({"error": "Documento não encontrado na organização"}), 404

# Lê o arquivo criptografado
file_path = document.file_handle

# Recupera a chave criptografada do arquivo
print(f"Encrypted file key recuperado para descriptografia: {document.encrypted_file_key}")
encrypted_file_key = document.encrypted_file_key
iv = document.iv # Certifique-se de que o iv está sendo obtido corretamente
tag = document.tag # Certifique-se de que o tag está sendo obtido corretamente
ephemeral_public_key = document.ephemeral_public_key # Obtém a chave pública efêmera armazenada
print(f"IV recuperado: {iv}")
print(f"Tag recuperado: {tag}")

# Verifica se os dados de criptografia estão presentes
if not encrypted_file_key or not iv or not tag or not ephemeral_public_key:
    return {'error': 'Cryptography data not found for this document'}, 404

decrypted_file_key = decrypt_file_key_with_ec_master(encrypted_file_key, iv, tag, ephemeral_public_key)
print(f"\n file key recuperada da encriptação : {decrypted_file_key}")

# Retornar metadados do documento e a chave criptografada
metadata = {
    "document_id": document.id,
    "document_name": document.name,
    "create_date": document.create_date,
    "creator": document.creator,
    "organization_id": document.organization_id,
    "file_handle": document.file_handle,
    "file_key": decrypted_file_key.decode('utf-8'),
    "encryption_vars": json.dumps(document.encryption_vars)
}

# Certifique-se de que todos os dados são serializáveis
# Convertendo valores como datetime para string, por exemplo
metadata["create_date"] = metadata["create_date"].isoformat() if isinstance(metadata["create_date"],
                                                                              datetime) else metadata[
    "create_date"]
```

code snippet for the retrieval of document's metadata

- download the document, which requires the file handle of document that is retrieved when you get the document metadata;

```
def download_document(file_handle):
    db.session.commit()'''
    # Busca o documento no banco de dados
    document = Document.query.filter_by(file_handle=file_handle).first()
    print(f"\nfile fetched : {document.name}")
    if not document:
        return {'error': 'File not found in database'}, 404

    # Recupera a chave criptografada e os dados de criptografia
    print(f"Encrypted file key recuperado para descriptografia: {document.encrypted_file_key}")
    encrypted_file_key = document.encrypted_file_key
    iv = document.iv # Certifique-se de que o iv está sendo obtido corretamente
    tag = document.tag # Certifique-se de que o tag está sendo obtido corretamente
    ephemeral_public_key = document.ephemeral_public_key # Obtém a chave pública
    print(f"IV recuperado: {iv}")
    print(f"Tag recuperado: {tag}")

    # Verifica se os dados de criptografia estão presentes
    if not encrypted_file_key or not iv or not tag or not ephemeral_public_key:
        return {'error': 'Cryptography data not found for this document'}, 404

    # Descriptografa a chave do arquivo usando a função de descriptografia
    decrypted_file_key = decrypt_file_key_with_ec_master(encrypted_file_key,
    print(f"\n file key recupada da encriptação : {decrypted_file_key}")
    # Define o caminho do arquivo usando o file_handle
    file_path = f"./api/uploads/{document.name}"

    # Verifica se o arquivo existe no caminho
    if not os.path.exists(file_path):
        return {'error': 'File not found on server'}, 404

    # Abre o arquivo em modo binário e o retorna na resposta
    with open(file_path, 'rb') as file:
        file_data = file.read()

    # Retorna tanto o conteúdo do arquivo quanto a chave descriptada
    return {
        # 'file_key': decrypted_file_key.decode('utf-8'),
        'file_data': file_data, # Dados binários do arquivo
        'file_name': document.name # Nome do arquivo
    }, 200 # Retorna uma tupla com resposta e código de status
```

code snippet of the document's download

- delete the document metadata.

```

ect-108130_105925 / delivery2 / repo-app / api / controllers.py
me 1414 lines (1136 loc) · 60.1 KB Raw Copy Download Edit
class SessionController:
    def delete_document_from_organization(session_key, nonce, document_name):
        "new_nonce": new_nonce}, 404

    # Verificar se os dados de criptografia estão presentes
    encrypted_file_key = document.encrypted_file_key
    iv = document.iv
    tag = document.tag
    ephemeral_public_key = document.ephemeral_public_key

    if not encrypted_file_key or not iv or not tag or not ephemeral_public_key:
        return {'error': 'Cryptography data not found for this document',
                "new_nonce": new_nonce}, 404

    # Descriptografar a chave do arquivo
    try:
        decrypted_file_key = decrypt_file_key_with_ec_master(encrypted_file_key, iv, tag, ephemeral_public_key)
    except Exception as e:
        return {'error': f'Failed to decrypt file key: {str(e)}'}, 500

    # Limpar os dados do documento no banco de dados
    try:
        file_handle = document.file_handle
        document.file_handle = None
        document.encrypted_file_key = None
        encryption_metadata = document.encryption_vars
        document.encryption_vars = None
        document.deleter = subject.username # Registrar o deleter
        db.session.commit()

    return {
        "success": True,
        "message": "Document content deleted successfully",
        "document name": document.name,
        "file_key": decrypted_file_key.hex(),
        "file_handle": file_handle,
        "encryption_metadata": encryption_metadata,
        "new_nonce": new_nonce
    }, 200

```

code snippet for the removal of the document metadata

Authentication, Re-authentication and session token

The authentication process is performed by sending a request where in the payload, is send a signed nonce, randomly generated identifier from the client side, signed with the private key in the subject credentials file, then in the server-side, it verifies the signature of the nonce with the subject public (previously stored) to verify the authenticity of the subject(as a user) to perform the authentication, this allows the authentication to rely not with a password directly, but with a key pair verification that works smoothly well as long as the credentials are shared properly between client-server

```
@staticmethod
def create_session():
    data = request.json

    # Extract encrypted key info
    encrypted_key_info = request.headers.get("X-Encrypted-Key-Info")
    if not encrypted_key_info:
        return jsonify({"error": "Encrypted key info is missing"}), 400
    key_info = json.loads(encrypted_key_info)
    private_key_path = "private_key.pem"

    # Decrypt the ChaCha20 key and nonce
    encrypted_key = binascii.unhexlify(key_info["key"])
    encrypted_nonce = binascii.unhexlify(key_info["nonce"])
    chacha_key = decrypt_with_private_key(private_key_path, encrypted_key)
    chacha_nonce = decrypt_with_private_key(private_key_path, encrypted_nonce)

    # Decrypt the payload
    encrypted_payload = request.json.get("encrypted_payload")
    if not encrypted_payload:
        return jsonify({"error": "Encrypted payload is missing"}), 400
    encrypted_payload = binascii.unhexlify(encrypted_payload)
    decrypted_payload = decrypt_with_chacha20(chacha_key, chacha_nonce, encrypted_payload)
    data = json.loads(decrypted_payload)

    # Extract signed_nonce and nonce from the payload
    signed_nonce = binascii.unhexlify(data.get("signed_nonce", ""))
    nonce = data.get("nonce", "")
    if not signed_nonce or not nonce:
        return jsonify({"error": "Missing signed_nonce or nonce in the payload"}), 400

    # Fetch the subject by username
    subject = Subject.query.filter_by(username=data.get("username")).first()
    if not subject:
        return jsonify({"error": "Subject not found"}), 404
```

```

subject = Subject.query.filter_by(username=data.get("username")).first()
if not subject:
    return jsonify({"error": "Subject not found"}), 404

# Verify the signature
try:
    public_key_pem = subject.public_key.encode()
    public_key = load_pem_public_key(public_key_pem, backend=default_backend())

    # Verify the signature
    public_key.verify(
        signed_nonce, # Signature
        nonce.encode(), # Original data
        ec.ECDSA(hashes.SHA256()) # Same algorithm used to sign
    )
except Exception as e:
    return jsonify({"error": f"Signature validation failed: {str(e)}"}), 400

# Check if an AuthenticationID with the same nonce exists
existing_auth_id = AuthenticationID.query.filter_by(nonce=nonce).join(Subject).filter(
    Subject.username == subject.username).first()
if existing_auth_id:
    return jsonify({"error": "Nonce already exists for this user"}), 400

new_auth_id = AuthenticationID(nonce=nonce, subject=subject)
db.session.add(new_auth_id)
db.session.commit()

# Generate JWT as session_key
created_at = datetime.now(timezone.utc)
expiration_time = created_at + timedelta(minutes=30) # Token valid for 15 minutes
payload = {
    "session_id": new_auth_id.id, # Unique session identifier
    "organization_name": data.get("organization_name"), # Organization name
    "subject_username": subject.username, # Username of the subject
    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

```

```

    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")

# Obter a organização associada à sessão
organization_name = data.get("organization_name")
organization = Organization.query.filter_by(name=organization_name).first()
if not organization:
    return jsonify({"error": "Organization not found"}), 404

# Antes de criar a nova sessão, deletar as sessões anteriores do mesmo subject na mesma organização
existing_sessions = Session.query.filter_by(subject_id=subject.id, organization_id=organization.id).all()
for session in existing_sessions:
    db.session.delete(session)
db.session.commit()

# Save the session to the database
new_session = Session(
    session_key=session_key, # Store the JWT here
    password=data.get("password"),
    credentials=data.get("credentials"),
    organization_id=organization.id,
    subject=subject,
    created_at=created_at # Store the creation time for server-side expiration
)

db.session.add(new_session)
db.session.commit()

```

code snippets for the session creation

When authenticating in the repository, by creating a session, the session context is stored in the file that the user provides as an argument of the *rep_create_session* command. This session context contains, among all relevant information regarding the current subject session active, the session token, that is used in the **authenticated** and **authorized** commands

After 30 minutes from the previous creation of session, the current session becomes expired and is deleted from the DB of the server-side application, making necessary to the client to re-authenticate, performed by the same command

To control the session of a subject, whenever the subject authenticates or re-authenticates, it creates a session token using JWT to control the access of an authenticated subject to any command-endpoint that needs a token validation, the two screenshots below provide the methods used to validate the session token and identify the subject's session related with.


```
def is_session_valid(session):
    expiration_time = timedelta(seconds=30*60) # 15 minutes

    now = datetime.now(timezone.utc)

    if session.created_at.tzinfo is None:
        session_created_at = session.created_at.replace(tzinfo=timezone.utc)
    else:
        session_created_at = session.created_at

    return now - session_created_at < expiration_time
```

```
def check_session(session_key):
    try:
        # Decode and validate the JWT signature
        payload = jwt.decode(
            session_key,
            SessionController.SECRET_KEY,
            algorithms=["HS256"]
        )
    except jwt.ExpiredSignatureError:
        # Remove a sessão se o token JWT estiver expirado
        session = Session.query.filter_by(session_key=session_key).first()
        if session:
            db.session.delete(session)
            db.session.commit()
            return None, "Session token has expired."
    except jwt.InvalidTokenError as e:
        return None, f"Invalid session token: {str(e)}"

    # Busca a sessão no banco usando a chave de sessão descryptografada
    session = Session.query.filter_by(session_key=session_key).first()
    if not session:
        return None # Se não encontrar a sessão, retorna None

    # Verificar se a sessão é válida
    is_valid = is_session_valid(session)
    if not is_valid:
        # Se a sessão estiver expirada, removê-la do banco de dados
        db.session.delete(session)
        db.session.commit()
        return None # Sessão inválida ou expirada

    # A sessão foi encontrada e é válida
    return session
```

The payload of the token receives a set of information to keep the uniqueness of the token to the subject, such as the organization name, the subject username, the date-time of the token creation, a session ID and jti generated with uuid4, that provides a 128 bits (which 122 are randomly generated) unique universal identifier.

Then the payload is encoded using the HS256 algorithm and a secret key that only the server has access of and the newly generated token is returned as a response to the client to be used in future authenticated operations

```
# Generate JWT as session_key
created_at = datetime.now(timezone.utc)
expiration_time = created_at + timedelta(minutes=30) # Token valid for 30 minutes
payload = {
    "session_id": new_auth_id.id, # Unique session identifier
    "organization_name": data.get("organization_name"), # Organization name
    "subject_username": subject.username, # Username of the subject
    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")
```

code snippet for the session creation with 30 minutes of lifespan

Document encryption/decryption management

When a subject wants to upload a new document to the organization, the command line client application assumes that the document is 'raw', i.e. not encrypted yet, then it performs a encryption over the document using a symmetric encryption, by default it was used ChaCha20, and sends the encryption metadata(the algorithm, the nonce and key used in the ChaCha20 encryption), within the encrypted document as encrypted vars to the server side.

In the server-side, the flask app encrypts the file key with ECC encryption, using the master key pair that only the server have access to the private key to decrypt the same file, then it proceeds to store the encrypted document in the local file system and stores the encryption vars, encrypted file key and the document's metadata in the DB

As for retrieving the document, the reverse process is performed, firstly the server-side retrieve the encryption vars and decrypts the file key stored in the DB, using the private master key, and returns it, this occurs when its performs the read document metadata command, then after downloading the document with the file handle, decrypts de document, using the encryption metadata stored in a file when performing the *rep_get_doc_metadata* command, with *rep_decrypt_file* command.

```

# Carrega e criptografa o arquivo
with open(data['file'], 'rb') as file:
    file_data = file.read()

# Usa ChaCha20 para criptografar
encrypted_file_data, chacha_key, nonce = encrypt_file_with_chacha20(file_data)
file_handle = hashlib.sha256(encrypted_file_data).hexdigest()

# Cria o dicionário para 'encryption_vars'
encryption_vars = {
    'nonce': nonce.hex(),
    'alg': "ChaCha20"
}

# Define os cabeçalhos para enviar a session_key
nonce = session_data["session_context"]["nonce"].encode('utf-8')
headers = {
    "X-Session-Key": session_key,
    "X-Nonce": nonce
}

# Faz a requisição POST para enviar o documento criptografado
response = requests.post(
    url,
    files={'file': encrypted_file_data},
    data={
        'file_name': data['document_name'], # Nome do documento
        'file_handle': file_handle,
        'file_encryption_key': chacha_key.hex(),
        'encryption_vars': json.dumps(encryption_vars) # Converte para JSON string
    },
    headers=headers # Inclui os cabeçalhos com a session_key
)

```

code snippet for the document upload and encryption on the client side

```

organization = session.organization
subject = session.subject

# Salva o arquivo de forma segura
filename = secure_filename(file.filename)
filepath = os.path.join(app.config['UPLOAD_FOLDER'], file_name)
file.seek(0) # Move o ponteiro de leitura do arquivo para o início antes de salvar
file.save(filepath)
print(f"Encrypting vars received: {json.loads(encryption_vars)}")

# Carrega a chave pública para a criptografia da chave do arquivo
public_key = load_ec_public_key(private_key_path)
print(f" file key antes de encriptação: {file_encryption_key}")
# Criptografa a chave do arquivo com a chave pública mestre
encrypted_file_key, ephemeral_public_key, iv, tag = encrypt_file_key_with_ec_master(
    file_encryption_key, public_key
)
print(f"Encrypted file key durante criptografia: {encrypted_file_key}")
print(f"IV gerado: {iv}")
print(f"Tag gerado: {tag}")

# Serializa a chave pública efêmera para armazenamento no banco de dados
ephemeral_public_key_serialized = ephemeral_public_key.public_bytes(
    encoding=serialization.Encoding.DER,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Cria um novo documento, incluindo o campo `encryption_vars`
new_document = Document(
    # document_handle=file_name,
    name=file_name,
    create_date=datetime.now(),
    creator=subject.username,
    file_handle=file_handle,
    acl={},
    organization_id=organization.id,
    encrypted_file_key=encrypted_file_key, # Salva a chave de criptografia criptografada
    iv=iv, # Armazena o IV diretamente
    tag=tag, # Armazena o TAG diretamente
    ephemeral_public_key=ephemeral_public_key_serialized, # Armazena a chave pública efêmera
    encryption_vars=json.loads(encryption_vars) # Converte o JSON de string para dicionário e armazena
)

# Adiciona e salva o documento no banco de dados
db.session.add(new_document)
db.session.commit()

```

code snippet for the document upload and encryption on the server side

```

document = Document.query.filter_by(file_handle=file_handle).first()
print(f"\nfile fetched : {document.name}")
if not document:
    return {'error': 'File not found in database'}, 404

# Recupera a chave criptografada e os dados de criptografia
print(f"Encrypted file key recuperado para descriptografia: {document.encrypted_file_key}")
encrypted_file_key = document.encrypted_file_key
iv = document.iv # Certifique-se de que o iv está sendo obtido corretamente
tag = document.tag # Certifique-se de que o tag está sendo obtido corretamente
ephemeral_public_key = document.ephemeral_public_key # Obtém a chave pública efêmera armazenada
print(f"IV recuperado: {iv}")
print(f"Tag recuperado: {tag}")

# Verifica se os dados de criptografia estão presentes
if not encrypted_file_key or not iv or not tag or not ephemeral_public_key:
    return {'error': 'Cryptography data not found for this document'}, 404

# Descriptografa a chave do arquivo usando a função de descriptografia
decrypted_file_key = decrypt_file_key_with_ec_master(encrypted_file_key, iv, tag, ephemeral_public_key)
print(f"\n file key recupada da encryptação : {decrypted_file_key}")
# Define o caminho do arquivo usando o file_handle
file_path = f"./api/uploads/{document.name}"

# Verifica se o arquivo existe no caminho
if not os.path.exists(file_path):
    return {'error': 'File not found on server'}, 404

# Abre o arquivo em modo binário e o retorna na resposta
with open(file_path, 'rb') as file:
    file_data = file.read()

# Retorna tanto o conteúdo do arquivo quanto a chave descriptada
return {
    # 'file_key': decrypted_file_key.decode('utf-8'),
    'file_data': file_data, # Dados binários do arquivo
    'file_name': document.name # Nome do arquivo
}, 200 # Retorna uma tupla com resposta e código de status

```

code snippet for the document download server side

```

def decrypt_file(encrypted_file, encryption_metadata_file):
    """
    Descriptografa um arquivo criptografado usando os metadados fornecidos.

    :param encrypted_file: Caminho para o arquivo criptografado.
    :param encryption_metadata_file: Caminho para o arquivo JSON contendo os metadados de criptografia.
    :return: O caminho do arquivo descriptografado.
    """
    try:
        # Ler o arquivo de metadados
        with open(encryption_metadata_file, 'r') as meta_file:
            metadata = json.load(meta_file)

        # Obter a chave e os parâmetros do algoritmo dos metadados
        file_key = bytes.fromhex(metadata.get("file_key"))
        encryption_vars = json.loads(metadata.get("encryption_vars"))
        alg = encryption_vars.get("alg")

        if not file_key or not alg:
            raise ValueError("Metadados incompletos: 'file_key' ou 'alg' ausentes.")

        # Configurar o algoritmo de descriptografia com base em `alg`
        if alg == "ChaCha20":
            nonce = bytes.fromhex(encryption_vars.get("nonce"))
            if not nonce:
                raise ValueError("Metadados incompletos: 'nonce' ausente para ChaCha20.")
            algorithm = algorithms.ChaCha20(file_key, nonce)
            cipher = Cipher(algorithm, mode=None)

        elif alg == "AES-GCM":
            nonce = bytes.fromhex(encryption_vars.get("nonce"))
            tag = bytes.fromhex(encryption_vars.get("tag"))
            if not nonce or not tag:
                raise ValueError("Metadados incompletos: 'nonce' ou 'tag' ausentes para AES-GCM.")
            algorithm = algorithms.AES(file_key)
            cipher = Cipher(algorithm, mode=modes.GCM(nonce, tag))

        elif alg == "AES-CBC":
            iv = bytes.fromhex(encryption_vars.get("iv"))
            if not iv:
                raise ValueError("Metadados incompletos: 'iv' ausente para AES-CBC.")
            algorithm = algorithms.AES(file_key)
            cipher = Cipher(algorithm, mode=modes.CBC(iv))

        else:
            raise ValueError(f"Algoritmo de criptografia '{alg}' não suportado.")

        # Criar o decodificador
        decryptor = cipher.decryptor()

        # Ler o conteúdo do arquivo criptografado
        with open(encrypted_file, 'rb') as enc_file:
            encrypted_data = enc_file.read()

        # Descriptografar os dados
        decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()
        print(decrypted_data.decode("utf-8"))
        # Definir o caminho de saída para o arquivo descriptografado
        output_file = f"{os.path.splitext(encrypted_file)[0]}_decrypted"
        with open(output_file, 'wb') as out_file:
            out_file.write(decrypted_data)

        print(f"Arquivo descriptografado salvo em: {output_file}")
        return output_file

```

code snippet for the document's decryption (client-side)

Secured communication (partially implemented)

The use of the a key pair, previously generated with ECC, is employed to secure the client-server communication, where the client encrypts the payload and headers(particularly important since its where the session token is sended in the request) of the request using the public key(that is obviously is publicly distributed for the clients) and the server receives the encrypted payload and header and decrypts using the private key(that only the server have access to).

However, due to the lack of time to the development team, there's still some command-endpoints communication in the project that still doesn't have this implemented, most of them are implemented with the secure communication, that's why this is a partially implemented feature

```
ame 1414 lines (1136 loc) · 60.1 KB

def create_session():

    # Extract encrypted key info
    encrypted_key_info = request.headers.get("X-Encrypted-Key-Info")
    if not encrypted_key_info:
        return jsonify({"error": "Encrypted key info is missing"}), 400
    key_info = json.loads(encrypted_key_info)
    private_key_path = "private_key.pem"

    # Decrypt the ChaCha20 key and nonce
    encrypted_key = binascii.unhexlify(key_info["key"])
    encrypted_nonce = binascii.unhexlify(key_info["nonce"])
    chacha_key = decrypt_with_private_key(private_key_path, encrypted_key)
    chacha_nonce = decrypt_with_private_key(private_key_path, encrypted_nonce)

    # Decrypt the payload
    encrypted_payload = request.json.get("encrypted_payload")
    if not encrypted_payload:
        return jsonify({"error": "Encrypted payload is missing"}), 400
    encrypted_payload = binascii.unhexlify(encrypted_payload)
    decrypted_payload = decrypt_with_chacha20(chacha_key, chacha_nonce, encrypted_payload)
    data = json.loads(decrypted_payload)

    # Extract signed_nonce and nonce from the payload
    signed_nonce = binascii.unhexlify(data.get("signed_nonce", ""))
    nonce = data.get("nonce", "")
    if not signed_nonce or not nonce:
        return jsonify({"error": "Missing signed_nonce or nonce in the payload"}), 400

    # Fetch the subject by username
    subject = Subject.query.filter_by(username=data.get("username")).first()
    if not subject:
        return jsonify({"error": "Subject not found"}), 404

    # Verify the signature
    try:
        public_key_pem = subject.public_key.encode()
```



```

# Envia o payload para criar uma sessão
url = f"http://{state['REP_ADDRESS']}/sessions"

key = os.urandom(32)
nonce = os.urandom(16)

encrypted_payload = encrypt_with_chacha20(key, nonce, json.dumps(payload))

# Encrypt ChaCha20 key and nonce with the public key
public_key_path = state['REP_PUB_KEY']
encrypted_key = encrypt_with_public_key(public_key_path, key)
encrypted_nonce = encrypt_with_public_key(public_key_path, nonce)

# Prepare the JSON for the headers

encryption_header = {
    "key": encrypted_key.hex(),
    "nonce": encrypted_nonce.hex()
}
headers = {
    #"X-Nonce": encrypted_nonce_header.hex(),
    "X-Encrypted-Key-Info": json.dumps(encryption_header) # Send JSON as a string in the header
}

response = requests.post(url, json={"encrypted_payload": encrypted_payload.hex()}, headers=headers)

if response.status_code == 201:
    # Obtém a resposta e criptografa a session_key
    response_data = response.json()

```

Code snippet of how the server-side (1st figure) receives the encrypted message(payload and header) from the client (2nd figure)

Nonce control for replay attacks prevention (partially implemented)

Protection over replay attacks with NONCE, a randomly generated identifier for a single operation, that is provided by the server to the client, the very first one is provided when the subject is authenticated, and must be used in the next request made by the client, the server checks the existence of the nonce in the DB, checks if it was already used, perform their operation of the request and returns a new NONCE to be used by the client in the next request that needs it

This approach allows the server to control the flow of each operation of the subject, by being the one who generates, distributes and validates the Nonce to any subject

However, due to the lack of time to the development team, there's still some command-endpoints communication in the project that still doesn't have this implemented, most of them are implemented with replay attack prevention using nonce control, that's why this is a partially implemented feature

```

@staticmethod
def delete_document_from_organization(session_key, nonce, document_name):
    # Obter a sessão associada à session key
    session = check_session(session_key)
    if session is None:
        return {"error": "Sessão inválida ou não encontrada"}, 404

    existing_nonce = Nonce.query.filter_by(nonce=nonce).first()
    if not existing_nonce:
        return jsonify({"error": "Nonce inválido ou não encontrado."}), 400

    if existing_nonce.used:
        return jsonify({"error": "Nonce já utilizado. Replay detectado!"}), 400

    # Marcar o nonce como usado
    existing_nonce.used = True
    db.session.commit()

    # Gerar um novo nonce exclusivo
    new_nonce = ''.join(random.choices(string.ascii_letters + string.digits, k=32))
    while Nonce.query.filter_by(nonce=new_nonce).first():
        new_nonce = ''.join(random.choices(string.ascii_letters + string.digits, k=32))

    # Salvar o novo nonce
    new_nonce_entry = Nonce(nonce=new_nonce, used=False)
    db.session.add(new_nonce_entry)
    db.session.commit()

```

How the server-side controllers check the existence of a Nonce in order to avoid replay attacks

```

# Descriptografar a chave do arquivo
try:
    decrypted_file_key = decrypt_file_key_with_ec_master(encrypted_file_key, iv, tag, ephemeral_public_key)
except Exception as e:
    return {'error': f'Failed to decrypt file key: {str(e)}'}, 500

# Limpar os dados do documento no banco de dados
try:
    file_handle = document.file_handle
    document.file_handle = None
    document.encrypted_file_key = None
    encryption_metadata = document.encryption_vars
    document.encryption_vars = None
    document.deleter = subject.username # Registrar o deleter
    db.session.commit()

    return {
        "success": True,
        "message": "Document content deleted successfully",
        "document name": document.name,
        "file_key": decrypted_file_key.hex(),
        "file_handle": file_handle,
        "encryption_metadata": encryption_metadata,
        "new_nonce": new_nonce
    }, 200
except Exception as e:
    db.session.rollback()
    return {"success": False, "message": f"Failed to update document: {str(e)}"}, 500

```

```

if not has_permission_in_document(session_key, "DOC_DELETE", document_name):
    print(f"\nnew nonce : {new_nonce}")
    return {"success": False,
        "message": "Subject must have DOC_DELETE permission to perform this operation and the Role must be present in the ACL of the document",
        "new_nonce": new_nonce}, 404

# Verificar se os dados de criptografia estão presentes
encrypted_file_key = document.encrypted_file_key
iv = document.iv
tag = document.tag
ephemeral_public_key = document.ephemeral_public_key

if not encrypted_file_key or not iv or not tag or not ephemeral_public_key:
    return {'error': 'Cryptography data not found for this document',
        "new_nonce": new_nonce}, 404

```

The server-side always ensures to return a new usable Nonce to the subject, even when a operation fails

Role-Permission access control

The server-side application performs role-permission check to verify if the subject that request a operation that requires a specific permission have the required permission to perform the operation

Ex : check if the current subject have the DOC_NEW permission when trying to upload a new document to the organization

```
def upload_document_to_organization(session_key, nonce, file_name, file, file_handle, file_encryption_key,
                                   encryption_vars,
                                   private_key_path="master_key.pem.pub"):
    # Verifica a sessão e a organização correspondente
    session = check_session(session_key)
    if session is None:
        return {"error": "Sessão inválida ou não encontrada"}, 404

    existing_nonce = Nonce.query.filter_by(nonce=nonce).first()
    if not existing_nonce:
        return jsonify({"error": "Nonce inválido ou não encontrado."}), 400

    if existing_nonce.used:
        return jsonify({"error": "Nonce já utilizado. Replay detectado!"}), 400

    # Marcar o nonce como usado
    existing_nonce.used = True
    db.session.commit()

    # Gerar um novo nonce exclusivo
    new_nonce = ''.join(random.choices(string.ascii_letters + string.digits, k=32))
    while Nonce.query.filter_by(nonce=new_nonce).first():
        new_nonce = ''.join(random.choices(string.ascii_letters + string.digits, k=32))

    # Salvar o novo nonce
    new_nonce_entry = Nonce(nonce=new_nonce, used=False)
    db.session.add(new_nonce_entry)
    db.session.commit()

    if not has_permission(session_key, "DOC_NEW"):
        print(f"\nnew nonce : {new_nonce}")
        return {"message": "Subject must have DOC_NEW permission to perform this operation",
                "new_nonce": new_nonce}, 404
```

code snippet for the retrieval of the document's upload, where it checks if the subject with the session has at least one with the required permission

It also performed verification on the level of ACLs of a document when performing an operation over a specific document in the organization.

Ex : check if the current subject has the DOC_READ permission when trying to read a document metadata to the organization.

```
@staticmethod
def get_document_metadata(session_key, document_name):
    # Verifica a sessão e a organização correspondente
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    if not has_permission_in_document(session_key, "DOC_READ", document_name):
        return jsonify({"error": "Subject must have DOC_READ permission to perform this operation and the Role must be present in the ACL of the document"}), 404
```

code snippet for the retrieval of the document's metadata, where it checks if the subject with the session has at least one with the required permission and also checks if the role is in the document's ACL

The subject can acquire access to these permissions by assuming a role that has the required permission(s), assuming that the role exists in the organization(and in the ACL of a document if that's the case) and it's not suspended.

```

def assume_role(session_key, role_name):
    # Verificar a sessão
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter o subject associado à sessão
    subject = session.subject
    if not subject:
        return jsonify({"error": "Subject não encontrado para esta sessão"}), 404

    # Buscar a role na organização
    role = Role.query.filter_by(name=role_name, organization_id=session.organization_id).first()
    if not role:
        return jsonify({"error": f"Role '{role_name}' não encontrada na organização"}), 404

    # Verificar se a role está acessível para o subject
    if role not in subject.accessible_roles:
        return jsonify({"error": f"Role '{role_name}' não está acessível para o subject '{subject.username}'"}), 403

    # Verificar se a role já está associada ao subject
    if role in subject.roles:
        return jsonify({"message": f"Role '{role_name}' já está associada ao subject '{subject.username}'"}), 200

    # Associar a role ao subject
    subject.roles.append(role)
    db.session.commit()

```

code snippet for the assume role function

A role in the organization can have multiply (or none) permissions, but it must be made available(give permission to a specific subject) to a subject by a subject that have this permission (usually the manager)

Role-Permission management

The application as a whole provides a set of commands to perform the Role-Permission management to create a role, add/remove a permission to a role, add/remove permission(access) of a role to a given subject, add/remove a permission of a role at the scope of a document's ACL and suspend a role at the scope of the organization.

```
@staticmethod
def add_role(session_key, new_role):
    # Verificar a sessão e obter as informações
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter a organização associada à sessão
    organization = session.organization
    if not organization:
        return jsonify({"error": "Organização não encontrada"}), 404

    if not has_permission(session_key, "ROLE_NEW"):
        return {"error": "Subject must have ROLE_NEW permission to perform this operation"},

    # Verificar se a role já existe na organização
    role = Role.query.filter_by(name=new_role, organization_id=organization.id).first()
    if role:
        return jsonify({"error": "Role já existe na organização"}), 400

    # Criar a nova role
    role = Role(name=new_role, organization_id=organization.id)

    # Adicionar a role à organização (sem associar a nenhum subject ainda)
    db.session.add(role)
    db.session.commit()

    return jsonify({"message": f"Role '{new_role}' criada com sucesso!"}), 200
```

add a new role to the organization

```

@staticmethod
def add_permission_to_role(session_key, role_name, permission_name):
    # Verificar a sessão
    session = check_session(session_key)
    if session is None:
        print("[DEBUG] Sessão inválida ou não encontrada.")
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter a organização associada à sessão
    organization = session.organization
    if not organization:
        print("[DEBUG] Organização não encontrada para a sessão.")
        return jsonify({"error": "Organização não encontrada para esta sessão"}), 404

    print(f"[DEBUG] Organização ID: {organization.id}, Nome: {organization.name}")

    if not has_permission(session_key, "ROLE_MOD"):
        return {"error": "Subject must have ROLE_MOD permission to perform this operation"}, 404

    # Buscar a role na organização
    print(f"[DEBUG] Procurando role com nome '{role_name}' e organização ID '{organization.id}'.")
    role = Role.query.filter_by(name=role_name, organization_id=organization.id).first()
    if not role:
        print(
            f"[DEBUG] Role '{role_name}' não encontrada na organização '{organization.name}' (ID: {organization.id}).")
        return jsonify({"error": f"Role '{role_name}' não encontrada na organização"}), 404

    print(f"[DEBUG] Role encontrada: {role.name} (ID: {role.id})")

    # Buscar a permissão pela tabela de permissões
    permission = Permission.query.filter_by(name=permission_name).first()
    if not permission:
        print(f"[DEBUG] Permissão '{permission_name}' não encontrada.")
        return jsonify({"error": f"Permissão '{permission_name}' não encontrada"}), 404

    print(f"[DEBUG] Permissão encontrada: {permission.name} (ID: {permission.id})")

    # Verificar se a permissão já está associada à role
    if any(rp.permission_id == permission.id for rp in role.permissions):
        print(f"[DEBUG] Permissão '{permission_name}' já está associada à role '{role_name}'.")
        return jsonify({"message": f"Permissão '{permission_name}' já está associada à role '{role_name}'"}), 200

    # Associar a permissão à role
    role_permission = RolePermission(role_id=role.id, permission_id=permission.id)
    db.session.add(role_permission)
    db.session.commit()

    print(f"[DEBUG] Permissão '{permission_name}' associada com sucesso à role '{role_name}'.")
    return jsonify({"message": f"Permissão '{permission_name}' associada com sucesso à role '{role_name}'"}), 200

```

add a permission to role

```

@staticmethod
def add_access_of_role_to_subject(session_key, role_name, username):
    # Verificar a sessão
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter a organização associada à sessão
    organization = session.organization
    if not organization:
        return jsonify({"error": "Organização não encontrada para esta sessão"}), 404

    if not has_permission(session_key, "ROLE_MOD"):
        return {"error": "Subject must have ROLE_MOD permission to perform this operation"}, 404

    # Buscar a role na organização
    role = Role.query.filter_by(name=role_name, organization_id=organization.id).first()
    if not role:
        return jsonify({"error": f"Role '{role_name}' não encontrada na organização"}), 404

    # Buscar o subject pelo username
    subject = Subject.query.filter_by(username=username).first()
    if not subject:
        return jsonify({"error": f"Subject com username '{username}' não encontrado"}), 404

    # Verificar se o subject pertence à organização
    if organization not in subject.organizations:
        return jsonify({"error": f"Subject '{username}' não pertence à organização"}), 400

    # Verificar se a role já está acessível ao subject
    if role in subject.accessible_roles:
        return jsonify({"message": f"Role '{role_name}' já está acessível ao subject '{username}'"}), 200

    # Tornar a role acessível ao subject
    subject.accessible_roles.append(role)
    db.session.commit()

    return jsonify({"message": f"Role '{role_name}' tornou-se acessível ao subject '{username}' com sucesso!"}), 200
@staticmethod

```

give access (permission) of a given role to a subject


```

me 1414 lines (1130 loc) + 60.1 KB
def change_doc_acl(session_key, role_name, permission_name, document_name, operation):

    organization = session.organization
    if not organization:
        return jsonify({"error": "Organização não encontrada para esta sessão"}), 404

    if not has_permission(session_key, "DOC_ACL"):
        return {"error": "Subject must have DOC_ACL permission to perform this operation"}, 404

    # Buscar o documento pelo nome e organização
    document = Document.query.filter_by(name=document_name, organization_id=organization.id).first()
    if not document:
        return jsonify({"error": f"Documento '{document_name}' não encontrado"}), 404

    # Buscar a role pelo nome e organização
    role = Role.query.filter_by(name=role_name, organization_id=organization.id).first()
    if not role:
        return jsonify({"error": f"Role '{role_name}' não encontrada"}), 404

    # Validar a permissão
    valid_permissions = ["DOC_DELETE", "DOC_READ"]
    if permission_name not in valid_permissions:
        return jsonify({"error": f"Permissão '{permission_name}' não é válida. Use {valid_permissions}."}), 400

    # Validar a operação
    if operation not in ["+", "-"]:
        return jsonify({"error": "Operação inválida. Use '+' para adicionar ou '-' para remover"}), 400

    # Inicializar ou obter o ACL do documento
    acl = document.acl or {}

    # Garantir que a permissão existe no ACL
    acl.setdefault(permission_name, [])

    if operation == "+":
        if role_name not in acl[permission_name]:
            acl[permission_name].append(role_name)
    elif operation == "-":
        if role_name in acl[permission_name]:
            acl[permission_name].remove(role_name)

```

```

# validar a operação
if operation not in ["+", "-"]:
    return jsonify({"error": "Operação inválida. Use '+' para adicionar ou '-' para remover"}), 400

# Inicializar ou obter o ACL do documento
acl = document.acl or {}

# Garantir que a permissão existe no ACL
acl.setdefault(permission_name, [])

if operation == "+":
    if role_name not in acl[permission_name]:
        acl[permission_name].append(role_name)
elif operation == "-":
    if role_name in acl[permission_name]:
        acl[permission_name].remove(role_name)

print(f"\n[DEBUG] ACL antes de salvar: {acl}")

# Atualizar o ACL do documento e marcar como modificado
document.acl = acl
flag_modified(document, "acl") # Informar ao SQLAlchemy que o campo foi modificado
db.session.commit()

print(f"[DEBUG] ACL persistido no banco: {document.acl}")
return jsonify({"message": f"ACL atualizado com {document.acl}"}), 200

except Exception as e:
    db.session.rollback()
    return jsonify({"error": f"Erro ao atualizar o ACL: {str(e)}"}), 500

```

modify the ACL of a document

```

@staticmethod
def suspend_role(session_key, role_name):
    # Verificar a sessão
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter a organização associada à sessão
    organization = session.organization
    if not organization:
        return jsonify({"error": "Organização não encontrada para esta sessão"}), 404

    if not has_permission(session_key, "ROLE_DOWN"):
        return {"error": "Subject must have ROLE_DOWN permission to perform this operation"},

    # Buscar a role na organização
    role = Role.query.filter_by(name=role_name, organization_id=organization.id).first()
    if not role:
        return jsonify({"error": f"Role '{role_name}' não encontrada na organização"}), 404

    # Verificar se a role já está suspensa
    if role.is_suspended:
        return jsonify({"message": f"Role '{role_name}' já está suspensa"}), 200

    # Suspender a role
    role.is_suspended = True
    db.session.commit()

    return jsonify({"message": f"Role '{role_name}' suspensa com sucesso!"}), 200

```

suspend a role from being assumed until is reactivated

It's also provided a set of commands to list information of a given role, such as list the permissions of a given role or list the subjects that are using this given role

```
def list_subject_roles(session_key, p_username):
    # Verificar a sessão
    session = check_session(session_key)
    if session is None:
        return jsonify({"error": "Sessão inválida ou não encontrada"}), 404

    # Obter o subject associado à sessão
    session_subject = session.subject
    if not session_subject:
        return jsonify({"error": "Subject não encontrado para esta sessão"}), 404

    # Buscar o subject pelo username
    subject = Subject.query.filter_by(username=p_username).first()
    if not subject:
        return jsonify({"error": f"Subject com username '{p_username}' não encontrado"}), 404

    # Obter as roles associadas ao subject
    user_roles = subject.roles # Relacionamento direto do modelo Subject
    if not user_roles:
        return jsonify({"message": "Nenhuma role associada ao subject"}), 200

    # Retornar a lista de nomes das roles associadas
    return jsonify([role.name for role in user_roles]), 200
```

code snippet of the listing of roles for a designated user

Context of the application for security analysis

For contextualization, the application developed by the team generates and controls user sessions (namely subjects) by JWT tokens. For these tokens, the management is performed exclusively by the application (the server) and the clients only stores the session context locally and use the session token to perform operations over api endpoints that require session token

Security Evaluation (V3 - Session Management)

3.1 Fundamental Session Management Security

3.1.1 Verify the application never reveals session tokens in URL parameters.

Given that all endpoints requests the session token as a header component, so its not revealed in the URL, which leads no space to leak the session token if intercepting the URL in a request, given the example below, which ensures that the this control is properly fulfilled

```
Blame 912 lines (740 loc) · 41.7 KB

def add_subject_route():
    # Obtém os cabeçalhos criptografados
    encrypted_session_key = request.headers.get("X-Session-Key")
    encrypted_key = request.headers.get("X-Encrypted-Key")
    encrypted_nonce = request.headers.get("X-Encrypted-Nonce")
    nonce_header = request.headers.get("X-Nonce")

    if not all([encrypted_session_key, encrypted_key, encrypted_nonce, nonce_header]):
        logger.warning("Missing required headers for adding subject.")
        return jsonify({"error": "Todos os cabeçalhos são obrigatórios"}), 400

    # Obtém o payload criptografado
    encrypted_payload = request.get_data()

    # Descriptografar chave e nonce ChaCha20 com a chave privada
    private_key_path = "private_key.pem" # Caminho para a chave privada
    chacha_key = decrypt_with_private_key(private_key_path, binascii.unhexlify(encrypted_key))
    chacha_nonce = decrypt_with_private_key(private_key_path, binascii.unhexlify(encrypted_nonce))

    # Descriptografar session_key e payload com ChaCha20
    session_key = decrypt_with_chacha20(chacha_key, chacha_nonce, binascii.unhexlify(encrypted_session_key)).decode('utf-8')
    payload_json = decrypt_with_chacha20(chacha_key, chacha_nonce, binascii.unhexlify(encrypted_payload)).decode('utf-8')
    data = json.loads(payload_json)

    # Extrai os campos do payload descriptografado
    username = data.get("username")
    name = data.get("name")
    email = data.get("email")
    public_key = data.get("public_key")

    if not all([session_key, username, name, email, public_key]):
        logger.warning("Missing required fields for adding subject.")
        print(payload_json)
        return jsonify(
            {"error": "Todos os campos são obrigatórios: session_key, username, name, email"}), 400

    # Chama o controlador para adicionar o sujeito
    logger.info(f"Adding subject to organization with session key: {session_key}")
    result, status_code = SessionController.add_subject_to_organization(
        session_key, nonce_header, username, name, email, public_key
    )
    return jsonify(result), status_code
except Exception as e:
    logger.error(f"Erro ao processar a requisição: {str(e)}")
    return jsonify({"error": f"Erro interno: {str(e)}"}), 500
```

Right below are all the endpoints mapping that were used in the application, where it can be noticed that none of them use the session token in the URL

```
@main_bp.route('/sessions/assume_role', methods=['POST'])
```

```
@main_bp.route('/sessions/roles/add', methods=['POST'])
```

```
@main_bp.route('/sessions/roles', methods=['GET'])
```

```
@main_bp.route('/sessions/release_role', methods=['POST'])
```

```
@main_bp.route('/sessions/list_role_subjects', methods=['GET'])
```

```
@main_bp.route('/sessions/list_subject_roles', methods=['GET'])
```

```
@main_bp.route('/sessions/list_role_permissions', methods=['GET'])
```

```
@main_bp.route('/sessions/list_permission_roles', methods=['GET'])
```

```
def list_permission_roles_route():
```

```
@main_bp.route('/sessions/subjects', methods=['GET'])
```

```
@main_bp.route('/organization/document/acl', methods=['POST'])
```

```
def check_document_acl():
```

```
@main_bp.route('/organization/roles/add_permission', methods=['POST'])
```

```
@main_bp.route('/organization/roles/remove_permission', methods=['POST'])
```

```
@main_bp.route('/organization/roles/suspend_role', methods=['POST'])
```

```
@main_bp.route('/organization/roles/reactivate_role', methods=['POST'])
```

```
@main_bp.route('/organization/roles/add_access', methods=['POST'])
```

```
@main_bp.route('/organization/roles/remove_access', methods=['POST'])
```

```
@main_bp.route('/sessions/documents', methods=['GET'])
```

```
@main_bp.route('/add_subject', methods=['POST'])
```

```
@main_bp.route('/add_document', methods=['POST'])
```

```
@main_bp.route('/document/metadata', methods=['GET'])
```

```
@main_bp.route('/delete_document/<string:document_name>', methods=['DELETE'])
```

3.2 Session Binding

Overall security consideration for this session binding's controls :

Additionally to the security and technical evaluation and considerations of the controls below, it's important to clarify that in order to achieve fully security when it comes to binding and distributing session context, and their token, a proper protocol security communication must be applied between the client-server exchange of data, being the most used and accepted in the security ecosystem, a TLS protocol or a similar one.

3.2.1 Verify the application generates a new session token on user authentication.

Given that, whenever a user creates a new session, after all the proper verifications in the authentication operation, the server always ensures that the new session being created is unique for that user, by deleting all previous sessions alongside with their session tokens for the user

Invalidate Previous Sessions: The server deletes all prior sessions associated with the user, including their corresponding session tokens, ensuring no old tokens remain active.

```
# Antes de criar a nova sessão, deletar as sessões anteriores do mesmo subject na mesma organização
existing_sessions = Session.query.filter_by(subject_id=subject.id, organization_id=organization.id).all()
for session in existing_sessions:
    db.session.delete(session)
db.session.commit()
```

This allows the application to generate a new session token (session key) without worrying with this new session token already being used in previous session or taking risk of allowing old session tokens to be used:

```
existing_auth_id = AuthenticationID.query.filter_by(nonce=nonce).join(Subject).filter(
    Subject.username == subject.username).first()
if existing_auth_id:
    return jsonify({"error": "Nonce already exists for this user"}), 400

new_auth_id = AuthenticationID(nonce=nonce, subject=subject)
db.session.add(new_auth_id)
db.session.commit()
```

The newly generated token is also generated, in the payload with data that ensures his uniqueness such as the date-time of creation(that is used in the expiration

verification), the sequential generated authentication ID and a 128 randomly generated component with UUID4

```
# Generate JWT as session_key
created_at = datetime.now(timezone.utc)
expiration_time = created_at + timedelta(minutes=30) # Token valid for 15 minutes
payload = {
    "session_id": new_auth_id.id, # Unique session identifier
    "organization_name": data.get("organization_name"), # Organization name
    "subject_username": subject.username, # Username of the subject
    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")
```

```
new_session = Session(
    session_key=session_key, # Store the JWT here
    password=data.get("password"),
    credentials=data.get("credentials"),
    organization_id=organization.id,
    subject=subject,
    created_at=created_at # Store the creation time for server-side expiration
)

db.session.add(new_session)
db.session.commit()
```

The behavior of the new token can be verified by copy-pasting the same token of the previous session (obviously after creating a new session) into the current session_token field at the session_file and trying to execute any command that requires a session token, it should receive a message like the below :

```
(venv) [tk@bismarck cmd-client]$ bash rep_create_session "Org4" "anon3" "password" "credentials" "session_file"
0
(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
{'error': 'Sessão inválida ou não encontrada'}
```

3.2.2 Verify that session tokens possess at least 64 bits of entropy

[Entropy](#) measures the randomness or unpredictability of a system. Higher entropy means more randomness.

High entropy ensures session tokens are resistant to brute-force attacks, making it computationally infeasible to guess or forge a valid token.

In the developed system, the generation of the JWT session token uses, in the payload, a JTI (one of the JWT token claims) with a unique identifier, that is generated using the

uuid.uuid4()

method,

```
# Generate JWT as session_key
created_at = datetime.now(timezone.utc)
expiration_time = created_at + timedelta(minutes=30) # Token valid for 15 minutes
payload = {
    "session_id": new_auth_id.id, # Unique session identifier
    "organization_name": data.get("organization_name"), # Organization name
    "subject_username": subject.username, # Username of the subject
    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")
```

that

generates, according to the [documentation](#), out of the total of the 128 bits generated from the uuid4 method, 122 of randomly generated bits

4.4. Algorithms for Creating a UUID from Truly Random or Pseudo-Random Numbers

The version 4 UUID is meant for generating UUIDs from truly-random or pseudo-random numbers.

The algorithm is as follows:

- o Set the two most significant bits (bits 6 and 7) of the clock_seq_hi_and_reserved to zero and one, respectively.
- o Set the four most significant bits (bits 12 through 15) of the time_hi_and_version field to the 4-bit version number from [Section 4.1.3](#).
- o Set all the other bits to randomly (or pseudo-randomly) chosen values.

This ensures, a number higher than 64 bits, 122 on total, are randomly generated, which means that the JWT session token have at least 122 bits of entropy

3.2.3 Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML 5 session storage

Although this project uses a command-line client instead of a browser-based client, the principle of securely storing session tokens remains applicable. Tokens must be stored securely to prevent unauthorized access and ensure confidentiality, integrity, and availability.

Current Implementation and Issues:

1 - Storage in Plain Text Files:

Session tokens are stored in a plain-text session context file, which introduces a significant risk of token exposure to anyone with access to the file system.

This approach fails to meet the requirements of secure storage, as it does not provide encryption or access control, which compromise the security of confidentiality of the session context for the application as the content of the file is saved as simple text, in the absence of the access control over the file, the plain text allows anyone to simply retrieve the session token.

```
if response.status_code == 201:
    # Obtém a resposta e criptografa a session_key
    response_data = response.json()
    #session_key = response_data["session_context"]["session_token"]

    ### Nao encryptar aq :

    '''
    # Usa o caminho para a chave pública
    public_key_path = "../public_key.pem"
    encrypted_session_key = encrypt_session_key(session_key, public_key_path)

    # Substitui a chave de sessão pela versão criptografada
    response_data["session_context"]["session_key"] = encrypted_session_key'''

    # Salva a resposta atualizada no arquivo de sessão
    with open(session_file, 'w') as file:
        json.dump(response_data, file, indent=4)
    return 0
else:
    # Caso falhe na criação da sessão
    print(f"Failed to create session: {response.json()} {response.status_code}")
    return 1
```

On top of that, The session context file is stored in the same directory as the `client.py` application, which is a poor practice as it increases the risk of accidental exposure or unauthorized access, given the fact that it is a location that is easy and simple to identify by an attacker (once knowing how the session file is created) who have the access of the system or even in contexts of a codebase leak (for example).

Another potential violation of security is the Unix file system access control, the file is created without customized access control, one that would properly limit the read/modify access of the file to the owner, resulting in group and other users having read access. This violates the confidentiality of the session token, as it could be read by unauthorized users that have access to the file system.

```
[~/U/s/e/s/d/r/cmd-client] : ls -l session_file
-rw-r--r-- 1 tk tk 539 dez 19 22:53 session_file
```

With all these analyses, regarding the storing and access of the session token in the command line client, it's safe to assume that it does not fulfill correctly security requisite of the control

possible solutions to mitigate these problems:

1 - Secure Token Storage:

- Implement **encrypted storage** for the session token. One approach is to use the `python-keyring` library to store the session token securely as a credential.
- Alternatively, and a way more secure and recommended in the security ecosystem, use a system-level secure storage solution (e.g., macOS Keychain, Windows Credential Manager, or Linux secret stores).

2 - Secure File Permissions:

- If token storage in a file is unavoidable, ensure file access control is restricted. Use `chmod` or equivalent commands to set file permissions to `600` (read/write access only for the owner).
- Programmatically enforce secure file permissions during file creation.

3 - Move Storage Location:

- If possible, relocate the session context file to a secure, system-protected directory (e.g., `$HOME/.appname/session_context` or equivalent).
- This path/location should be protected (as a secured and controlled secret whiting any secret manager) or at least be directory/file-system partition encrypted

3.2.4 Verify that session tokens are generated using approved cryptographic algorithms

For the token that is generated in the authentication, the HS256 is used to hash the payload of my newly generated JWT token

```
session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")
```

HS256 means HMAC with SHA-256:

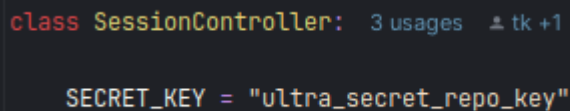
HMAC (Hash-based Message Authentication Code) is a cryptographic construct that combines a hash function (SHA-256, in this case) with a secret key that is secured in the organization server.

SHA-256 is a [NIST-approved](#) algorithm that is widely recognized as secure for authentication.

While the HS256 algorithm itself is considered secure, the current implementation introduces several weaknesses related to key management, which undermine the security guarantees of the cryptographic process.

Issues Identified:

Key Disclosure:



```
class SessionController: 3 usages  ± tk +1  
  
    SECRET_KEY = "ultra_secret_repo_key"
```

The HMAC key is hard-coded into the application code. This introduces a significant risk:

- If the codebase is leaked (e.g., through a repository compromise), the key could be exposed.
- Reverse engineering of the application could reveal the key.
 - All these disclosure problems increase the chances of an attacker perform a operation that bypass the integrity verification by simply forging his own token with the leaked key

Hard-coded keys fail to comply with secure key storage practices required for L3

Insufficient Key Length:

- Keys shorter than 256 bits are generally considered weak, especially when used in modern cryptographic algorithms like HMAC with SHA-256 or higher. A short key increases the risk of brute-force attacks, where an attacker can attempt many possible keys until the correct one is found.

Lack of Key Rotation:

- No mechanism is implemented to periodically rotate(change dynamically) the encoding key.
- Without rotation, a compromised key(that, for instance, was leaked to an attacker) could remain valid indefinitely, increasing the risk of long-term exploitation.

Lack of Centralized Key Management

- Hardcoding a key is, [by definition](#), a security problem, as it bypasses best practices for secure key management. Keys should be stored in secure environments, such as:
 - Environment variables.
 - Secret management tools (e.g., AWS Secrets Manager, Azure Key Vault, HashiCorp Vault).

Recommendations for Mitigation:

1. **Use a Secure Key Storage Mechanism:**
 - Implement a secret management solution (e.g., AWS Secrets Manager, Azure Key Vault, or HashiCorp Vault) to securely store the HMAC key.
 - Avoid hardcoding keys in application code or configuration files.
2. **Upgrade Key Length:**
 - Use a cryptographic key of at least 256 bits in length to meet modern security standards.
3. **Implement Key Rotation:**
 - Develop a mechanism to rotate keys periodically. For JWTs, implement a way to verify tokens using both the old and new keys during the transition period.

Final consideration : Even though, the HS256 is approved by the NIST, it's worth to mention that other algorithms should also be considered, since they are more reliable in terms of security and attack prevention, such as RS256 or ES256

3.3 Session Termination

3.3.1 Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties

Implementation Context: The project guidelines do not include a logout feature or command. Therefore, this evaluation focuses solely on session expiration and how the application handles invalidation of expired tokens.

Observed Behavior:

1. **Session Expiration:**

- When a session token expires, the server removes the session token and its associated session context from the session store.
- Any attempt to use an expired session token results in an "invalid session" or "session not found" error response.

2. **Server-Side Validation:**

- The application relies exclusively on server-side validation, ensuring that session tokens are checked against the session store.
- This approach prevents the reuse of expired tokens, even if they are cached or stored on the client side.

3. **No Downstream Relying Parties:**

- The application operates as a standalone system without any relying parties. As such, the evaluation is limited to internal session management.

Security Strengths:

- **Token Expiration:** Expired tokens are effectively invalidated, ensuring they cannot be used for unauthorized access.
- **Server-Side Validation:** This eliminates the risk of token reuse from client-side caching or manipulation.

Identified Gaps and Risks:

1. **Lack of Logout Functionality:**

- Without a logout feature, users cannot explicitly terminate sessions before token expiration.
- This increases the risk of token misuse if the session token is exposed during its validity period.

2. **Potential for Session Hijacking:**

- The absence of logout means that if an attacker obtains an active session token (e.g., via theft or interception), they can use it until it expires.

Recommendations:

1. Reconsider Logout Requirements:

- Even though the guidelines do not specify a logout feature, adding one is a best practice for improving security. Logout should:
 - Remove the session token from the session store immediately.
 - Ensure that clients are instructed to delete any cached session tokens or session contexts.

2. Mitigate Token Misuse Risks:

- Use short-lived access tokens combined with refresh tokens to minimize the impact of exposed tokens.
- Consider logging token usage (e.g., IP or device checks) to detect suspicious activity and trigger token invalidation.

3. Client-Side Improvements:

- Encourage or enforce secure client-side storage mechanisms to reduce the risk of token theft or caching.

4. Explicitly Handle Session Hijacking Scenarios:

- Implement additional server-side controls, such as:
 - Binding tokens to specific IP addresses or devices.
 - Adding an idle timeout to limit session duration when inactive.

Evaluation Summary:

- The application **satisfies the expiration requirement** of this control, as expired tokens are invalidated server-side.
- However, the **lack of a logout feature** introduces security risks, particularly in scenarios involving token exposure or session hijacking.

A screenshot of a failed attempt to use an expired token demonstrates this behavior.

```
(venv) [tk@bismarck cmd-client]$ bash rep_create_session "0rg4" "anon3" "password" "credentials" "session_file"
(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
[{'email': 'anon3@example.com', 'full_name': 'anon3', 'id': 1, 'username': 'anon3'}]
(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
{'error': 'Sessão inválida ou não encontrada'}
(venv) [tk@bismarck cmd-client]$
```

3.3.2 If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period

Implementation Details:

1. Session Expiration Mechanism:

- The application enforces session expiration on the server side.
- Sessions automatically expire 30 minutes after their creation, requiring the user to re-authenticate. Expired sessions are removed from the session store.
- Re-authentication ensures users must provide valid credentials to establish a new session.

2. Idle Period Simulation for Testing:

- The expiration mechanism has been tested using a simulated time advance (**SIMULATE_TIME_ADVANCE**) to validate that sessions are correctly invalidated after 30 minutes.
- The test involves advancing the system clock within the validation function and executing a command with the session token, expecting an "invalid session" response.

follow the test bellow to check the behaviour expected

```
(venv) [tk@bismarck cmd-client]$ bash rep_create_org "Org4" "anon3" "anon3" "anon3@example.com" "credentials"
INFO - Organization created successfully.
{'status': 'success', 'message': 'Organization created successfully.'}
(venv) [tk@bismarck cmd-client]$ bash rep_create_session "Org4" "anon3" "password" "credentials" "session_file"
INFO - Session created successfully.
{'email': 'anon3@example.com', 'full_name': 'anon3', 'id': 1, 'username': 'anon3'}
(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
[{'email': 'anon3@example.com', 'full_name': 'anon3', 'id': 1, 'username': 'anon3'}]
(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
{'error': 'Sessão inválida ou não encontrada'}
```

It's important to specify that, since the verification of expiration for the session token is performed with a timezone reference, to test if the server side that really validates the session as expired after 30 minutes (without needing to wait all the 30 minutes) a simulation within the method that checks the expiration is needed to check if it works as expected, with a simulated time advance by modifying the `is_session_valid()` method as its shown below (this method can be found at `repo-app/api/utlis.py`):


```

SIMULATE_TIME_ADVANCE = timedelta(minutes=35) # Simula 35 minutos no futuro

def is_session_valid(session): 1usage  tk *
    expiration_time = timedelta(seconds=30*60) # 30 minutos
    now = datetime.now(timezone.utc) + SIMULATE_TIME_ADVANCE # Simula avanço no tempo

    if session.created_at.tzinfo is None:
        session_created_at = session.created_at.replace(tzinfo=timezone.utc)
    else:
        session_created_at = session.created_at

    return now - session_created_at < expiration_time

```

The code below to check as simulation :

```

SIMULATE_TIME_ADVANCE = timedelta(minutes=35)

def is_session_valid(session):
    expiration_time = timedelta(seconds=30*60)
    now = datetime.now(timezone.utc) + SIMULATE_TIME_ADVANCE
    if session.created_at.tzinfo is None:
        session_created_at =
session.created_at.replace(tzinfo=timezone.utc)
    else:
        session_created_at = session.created_at
    return now - session_created_at < expiration_time

```

Then it can be executed any authenticated or authorized command with the current session token and it will return a invalid session message

```

(venv) [tk@bismarck cmd-client]$ bash rep_list_subjects "session_file"
INFO    - Setting REP_ADDRESS from Environment to:
INFO    - Loading REP_PUB_KEY from env variable:
{'error': 'Sessão inválida ou não encontrada'}

```

After this validation it can return to the original code to be verified

```
def is_session_valid(session): 1 usage 2 tk
    expiration_time = timedelta(seconds=30*60) # 15 minutes

    now = datetime.now(timezone.utc)

    if session.created_at.tzinfo is None:
        session_created_at = session.created_at.replace(tzinfo=timezone.utc)
    else:
        session_created_at = session.created_at

    return now - session_created_at < expiration_time
```

Identified Gaps and Risks:

Inactivity Timeout:

- The current design treats all sessions the same, whether actively used or idle. For **L3 compliance**, inactivity timeout should be shorter than the periodic reauthentication interval (e.g., 15 minutes for inactivity versus 30 minutes for active use).
- This could leave active sessions vulnerable to misuse during the active 30-minute window.

Absence of Two-Factor Authentication (2FA):

- Re-authentication relies solely on the subject credentials. Without 2FA, the system remains vulnerable to compromised credentials.
- Without a secondary pattern of authentication, the disclosure of credentials, if happened, leaves the session access compromised to anyone that have access of the leaked credentials

Recommendations for Improvement:

Adjust the implementation an Inactivity Timeout:

- Reduce the timeout for idle sessions to 15 minutes, as recommended for L3 compliance. This limits the risk window for inactive sessions being exploited.

Implementing Two-Factor Authentication (2FA): the implementation of **2FA** is highly recommended as part of the re-authentication process. Once 2FA is

integrated, users will be required to authenticate with a second factor, such as an SMS code or an authenticator app, when their session is resumed after inactivity.

3.3.3 Verify that the application gives the option to terminate all other active sessions after a successful password change (including change via password reset/recovery), and that this is effective across the application, federated login (if present), and any relying parties

The application guidelines does not support password changes, in this case the change of the key pair credentials of a subject. Therefore, the control described in **3.3.3**, which mandates the termination of active sessions following a password change, is not applicable to this project.

Additionally, the project does not include **federated login** or any **relying parties** for authentication, so there is no need to terminate sessions across external systems.

While password management and session termination after password changes are important for applications relying on passwords, in this case, the security measures focus on key pair credentials.

Since the project does not involve password changes, federated logins, or relying parties, this control is **not applicable** to the current design of the application. However, it is essential to ensure that, if credentials like key pairs are changed, the system **does invalidate any previous sessions** to maintain security.

3.3.4 Verify that users are able to view and (having re-entered login credentials) log out of any or all currently active sessions and devices.

The application guidelines does not specify a feature that allows users to view or log out of active sessions or devices. The project guidelines do not specify any command or functionality for logging out, and session invalidation is solely handled through **session expiration** verification.

Since the project guidelines does not include the ability for users to view active sessions or log out manually, and the expiration of sessions is automatically managed by the system, this control does not apply to the current implementation.

While the lack of a logout feature and session management interface does not impact the current session security model, this could be considered for future

development, depending on evolving user requirements.

Given that the application guidelines does not provide functionality for users to view or log out of active sessions or devices, and session invalidation is solely based on session expiration, **this control is not applicable** to the current project. However, it may be a consideration for future versions of the application, should the need for manual session management arise.

Recommendations for Future Implementation:

1. Session Management Interface:

- Implement a feature that allows users to view all currently active sessions, including details such as:
 - Session creation timestamp.
 - Device or client used to create the session.
 - Last activity timestamp (if available).
- This would align with the ASVS L3 requirement and enhance transparency.

2. Manual Logout Mechanism:

- Introduce a "Logout from All Sessions" option that requires users to re-enter their credentials. This should terminate all active sessions for the user, including the current one.

3. Device Awareness:

- Track sessions by device or client, allowing users to log out of specific sessions selectively. This feature could leverage unique identifiers for each client.

4. Credential Re-entry Requirement:

- Require users to re-enter their credentials to confirm logout actions, ensuring the legitimacy of the request.

3.4 Cookie-based Session Management

Given the fact that the guidelines of the project does not specify a feature, command or any usage for session cookies, cookies were not implemented in the session management, which makes this control unsuitable to be used in the evaluation of this project

3.5 Token-based Session Management

3.5.1 Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.

The control deals with the revocation of OAuth tokens, but was not evaluated, as the application uses JWTs (JSON Web Tokens) rather than OAuth tokens. Since the session management in the project is based on JWTs, the concerns related to the revocation of OAuth tokens do not apply to the scope of this implementation.

Analysis:

1. OAuth Token Revocation vs. JWT Expiration:

- OAuth tokens are typically long-lived and associated with linked applications or relying parties. Their revocation allows users to control access to their resources.
- JWTs are stateless by design, meaning they cannot be revoked once issued, except by implementing server-side validation mechanisms (e.g., token blacklisting or session store invalidation).

2. Implications of Not Using OAuth:

- The absence of OAuth tokens eliminates the need for token revocation mechanisms.

3.5.2 Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.

Context:

The project employs **JWTs (JSON Web Tokens)** for session management and authentication. These tokens are dynamically generated, contain expiration times, and are cryptographically signed to ensure their integrity. Static API secrets or keys are not used in this implementation.

Compliance with Control:

- The application's use of dynamically generated JWTs fulfills the intent of this control, as it avoids reliance on static secrets or API keys for session management.

3.5.3 Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.

This control ensures that **stateless session tokens**, such as JSON Web Tokens (JWTs), are protected against various attacks by implementing strong security mechanisms. Stateless tokens do not rely on server-side storage to maintain session state, so their integrity and confidentiality must be assured using cryptographic measures.

What the Control Requires For stateless session tokens, the following protections must be implemented:

1. **Digital Signatures:**
 - Tokens must be signed (e.g., using HMAC or asymmetric algorithms like RS256) to ensure their integrity and prevent tampering.
2. **Encryption:**
 - Tokens should be encrypted if they contain sensitive information to protect against unauthorized access. This can involve encrypting payloads with algorithms like AES.
3. **Replay Prevention:**
 - Implement countermeasures, such as unique identifiers (jti claims) or time-based expiration (**exp** claims), to prevent replay attacks.
4. **Protection Against Specific Threats:**
 - **Tampering:** Ensured by verifying the digital signature.
 - **Enveloping:** Prevent attackers from wrapping a legitimate token with malicious data by validating token structures and claims.
 - **Null Cipher Attacks:** Avoid using insecure algorithms (e.g., "none" algorithm in JWTs).
 - **Key Substitution Attacks:** Use secure key management practices to ensure only trusted keys are used for signing and validation.

Detailed Analysis of Session Token Management of the project centered on the control requirements

1 - Digital Signatures

Purpose: Protect the integrity of the session token by ensuring it cannot be tampered with.

- **Implementation Details:**

- JWTs are signed using HMAC with the **HS256** algorithm and a static **SECRET_KEY**.
- Signing ensures that any unauthorized modifications to the token would render it invalid.
- **Potential Concerns:**
 - The use of a hardcoded **SECRET_KEY** may weaken security, as it was stated at 3.2.4, especially if it is not rotated/changed periodically or securely managed.

```
SECRET_KEY = "ultra_secret_repo_key"
```

- Using **HS256** requires shared knowledge of the key between signing and verification points, which increases the risk of key exposure if multiple services are involved.

```
def check_session(session_key):
    try:
        # Decode and validate the JWT signature
        payload = jwt.decode(
            session_key,
            SessionController.SECRET_KEY,
            algorithms=["HS256"]
        )
```

2 - Encryption

Purpose: Protect sensitive information in the session token payload.

- **Implementation Details:**
 - There is no encryption applied to the JWT payload itself, since the session token is generated by a NIST approved hashing algorithm but not an encryption one, which means any sensitive data (e.g., **organization_name** and **subject_username**) would be visible if intercepted.
 - However, the payload avoids storing sensitive data like passwords, reducing potential risk.

```

new_auth_id = AuthenticationID(nonce=nonce, subject=subject)
db.session.add(new_auth_id)
db.session.commit()

# Generate JWT as session_key
created_at = datetime.now(timezone.utc)
expiration_time = created_at + timedelta(minutes=30) # Token valid for 15 minutes
payload = {
    "session_id": new_auth_id.id, # Unique session identifier
    "organization_name": data.get("organization_name"), # Organization name
    "subject_username": subject.username, # Username of the subject
    "iat": created_at, # Issued at
    "exp": expiration_time, # Expiration time
    "jti": str(uuid.uuid4()), # Unique identifier for this token
}

session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")

```

- **Potential Concerns:**

- The absence of payload encryption could expose data to unauthorized access if the JWT is intercepted, especially over insecure communication channels.
- This problem increases the liability and burden over the communication channels and protocols, that will require a high-level of confidentiality communication to protect the session token from being compromised (such as the use of TLS)

3 - Replay Prevention

Purpose: Prevent the reuse of a valid token or nonce to execute unauthorized operations.

- **Implementation Details:**

- A unique **nonce** is generated and validated during session creation to ensure it has not been reused (controlled outside the scope of the session).
- Tokens include a **jti** (JWT ID) claim, which serves as a unique identifier for the token.
- The **exp** claim enforces token expiration, limiting the window for replay attacks.
- The application also deletes old sessions associated with the same user and organization during new session creation.

- **Potential Concerns:**

- There is no explicit mechanism to track or invalidate replayed operations using the session token(i.e. using specifically the JWT token claims) on the server side, leaving some replay attack vectors unaddressed if needed to use the jwt token for this purpose.
- If an attacker captures a valid token before expiration, replay attacks could still occur within the validity period

```

object-108130_105925 / delivery2 / repo-app / api / controllers.py

name 1414 lines (1136 loc) · 60.1 KB

def check_session(session_key):
    SessionController.SECRET_KEY,
    algorithms=["HS256"]
    )
except jwt.ExpiredSignatureError:
    # Remove a sessão se o token JWT estiver expirado
    session = Session.query.filter_by(session_key=session_key).first()
    if session:
        db.session.delete(session)
        db.session.commit()
    return None, "Session token has expired."
except jwt.InvalidTokenError as e:
    return None, f"Invalid session token: {str(e)}"

# Busca a sessão no banco usando a chave de sessão descryptografada
session = Session.query.filter_by(session_key=session_key).first()
if not session:
    return None # Se não encontrar a sessão, retorna None

# Verificar se a sessão é válida
is_valid = is_session_valid(session)
if not is_valid:
    # Se a sessão estiver expirada, removê-la do banco de dados
    db.session.delete(session)
    db.session.commit()
    return None # Sessão inválida ou expirada

# A sessão foi encontrada e é válida
return session

```

4 - Protection Against Tampering

Purpose: Ensure the session token cannot be modified without detection.

- **Implementation Details:**

- The JWT signature ensures integrity and detects tampering during validation.
- The token signature is verified using the server-side **SECRET_KEY** during the **check_session** function.
- If the validation results in a failure, it indicates that the token suffered some sort of tampering

```

object-108130_105925 / delivery2 / repo-app / api / controllers.py
name 1414 lines (1136 loc) · 60.1 KB

def check_session(session_key):
    SessionController.SECRET_KEY,
    algorithms=["HS256"]
    )
except jwt.ExpiredSignatureError:
    # Remove a sessão se o token JWT estiver expirado
    session = Session.query.filter_by(session_key=session_key).first()
    if session:
        db.session.delete(session)
        db.session.commit()
        return None, "Session token has expired."
except jwt.InvalidTokenError as e:
    return None, f"Invalid session token: {str(e)}"

# Busca a sessão no banco usando a chave de sessão descriptografada
session = Session.query.filter_by(session_key=session_key).first()
if not session:
    return None # Se não encontrar a sessão, retorna None

# Verificar se a sessão é válida
is_valid = is_session_valid(session)
if not is_valid:
    # Se a sessão estiver expirada, removê-la do banco de dados
    db.session.delete(session)
    db.session.commit()
    return None # Sessão inválida ou expirada

# A sessão foi encontrada e é válida
return session

```

- **Potential Concerns:**

- The reliance on **HS256** ties security directly to the protection of the static secret.
- If the **SECRET_KEY** is exposed or guessed, the token becomes vulnerable to tampering or forgery.

```
SECRET_KEY = "ultra_secret_repo_key"
```

5 - Null Cipher and Weak Algorithm Prevention

Purpose: Prevent the use of insecure algorithms for token generation and validation.

- **Implementation Details:**

- The implementation explicitly uses **HS256** as the signing algorithm.
- There is no allowance for the "none" algorithm or other insecure options, as seen in the **check_session** function.

```
session_key = jwt.encode(payload, SessionController.SECRET_KEY, algorithm="HS256")
```

- **Potential Concerns:**

- **HS256** is secure when used correctly, but stronger algorithms like **RS256** (with a key pair) could provide better protection against key substitution and scalability.

6 - Key Substitution Attacks

Purpose: Prevent attackers from substituting keys to forge tokens.

- **Implementation Details:**

- A single **SECRET_KEY** is used to sign and validate tokens, one key that only the server has access to use it.

```
SECRET_KEY = "ultra_secret_repo_key"
```

- No mechanisms are described for rotating keys or using distinct keys per tenant/organization.

- **Potential Concerns:**

- Without a robust key management system, the static **SECRET_KEY** is a single point of failure.
- If the key is leaked or compromised, all tokens become vulnerable and the risk of receiving a forged token increases.

The session token generation and handling fulfill some aspects of the ASVS control requirements but have several potential gaps:

- Strong signature verification (HS256) protects against tampering but relies on the static **SECRET_KEY**.
 - This problem was already well addressed at 3.2.4
- Replay prevention is partially addressed via **nonce** validation, but not implemented via token claims(although is also partially implemented via a custom nonce verification in some client-server communication points), and **jti** claims but lacks server-side tracking of token uniqueness.
- Encryption could be more ensured and implemented for sensitive payloads, avoiding leaving data exposed in case of token interception.
- Key(secret key for HS256 encoding algorithm) management practices are insufficient to mitigate substitution or leakage risks.
 - This problem was already well addressed at 3.2.4

Recommendations for L3 Compliance:

- 1. Transition to Asymmetric Algorithms:**
 - Use RS256 or ES256 for signing JWTs to separate signing and verification responsibilities.
- 2. Implement Key Management Best Practices:**
 - Store signing keys securely using a secret manager.
 - Rotate keys periodically and enforce secure key access policies.
- 3. Enhance Replay Prevention:**
 - Perform verifications over the JWT token claims to identify potential replay attacks
- 4. Overall recommendation**
 - Enforce the use of TLS to ensure the completeness of confidential communication

3.6 Federated Re-authentication

Not properly suitable for evaluation for this project. The control in question addresses scenarios involving federated authentication systems, where the authentication process is outsourced to an external Identity Provider (IdP) or Credential Service Provider (CSP). This includes protocols such as OAuth 2.0, OpenID Connect, or SAML, where the application acts as a Relying Party (RP) and depends on external systems to validate user identities and manage re-authentication.

No Federated Authentication Involved

The guidelines of the project specify that the own application manages authentication and re-authentication internally, without relying on third-party systems or protocols like OAuth 2.0 or OpenID Connect.

All aspects of user authentication, session token management, and validation are performed solely by the application.

No External Identity Provider (IdP) or CSP:

- The application does not delegate user identity verification or session management to an external service.
- Re-authentication is handled directly within your application, bypassing the need for federated controls.

Re-authentication Is Internal:

Any need for re-authentication is enforced by the own application, not through a federated system requiring interaction with an external entity.

In conclusion, considering the points exposed above, this control is not suitable to used in the evaluation for this project

3.7 Defenses Against Session Management Exploits

3.7.1 Verify the application ensures a full, valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.

This control aims to ensure that sensitive operations are performed only under a valid session or after additional user verification to prevent unauthorized actions.

Applicability Assessment

- This control is not applicable to the current project given the fact that (according to the project guidelines):

1. The application validates all operations using a session token without requiring re-authentication or secondary verification, as stated in the guidelines of the project.
2. There are no explicitly defined sensitive operations (e.g., credential changes or financial transactions) in the project's functionality.

As per the project's design and guidelines, the security model relies on centralized session token validation, and additional re-authentication measures are not implemented, nor expected. Therefore, this control is not relevant for evaluation or enforcement.

Conclusion

Therefore, after all the feature and technical description of the project, alongside with the security ASVS evaluation of the 3rd chapter over the developed system, its clear to assume that:

- The project implemented most of the features expected on the project guidelines, specially the features related to document management, authentication, session control and access control
 - Some requirements related to security the project still need be implemented in their totality, but the ones implemented function well as expected
- The security evaluation of the V3 chapter of Session Management over the developed system proves that the application as whole follow most of the expected requirements of the ASVS chapter, but further security and technical improvement is needed in other to improve aspects of confidentiality, integrity and availability related to the session control of the application and to meet other security standards that were not achieved at the evaluation process