Authentication and OAuth

Authentication is one of the most critical aspects of modern web applications.

Part 1: Core Authentication Concepts

What is Authentication?

Authentication is the process of verifying the identity of a user, device, or system attempting to access a resource. In simple terms, it answers the question: "Are you who you claim to be?"

There are several authentication approaches:

1. Basic Authentication This is the simplest form where credentials (username and password) are sent with each request, usually in an HTTP header.

2. Session-based Authentication After successful login, the server creates a session and sends a session ID to the client (typically stored in a cookie).

3. Token-based Authentication After authentication, the server issues a token (like JWT) that the client includes in subsequent requests.

4. OAuth and OAuth 2.0 Authorization frameworks that enable third-party applications to access resources on behalf of users without sharing credentials.

Part 2: Basic Authentication

Concept

Basic Authentication involves sending a username and password with each request. The credentials are encoded (not encrypted) in Base64 format and included in the HTTP Authorization header.

Flow

1. The client sends a request to a protected resource
2. The server responds with a 401 status code and a WWW-Authenticate header
3. The client includes credentials in the Authorization header in subsequent requests
4. The server validates the credentials and grants access if valid

FastAPI Example

```python
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import HTTPBasic, HTTPBasicCredentials
import secrets

app = FastAPI()
security = HTTPBasic()

# In a real application, you would store these in a secure database
# and hash the passwords
USER_DB = {
    "alice": "password123",
    "bob": "securepassword"
}
```

```python
def authenticate_user(credentials: HTTPBasicCredentials = Depends(security)):
    """Authenticate user with basic auth"""
    is_username_correct = False
    is_password_correct = False

    if credentials.username in USER_DB:
        is_username_correct = True
        if secrets.compare_digest(credentials.password, USER_DB[credentials.username]):
            is_password_correct = True

    if not (is_username_correct and is_password_correct):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid credentials",
            headers={"WWW-Authenticate": "Basic"},
        )

    return credentials.username

@app.get("/user/me")
def get_current_user(username: str = Depends(authenticate_user)):
    """Return the current user"""
    return {"username": username}
```

Advantages and Limitations

Advantages:

- Simple to implement
- Supported by most HTTP clients

Limitations:

- Credentials sent with every request (even if encoded)
- No easy way to log out (except clearing browser cache)
- No way to implement granular permissions
- Credentials could be intercepted if not using HTTPS

Part 3: Session-Based Authentication

Concept

Session-based authentication creates a server-side session after successful login. The server generates a session ID and sends it to the client, usually as a cookie. This session ID is used to identify the user in subsequent requests.

Flow

1. User submits credentials (username and password)
2. Server verifies credentials
3. If valid, server creates a session and stores it (in memory, database, etc.)
4. Server sends a session ID to the client (as a cookie)
5. Client includes this cookie in subsequent requests
6. Server validates the session ID and grants access if valid

Side Note:

To better understand the below example:

The `secrets.compare_digest` function is a secure method for comparing two strings (or byte sequences) to determine if they are equal. It is designed to mitigate **timing attacks**, which are a type of side-channel attack where an attacker can infer information about the data being compared based on the time it takes to perform the comparison.

Why Use `secrets.compare_digest`?

In a typical string comparison operation (e.g., `==` in Python), the comparison stops as soon as a mismatch is found. This means that the time taken to compare two strings can vary depending on how many characters match at the beginning. An attacker could exploit this timing difference to guess the correct value incrementally, character by character.

`secrets.compare_digest`, on the other hand, performs the comparison in constant time, regardless of how many characters match. This ensures that no information about the strings being compared is leaked through timing differences.

Syntax

```
secrets.compare_digest(a, b)
```

- `a` and `b`: The two strings or byte sequences to compare.
- Returns: `True` if the inputs are equal, `False` otherwise.

FastAPI Example

```
from fastapi import FastAPI, Depends, HTTPException, status, Response, Request
from fastapi.security import HTTPBasic, HTTPBasicCredentials
import secrets
import uuid
from typing import Dict

app = FastAPI()
security = HTTPBasic()

# In a real application, you would store these in a secure database
# and hash the passwords
USER_DB = {
    "alice": "password123",
    "bob": "securepassword"
}

# Session storage (in a real app, use Redis or a database)
SESSIONS: Dict[str, str] = {}

@app.post("/login")
def login(credentials: HTTPBasicCredentials = Depends(security), response: Response =
None):
    """Log in and create a session"""
    is_username_correct = False
    is_password_correct = False

    if credentials.username in USER_DB:
        is_username_correct = True
        if secrets.compare_digest(credentials.password, USER_DB[credentials.username]):
            is_password_correct = True
    if not (is_username_correct and is_password_correct):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid credentials",
            headers={"WWW-Authenticate": "Basic"},
        )

    # Creating a session for the user...
    session_id = str(uuid.uuid4())
    SESSIONS[session_id] = credentials.username

    # Set cookie
    response.set_cookie(
        key="session_id",
        value=session_id,
        httponly=True,  # Prevents JavaScript access
        max_age=1800,   # 30 minutes
        secure=True,    # Only sent over HTTPS
        samesite="lax"  # Protection against CSRF
    )

    return {"message": "Successfully logged in"}

def get_current_user(request: Request):
    """Get current user from session"""
    session_id = request.cookies.get("session_id")
    if not session_id or session_id not in SESSIONS:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Not authenticated"
        )
    return SESSIONS[session_id]

@app.get("/user/me")
def read_current_user(username: str = Depends(get_current_user)):
    """Return the current user"""
    return {"username": username}

@app.post("/logout")
def logout(response: Response, username: str = Depends(get_current_user)):
    """Log out and delete session"""
    # Find the session ID associated with this user
    session_ids = [sid for sid, user in SESSIONS.items() if user == username]

    # Remove sessions from storage
    for sid in session_ids:
        SESSIONS.pop(sid, None)

    # Clear the cookie
    response.delete_cookie(key="session_id")
    return {"message": "Successfully logged out"}
```

Advantages and Limitations

Advantages:

- Server has complete control over the session lifecycle
- Easy to implement logout by invalidating the session
- Session data stored server-side (more secure)

Limitations:

- Requires server-side storage
- Scaling can be challenging (session replication across servers)
- Vulnerable to CSRF attacks if not implemented properly

Part 4: Token-Based Authentication (JWT)

Concept

Token-based authentication, particularly using JSON Web Tokens (JWT), issues a signed token to the client after successful authentication. The client includes this token in the Authorization header for subsequent requests. JWTs can contain claims (user information) that the server can verify without database lookups.

Flow

1. User submits credentials
2. Server verifies credentials
3. If valid, server generates a JWT containing user claims and signs it
4. Server sends the JWT to the client
5. Client stores the JWT (usually in memory or local storage)
6. Client sends the JWT in the Authorization header for subsequent requests
7. Server validates the token signature and grants access if valid

FastAPI Example

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from datetime import datetime, timedelta
from pydantic import BaseModel
from typing import Optional

# Configuration
SECRET_KEY = "a_very_secret_key_that_should_be_secured_in_production"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

# In a real app, store users in a database with hashed passwords
USERS_DB = {
    "alice": {
        "username": "alice",
        "hashed_password": "$2b$12$MnjbRTl3QBzMiMt4Qg13q.3nwOQcQkiYj1k3R4t5P4rCMgczXGnj2",
# "password123"
    },
    "bob": {
        "username": "bob",
        "hashed_password": "$2b$12$DaE5nY6OSGZk3vpiH1EZvuNsElRCeJFr4Xboh67HwUIZdXy2gZNtS",
# "securepassword"
    }
}

app = FastAPI()

# Password hashing context
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# OAuth2 scheme for token retrieval
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# Models
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: Optional[str] = None

class User(BaseModel):
    username: str

class UserInDB(User):
    hashed_password: str

# Helper functions
def verify_password(plain_password, hashed_password):
    """Verify if plain password matches the hashed one"""
    return pwd_context.verify(plain_password, hashed_password)

def get_user(db, username: str):
    """Get user from database"""
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)
    return None

def authenticate_user(db, username: str, password: str):
    """Authenticate user against database"""
    user = get_user(db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    """Create a JWT token"""
    to_encode = data.copy()
```

```python
    # Set expiration time
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)

    to_encode.update({"exp": expire})

    # Create token
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_user(token: str = Depends(oauth2_scheme)):
    """Get the current user from the token"""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        # Decode and verify token
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")

        if username is None:
            raise credentials_exception

        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    # Get user from database
    user = get_user(USERS_DB, username=token_data.username)
    if user is None:
        raise credentials_exception

    return user

# Routes
@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends()):
    """Login and get JWT token"""
    # Authenticate user
    user = authenticate_user(USERS_DB, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Create access token
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/users/me", response_model=User)
async def read_users_me(current_user: User = Depends(get_current_user)):
    """Get current user information"""
    return current_user
```

Advantages and Limitations

Advantages:

- Stateless authentication (no server-side storage needed)
- Good for microservices and distributed systems
- Can contain user claims, reducing database lookups
- Well-suited for mobile applications

Limitations:

- Cannot be invalidated before expiration (unless using a blacklist)
- Size limitations (should be kept small)
- Need to handle token renewal
- Requires secure storage on client-side