Chapter 20: Security in System Design

Introduction: Security is Not Optional

Security must be designed into the system from the start, not added later.

Cost of Security Breach:

Equifax (2017):
├─ 147 million users affected
├─ $700 million settlement
├─ Reputation damage
└─ CEO resigned

Target (2013):
├─ 40 million credit cards stolen
├─ $18.5 million settlement
└─ Years of reputation damage

Prevention cost: $1-10 million
Breach cost: $100+ million

Security is an investment, not an expense!

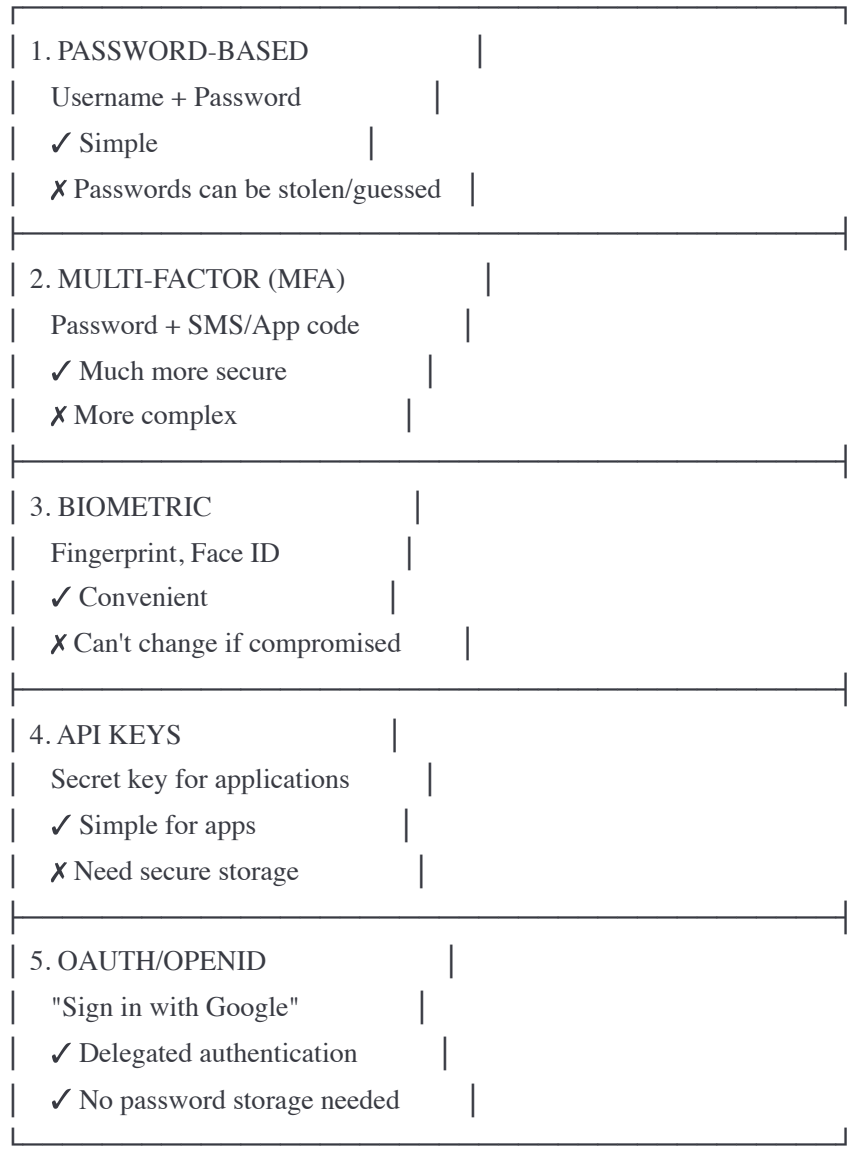
---

1. Authentication vs Authorization

Authentication (AuthN): "Who are you?"

Definition: Verifying identity of a user/service.

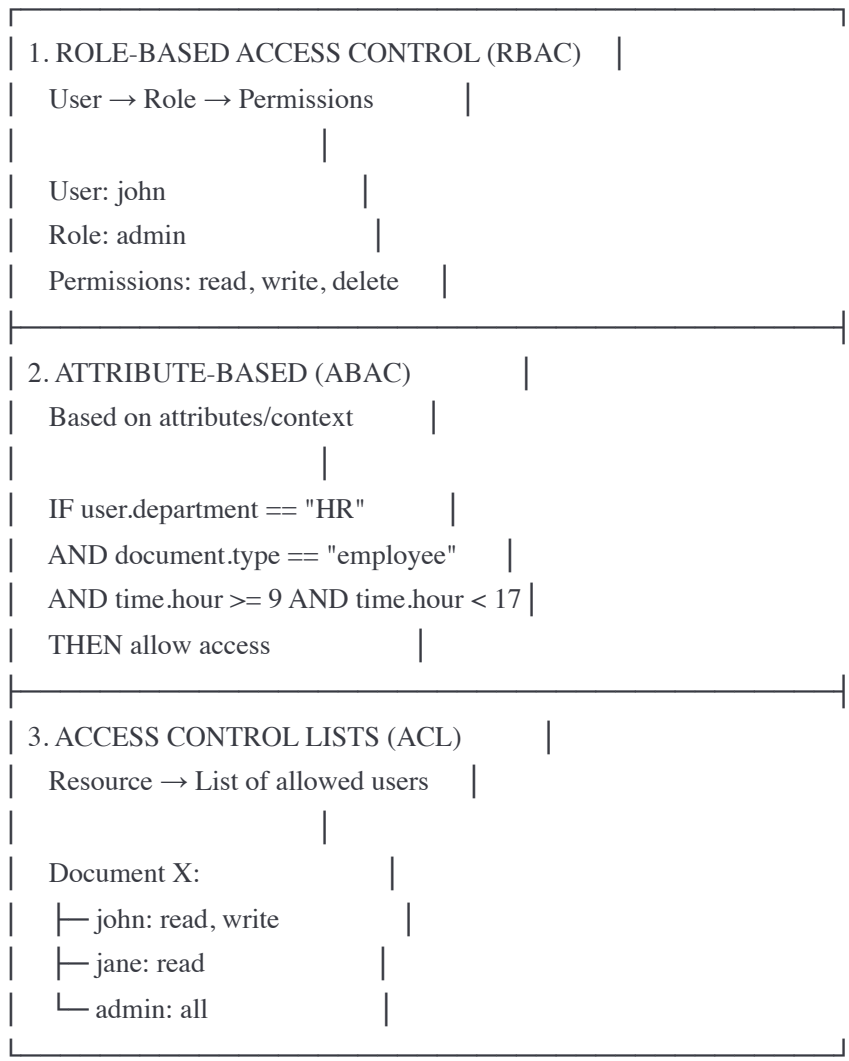
Authentication Methods:



Authorization (AuthZ): "What can you do?"

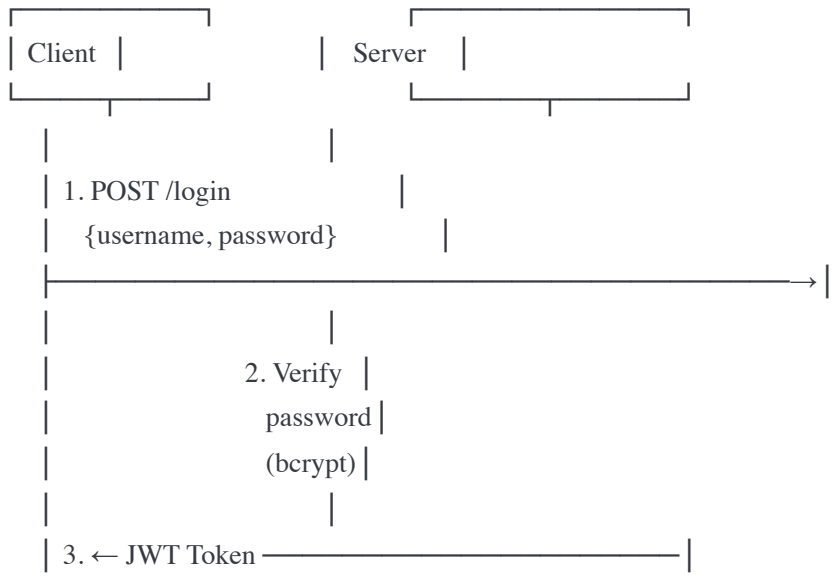
Definition: Determining what resources/actions user can access.

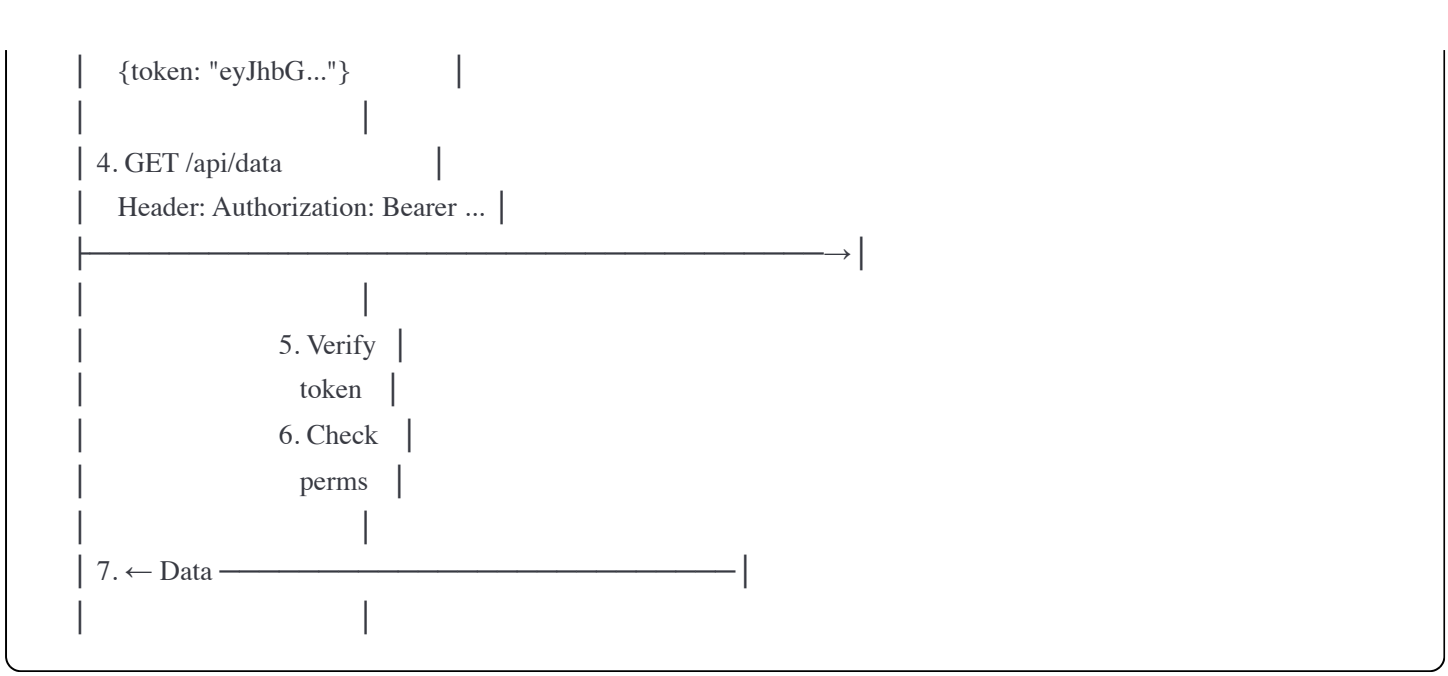
Authorization Models:



Authentication Flow

Login Flow:





Complete Authentication Implementation

javascript

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const express = require('express');

const app = express();
const SECRET_KEY = process.env.JWT_SECRET || 'your-secret-key';

// User database (simplified)
const users = new Map([
  ['john@example.com', {
    id: 1,
    email: 'john@example.com',
    // Password: "password123" (hashed)
    passwordHash: '$2b$10$rBV2uKh3KwY4YjH5oHxM0OqXkW5rJ6mF7nYH8L0xK2KxYqF3KqP8G',
    role: 'admin'
  }]
]);

// 1. Register User
app.post('/register', async (req, res) => {
  const { email, password, name } = req.body;

  // Validation
  if (!email || !password) {
    return res.status(400).json({ error: 'Email and password required' });
  }

  if (password.length < 8) {
    return res.status(400).json({ error: 'Password must be 8+ characters' });
  }

  // Check if user exists
  if (users.has(email)) {
    return res.status(409).json({ error: 'User already exists' });
  }

  try {
    // Hash password (bcrypt with salt)
    const passwordHash = await bcrypt.hash(password, 10);

    const user = {
      id: users.size + 1,
      email,
      name,
      passwordHash,
      role: 'user',
```

```
      createdAt: new Date().toISOString()
    };

    users.set(email, user);

    // Don't return password hash!
    const { passwordHash: _, ...userResponse } = user;

    res.status(201).json(userResponse);

  } catch (error) {
    res.status(500).json({ error: 'Registration failed' });
  }
});

// 2. Login (Issue JWT)
app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  const user = users.get(email);

  if (!user) {
    // Don't reveal if user exists
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  try {
    // Compare password with hash
    const isValid = await bcrypt.compare(password, user.passwordHash);

    if (!isValid) {
      // Log failed attempt (for security monitoring)
      console.log(`Failed login attempt for ${email}`);
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    // Create JWT token
    const token = jwt.sign(
      {
        userId: user.id,
        email: user.email,
        role: user.role
      },
      SECRET_KEY,
      {
        expiresIn: '24h',
        issuer: 'my-app',
```

```
      subject: user.id.toString()
    }
  );

  // Create refresh token (longer-lived)
  const refreshToken = jwt.sign(
    { userId: user.id },
    SECRET_KEY,
    { expiresIn: '7d' }
  );

  res.json({
    token,
    refreshToken,
    expiresIn: 86400  // 24 hours in seconds
  });

 } catch (error) {
  res.status(500).json({ error: 'Login failed' });
 }
});

// 3. Authentication Middleware
function authenticate(req, res, next) {
 const authHeader = req.headers.authorization;

 if (!authHeader || !authHeader.startsWith('Bearer ')) {
  return res.status(401).json({ error: 'No token provided' });
 }

 const token = authHeader.substring(7);

 try {
  // Verify and decode JWT
  const decoded = jwt.verify(token, SECRET_KEY);

  // Attach user info to request
  req.user = decoded;

  next();

 } catch (error) {
  if (error.name === 'TokenExpiredError') {
   return res.status(401).json({ error: 'Token expired' });
  }

  res.status(401).json({ error: 'Invalid token' });
```

```
  }
}

// 4. Authorization Middleware
function authorize(...allowedRoles) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ error: 'Not authenticated' });
    }

    if (!allowedRoles.includes(req.user.role)) {
      return res.status(403).json({
        error: 'Forbidden',
        message: `Requires one of: ${allowedRoles.join(', ')}`
      });
    }

    next();
  };
}

// Protected endpoints
app.get('/api/profile', authenticate, (req, res) => {
  // Any authenticated user
  res.json({
    userId: req.user.userId,
    email: req.user.email
  });
});

app.get('/api/admin/users', authenticate, authorize('admin'), (req, res) => {
  // Only admins
  const allUsers = Array.from(users.values()).map(u => {
    const { passwordHash, ...safe } = u;
    return safe;
  });

  res.json(allUsers);
});

app.delete('/api/admin/users/:id', authenticate, authorize('admin'), (req, res) => {
  // Only admins can delete users
  // Delete user logic...
  res.json({ success: true });
});

// 5. Refresh Token
```

```
app.post('/refresh', (req, res) => {
  const { refreshToken } = req.body;

  if (!refreshToken) {
    return res.status(401).json({ error: 'No refresh token' });
  }

  try {
    const decoded = jwt.verify(refreshToken, SECRET_KEY);

    // Issue new access token
    const newToken = jwt.sign(
      {
        userId: decoded.userId,
        email: decoded.email,
        role: decoded.role
      },
      SECRET_KEY,
      { expiresIn: '24h' }
    );

    res.json({ token: newToken });

  } catch (error) {
    res.status(401).json({ error: 'Invalid refresh token' });
  }
});
```

---

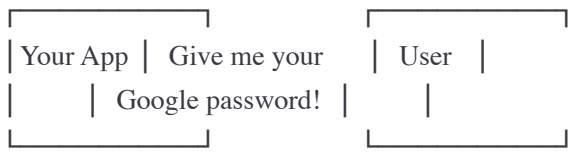
2. OAuth 2.0

What is OAuth?

OAuth 2.0: Delegated authorization protocol.

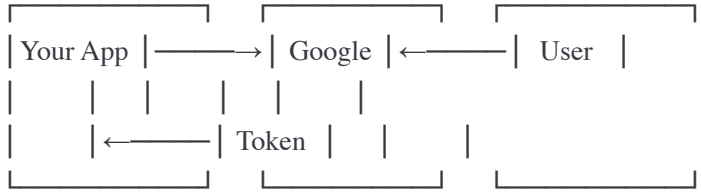
Use Case: "Sign in with Google"

Without OAuth:



✗ Never share passwords!

With OAuth:



✓ Google handles authentication

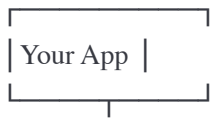
✓ Your app gets token

✓ User never shares password

OAuth 2.0 Flow (Authorization Code)

Complete OAuth Flow:

1. User Clicks "Sign in with Google"

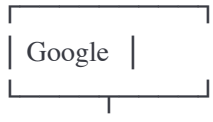


Redirect to Google

↓

```
https://accounts.google.com/oauth/authorize?
 client_id=YOUR_CLIENT_ID
 &redirect_uri=https://yourapp.com/callback
 &response_type=code
 &scope=email profile
```

2. User Logs in to Google



User authenticates

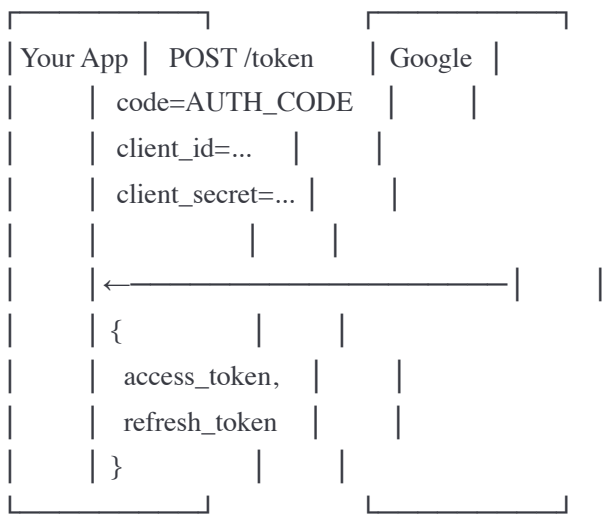
User approves permissions

↓

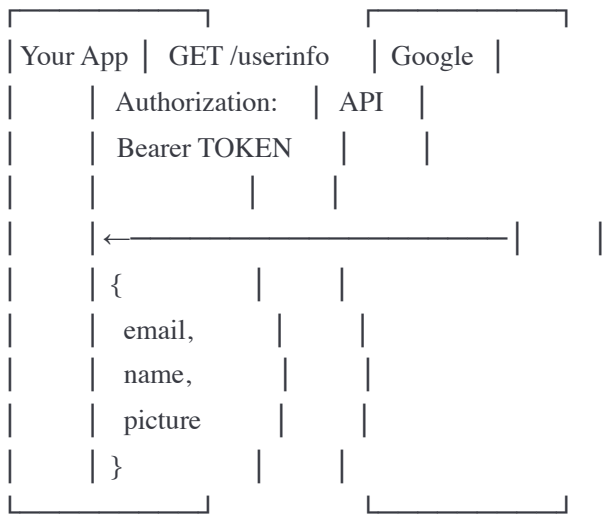
3. Google Redirects Back

https://yourapp.com/callback?code=AUTH_CODE_HERE

4. Your App Exchanges Code for Token



5. Use Token to Access User Data



OAuth 2.0 Implementation

javascript

```
const express = require('express');
const axios = require('axios');
const crypto = require('crypto');

const app = express();

// OAuth Configuration
const OAUTH_CONFIG = {
  google: {
    clientId: process.env.GOOGLE_CLIENT_ID,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    redirectUri: 'http://localhost:3000/auth/google/callback',
    authUrl: 'https://accounts.google.com/o/oauth2/v2/auth',
    tokenUrl: 'https://oauth2.googleapis.com/token',
    userInfoUrl: 'https://www.googleapis.com/oauth2/v2/userinfo',
    scope: 'email profile'
  }
};

// Store for state tokens (prevent CSRF)
const stateTokens = new Map();

// 1. Initiate OAuth Flow
app.get('/auth/google', (req, res) => {
  // Generate random state token (CSRF protection)
  const state = crypto.randomBytes(16).toString('hex');
  stateTokens.set(state, Date.now());

  // Cleanup old tokens (after 10 minutes)
  for (const [token, timestamp] of stateTokens) {
    if (Date.now() - timestamp > 600000) {
      stateTokens.delete(token);
    }
  }

  // Build authorization URL
  const authUrl = new URL(OAUTH_CONFIG.google.authUrl);
  authUrl.searchParams.set('client_id', OAUTH_CONFIG.google.clientId);
  authUrl.searchParams.set('redirect_uri', OAUTH_CONFIG.google.redirectUri);
  authUrl.searchParams.set('response_type', 'code');
  authUrl.searchParams.set('scope', OAUTH_CONFIG.google.scope);
  authUrl.searchParams.set('state', state);

  // Redirect user to Google
  res.redirect(authUrl.toString());
});
```

```
// 2. Handle OAuth Callback
app.get('/auth/google/callback', async (req, res) => {
  const { code, state } = req.query;

  // Verify state token (CSRF protection)
  if (!state || !stateTokens.has(state)) {
    return res.status(400).json({ error: 'Invalid state token' });
  }

  stateTokens.delete(state);

  if (!code) {
    return res.status(400).json({ error: 'No authorization code' });
  }

  try {
    // Exchange code for access token
    const tokenResponse = await axios.post(
      OAUTH_CONFIG.google.tokenUrl,
      {
        code,
        client_id: OAUTH_CONFIG.google.clientId,
        client_secret: OAUTH_CONFIG.google.clientSecret,
        redirect_uri: OAUTH_CONFIG.google.redirectUri,
        grant_type: 'authorization_code'
      }
    );

    const { access_token, refresh_token } = tokenResponse.data;

    // Get user info from Google
    const userResponse = await axios.get(
      OAUTH_CONFIG.google.userInfoUrl,
      {
        headers: {
          Authorization: `Bearer ${access_token}`
        }
      }
    );

    const googleUser = userResponse.data;

    // Find or create user in our database
    let user = await db.findUserByEmail(googleUser.email);

    if (!user) {
```

```
      user = await db.createUser({
        email: googleUser.email,
        name: googleUser.name,
        picture: googleUser.picture,
        authProvider: 'google',
        googleId: googleUser.id
      });
    }

    // Create our own JWT token
    const appToken = jwt.sign(
      {
        userId: user.id,
        email: user.email,
        role: user.role
      },
      SECRET_KEY,
      { expiresIn: '24h' }
    );

    // Store OAuth tokens securely (encrypted)
    await db.storeOAuthTokens(user.id, {
      provider: 'google',
      accessToken: access_token,
      refreshToken: refresh_token
    });

    // Redirect to app with our token
    res.redirect(`/dashboard?token=${appToken}`);

  } catch (error) {
    console.error('OAuth error:', error);
    res.status(500).json({ error: 'OAuth authentication failed' });
  }
});

// 3. Use OAuth Token to Access Google APIs
app.get('/api/google-calendar', authenticate, async (req, res) => {
  // Get stored OAuth tokens
  const tokens = await db.getOAuthTokens(req.user.userId, 'google');

  if (!tokens) {
    return res.status(401).json({ error: 'Google not connected' });
  }

  try {
    // Call Google Calendar API with OAuth token
```

```
    const response = await axios.get(
      'https://www.googleapis.com/calendar/v3/calendars/primary/events',
      {
        headers: {
          Authorization: `Bearer ${tokens.accessToken}`
        }
      }
    );

    res.json(response.data);

  } catch (error) {
    if (error.response?.status === 401) {
      // Access token expired, refresh it
      const newTokens = await refreshOAuthToken(tokens.refreshToken);

      // Retry with new token
      const response = await axios.get(
        'https://www.googleapis.com/calendar/v3/calendars/primary/events',
        {
          headers: {
            Authorization: `Bearer ${newTokens.access_token}`
          }
        }
      );

      res.json(response.data);
    } else {
      throw error;
    }
  }
});

async function refreshOAuthToken(refreshToken) {
  const response = await axios.post(
    OAUTH_CONFIG.google.tokenUrl,
    {
      refresh_token: refreshToken,
      client_id: OAUTH_CONFIG.google.clientId,
      client_secret: OAUTH_CONFIG.google.clientSecret,
      grant_type: 'refresh_token'
    }
  );

  // Update stored tokens
  await db.updateOAuthTokens(userId, response.data);
```

```
  return response.data;
}
```

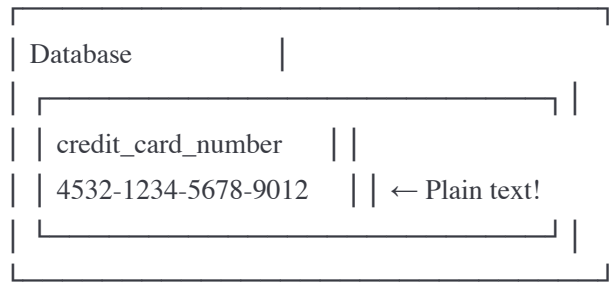
---

3. Encryption

Encryption at Rest

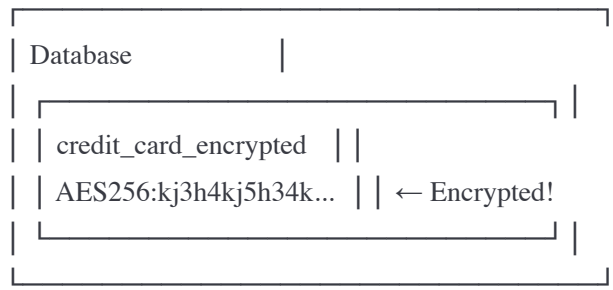
Encrypt data in storage.

Without Encryption:



Attacker gets database backup → Sees all credit cards! ✗

With Encryption:



Attacker gets database → Sees gibberish (useless without key)

Implementation:

```
javascript
```

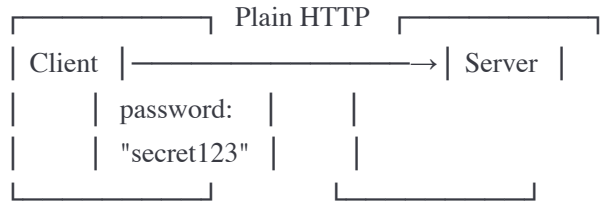
```
const crypto = require('crypto');

class EncryptionService {
  constructor(encryptionKey) {
    // Key should be 32 bytes for AES-256
    this.key = Buffer.from(encryptionKey, 'hex');
    this.algorithm = 'aes-256-gcm';
  }

  encrypt(plaintext) {
    // Generate random IV (initialization vector)
    const iv = crypto.randomBytes(16);

    // Create cipher
    const cipher = crypto.createCipheriv(this.algorithm, this.key, iv);

    // Encrypt
    let encrypted = cipher.update(plaintext, 'utf8', 'hex');
    encrypted += cipher.final('hex');

    // Get authentication tag (for integrity)
    const authTag = cipher.getAuthTag();

    // Return IV + encrypted data + auth tag
    return {
      iv: iv.toString('hex'),
      encrypted: encrypted,
      authTag: authTag.toString('hex')
    };
  }

  decrypt(encryptedData) {
    const { iv, encrypted, authTag } = encryptedData;

    // Create decipher
    const decipher = crypto.createDecipheriv(
      this.algorithm,
      this.key,
      Buffer.from(iv, 'hex')
    );

    // Set auth tag
    decipher.setAuthTag(Buffer.from(authTag, 'hex'));

    // Decrypt
    let decrypted = decipher.update(encrypted, 'hex', 'utf8');
```

```
    decrypted += decipher.final('utf8');

    return decrypted;
  }
}

// Usage
const encryptionKey = process.env.ENCRYPTION_KEY; // 32-byte hex string
const encryption = new EncryptionService(encryptionKey);

// Storing credit card
async function storeCreditCard(userId, cardNumber) {
  // Encrypt before storing
  const encrypted = encryption.encrypt(cardNumber);

  await db.query(
    'INSERT INTO credit_cards (user_id, card_data_iv, card_data_encrypted, card_data_tag) VALUES (?, ?, ?, ?)',
    [userId, encrypted.iv, encrypted.encrypted, encrypted.authTag]
  );

  console.log('Stored encrypted credit card');
}

// Retrieving credit card
async function getCreditCard(userId) {
  const result = await db.query(
    'SELECT card_data_iv, card_data_encrypted, card_data_tag FROM credit_cards WHERE user_id = ?',
    [userId]
  );

  if (!result) return null;

  // Decrypt
  const decrypted = encryption.decrypt({
    iv: result.card_data_iv,
    encrypted: result.card_data_encrypted,
    authTag: result.card_data_tag
  });

  return decrypted;
}

// Store: "4532-1234-5678-9012"
// Database: "a3f2...b8c1" (encrypted)
// Retrieve: "4532-1234-5678-9012" (decrypted)
```

Encryption in Transit (TLS/SSL)

Encrypt data over network.

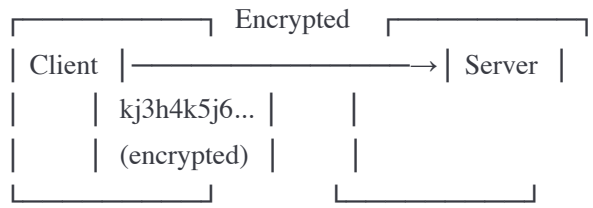
Without HTTPS:



↑

Attacker can intercept and read! ✗

With HTTPS:



↑

Attacker sees gibberish! ✓

SSL/TLS Handshake:

1. Client Hello
   Client → Server: "Let's use TLS 1.3"

2. Server Hello
   Server → Client: "OK, here's my certificate"
   Certificate contains: Public key

3. Client Verifies Certificate
   Checks: Is it signed by trusted CA?
   Is domain correct?
   Is it expired?

4. Key Exchange
   Client generates session key
   Encrypts with server's public key
   Sends to server

5. Encrypted Communication
   All data encrypted with session key
   Both client and server can decrypt

HTTPS Configuration (Node.js):

javascript

```
const https = require('https');
const fs = require('fs');
const express = require('express');

const app = express();

// Load SSL certificate
const options = {
  key: fs.readFileSync('path/to/private-key.pem'),
  cert: fs.readFileSync('path/to/certificate.pem'),
  ca: fs.readFileSync('path/to/ca-bundle.pem')  // Optional: CA bundle
};

// Create HTTPS server
https.createServer(options, app).listen(443, () => {
  console.log('HTTPS server running on port 443');
});

// Force HTTPS redirect
app.use((req, res, next) => {
  if (req.protocol !== 'https') {
    return res.redirect(`https://${req.hostname}${req.url}`);
  }
  next();
});

// Secure headers
app.use((req, res, next) => {
  // HSTS: Force HTTPS for future requests
  res.setHeader('Strict-Transport-Security', 'max-age=31536000; includeSubDomains');

  // Prevent clickjacking
  res.setHeader('X-Frame-Options', 'DENY');

  // XSS protection
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-XSS-Protection', '1; mode=block');

  // CSP: Content Security Policy
  res.setHeader('Content-Security-Policy', "default-src 'self'");

  next();
});
```

4. API Security

API Key Authentication

javascript

```
const crypto = require('crypto');

class APIKeyManager {
  constructor() {
    this.keys = new Map();  // In production: use database
  }

  generateAPIKey(userId) {
    // Generate cryptographically secure random key
    const apiKey = crypto.randomBytes(32).toString('hex');

    // Hash before storing (like passwords)
    const hash = crypto.createHash('sha256').update(apiKey).digest('hex');

    this.keys.set(hash, {
      userId,
      createdAt: new Date(),
      lastUsed: null,
      usageCount: 0
    });

    // Return raw key to user (only time they see it!)
    return apiKey;
  }

  async validateAPIKey(apiKey) {
    // Hash provided key
    const hash = crypto.createHash('sha256').update(apiKey).digest('hex');

    const keyData = this.keys.get(hash);

    if (!keyData) {
      return { valid: false };
    }

    // Update usage stats
    keyData.lastUsed = new Date();
    keyData.usageCount++;

    return {
      valid: true,
      userId: keyData.userId
    };
  }

  revokeAPIKey(apiKey) {
```

```
    const hash = crypto.createHash('sha256').update(apiKey).digest('hex');
    this.keys.delete(hash);
  }
}

const apiKeyManager = new APIKeyManager();

// Generate API key for user
app.post('/api/keys', authenticate, (req, res) => {
  const apiKey = apiKeyManager.generateAPIKey(req.user.userId);

  res.json({
    apiKey,
    message: 'Save this key securely. You won\'t see it again!'
  });
});

// Middleware to validate API key
async function validateAPIKey(req, res, next) {
  const apiKey = req.headers['x-api-key'];

  if (!apiKey) {
    return res.status(401).json({ error: 'API key required' });
  }

  const validation = await apiKeyManager.validateAPIKey(apiKey);

  if (!validation.valid) {
    return res.status(401).json({ error: 'Invalid API key' });
  }

  req.user = { userId: validation.userId };
  next();
}

// Protected API endpoint
app.get('/api/data', validateAPIKey, (req, res) => {
  res.json({ data: 'Secret data', userId: req.user.userId });
});
```

Rate Limiting for API Security

```
javascript
```

```
class RateLimiter {
  constructor(redis) {
    this.redis = redis;
  }

  async checkLimit(identifier, limit = 100, windowMs = 60000) {
    const key = `rate_limit:${identifier}`;
    const now = Date.now();
    const windowStart = now - windowMs;

    // Use Redis sorted set for sliding window
    const multi = this.redis.multi();

    // Remove old entries
    multi.zremrangebyscore(key, 0, windowStart);

    // Add current request
    multi.zadd(key, now, `${now}-${Math.random()}`);

    // Count requests in window
    multi.zcard(key);

    // Set expiry
    multi.expire(key, Math.ceil(windowMs / 1000));

    const results = await multi.exec();
    const count = results[2][1];  // Result of zcard

    if (count > limit) {
      const oldestEntry = await this.redis.zrange(key, 0, 0, 'WITHSCORES');
      const resetTime = parseInt(oldestEntry[1]) + windowMs;

      return {
        allowed: false,
        limit,
        remaining: 0,
        resetAt: resetTime
      };
    }

    return {
      allowed: true,
      limit,
      remaining: limit - count,
      resetAt: now + windowMs
    };
```

```
  }
}

// Middleware
const rateLimiter = new RateLimiter(redisClient);

async function rateLimitMiddleware(req, res, next) {
  // Rate limit by API key or IP
  const identifier = req.headers['x-api-key'] || req.ip;

  const result = await rateLimiter.checkLimit(identifier, 100, 60000);

  // Set headers
  res.setHeader('X-RateLimit-Limit', result.limit);
  res.setHeader('X-RateLimit-Remaining', result.remaining);
  res.setHeader('X-RateLimit-Reset', result.resetAt);

  if (!result.allowed) {
    res.setHeader('Retry-After', Math.ceil((result.resetAt - Date.now()) / 1000));
    return res.status(429).json({ error: 'Rate limit exceeded' });
  }

  next();
}

app.use('/api', rateLimitMiddleware);
```

---

5. Security Best Practices

SQL Injection Prevention

```
javascript
```

```javascript
// ❌ VULNERABLE to SQL injection
app.get('/users', (req, res) => {
  const userId = req.query.id;

  // NEVER do this!
  const query = `SELECT * FROM users WHERE id = ${userId}`;
  db.query(query);
});

// Attacker sends: ?id=1 OR 1=1
// Query becomes: SELECT * FROM users WHERE id = 1 OR 1=1
// Returns ALL users! ✗

// ✓ SAFE: Use parameterized queries
app.get('/users', async (req, res) => {
  const userId = req.query.id;

  // Parameterized query
  const result = await db.query(
    'SELECT * FROM users WHERE id = ?',
    [userId]  // Parameter safely escaped
  );

  res.json(result);
});
```

XSS (Cross-Site Scripting) Prevention

javascript

```javascript
// ❌ VULNERABLE to XSS
app.get('/search', (req, res) => {
  const query = req.query.q;

  // Directly rendering user input
  res.send(`<h1>Results for: ${query}</h1>`);
});

// Attacker sends: ?q=<script>alert('XSS')</script>
// Page executes attacker's script! ✗

// ✓ SAFE: Escape user input
const escapeHtml = require('escape-html');

app.get('/search', (req, res) => {
  const query = escapeHtml(req.query.q);

  res.send(`<h1>Results for: ${query}</h1>`);
});

// Or use templating engine that auto-escapes
app.set('view engine', 'ejs');

app.get('/search', (req, res) => {
  res.render('search', { query: req.query.q });  // Auto-escaped
});
```

CSRF (Cross-Site Request Forgery) Prevention

javascript

```
const csrf = require('csurf');
const cookieParser = require('cookie-parser');

app.use(cookieParser());

// CSRF protection
const csrfProtection = csrf({ cookie: true });

// Apply to state-changing operations
app.get('/form', csrfProtection, (req, res) => {
  // Send CSRF token to client
  res.render('form', { csrfToken: req.csrfToken() });
});

app.post('/transfer', csrfProtection, (req, res) => {
  // CSRF token automatically verified
  // If invalid, returns 403

  // Process transfer
  res.json({ success: true });
});

// Client must include CSRF token in form:
// <input type="hidden" name="_csrf" value="<%= csrfToken %>">
```

Input Validation

javascript

```
const Joi = require('joi');

// Define schemas
const userSchema = Joi.object({
  email: Joi.string().email().required(),
  password: Joi.string().min(8).required(),
  name: Joi.string().min(2).max(100),
  age: Joi.number().integer().min(0).max(150)
});

function validate(schema) {
  return (req, res, next) => {
    const { error, value } = schema.validate(req.body);

    if (error) {
      return res.status(400).json({
        error: 'Validation failed',
        details: error.details.map(d => ({
          field: d.path.join('.'),
          message: d.message
        }))
      });
    }

    // Replace req.body with validated value
    req.body = value;
    next();
  };
}

// Apply validation
app.post('/register', validate(userSchema), async (req, res) => {
  // req.body is now validated and sanitized
  const user = await createUser(req.body);
  res.json(user);
});
```

Key Takeaways

1. **Authentication:**
   - Password hashing (bcrypt, never plain text)
   - JWT for stateless authentication
   - OAuth for delegated authentication

- Multi-factor authentication

2. **Authorization:**
   - RBAC (role-based)
   - Check permissions on every request
   - Principle of least privilege
3. **Encryption:**
   - At rest: AES-256 for stored data
   - In transit: TLS/HTTPS always
   - Key management critical
4. **API Security:**
   - API keys for applications
   - Rate limiting
   - Input validation
   - CSRF protection
5. **Best Practices:**
   - Never trust user input
   - Defense in depth (layers)
   - Principle of least privilege
   - Security headers
   - Regular audits

Practice Problems

1. Design authentication system for banking app (high security requirements)
2. How would you securely store and transmit credit card information?
3. Design API authentication for public API (rate limits, quotas, abuse prevention)

Ready for Chapter 21: Real-Time Systems?