Table of Contents

# Bab 16: Security Best Practices di TypeScript

## Penjelasan Materi

Security Best Practices di TypeScript melibatkan penerapan praktik-praktik keamanan dalam pengembangan aplikasi. TypeScript menyediakan fitur-fitur yang dapat membantu mencegah kerentanan keamanan melalui type safety, namun kita juga perlu menerapkan praktik keamanan lainnya untuk melindungi aplikasi dari berbagai ancaman.

## Analogi yang Mudah Dipahami

Bayangkan security practices seperti sistem keamanan rumah: - Type Safety seperti kunci pintu yang hanya menerima kunci yang tepat - Input Validation seperti satpam yang memeriksa tamu sebelum masuk - Authentication seperti kartu akses ke area tertentu - Authorization seperti daftar orang yang diizinkan masuk ke ruangan khusus - Encryption seperti brankas untuk menyimpan barang berharga

## Point Penting

1. **Type Safety**
   * Strict type checking
   * No implicit any
   * Strict null checks
   * Type guards
2. **Input Validation**
   * Data sanitization
   * Type validation
   * Schema validation
   * Input boundaries
3. **Authentication & Authorization**
   * JWT handling
   * Session management
   * Role-based access
   * Permission checks
4. **Data Security**
   * Encryption
   * Hashing
   * Secure storage
   * Data masking
5. **Security Headers**
   * CORS
   * CSP
   * XSS Protection
   * CSRF Protection

## Contoh Kode dan Penjelasan

```typescript // 1. Type Safety Configuration // tsconfig.json { “compilerOptions”: { “strict”: true, “noImplicitAny”: true, “strictNullChecks”: true, “strictFunctionTypes”: true, “strictBindCallApply”: true, “strictPropertyInitialization”: true, “noImplicitThis”: true, “alwaysStrict”: true } }

// 2. Input Validation dengan Zod import { z } from ‘zod’;

const UserSchema = z.object({ username: z.string().min(3).max(50), email: z.string().email(), password: z.string().min(8).regex(/[A-Z]/).regex(/[0-9]/) });

type User = z.infer;

function createUser(data: unknown): User { const validatedData = UserSchema.parse(data); return validatedData; }

// 3. Secure Authentication interface TokenPayload { userId: string; role: string; exp: number; }

class AuthService { private readonly secretKey: string;

constructor(secretKey: string) {  
 this.secretKey = secretKey;  
}  
  
createToken(payload: Omit<TokenPayload, "exp">): string {  
 const token = {  
 ...payload,  
 exp: Date.now() + 24 \* 60 \* 60 \* 1000 // 24 hours  
 };  
   
 // Implementasi JWT signing  
 return "signed.jwt.token";  
}  
  
verifyToken(token: string): TokenPayload {  
 // Implementasi JWT verification  
 if (!this.isValidToken(token)) {  
 throw new Error("Invalid token");  
 }  
 return {} as TokenPayload;  
}  
  
private isValidToken(token: string): boolean {  
 // Implementasi validasi token  
 return true;  
}

}

// 4. Authorization Middleware type Permission = “read” | “write” | “admin”;

interface AuthenticatedRequest extends Request { user?: TokenPayload; }

function requirePermission(permission: Permission) { return function( req: AuthenticatedRequest, res: Response, next: NextFunction ) { if (!req.user) { return res.status(401).json({ error: “Unauthorized” }); }

if (!hasPermission(req.user, permission)) {  
 return res.status(403).json({ error: "Forbidden" });  
 }  
  
 next();  
};

}

function hasPermission(user: TokenPayload, permission: Permission): boolean { // Implementasi cek permission return true; }

// 5. Secure Password Handling import \* as bcrypt from ‘bcrypt’;

class PasswordService { private readonly SALT\_ROUNDS = 10;

async hashPassword(password: string): Promise<string> {  
 return bcrypt.hash(password, this.SALT\_ROUNDS);  
}  
  
async verifyPassword(  
 password: string,  
 hashedPassword: string  
): Promise<boolean> {  
 return bcrypt.compare(password, hashedPassword);  
}

}

// 6. Data Encryption import \* as crypto from ‘crypto’;

class EncryptionService { private readonly algorithm = ‘aes-256-gcm’; private readonly key: Buffer; private readonly iv: Buffer;

constructor(secretKey: string) {  
 this.key = crypto.scryptSync(secretKey, 'salt', 32);  
 this.iv = crypto.randomBytes(16);  
}  
  
encrypt(text: string): { encryptedData: string; tag: string } {  
 const cipher = crypto.createCipheriv(  
 this.algorithm,  
 this.key,  
 this.iv  
 );  
   
 let encrypted = cipher.update(text, 'utf8', 'hex');  
 encrypted += cipher.final('hex');  
   
 return {  
 encryptedData: encrypted,  
 tag: cipher.getAuthTag().toString('hex')  
 };  
}  
  
decrypt(  
 encryptedData: string,  
 tag: string  
): string {  
 const decipher = crypto.createDecipheriv(  
 this.algorithm,  
 this.key,  
 this.iv  
 );  
   
 decipher.setAuthTag(Buffer.from(tag, 'hex'));  
   
 let decrypted = decipher.update(encryptedData, 'hex', 'utf8');  
 decrypted += decipher.final('utf8');  
   
 return decrypted;  
}

}

// 7. Secure Headers Configuration import helmet from ‘helmet’; import express from ‘express’;

const app = express();

app.use(helmet()); app.use(helmet.contentSecurityPolicy({ directives: { defaultSrc: [“‘self’”], scriptSrc: [“‘self’”, “‘unsafe-inline’”], styleSrc: [“‘self’”, “‘unsafe-inline’”], imgSrc: [“‘self’”, “data:”, “https:”], connectSrc: [“‘self’”], fontSrc: [“‘self’”], objectSrc: [“‘none’”], mediaSrc: [“‘self’”], frameSrc: [“‘none’”] } }));

// 8. CSRF Protection import csurf from ‘csurf’;

const csrfProtection = csurf({ cookie: true });

app.post(‘/api/data’, csrfProtection, (req, res) => { // Protected route });

// 9. Rate Limiting import rateLimit from ‘express-rate-limit’;

const limiter = rateLimit({ windowMs: 15 \* 60 \* 1000, // 15 minutes max: 100 // limit each IP to 100 requests per windowMs });

app.use(‘/api/’, limiter);

// 10. Secure File Upload interface FileValidationResult { isValid: boolean; error?: string; }

class FileValidator { private readonly ALLOWED\_TYPES = [‘image/jpeg’, ‘image/png’]; private readonly MAX\_SIZE = 5 \* 1024 \* 1024; // 5MB

validateFile(file: Express.Multer.File): FileValidationResult {  
 if (!this.ALLOWED\_TYPES.includes(file.mimetype)) {  
 return {  
 isValid: false,  
 error: "Invalid file type"  
 };  
 }  
  
 if (file.size > this.MAX\_SIZE) {  
 return {  
 isValid: false,  
 error: "File too large"  
 };  
 }  
  
 return { isValid: true };  
}  
  
sanitizeFilename(filename: string): string {  
 return filename  
 .replace(/[^a-zA-Z0-9.-]/g, '')  
 .replace(/\.{2,}/g, '.');  
}

} ```

## Cara Kerja Security Practices

1. **Prevention**:
   * Input validation
   * Type checking
   * Access control
   * Data sanitization
2. **Detection**:
   * Logging
   * Monitoring
   * Auditing
   * Error tracking
3. **Response**:
   * Error handling
   * Security headers
   * Rate limiting
   * Incident response

## Tips dan Trik

1. **Secure Configuration**

* // ✅ Gunakan environment variables  
  const config = {  
   jwtSecret: process.env.JWT\_SECRET,  
   dbConnection: process.env.DB\_CONNECTION,  
   apiKeys: process.env.API\_KEYS?.split(',')  
  };  
    
  // Validasi konfigurasi  
  if (!config.jwtSecret) {  
   throw new Error('JWT\_SECRET is required');  
  }

1. **Safe Type Assertions**

* // ✅ Gunakan type guards  
  function isUser(obj: unknown): obj is User {  
   return (  
   typeof obj === 'object' &&  
   obj !== null &&  
   'id' in obj &&  
   'username' in obj  
   );  
  }  
    
  function processUser(data: unknown) {  
   if (!isUser(data)) {  
   throw new Error('Invalid user data');  
   }  
   // Safe to use data as User  
  }

1. **Secure Error Handling**

* // ✅ Jangan ekspos detail error ke client  
  class ApiError extends Error {  
   constructor(  
   public statusCode: number,  
   public message: string,  
   public internal?: Error  
   ) {  
   super(message);  
   }  
    
   toResponse() {  
   return {  
   error: this.message,  
   status: this.statusCode  
   };  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Unsafe Type Assertions**

* // ❌ Buruk: Type assertion tidak aman  
  const userData = JSON.parse(input) as User;  
    
  // ✅ Baik: Validasi data  
  const userData = UserSchema.parse(JSON.parse(input));

1. **Hardcoded Secrets**

* // ❌ Buruk: Secret hardcoded  
  const API\_KEY = "1234567890";  
    
  // ✅ Baik: Gunakan environment variables  
  const API\_KEY = process.env.API\_KEY;  
  if (!API\_KEY) {  
   throw new Error("API\_KEY is required");  
  }

1. **Insufficient Input Validation**

* // ❌ Buruk: Validasi minimal  
  app.post('/api/users', (req, res) => {  
   const user = req.body;  
   // Process user  
  });  
    
  // ✅ Baik: Validasi lengkap  
  app.post('/api/users', (req, res) => {  
   try {  
   const user = UserSchema.parse(req.body);  
   // Process validated user  
   } catch (error) {  
   res.status(400).json({  
   error: "Invalid input"  
   });  
   }  
  });

### Solusi:

1. Implementasi strict type checking
2. Gunakan schema validation
3. Terapkan proper authentication
4. Enkripsi data sensitif
5. Implementasi security headers