

Node.js Security Interview Questions - All Levels

A comprehensive collection of security interview questions for Node.js developers, organized by experience level.

Table of Contents

1. [Junior Level Questions](#)
 2. [Middle Level Questions](#)
 3. [Senior Level Questions](#)
 4. [Practical Scenarios](#)
-

Junior Level Questions

1. What is HTTPS and why is it important?

Answer:

HTTPS (HTTP Secure) is the secure version of HTTP that encrypts data between client and server using TLS/SSL.

Importance:

- Protects sensitive data (passwords, credit cards)

- Prevents man-in-the-middle attacks
- Improves SEO rankings
- Required for modern browser features

2. What are environment variables and why should secrets not be hardcoded?

Answer:

Environment variables store configuration outside the code.

Why not hardcode:

- Secrets in code can be exposed in version control
- Different environments need different credentials
- Makes secret rotation difficult
- Violates security best practices

```
// Bad
const apiKey = 'sk_live_12345...';

// Good
const apiKey = process.env.API_KEY;
```

3. What is SQL Injection and how do you prevent it?

Answer:

SQL Injection occurs when malicious SQL code is inserted into queries.

Prevention:

```
// Vulnerable
db.query(`SELECT * FROM users WHERE id =
${req.params.id}`);

// Safe - Using parameterized queries
db.query('SELECT * FROM users WHERE id = ?',
[req.params.id]);

// With ORM (Sequelize)
User.findById(req.params.id); // Automatically safe
```

4. What is XSS (Cross-Site Scripting)?

Answer:

XSS allows attackers to inject malicious scripts into web pages viewed by other users.

Types:

- **Stored XSS:** Malicious script stored in database
- **Reflected XSS:** Script reflected from user input
- **DOM-based XSS:** Client-side script manipulation

Prevention:

- Sanitize user input
- Use Content Security Policy (CSP)
- Escape output
- Use secure templating engines

5. What is the purpose of the `helmet` package?

Answer:

Helmet helps secure Express apps by setting various HTTP security headers.

```
const helmet = require('helmet');
app.use(helmet());

// Sets headers like:
// - X-Content-Type-Options: nosniff
// - X-Frame-Options: SAMEORIGIN
// - Strict-Transport-Security
// - X-XSS-Protection
```

6. What is CORS and why is it important?

Answer:

CORS (Cross-Origin Resource Sharing) controls which domains can access your API.

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

// Allow specific origin
app.use(cors({
  origin: 'https://myapp.com',
  credentials: true
}));
```

7. What is password hashing and which libraries should you use?

Answer:

Hashing converts passwords into irreversible strings.

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

// Hash password
const hash = await bcrypt.hash(password, 10);

// Verify password
const isValid = await bcrypt.compare(password, hash);
```

Never use:

- Plain text storage
- MD5 or SHA1 for passwords
- Weak hashing algorithms

8. What is the difference between authentication and authorization?

Answer:

- **Authentication:** Verifying who you are (login)
- **Authorization:** Verifying what you can access (permissions)

```
// Authentication
const token = jwt.sign({ userId: user.id }, secret);
```

```
// Authorization
const hasPermission = user.role === 'admin';
```

9. What are JWT tokens and what do they contain?

Answer:

JWT (JSON Web Token) is a secure way to transmit information between parties.

Structure:

```
header.payload.signature
```

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

// Create token
const token = jwt.sign(
  { userId: 123, email: 'user@example.com' },
  process.env.JWT_SECRET,
  { expiresIn: '1h' }
);

// Verify token
const decoded = jwt.verify(token,
  process.env.JWT_SECRET);
```

10. Why should you validate user input?

Answer:

Input validation prevents security vulnerabilities and data corruption.

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

const schema = Joi.object({
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(0).max(120)
});

const { error, value } = schema.validate(req.body);
```

Middle Level Questions

11. Explain rate limiting and implement it in Express

Answer:

Rate limiting restricts the number of requests a client can make.

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per
windowMs
  message: 'Too many requests, please try again later'
});

app.use('/api/', limiter);

// Stricter for login
const loginLimiter = rateLimit({
```

```
windowMs: 15 * 60 * 1000,  
max: 5,  
skipSuccessfulRequests: true  
});  
  
app.post('/login', loginLimiter, loginController);
```

12. What is CSRF and how do you prevent it?

Answer:

CSRF (Cross-Site Request Forgery) tricks users into performing unwanted actions.

```
const csrf = require('csrf');  
const csrfProtection = csrf({ cookie: true });  
  
app.get('/form', csrfProtection, (req, res) => {  
  res.render('form', { csrfToken: req.csrfToken() });  
});  
  
app.post('/submit', csrfProtection, (req, res) => {  
  // Process form  
});
```

Prevention methods:

- CSRF tokens
- SameSite cookies
- Double submit cookies
- Custom request headers

13. How do you implement secure session management?

Answer:

```
const session = require('express-session');
const RedisStore = require('connect-redis')(session);

app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: true, // HTTPS only
    httpOnly: true, // No JavaScript access
    maxAge: 1000 * 60 * 60 * 24, // 24 hours
    sameSite: 'strict' // CSRF protection
  }
}));
```

Best practices:

- Use secure, random session IDs
- Store sessions server-side (Redis, database)
- Set appropriate expiration
- Regenerate session on login
- Implement session timeout

14. What is NoSQL Injection and how do you prevent it?

Answer:

NoSQL injection exploits NoSQL database queries.

```
// Vulnerable MongoDB query
User.find({ username: req.body.username });

// Attack: { "username": { "$gt": "" } } returns all
users

// Prevention 1: Type checking
const username = String(req.body.username);
User.find({ username });

// Prevention 2: Sanitization
const mongoSanitize = require('express-mongo-sanitize');
app.use(mongoSanitize());

// Prevention 3: Validation
const schema = Joi.object({
  username: Joi.string().alphanum().required()
});
```

15. How do you securely store and handle API keys?

Answer:

```
// .env file
API_KEY=your_secret_key
API_SECRET=your_secret
```

```
// Load with dotenv
require('dotenv').config();

// Use in code
const apiKey = process.env.API_KEY;

// Never log secrets
console.log(`API Key: ${apiKey.substring(0, 4)}...`);
// Only show first 4 chars

// For production
// - Use secret management services (AWS Secrets
// Manager, HashiCorp Vault)
// - Rotate keys regularly
// - Use different keys per environment
// - Implement key rotation strategies
```

16. Explain JWT security best practices

Answer:

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

// Best practices:
// 1. Use strong secret
const secret = crypto.randomBytes(64).toString('hex');

// 2. Set appropriate expiration
const accessToken = jwt.sign(
  { userId: user.id },
  process.env.ACCESS_TOKEN_SECRET,
```

```
{ expiresIn: '15m' } // Short-lived
);

// 3. Use refresh tokens
const refreshToken = jwt.sign(
  { userId: user.id },
  process.env.REFRESH_TOKEN_SECRET,
  { expiresIn: '7d' }
);

// 4. Don't store sensitive data
// Bad: { userId: 1, password: 'hash', ssn: '123-45-6789' }
// Good: { userId: 1, role: 'user' }

// 5. Verify algorithm
jwt.verify(token, secret, { algorithms: ['HS256'] });

// 6. Implement token blacklist for logout
// Store revoked tokens in Redis with expiration
```

17. How do you implement proper error handling without exposing sensitive information?

Answer:

```
// Custom error class
class AppError extends Error {
  constructor(message, statusCode, isOperational = true) {
    super(message);
  }
}
```

```
this.statusCode = statusCode;
this.isOperational = isOperational;
}
}

// Error handling middleware
app.use((err, req, res, next) => {
  // Log full error for debugging
  console.error(err.stack);

  // Send safe error to client
  if (process.env.NODE_ENV === 'production') {
    // Don't expose internal errors
    if (err.isOperational) {
      res.status(err.statusCode).json({
        status: 'error',
        message: err.message
      });
    } else {
      // Programming or unknown errors
      res.status(500).json({
        status: 'error',
        message: 'Something went wrong'
      });
    }
  } else {
    // Development: show full error
    res.status(err.statusCode || 500).json({
      status: 'error',
      message: err.message,
      stack: err.stack
    });
  }
});
```

```

    }
  });
};

```

18. What is Content Security Policy (CSP) and how do you implement it?

Answer:

CSP prevents XSS by controlling resource loading.

```

const helmet = require('helmet');

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'"],
      scriptSrc: ["'self'", 'trusted-cdn.com'],
      imgSrc: ["'self'", 'data:', 'https:'],
      connectSrc: ["'self'", 'api.example.com'],
      fontSrc: ["'self'", 'fonts.gstatic.com'],
      objectSrc: ["'none'"],
      upgradeInsecureRequests: []
    }
  })
);

```

19. How do you prevent timing attacks in authentication?

Answer:

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

// Vulnerable: Direct comparison
if (userToken === providedToken) {
  // Different response times reveal information
}

// Secure: Constant-time comparison
function safeCompare(a, b) {
  const bufA = Buffer.from(a);
  const bufB = Buffer.from(b);

  if (bufA.length !== bufB.length) {
    return false;
  }

  return crypto.timingSafeEqual(bufA, bufB);
}

// Use bcrypt for passwords (built-in protection)
const isValid = await bcrypt.compare(password, hash);
```

20. Explain dependency security and how to manage it

Answer:

```
# Check for vulnerabilities
npm audit

# Fix automatically
npm audit fix
```

```
# Check for outdated packages
npm outdated

# Use tools
npm install -g snyk
snyk test

# Keep dependencies updated
npm update

# Lock file security
# Always commit package-lock.json
```

Best practices:

- Regularly update dependencies
- Use `npm audit` in CI/CD
- Review dependency changes
- Use minimal dependencies
- Pin major versions
- Monitor security advisories

Senior Level Questions

21. Design a comprehensive authentication and authorization system

Answer:

```
// Multi-layer security architecture

// 1. Authentication Layer
class AuthService {
  async register(email, password) {
    // Validate input
    const validation = await this.validateInput({
email, password });

    // Check existing user
    const exists = await User.findByEmail(email);
    if (exists) throw new AppError('Email already
registered', 400);

    // Hash password with salt
    const salt = await bcrypt.genSalt(12);
    const hashedPassword = await bcrypt.hash(password,
salt);

    // Create user with email verification token
    const verificationToken =
crypto.randomBytes(32).toString('hex');
    const user = await User.create({
      email,
      password: hashedPassword,
      verificationToken,
      verificationExpires: Date.now() + 24 * 60 * 60 *
1000
    });
  }
};
```

```
// Send verification email
await this.sendVerificationEmail(email,
verificationToken);

return { message: 'Registration successful. Please
verify your email.' };
}

async login(email, password, ip, userAgent) {
  // Rate limiting check
  await this.checkRateLimit(ip, email);

  // Find user
  const user = await
User.findByEmail(email).select('+password');
  if (!user) {
    await this.logFailedAttempt(email, ip);
    throw new AppError('Invalid credentials', 401);
  }

  // Check if account is locked
  if (user.isLocked()) {
    throw new AppError('Account locked. Try again
later.', 423);
  }

  // Verify password
  const isValid = await bcrypt.compare(password,
user.password);
  if (!isValid) {
    await user.incrementLoginAttempts();
    throw new AppError('Invalid credentials', 401);
  }
}
```

```
}

// Check 2FA if enabled
if (user.twoFactorEnabled) {
  const tempToken = await
this.createTempToken(user.id);
  return { requires2FA: true, tempToken };
}

// Reset login attempts
await user.resetLoginAttempts();

// Generate tokens
const tokens = await this.generateTokenPair(user);

// Log successful login
await this.logSuccessfulLogin(user.id, ip,
userAgent);

// Store refresh token
await this.storeRefreshToken(user.id,
tokens.refreshToken);

return tokens;
}

async generateTokenPair(user) {
  const accessToken = jwt.sign(
    {
      userId: user.id,
      email: user.email,
      role: user.role,
```

```

        permissions: user.permissions
      },
      process.env.ACCESS_TOKEN_SECRET,
      { expiresIn: '15m', algorithm: 'HS256' }
    );

    const refreshToken = jwt.sign(
      { userId: user.id, tokenVersion:
user.tokenVersion },
      process.env.REFRESH_TOKEN_SECRET,
      { expiresIn: '7d', algorithm: 'HS256' }
    );

    return { accessToken, refreshToken };
  }
}

// 2. Authorization Layer
class AuthorizationMiddleware {
  requireAuth(req, res, next) {
    try {
      const token = req.headers.authorization?.split('
')[1];
      if (!token) {
        throw new AppError('No token provided', 401);
      }

      // Check token blacklist
      if (this.isTokenBlacklisted(token)) {
        throw new AppError('Token has been revoked',
401);
      }
    }
  }
}

```

```

const decoded = jwt.verify(
  token,
  process.env.ACCESS_TOKEN_SECRET,
  { algorithms: ['HS256'] }
);

req.user = decoded;
next();
} catch (error) {
  next(new AppError('Invalid token', 401));
}
}

requireRole(...roles) {
  return (req, res, next) => {
    if (!req.user) {
      return next(new AppError('Not authenticated',
401));
    }

    if (!roles.includes(req.user.role)) {
      return next(new AppError('Insufficient
permissions', 403));
    }

    next();
  };
}

requirePermission(...permissions) {
  return async (req, res, next) => {

```

```

    const user = await
User.findById(req.user.userId);

    const hasPermission =
permissions.every(permission =>
    user.permissions.includes(permission)
);

    if (!hasPermission) {
        return next(new AppError('Insufficient
permissions', 403));
    }

    next();
};
}
}

// 3. Two-Factor Authentication
class TwoFactorService {
    async enable2FA(userId) {
        const secret = speakeasy.generateSecret({
            name: `MyApp (${user.email})`
        });

        await User.updateById(userId, {
            twoFactorSecret: secret.base32,
            twoFactorEnabled: false // Enable after
verification
        });

        // Generate QR code

```

```

    const qrCode = await
QRCode.toDataURL(secret.otppauth_url);

    return { secret: secret.base32, qrCode };
}

async verify2FA(userId, token) {
    const user = await User.findById(userId);

    const verified = speakeasy.totp.verify({
        secret: user.twoFactorSecret,
        encoding: 'base32',
        token,
        window: 2 // Allow 2 time steps variance
    });

    if (!verified) {
        throw new AppError('Invalid 2FA code', 401);
    }

    return true;
}
}

// 4. Account Security Features
class AccountSecurityService {
    async changePassword(userId, currentPassword,
newPassword) {
        const user = await
User.findById(userId).select('+password');

        // Verify current password

```

```
const isValid = await
bcrypt.compare(currentPassword, user.password);
if (!isValid) {
  throw new AppError('Current password is
incorrect', 401);
}

// Check password history (prevent reuse)
const isInHistory = await
this.checkPasswordHistory(userId, newPassword);
if (isInHistory) {
  throw new AppError('Cannot reuse recent
passwords', 400);
}

// Hash new password
const hash = await bcrypt.hash(newPassword, 12);

// Update password and increment token version
await User.updateById(userId, {
  password: hash,
  tokenVersion: user.tokenVersion + 1,
  passwordChangedAt: new Date()
});

// Add to password history
await this.addToPasswordHistory(userId, hash);

// Invalidate all existing tokens
await this.invalidateAllUserTokens(userId);

// Notify user
```

```
    await this.sendPasswordChangedEmail(user.email);

    return { message: 'Password changed successfully'
};
}

async setupPasswordPolicy() {
    return {
        minLength: 12,
        requireUppercase: true,
        requireLowercase: true,
        requireNumbers: true,
        requireSpecialChars: true,
        maxAge: 90 * 24 * 60 * 60 * 1000, // 90 days
        preventReuse: 5 // Last 5 passwords
    };
}
}

// Usage
app.post('/register', authController.register);
app.post('/login', authController.login);
app.post('/refresh', authController.refreshToken);
app.post('/logout', auth.requireAuth,
authController.logout);

// Protected routes
app.get('/admin',
    auth.requireAuth,
    auth.requireRole('admin'),
    adminController.dashboard
);
```

```
app.post('/users/:id/delete',
  auth.requireAuth,
  auth.requirePermission('users:delete'),
  userController.delete
);
```

22. How do you handle secrets in a microservices architecture?

Answer:

```
// 1. Secret Management Service Integration
class SecretManager {
  constructor() {
    this.cache = new Map();
    this.cacheTimeout = 5 * 60 * 1000; // 5 minutes
  }

  async getSecret(secretName) {
    // Check cache
    const cached = this.cache.get(secretName);
    if (cached && cached.expiresAt > Date.now()) {
      return cached.value;
    }

    // Fetch from secret manager
    let secret;

    if (process.env.USE_AWS_SECRETS) {
      secret = await this.getFromAWS(secretName);
    }
  }
}
```

```

    } else if (process.env.USE_VAULT) {
        secret = await this.getFromVault(secretName);
    } else {
        // Local development
        secret = process.env[secretName];
    }

    // Cache with expiration
    this.cache.set(secretName, {
        value: secret,
        expiresAt: Date.now() + this.cacheTimeout
    });

    return secret;
}

async getFromAWS(secretName) {
    const AWS = require('aws-sdk');
    const client = new AWS.SecretsManager({
        region: process.env.AWS_REGION
    });

    const data = await client.getSecretValue({
        SecretId: secretName
    }).promise();

    return JSON.parse(data.SecretString);
}

async getFromVault(secretPath) {
    const vault = require('node-vault')({
        endpoint: process.env.VAULT_ADDR,

```

```

        token: process.env.VAULT_TOKEN
    });

    const result = await vault.read(secretPath);
    return result.data;
}

async rotateSecret(secretName, newValue) {
    // Update in secret manager
    await this.updateInSecretManager(secretName,
newValue);

    // Invalidate cache
    this.cache.delete(secretName);

    // Notify other services
    await this.notifySecretRotation(secretName);
}
}

// 2. Service-to-Service Authentication
class ServiceAuthManager {
    async generateServiceToken(serviceName) {
        return jwt.sign(
            {
                service: serviceName,
                type: 'service',
                permissions:
this.getServicePermissions(serviceName)
            },
            await secretManager.getSecret('SERVICE_SECRET'),
            { expiresIn: '1h' }

```

```

    );
}

async verifyServiceToken(token) {
  try {
    const decoded = jwt.verify(
      token,
      await secretManager.getSecret('SERVICE_SECRET')
    );

    if (decoded.type !== 'service') {
      throw new Error('Invalid service token');
    }

    return decoded;
  } catch (error) {
    throw new AppError('Service authentication
failed', 401);
  }
}

// 3. mTLS for Service Communication
class MTLSClient {
  constructor() {
    this.httpsAgent = new https.Agent({
      cert: fs.readFileSync('/etc/certs/service.crt'),
      key: fs.readFileSync('/etc/certs/service.key'),
      ca: fs.readFileSync('/etc/certs/ca.crt'),
      rejectUnauthorized: true
    });
  }
}

```

```

    async callService(url, data) {
      return axios.post(url, data, {
        httpsAgent: this.httpsAgent
      });
    }
  }
}

```

23. Implement a secure file upload system with validation

Answer:

```

const multer = require('multer');
const path = require('path');
const crypto = require('crypto');
const sharp = require('sharp');

class SecureFileUpload {
  constructor() {
    this.allowedMimeTypes = {
      images: ['image/jpeg', 'image/png', 'image/gif',
        'image/webp'],
      documents: ['application/pdf',
        'application/msword']
    };
  }

  this.maxFileSize = 5 * 1024 * 1024; // 5MB
}

createUploadMiddleware() {
  const storage = multer.diskStorage({

```

```

    destination: (req, file, cb) => {
      const uploadPath = path.join(__dirname,
'../uploads', req.user.id);
      fs.mkdirSync(uploadPath, { recursive: true });
      cb(null, uploadPath);
    },
    filename: (req, file, cb) => {
      // Generate secure random filename
      const randomName =
crypto.randomBytes(16).toString('hex');
      const ext = path.extname(file.originalname);
      cb(null, `${randomName}${ext}`);
    }
  });

```

```

return multer({
  storage,
  limits: {
    fileSize: this.maxFileSize,
    files: 5
  },
  fileFilter: (req, file, cb) => {
    this.validateFile(file, cb);
  }
});

```

```

validateFile(file, cb) {
  // 1. Check MIME type
  const mimeType = file.mimetype;
  const allowedTypes = [
    ...this.allowedMimeTypes.images,

```

```

        ...this.allowedMimeTypes.documents
    ];

    if (!allowedTypes.includes(mimeType)) {
        return cb(new AppError('Invalid file type',
400));
    }

    // 2. Check file extension
    const ext =
path.extname(file.originalname).toLowerCase();
    const allowedExts = ['.jpg', '.jpeg', '.png',
'.gif', '.webp', '.pdf'];

    if (!allowedExts.includes(ext)) {
        return cb(new AppError('Invalid file extension',
400));
    }

    // 3. Sanitize filename
    const sanitized = file.originalname.replace(/[^a-
zA-Z0-9.-]/g, '_');
    file.originalname = sanitized;

    cb(null, true);
}

async processUploadedFile(file, type) {
    // 1. Scan for malware (use antivirus service)
    await this.scanForMalware(file.path);

    // 2. Verify file content matches MIME type

```

```
    const fileType = await
this.detectFileType(file.path);
    if (fileType !== file.mimetype) {
        fs.unlinkSync(file.path);
        throw new AppError('File content does not match
extension', 400);
    }

    // 3. Process based on file type
    if
(this.allowedMimeTypes.images.includes(file.mimetype))
{
        await this.processImage(file.path);
    }

    // 4. Generate CDN URL or signed URL
    const url = await this.uploadToStorage(file.path);

    // 5. Save metadata to database
    const fileRecord = await File.create({
        userId: file.userId,
        originalName: file.originalname,
        filename: file.filename,
        mimetype: file.mimetype,
        size: file.size,
        url,
        uploadedAt: new Date()
    });

    // 6. Delete local file
    fs.unlinkSync(file.path);
```

```
    return fileRecord;
}

async processImage(imagePath) {
    // Strip EXIF data for privacy
    // Resize and optimize
    await sharp(imagePath)
        .resize(2000, 2000, {
            fit: 'inside',
            withoutEnlargement: true
        })
        .jpeg({ quality: 85 })
        .toFile(imagePath + '.processed');

    // Replace original with processed
    fs.renameSync(imagePath + '.processed', imagePath);
}

async uploadToStorage(filePath) {
    // Upload to S3 with encryption
    const s3 = new AWS.S3();
    const fileStream = fs.createReadStream(filePath);
    const uploadParams = {
        Bucket: process.env.S3_BUCKET,
        Key: path.basename(filePath),
        Body: fileStream,
        ServerSideEncryption: 'AES256',
        ACL: 'private'
    };

    const result = await
s3.upload(uploadParams).promise();
```

```
    return result.Location;
}

async generateSignedUrl(fileId, expiresIn = 3600) {
    const file = await File.findById(fileId);

    const s3 = new AWS.S3();
    const url = s3.getSignedUrl('getObject', {
        Bucket: process.env.S3_BUCKET,
        Key: file.filename,
        Expires: expiresIn
    });

    return url;
}

async scanForMalware(filePath) {
    // Integrate with ClamAV or cloud antivirus service
    // This is a placeholder
    return true;
}

async detectFileType(filePath) {
    const { fileTypeFromFile } = await import('file-type');
    const type = await fileTypeFromFile(filePath);
    return type?.mime;
}
}

// Usage
const fileUpload = new SecureFileUpload();
```

```

const upload = fileUpload.createUploadMiddleware();

app.post('/upload',
  auth.requireAuth,
  upload.array('files', 5),
  async (req, res, next) => {
    try {
      const uploadedFiles = await Promise.all(
        req.files.map(file =>
fileUpload.processUploadedFile(file))
      );

      res.json({
        status: 'success',
        files: uploadedFiles
      });
    } catch (error) {
      next(error);
    }
  }
);

```

24. Design a comprehensive API security strategy

Answer:

```

// 1. API Gateway with multiple security layers
class APIGateway {
  constructor() {
    this.setupSecurityMiddleware();
  }
}

```

```

setupSecurityMiddleware() {
  // Layer 1: Basic security headers
  this.app.use(helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        styleSrc: ["'self'", "'unsafe-inline'"],
        scriptSrc: ["'self'"],
        imgSrc: ["'self'", 'data:', 'https:']
      }
    },
    hsts: {
      maxAge: 31536000,
      includeSubDomains: true,
      preload: true
    }
  }));

  // Layer 2: CORS with strict origin checking
  this.app.use(cors({
    origin: (origin, callback) => {
      const allowedOrigins =
process.env.ALLOWED_ORIGINS.split(',');
      if (!origin || allowedOrigins.includes(origin))
{
        callback(null, true);
      } else {
        callback(new Error('Not allowed by CORS'));
      }
    },
    credentials: true,
  }));
}

```

```

        methods: ['GET', 'POST', 'PUT', 'DELETE',
'PATCH'],
        allowedHeaders: ['Content-Type',
'Authorization'],
        exposedHeaders: ['X-Total-Count'],
        maxAge: 86400
    )));

// Layer 3: Request sanitization
this.app.use(express.json({ limit: '10kb' }));
this.app.use(mongoSanitize());
this.app.use(xss());

// Layer 4: Rate limiting (multiple tiers)
this.setupRateLimiting();

// Layer 5: API key/token validation
this.setupAuthenticationStrategies();

// Layer 6: Request validation
this.setupRequestValidation();

// Layer 7: Audit logging
this.setupAuditLogging();
}

setupRateLimiting() {
    const RedisStore = require('rate-limit-redis');
    const redisClient = require('./redis');

    // Global rate limit
    const globalLimiter = rateLimit({

```

```
    store: new RedisStore({ client: redisClient }),
    windowMs: 15 * 60 * 1000,
    max: 100,
    message: 'Too many requests from this IP'
  });

// Auth endpoints - stricter
const authLimiter = rateLimit({
  store: new RedisStore({ client: redisClient }),
  windowMs: 15 * 60 * 1000,
  max: 5,
  skipSuccessfulRequests: true
});

// Expensive operations - very strict
const expensiveLimiter = rateLimit({
  store: new RedisStore({ client: redisClient }),
  windowMs: 60 * 60 * 1000,
  max: 10
});

this.app.use('/api/', globalLimiter);
this.app.use('/api/auth/', authLimiter);
this.app.use('/api/reports/', expensiveLimiter);

// Per-user rate limiting
this.app.use(this.perUserRateLimit());
}

perUserRateLimit() {
  const limits = new Map();
```

```

return async (req, res, next) => {
  if (!req.user) return next();

  const userId = req.user.userId;
  const now = Date.now();
  const windowMs = 60 * 1000; // 1 minute
  const maxRequests = req.user.role === 'premium' ?
1000 : 100;

  if (!limits.has(userId)) {
    limits.set(userId, { count: 0, resetTime: now +
windowMs });
  }

  const userLimit = limits.get(userId);

  if (now > userLimit.resetTime) {
    userLimit.count = 0;
    userLimit.resetTime = now + windowMs;
  }

  userLimit.count++;

  if (userLimit.count > maxRequests) {
    return res.status(429).json({
      error: 'Rate limit exceeded for your account
tier'
    });
  }

  next();
};

```

```

}

setupAuthenticationStrategies() {
  // Strategy 1: JWT Bearer token
  const jwtAuth = async (req, res, next) => {
    const token = req.headers.authorization?.split('
')[1];
    if (!token) return next();

    try {
      const decoded = jwt.verify(token,
process.env.JWT_SECRET);
      req.user = decoded;
      req.authMethod = 'jwt';
    } catch (error) {
      // Invalid token - don't set user
    }

    next();
  };

  // Strategy 2: API Key
  const apiKeyAuth = async (req, res, next) => {
    const apiKey = req.headers['x-api-key'];
    if (!apiKey) return next();

    try {
      // Hash the API key
      const hashedKey = crypto
        .createHash('sha256')
        .update(apiKey)
        .digest('hex');
    }
  };
}

```

```
// Look up in database
const key = await APIKey.findOne({
  hashedKey,
  active: true,
  expiresAt: { $gt: new Date() }
});

if (key) {
  req.user = {
    userId: key.userId,
    role: 'api',
    permissions: key.permissions
  };
  req.authMethod = 'apiKey';

  // Update last used
  key.lastUsedAt = new Date();
  await key.save();
}
} catch (error) {
  // Invalid API key - don't set user
}

next();
};

// Strategy 3: OAuth 2.0
const oauthAuth = async (req, res, next) => {
  // Implement OAuth validation
  next();
};
```

```
// Apply all strategies
this.app.use(jwtAuth);
this.app.use(apiKeyAuth);
}

setupRequestValidation() {
  // Validate all requests against schemas
  this.app.use((req, res, next) => {
    // Get schema for route
    const schema = this.getSchemaForRoute(req.path,
req.method);

    if (schema) {
      const { error } = schema.validate(req.body);
      if (error) {
        return res.status(400).json({
          status: 'error',
          message: 'Validation error',
          details: error.details
        });
      }
    }

    next();
  });
}

setupAuditLogging() {
  this.app.use((req, res, next) => {
    const startTime = Date.now();
```

```

// Log after response
res.on('finish', async () => {
  const duration = Date.now() - startTime;

  await AuditLog.create({
    userId: req.user?.userId,
    method: req.method,
    path: req.path,
    statusCode: res.statusCode,
    duration,
    ip: req.ip,
    userAgent: req.headers['user-agent'],
    authMethod: req.authMethod,
    timestamp: new Date()
  });

  // Alert on suspicious activity
  if (res.statusCode === 401 || res.statusCode
=== 403) {
    await this.checkForAttackPatterns(req);
  }
});

next();
});
}

async checkForAttackPatterns(req) {
  const ip = req.ip;
  const recentFailures = await
AuditLog.countDocuments({
    ip,

```

```

        statusCode: { $in: [401, 403, 429] },
        timestamp: { $gt: new Date(Date.now() - 15 * 60 *
1000) }
    });

    if (recentFailures > 20) {
        // Block IP temporarily
        await this.blockIP(ip, 60 * 60 * 1000); // 1 hour

        // Send alert
        await this.sendSecurityAlert({
            type: 'potential_attack',
            ip,
            failures: recentFailures
        });
    }
}

```

// 2. Input Validation Framework

```

class ValidationFramework {
    static schemas = {
        'POST:/api/users': Joi.object({
            email: Joi.string().email().required(),
            password: Joi.string().min(12).required(),
            name: Joi.string().min(2).max(100).required()
        }),

        'PUT:/api/users/:id': Joi.object({
            email: Joi.string().email(),
            name: Joi.string().min(2).max(100)
        }).min(1),
    }
}

```

```

    'POST:/api/posts': Joi.object({
      title: Joi.string().min(1).max(200).required(),
      content:
Joi.string().min(1).max(10000).required(),
      tags: Joi.array().items(Joi.string()).max(10)
    })
  };

  static getSchema(route, method) {
    const key = `${method}:${route}`;
    return this.schemas[key];
  }
}

```

// 3. API Versioning Security

```

class APIVersioning {
  constructor() {
    // Different security requirements per version
    this.versionSecurity = {
      'v1': {
        deprecated: true,
        minTokenExpiry: 900, // 15 min
        allowedMethods: ['GET'] // Read-only
      },
      'v2': {
        minTokenExpiry: 3600, // 1 hour
        requiresEnhancedAuth: false
      },
      'v3': {
        minTokenExpiry: 3600,
        requiresEnhancedAuth: true,

```

```

        requires2FA: true
    }
};
}

versionMiddleware() {
    return (req, res, next) => {
        const version = req.headers['api-version'] ||
'v1';
        const security = this.versionSecurity[version];

        if (!security) {
            return res.status(400).json({
                error: 'Invalid API version'
            });
        }

        if (security.deprecated) {
            res.set('X-API-Deprecated', 'true');
            res.set('X-API-Sunset', '2024-12-31');
        }

        if (security.allowedMethods &&
!security.allowedMethods.includes(req.method)) {
            return res.status(405).json({
                error: 'Method not allowed in this API
version'
            });
        }

        req.apiVersion = version;
    }
}

```

```

    req.apiSecurity = security;
    next();
  };
}
}

```

25. Explain encryption strategies for data at rest and in transit

Answer:

```

// 1. Data in Transit - TLS Configuration
const https = require('https');
const fs = require('fs');

class SecureServer {
  createHTTPSServer() {
    const options = {
      key: fs.readFileSync('/path/to/private-key.pem'),
      cert:
fs.readFileSync('/path/to/certificate.pem'),
      ca: fs.readFileSync('/path/to/ca-cert.pem'),

      // TLS Configuration
      minVersion: 'TLSv1.3',
      maxVersion: 'TLSv1.3',
      ciphers: [
        'TLS_AES_128_GCM_SHA256',
        'TLS_AES_256_GCM_SHA384',
        'TLS_CHACHA20_POLY1305_SHA256'
      ].join(':'),

```

```

    // Security options
    honorCipherOrder: true,
    requestCert: false, // Set true for mTLS
    rejectUnauthorized: true,

    // Perfect Forward Secrecy
    dhparam: fs.readFileSync('/path/to/dhparam.pem')
  };

  return https.createServer(options, this.app);
}
}

```

// 2. Data at Rest - Database Encryption

```

class DatabaseEncryption {
  constructor() {
    this.algorithm = 'aes-256-gcm';
    this.keyLength = 32;
    this.ivLength = 16;
    this.tagLength = 16;

    // Load encryption key from secure source
    this.encryptionKey = this.loadEncryptionKey();
  }

  loadEncryptionKey() {
    // In production, load from KMS or secret manager
    const keyString = process.env.DB_ENCRYPTION_KEY;
    return Buffer.from(keyString, 'hex');
  }
}

```

```
encrypt(plaintext) {
  // Generate random IV for each encryption
  const iv = crypto.randomBytes(this.ivLength);

  const cipher = crypto.createCipheriv(
    this.algorithm,
    this.encryptionKey,
    iv
  );

  let ciphertext = cipher.update(plaintext, 'utf8',
'hex');
  ciphertext += cipher.final('hex');

  const authTag = cipher.getAuthTag();

  // Return IV + authTag + ciphertext
  return {
    iv: iv.toString('hex'),
    authTag: authTag.toString('hex'),
    ciphertext
  };
}

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

  const decipher = crypto.createDecipheriv(
    this.algorithm,
    this.encryptionKey,
    Buffer.from(iv, 'hex')
  );
```

```

    decipher.setAuthTag(Buffer.from(authTag, 'hex'));

    let plaintext = decipher.update(ciphertext, 'hex',
'utf8');
    plaintext += decipher.final('utf8');

    return plaintext;
}

// Field-level encryption for Mongoose
createEncryptionPlugin() {
    return function(schema, options) {
        const fieldsToEncrypt = options.fields || [];

        // Encrypt before saving
        schema.pre('save', function(next) {
            fieldsToEncrypt.forEach(field => {
                if (this[field] && !this[field +
'_encrypted']) {
                    const encrypted =
dbEncryption.encrypt(this[field]);
                    this[field + '_encrypted'] = encrypted;
                    this[field] = undefined; // Remove
plaintext
                }
            });
            next();
        });

        // Decrypt after reading
        schema.post('find', function(docs) {

```

```

docs.forEach(doc => {
  fieldsToEncrypt.forEach(field => {
    if (doc[field + '_encrypted']) {
      doc[field] =
dbEncryption.decrypt(doc[field + '_encrypted']);
      doc[field + '_encrypted'] = undefined; //
Remove encrypted
    }
  });
});
});
};
}
}

```

// 3. Key Management System

```

class KeyManagementSystem {
  constructor() {
    this.keys = new Map();
    this.keyRotationPeriod = 90 * 24 * 60 * 60 * 1000;
// 90 days
  }

```

```

  async generateDataKey() {
    // In production, use AWS KMS, Azure Key Vault, or
HashiCorp Vault
    const dataKey = crypto.randomBytes(32);
    const keyId =
crypto.randomBytes(16).toString('hex');

    // Encrypt data key with master key
    const encryptedDataKey = await

```

```

this.encryptWithMasterKey(dataKey);

    // Store encrypted data key
    await this.storeKey(keyId, encryptedDataKey, {
        createdAt: new Date(),
        expiresAt: new Date(Date.now() +
this.keyRotationPeriod)
    });

    return { keyId, dataKey };
}

async rotateKeys() {
    const expiredKeys = await this.getExpiredKeys();

    for (const oldKeyId of expiredKeys) {
        // Generate new key
        const { keyId: newKeyId, dataKey: newDataKey } =
            await this.generateDataKey();

        // Re-encrypt data with new key
        await this.reEncryptData(oldKeyId, newKeyId);

        // Archive old key (don't delete - needed for
backups)
        await this.archiveKey(oldKeyId);
    }
}

async encryptWithMasterKey(dataKey) {
    // Use AWS KMS
    const kms = new AWS.KMS();

```

```

const result = await kms.encrypt({
  KeyId: process.env.KMS_MASTER_KEY_ID,
  Plaintext: dataKey
}).promise();

return result.CiphertextBlob;
}
}

// 4. Encryption for File Storage
class FileEncryption {
  async encryptAndUploadFile(filePath) {
    // Generate unique key for file
    const fileKey = crypto.randomBytes(32);
    const iv = crypto.randomBytes(16);

    // Create cipher stream
    const cipher = crypto.createCipheriv('aes-256-cbc',
fileKey, iv);

    // Create streams
    const input = fs.createReadStream(filePath);
    const output = fs.createWriteStream(filePath +
'.encrypted');

    // Encrypt file
    await pipeline(input, cipher, output);

    // Encrypt file key with user's public key or
master key
    const encryptedFileKey = await
this.encryptFileKey(fileKey);

```

```

// Upload encrypted file to S3
const s3 = new AWS.S3();
const uploadResult = await s3.upload({
  Bucket: process.env.S3_BUCKET,
  Key: path.basename(filePath),
  Body: fs.createReadStream(filePath +
'.encrypted'),
  ServerSideEncryption: 'aws:kms',
  SSEKMSKeyId: process.env.KMS_KEY_ID,
  Metadata: {
    'encrypted-key':
encryptedFileKey.toString('base64'),
    'iv': iv.toString('base64')
  }
}).promise();

// Clean up
fs.unlinkSync(filePath);
fs.unlinkSync(filePath + '.encrypted');

return uploadResult.Location;
}

async downloadAndDecryptFile(s3Key, outputPath) {
  const s3 = new AWS.S3();

  // Get file and metadata
  const object = await s3.getObject({
    Bucket: process.env.S3_BUCKET,
    Key: s3Key
  }).promise();

```

```

    // Decrypt file key
    const encryptedFileKey = Buffer.from(
      object.Metadata['encrypted-key'],
      'base64'
    );
    const fileKey = await
this.decryptFileKey(encryptedFileKey);

    const iv = Buffer.from(object.Metadata['iv'],
'base64');

    // Decrypt file
    const decipher = crypto.createDecipheriv('aes-256-
cbc', fileKey, iv);

    const input = Readable.from(object.Body);
    const output = fs.createWriteStream(outputPath);

    await pipeline(input, decipher, output);

    return outputPath;
  }
}

// 5. Client-Side Encryption
class ClientSideEncryption {
  // For highly sensitive data, encrypt on client
  before sending

  // Client-side (browser)
  async encryptSensitiveData(data, userPassword) {

```

```
// Derive key from password
const salt = crypto.getRandomValues(new
Uint8Array(16));
const key = await window.crypto.subtle.deriveKey(
  {
    name: 'PBKDF2',
    salt,
    iterations: 100000,
    hash: 'SHA-256'
  },
  await window.crypto.subtle.importKey(
    'raw',
    new TextEncoder().encode(userPassword),
    'PBKDF2',
    false,
    ['deriveKey']
  ),
  { name: 'AES-GCM', length: 256 },
  false,
  ['encrypt']
);

// Encrypt data
const iv = crypto.getRandomValues(new
Uint8Array(12));
const encrypted = await
window.crypto.subtle.encrypt(
  { name: 'AES-GCM', iv },
  key,
  new TextEncoder().encode(data)
);
```

```
return {
    encrypted: btoa(String.fromCharCode(...new
Uint8Array(encrypted))),
    iv: btoa(String.fromCharCode(...iv)),
    salt: btoa(String.fromCharCode(...salt))
};
}
```

26. How do you implement security monitoring and incident response?

Answer:

```
// 1. Security Monitoring System
class SecurityMonitor {
    constructor() {
        this.alertThresholds = {
            failedLogins: 5,
            suspiciousActivity: 10,
            dataExfiltration: 1000 // KB
        };
    }

    async monitorSecurityEvents() {
        // Real-time security event processing
        this.setupEventListeners();
        this.setupAnomalyDetection();
        this.setupThreatIntelligence();
    }
}
```

```

setupEventListeners() {
  // Monitor authentication events
  events.on('auth:failed', async (data) => {
    await this.handleFailedAuth(data);
  });

  // Monitor suspicious patterns
  events.on('suspicious:activity', async (data) => {
    await this.handleSuspiciousActivity(data);
  });

  // Monitor data access
  events.on('data:access', async (data) => {
    await this.handleDataAccess(data);
  });
}

async handleFailedAuth(data) {
  const { userId, ip, timestamp } = data;

  // Count recent failures
  const recentFailures = await
SecurityEvent.countDocuments({
  type: 'failed_auth',
  $or: [{ userId }, { ip }],
  timestamp: { $gt: new Date(Date.now() - 15 * 60 *
1000) }
  });

  if (recentFailures >=
this.alertThresholds.failedLogins) {
    // Create security incident

```

```

await this.createIncident({
  type: 'brute_force_attempt',
  severity: 'high',
  userId,
  ip,
  details: { failureCount: recentFailures }
});

```

```

// Auto-response: block IP temporarily
await this.blockIP(ip, 3600000); // 1 hour

```

```

// Notify security team
await this.notifySecurityTeam({
  type: 'brute_force',
  ip,
  userId,
  failures: recentFailures
});

```

```

}
}

```

```

async setupAnomalyDetection() {
  // Detect unusual patterns
  setInterval(async () => {
    await this.detectAnomalies();
  }, 60000); // Every minute
}

```

```

async detectAnomalies() {
  // 1. Unusual access patterns
  const unusualAccess = await
this.detectUnusualAccessPatterns();

```

```
// 2. Impossible travel (access from different
locations)
    const impossibleTravel = await
this.detectImpossibleTravel();

// 3. Data exfiltration
    const dataExfiltration = await
this.detectDataExfiltration();

// 4. Privilege escalation attempts
    const privEscalation = await
this.detectPrivilegeEscalation();

// Create incidents for detected anomalies
const anomalies = [
    ...unusualAccess,
    ...impossibleTravel,
    ...dataExfiltration,
    ...privEscalation
];

for (const anomaly of anomalies) {
    await this.createIncident(anomaly);
}

}

async detectImpossibleTravel() {
    // Find users with logins from different locations
within short time
    const suspiciousLogins = await AuditLog.aggregate([
        {
```

```

    $match: {
      type: 'login',
      timestamp: { $gt: new Date(Date.now() - 60 *
60 * 1000) }
    }
  },
  {
    $group: {
      _id: '$userId',
      locations: { $push: { ip: '$ip', timestamp:
'$timestamp' } }
    }
  },
  {
    $match: {
      'locations.1': { $exists: true } // At least
2 logins
    }
  }
]);

const impossibleTravel = [];

for (const user of suspiciousLogins) {
  const locations = user.locations.sort((a, b) =>
    a.timestamp - b.timestamp
  );

  for (let i = 1; i < locations.length; i++) {
    const distance = await this.calculateDistance(
      locations[i-1].ip,
      locations[i].ip

```

```

    );

    const timeDiff = locations[i].timestamp -
locations[i-1].timestamp;
    const speedKmh = (distance / timeDiff) *
3600000;

    // If speed > 1000 km/h (impossible)
    if (speedKmh > 1000) {
        impossibleTravel.push({
            type: 'impossible_travel',
            severity: 'critical',
            userId: user._id,
            details: { distance, timeDiff, speedKmh }
        });
    }
}

return impossibleTravel;
}

async detectDataExfiltration() {
    // Monitor large data transfers
    const largeTransfers = await AuditLog.find({
        type: 'data_export',
        size: { $gt:
this.alertThresholds.dataExfiltration * 1024 },
        timestamp: { $gt: new Date(Date.now() - 60 * 60 *
1000) }
    });
}

```

```
return largeTransfers.map(transfer => ({
  type: 'data_exfiltration',
  severity: 'critical',
  userId: transfer.userId,
  details: { size: transfer.size, type:
transfer.dataType }
  }));
}
}

// 2. Incident Response System
class IncidentResponseSystem {
  async createIncident(incidentData) {
    const incident = await SecurityIncident.create({
      ...incidentData,
      status: 'open',
      createdAt: new Date(),
      assignedTo: await this.getOnCallEngineer()
    });

    // Execute automated response
    await this.executeAutomatedResponse(incident);

    // Notify team
    await this.notifyIncidentTeam(incident);

    // Create ticket in SIEM system
    await this.createSIEMTicket(incident);

    return incident;
  }
}
```

```

async executeAutomatedResponse(incident) {
  const playbook = this.getPlaybook(incident.type);

  for (const action of playbook.actions) {
    try {
      await this.executeAction(action, incident);

      await incident.addLog({
        action: action.name,
        status: 'success',
        timestamp: new Date()
      });
    } catch (error) {
      await incident.addLog({
        action: action.name,
        status: 'failed',
        error: error.message,
        timestamp: new Date()
      });
    }
  }
}

```

```

getPlaybook(incidentType) {
  const playbooks = {
    brute_force_attempt: {
      actions: [
        { name: 'block_ip', priority: 1 },
        { name: 'notify_user', priority: 2 },
        { name: 'require_password_reset', priority: 3
      ]
    }
  }
}

```

```

    },
    data_exfiltration: {
      actions: [
        { name: 'suspend_account', priority: 1 },
        { name: 'revoke_tokens', priority: 1 },
        { name: 'notify_security_team', priority: 1
},
        { name: 'preserve_evidence', priority: 2 }
      ]
    },
    impossible_travel: {
      actions: [
        { name: 'suspend_account', priority: 1 },
        { name: 'require_2fa_verification', priority:
2 },
        { name: 'notify_user', priority: 2 }
      ]
    }
  };

  return playbooks[incidentType] ||
playbooks.default;
}

async executeAction(action, incident) {
  switch (action.name) {
    case 'block_ip':
      await this.blockIP(incident.ip, 86400000); //
24 hours
      break;

    case 'suspend_account':

```

```
        await User.updateById(incident.userId, {
suspended: true });
        break;

    case 'revoke_tokens':
        await
this.revokeAllUserTokens(incident.userId);
        break;

    case 'require_password_reset':
        await User.updateById(incident.userId, {
            passwordResetRequired: true
        });
        break;

    case 'notify_user':
        await this.sendSecurityAlert(incident.userId,
incident);
        break;

    case 'preserve_evidence':
        await this.preserveForensicEvidence(incident);
        break;
    }
}

async preserveForensicEvidence(incident) {
    // Collect all relevant data
    const evidence = {
        incident,
        auditLogs: await AuditLog.find({
            userId: incident.userId,
```

```

        timestamp: {
            $gte: new Date(incident.createdAt - 3600000),
            $lte: new Date(incident.createdAt + 3600000)
        }
    )),
    sessions: await Session.find({ userId:
incident.userId }),
    userActions: await UserAction.find({
        userId: incident.userId,
        timestamp: {
            $gte: new Date(incident.createdAt - 86400000)
        }
    })
};

// Store in secure, immutable storage
const evidenceId = await
this.storeEvidence(evidence);

// Create chain of custody
await ChainOfCustody.create({
    evidenceId,
    incidentId: incident.id,
    collectedBy: 'automated_system',
    collectedAt: new Date(),
    hash: this.hashEvidence(evidence)
});

return evidenceId;
}
}

```

```
// 3. Security Logging
class SecurityLogger {
  constructor() {
    this.winston = winston.createLogger({
      level: 'info',
      format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.json()
      ),
      transports: [
        // File transport for all logs
        new winston.transports.File({
          filename: 'logs/security.log',
          maxsize: 10485760, // 10MB
          maxFiles: 30
        }),
        // Separate file for critical events
        new winston.transports.File({
          filename: 'logs/security-critical.log',
          level: 'error'
        }),
        // Send to external SIEM
        new winston.transports.Http({
          host: process.env.SIEM_HOST,
          port: process.env.SIEM_PORT,
          ssl: true
        })
      ]
    });
  }

  logSecurityEvent(event) {
```

```
const logEntry = {
  timestamp: new Date(),
  type: event.type,
  severity: event.severity,
  userId: event.userId,
  ip: event.ip,
  userAgent: event.userAgent,
  details: event.details,
  // Add context
  requestId: event.requestId,
  sessionId: event.sessionId
};

this.winston.log(event.severity, logEntry);

// Store in database for querying
SecurityEvent.create(logEntry);

// Send to real-time monitoring
this.sendToMonitoring(logEntry);
}
}
```

Practical Scenarios

Scenario 1: You discover a dependency vulnerability

Question: npm audit shows a high-severity vulnerability in an older version of `lodash`. What steps do you take?

Answer:

```
# 1. Assess the vulnerability
npm audit

# 2. Check if you're using the vulnerable functionality
grep -r "lodash" src/

# 3. Update the package
npm update lodash

# 4. If update doesn't fix it
npm audit fix

# 5. Force update (breaking changes possible)
npm audit fix --force

# 6. If still not fixed, check if it's a nested
dependency
npm ls lodash

# 7. Consider alternatives or wait for fix
# 8. Document the decision and monitor

# 9. Add to CI/CD pipeline
# package.json scripts
{
  "scripts": {
    "audit": "npm audit --audit-level=moderate",
    "precommit": "npm run audit"
```

```

    }
  }
}

```

Scenario 2: Implement passwordless authentication

Question: Design a secure magic link authentication system.

Answer:

```

class MagicLinkAuth {
  async sendMagicLink(email) {
    // 1. Validate email
    const user = await User.findByEmail(email);
    if (!user) {
      // Don't reveal if email exists
      return { message: 'If email exists, link has been
sent' };
    }

    // 2. Generate secure token
    const token =
crypto.randomBytes(32).toString('hex');
    const hashedToken = crypto.createHash('sha256')
      .update(token)
      .digest('hex');

    // 3. Store hashed token with expiration
    await MagicLinkToken.create({
      userId: user.id,
      token: hashedToken,
      expiresAt: new Date(Date.now() + 15 * 60 * 1000),

```

```

// 15 min
    used: false
  });

  // 4. Create magic link
  const magicLink =
`${process.env.APP_URL}/auth/verify?token=${token}`;

  // 5. Send email
  await emailService.send({
    to: email,
    subject: 'Your login link',
    html: `Click here to log in: <a
href="${magicLink}">Log in</a>
        This link expires in 15 minutes.`
  });

  return { message: 'If email exists, link has been
sent' };
}

async verifyMagicLink(token) {
  // 1. Hash provided token
  const hashedToken = crypto.createHash('sha256')
    .update(token)
    .digest('hex');

  // 2. Find valid token
  const magicToken = await MagicLinkToken.findOne({
    token: hashedToken,
    expiresAt: { $gt: new Date() },
    used: false
  });

```

```
    });

    if (!magicToken) {
        throw new AppError('Invalid or expired token',
401);
    }

    // 3. Mark as used
    magicToken.used = true;
    await magicToken.save();

    // 4. Generate JWT tokens
    const user = await
User.findById(magicToken.userId);
    const tokens = await
authService.generateTokenPair(user);

    // 5. Log successful login
    await auditLog.create({
        userId: user.id,
        type: 'magic_link_login',
        timestamp: new Date()
    });

    return tokens;
}
}
```

Scenario 3: Secure API key generation and management

Question: Implement a system for users to generate and manage API keys.

Answer:

```

class APIKeyService {
  async generateAPIKey(userId, name, permissions) {
    // 1. Generate secure key
    const key =
`sk_${process.env.ENV}_${crypto.randomBytes(32).toString(
  'hex')}`;

    // 2. Hash for storage (never store plaintext)
    const hashedKey = crypto.createHash('sha256')
      .update(key)
      .digest('hex');

    // 3. Store in database
    const apiKey = await APIKey.create({
      userId,
      name,
      hashedKey,
      keyPrefix: key.substring(0, 12), // For
identification
      permissions,
      createdAt: new Date(),
      lastUsedAt: null,
      expiresAt: new Date(Date.now() + 365 * 24 * 60 *
60 * 1000) // 1 year
    });

    // 4. Return key only once
    return {
      apiKey: key, // Show only once
      keyId: apiKey.id,

```

```
    prefix: apiKey.keyPrefix,
    message: 'Save this key, it won\'t be shown
again'
  };
}

async validateAPIKey(key) {
  // 1. Hash provided key
  const hashedKey = crypto.createHash('sha256')
    .update(key)
    .digest('hex');

  // 2. Find key
  const apiKey = await APIKey.findOne({
    hashedKey,
    active: true,
    expiresAt: { $gt: new Date() }
  });

  if (!apiKey) {
    return null;
  }

  // 3. Update last used
  apiKey.lastUsedAt = new Date();
  await apiKey.save();

  return {
    userId: apiKey.userId,
    permissions: apiKey.permissions
  };
}
```

```
async rotateAPIKey(keyId) {
  // Generate new key
  const oldKey = await APIKey.findById(keyId);
  const newKey = await this.generateAPIKey(
    oldKey.userId,
    oldKey.name,
    oldKey.permissions
  );

  // Set grace period for old key
  oldKey.expiresAt = new Date(Date.now() + 7 * 24 *
60 * 60 * 1000);
  oldKey.rotated = true;
  await oldKey.save();

  return newKey;
}
```

Additional Resources

- [OWASP Top 10](#)
- [Node.js Security Best Practices](#)
- [Snyk Security Guides](#)
- [npm Security Best Practices](#)