DocDesk - Information Security Risk Assessment

> **Project:** DocDesk – Medical Records Management System
> **Date:** 12 February 2026
> **Scope:** Backend API (Node.js/Express), Admin Portal (React), Mobile Frontend (React Native)

Executive Summary

DocDesk is a healthcare application that manages sensitive patient medical records, doctor information, prescriptions, appointments, and test results. A thorough code-level security audit of the entire project has identified **five major information security risks** that could lead to unauthorized access, data breaches, and compromise of protected health information (PHI).

Risk 1: Hardcoded Credentials & Secrets Exposure - Hirusha

Description

Email service credentials (Gmail App Password) are hardcoded directly in the source code within the `authController.js` file. The SMTP credentials (`user: "haritharashmikanawarathna@gmail.com"` , `pass: "jupy bhwr lygn yvfe"`) appear in plaintext in **three separate functions**: `forgotPassword` , `resendOTP` , and `getOTP` .

Affected Asset(s)

Asset	Type	Classification
Gmail SMTP credentials (App Password)	Authentication credential	Confidential
Email service (Nodemailer transporter)	Communication infrastructure	High value
Source code repository	Intellectual property	Internal

Evidence (Code Locations)

- `Docdesk-backend/controllers/authController.js` — Lines ~245–249 (`forgotPassword` function)
- `Docdesk-backend/controllers/authController.js` — Lines ~415–419 (`resendOTP` function)
- `Docdesk-backend/controllers/authController.js` — Lines ~470–475 (`getOTP` function)

```
// Hardcoded in source code (repeated 3 times)
const transporter = nodemailer.createTransport({
  service: "gmail",
  auth: {
    user: "haritharashmikanawarathna@gmail.com",
    pass: "jupy bhwr lygn yvfe",
  },
});
```

Impact

- **Severity: CRITICAL**
- If the repository is pushed to a public Git host or shared, the email credentials are immediately exposed.
- An attacker can use the compromised Gmail account to send phishing emails impersonating DocDesk.
- The attacker could intercept or send password reset OTPs to any user, enabling full account takeover.
- This violates the principle of **secrets management** (OWASP A07:2021 – Security Misconfiguration).

Recommendation

- Move all credentials to environment variables (`.env`) and access via `process.env`.
- Use a dedicated transactional email service (e.g., SendGrid, AWS SES) with API keys rotated regularly.
- Immediately rotate the exposed Gmail App Password.
- Add `.env` to `.gitignore` and audit Git history for previously committed secrets using tools like `git-secrets` or `truffleHog`.

Risk 2: Broken Access Control — No Role-Based Authorization Enforcement - Haritha

Description

While the backend defines user roles (`patient` , `doctor` , `admin`) and applies JWT authentication middleware, there is **no role-based authorization check** on any protected route. The `AuthMiddleware.js` only verifies that the JWT is valid—it does not check `req.userData.roles` to enforce whether a patient, doctor, or admin is allowed to perform the requested action.

Affected Asset(s)

Asset	Type	Classification
Patient medical records (PHI)	Protected Health Information	Highly Confidential
Patient personal data (NIC, email, address, health data)	PII / PHI	Highly Confidential
Doctor profiles and verification status	Professional data	Confidential
Admin operations (doctor verification, user deletion)	System administration	Critical

Evidence (Code Locations)

AuthMiddleware.js — Only verifies token, no role check:

```
const decoded = jwt.verify(token, process.env.ACCESS_TOKEN_SECRET);
req.userData = decoded;
next(); // No role check performed
```

Routes allow any authenticated user to perform any action:

- `GET /api/patients/` — returns ALL patients (accessible by any authenticated user, including other patients)
- `DELETE /api/patients/:id` — any authenticated user can delete any patient
- `DELETE /api/doctors/:id` — any authenticated user can delete any doctor
- `PATCH /api/doctors/verifyDoctor/:id` — any authenticated user can verify a doctor
- `PUT /api/doctors/:id` — any authenticated user can update any doctor profile

Impact

- **Severity: CRITICAL**
- A patient can access, modify, or delete other patients' records (horizontal privilege escalation).
- A patient can verify doctors, delete doctors, or perform admin-only operations (vertical privilege escalation).

- This is a direct violation of **OWASP A01:2021 – Broken Access Control**, the #1 web application security risk.
- In a healthcare context, this constitutes a potential HIPAA/data protection regulation violation.

Recommendation

- Implement role-based middleware (e.g., `authorizeRoles("admin")`) that checks `req.userData.roles` before granting access to protected endpoints.
- Apply the principle of least privilege: patients should only access their own data; doctors should only access patients who have granted them access; only admins should manage user accounts.
- Add ownership validation (e.g., ensure `req.userData._id === req.params.id` for self-operations).

Risk 3: Insecure Token & Session Management - Awishke

Description

Multiple vulnerabilities exist in how authentication tokens are generated, stored, and managed across the application:

1. **Refresh tokens are not properly stored in the database** — The `generateRefreshToken.js` uses incorrect `findOneAndUpdate` syntax (multiple update objects instead of one), meaning refresh tokens are never actually saved to the database.
2. **Refresh token is not validated against stored value** — The `refreshAccessGenerate.js` only verifies the JWT signature but never compares the refresh token against the one stored in the database, making token revocation impossible.
3. **Tokens stored in `localStorage` (admin portal)** — The admin portal stores both access and refresh tokens in browser `localStorage`, which is vulnerable to XSS attacks.
4. **No token blacklisting or logout invalidation** — There is no mechanism to invalidate tokens on logout.

Affected Asset(s)

Asset	Type	Classification
JWT Access Tokens	Authentication credential	Confidential
JWT Refresh Tokens	Authentication credential	Highly Confidential
User sessions (all roles)	Session state	Confidential

`localStorage` (admin portal browser)	Client-side storage	Vulnerable

Evidence (Code Locations)

Incorrect `findOneAndUpdate` — refresh token never saved:

```
// generateRefreshToken.js — BUG: three separate objects passed instead of one
updatedUser = await Doctor.findOneAndUpdate(
  { _id: payload._id },
  { doctorId: payload._id },     // This is treated as the update
  { refreshToken: refreshToken }, // This is treated as options (ignored)
  { new: true }                  // This is extra (ignored)
);
```

Refresh token not validated against DB:

```
// refreshAccessGenerate.js — Only checks JWT signature, not DB value
const decoded = jwt.verify(refreshToken, process.env.REFRESH_TOKEN_SECRET);
// ... finds user but NEVER checks if user.refreshToken === refreshToken
```

localStorage usage in admin portal:

```
// useLogin.js
localStorage.setItem("user", JSON.stringify(response.data));
```

Impact

- **Severity: HIGH**
- Stolen refresh tokens can never be revoked since they are not stored/validated against the database.
- An attacker with a valid refresh token has indefinite access (7-day validity, continuously renewable).
- XSS attacks on the admin portal can steal tokens directly from `localStorage`.
- Logging out does not actually invalidate the session on the server side.

Recommendation

- Fix the `findOneAndUpdate` call to correctly store refresh tokens in the database.
- Validate refresh tokens against the stored database value during token refresh.
- Implement token blacklisting on logout.
- Use `httpOnly` cookies for token storage in the admin portal instead of `localStorage`.
- Implement refresh token rotation (issue a new refresh token on each refresh, invalidating the old one).

Risk 4: Unrestricted CORS & Missing Security Headers - Randil

Description

The backend server uses `cors()` with **no configuration**, which allows requests from **any origin**. Additionally, the server lacks critical HTTP security headers (no Helmet.js or equivalent), HTTPS enforcement, and rate limiting. The admin portal and mobile frontend communicate over plain HTTP.

Affected Asset(s)

Asset	Type	Classification
Backend API server (Express)	Application infrastructure	Critical
All API endpoints (medical data, auth)	Service endpoints	Critical
Patient/Doctor data transmitted over network	Data in transit	Highly Confidential
Admin portal web application	Web interface	High value

Evidence (Code Locations)

Unrestricted CORS in server.js:

```
app.use(cors()); // No origin restriction — accepts ALL origins
```

No security headers — no Helmet.js or similar middleware present in `server.js` or `package.json`.

HTTP-only communication (no TLS):

```
// Dockdesk-admin-portal/constants/constants.js
export const baseUrl = "http://localhost:5000/api";

// Dockdesk-frontend/src/constants/constants.js
const base = Platform.OS === "android" ? "http://10.219.215.240" : "http://localho
```

No rate limiting on sensitive endpoints — Authentication routes (`/signin` , `/signup` , `/forgotPassword` , `/verifyOTP`) have no rate limiting, allowing brute-force attacks on passwords and OTPs.

Impact

- **Severity: HIGH**
- Cross-Site Request Forgery (CSRF) attacks can be performed from any malicious website against authenticated users.
- Man-in-the-Middle (MITM) attacks can intercept credentials, tokens, and medical data transmitted over HTTP.
- Brute-force attacks on 6-digit OTPs (only 1,000,000 combinations) are feasible without rate limiting.
- Missing security headers (CSP, X-Frame-Options, X-Content-Type-Options) expose the application to XSS, clickjacking, and MIME-sniffing attacks.
- This violates **OWASP A05:2021 – Security Misconfiguration**.

Recommendation

- Configure CORS with a strict allowlist of trusted origins:

```
app.use(cors({ origin: ["https://admin.docdesk.com"], credentials: true }));
```

- Add the `helmet` middleware for security headers.
- Enforce HTTPS in production (TLS termination at reverse proxy if needed).
- Implement rate limiting using `express-rate-limit` on authentication and OTP endpoints.
- Add account lockout after repeated failed login/OTP attempts.

Risk 5: Sensitive Data Over-Exposure in API Responses - Chamika

Description

Multiple API endpoints return **entire database documents** including sensitive fields such as hashed passwords, refresh tokens, internal IDs, and full medical history to any authenticated caller. The `getPatients` and `getDoctors` endpoints return **all records** with no filtering or field projection.

Affected Asset(s)

Asset	Type	Classification
Bcrypt password hashes	Authentication credential	Highly Confidential

Refresh tokens stored in user documents	Session credential	Highly Confidential
Full patient list with medical data	PHI / PII	Highly Confidential
OTP fields (reset password, email verification)	Temporary credentials	Confidential
Patient NIC (National Identity Card) numbers	Government-issued PII	Confidential

Evidence (Code Locations)

Returns ALL patients with ALL fields (including passwords, tokens, OTPs):

```
// Patient.Controller.js
const getPatients = async (req, res) => {
  const Patients = await Patient.find({}).sort({ createdAt: -1 });
  res.status(200).json(Patients); // No field filtering — exposes password, refres
};
```

Returns ALL doctors with ALL fields:

```
// Doctor.Controller.js
const getDoctors = async (req, res) => {
  const doctors = await DocModel.find({}).sort({ createdAt: -1 });
  res.status(200).json(doctors); // No field filtering — exposes password, refresh
};
```

Single patient/doctor endpoints also return full documents:

```
const getPatient = async (req, res) => {
  const patient = await Patient.findById(id);
  res.status(200).json(patient); // Full document including password hash
};
```

Impact

- **Severity: HIGH**
- Password hashes (bcrypt) are exposed to any authenticated user—while bcrypt is slow to crack, exposure is unnecessary and risky.
- Refresh tokens in responses allow session hijacking if intercepted.
- Leaked OTP values enable bypassing email verification and password reset flows.
- NIC numbers and full medical histories exposed to unauthorized roles.

- This violates **OWASP A03:2021 – Injection** (data leakage) and common data minimization principles.

Recommendation

- Use Mongoose `.select()` to exclude sensitive fields:

```
Patient.find({}).select("-password -refreshToken -resetPasswordOTP -emailVerif
```

- Create separate DTOs/response schemas for each endpoint that return only the necessary fields.
- Never expose password hashes, tokens, or OTP values in API responses.
- Implement pagination on list endpoints to prevent mass data extraction.
- Apply field-level access control based on the requester's role.

Summary Risk Matrix

#	Risk	Severity	Likelihood	OWASP Category	Primary Asset at Risk
1	Hardcoded Credentials & Secrets Exposure	**Critical**	High	A07:2021 – Security Misconfiguration	Email service credentials
2	Broken Access Control (No RBAC)	**Critical**	High	A01: 2021 – Broken Access Control	Patient medical records (PHI)
3	Insecure Token & Session Management	**High**	Medium	A07:2021 – Security Misconfiguration	JWT tokens & user sessions
4	Unrestricted CORS & Missing Security Headers	**High**	Medium	A05:2021 – Security Misconfiguration	Backend API & data in transit
5	Sensitive Data Over-Exposure in API Responses	**High**	High	A03:2021 – Sensitive Data Exposure	Passwords, tokens, PHI/PII

References

- OWASP Top 10 (2021)

- OWASP Top 10 (2021)
- OWASP API Security Top 10
- NIST SP 800-53 – Security and Privacy Controls
- CWE-798: Use of Hard-coded Credentials
- CWE-862: Missing Authorization