# CTF-cybercupAi Technical Report

## Table of Contents



---

## 1. Repository Overview

The repository is organized as a full-stack web application with the following structure:

- **client/**: Next.js application using the App Router for the frontend
- **server/**: Node.js/Express application for the backend API
- **docker-compose.yml**: Docker orchestration for client and server services
- **Top-level README.md**: Minimal project identifier (CTF-cybercupAi)

---

# 2. Technology Stack

## 2.1 Frontend Technologies

The frontend is built using modern React-based technologies:

- **Next.js (App Router)**: React framework with functional components and hooks
- **Custom Layout**: Implemented in `client/app/layout.js`, integrating `AuthProvider`, `Navbar`, and `Footer` components
- **State Management**: Client-side state and effects for authentication, challenge interaction, admin screens, leaderboard, and support forms
- **Styling**: Tailwind-style utility classes via `globals.css`
- **Google Authentication**: Google Identity Services script integrated directly in login and register pages

## 2.2 Backend Technologies

The backend implements a RESTful API architecture:

- **Node.js with Express**: Server implementation in `server/src/server.js`
- **MongoDB with Mongoose**: Database connection configured in `server/src/config/db.js` with model definitions
- **JWT Authentication**: HTTP cookie-based authentication using `jsonwebtoken`, `cookie-parser`, and custom middleware in `server/src/middleware/auth.js`
- **Password Security**: Bcrypt-based password hashing implemented in `server/src/routes/authRoutes.js`
- **Google OAuth**: Backend verification using `google-auth-library`

## 2.3 Email Support

Email functionality is implemented using:

- **Nodemailer**: Configured in `client/app/api/support/route.js` for sending support emails and automated replies

## 2.4 Docker Configuration

Containerization is achieved through:

- **Client Dockerfile**: `client/Dockerfile` builds and runs the frontend container
- **Server Dockerfile**: `server/Dockerfile` builds and runs the backend container
- **Docker Compose**: `docker-compose.yml` defines client and server services with standard port mappings

---

# 3. Backend Architecture

## 3.1 Server Entry Point

**File**: `server/src/server.js`

The main server file performs the following initialization:

- Loads environment variables using `dotenv`
- Establishes MongoDB connection via `connectDB()` from `server/src/config/db.js`
- Configures Express middleware:
  - `cors`: Configured with `origin: "http://localhost:3000"` and `credentials: true`
  - `express.json()`: Parses JSON request bodies
  - `cookieParser()`: Enables cookie parsing
- Mounts route modules:
  - `/api/auth` routes to `authRoutes`
  - `/api/problems` routes to `problemRoutes`
  - `/api/attempts` routes to `attemptRoutes`
  - `/api/users` routes to `userRoutes`
  - `/api/leaderboard` routes to `leaderboardRoutes`
  - `/api/stats` routes to `statsRoutes`
- Implements root route `GET /` with simple health check response
- Listens on `PORT` from environment variable or defaults to `5000`

## 3.2 Database Configuration

**File**: `server/src/config/db.js`

Database connection management:

- Uses `mongoose.connect(process.env.MONGO_URI)` to establish MongoDB connection
- Logs successful connections with database name
- Implements error handling with process exit (code 1) on connection failure

## 3.3 Authentication Middleware

**File**: `server/src/middleware/auth.js`

Two middleware functions provide authentication and authorization:

**auth middleware**:

- Reads JWT from `req.cookies[process.env.COOKIE_NAME]`
- Verifies token using `jwt.verify` with `process.env.JWT_SECRET`
- Queries User collection by `payload.userId`
- Attaches `req.user` object with `{ id, email, isAdmin, name }` if user exists
- Returns HTTP 401 for missing, invalid, or expired tokens

**adminOnly middleware**:

- Requires `req.user.isAdmin` to be truthy
- Returns HTTP 403 if user is not an administrator

### 3.4 Data Models

**User Model** (`server/src/models/User.js`):

Fields:

- `name` (String, required)
- `email` (String, required, unique)
- `passwordHash` (String, optional)
- `isAdmin` (Boolean, default: false)
- `authProvider` (String enum: "local" or "google", default: "local")
- `googleId` (String)
- `avatar` (String)
- Automatic timestamps: `createdAt`, `updatedAt`

**Problem Model** (`server/src/models/Problem.js`):

Fields:

- `title` (String, required)
- `description` (String, required)
- `flagAnswer` (String, required)
- `difficulty` (String enum: "easy", "medium", "hard", required)
- `createdBy` (ObjectId reference to User)
- Automatic timestamps: `createdAt`, `updatedAt`

**Attempt Model** (`server/src/models/Attempt.js`):

Fields:

- `userId` (ObjectId reference to User, required)
- `problemId` (ObjectId reference to Problem, required)
- `answers` (Array of Strings, required)
- `result` (String enum: "correct", "incorrect", required)

- Automatic timestamps: `createdAt`, `updatedAt`

Indexes:

- Compound index: `{ userId: 1, problemId: 1, createdAt: 1 }`
- Compound unique index: `{ userId: 1, problemId: 1, result: 1 }` with partial filter `result: "correct"` (ensures at most one correct attempt per user/problem pair)

### 3.5 Authentication Routes

**File**: `server/src/routes/authRoutes.js`

Dependencies: `bcryptjs`, `jsonwebtoken`, `User` model, `OAuth2Client` (Google)

**Utility Function**:

- `setTokenCookie(res, user)`: Signs JWT with `{ userId, isAdmin }` and `expiresIn: "7d"`, sets HTTP-only cookie with `sameSite: "lax"`, `secure: false`, and 7-day `maxAge`

**Endpoints**:

**POST /api/auth/register**:

- Request body: `{ name, email, password }`
- Validates email availability (returns HTTP 400 "Email already in use" if exists)
- Hashes password using bcrypt
- Creates user with `authProvider: "local"`
- Sets authentication cookie
- Returns user data: `{ id, name, email, isAdmin }`

**POST /api/auth/login**:

- Request body: `{ email, password }`
- Looks up user by email
- Returns HTTP 400 "Invalid credentials" if user not found
- Returns HTTP 400 with Google login instruction if `passwordHash` is missing or `authProvider === "google"`
- Compares password using bcrypt (HTTP 400 "Invalid credentials" on mismatch)
- Sets cookie and returns `{ id, name, email, isAdmin }` on success

**POST /api/auth/google**:

- Request body: `{ credential }` (Google ID token)
- Verifies token using `googleClient.verifyIdToken` with `GOOGLE_CLIENT_ID`
- Extracts `sub` (googleId), `email`, `name`, `picture` from payload

- Returns HTTP 400 if email is missing
- Creates or updates user:
    - New users: Sets `authProvider: "google"`, `googleId`, `avatar`, `passwordHash: null`
    - Existing users: Updates `googleId`, `authProvider` (if missing), and `avatar`
- Sets JWT cookie and returns `{ id, name, email, isAdmin }`

**POST /api/auth/logout**:

- Clears authentication cookie
- Returns "Logged out" message

**GET /api/auth/me** (protected by auth middleware):

- Returns current user data: `{ id, name, email, isAdmin }` from `req.user`

### 3.6 Problem Routes

**File**: `server/src/routes/problemRoutes.js`

**Public Routes** (no authentication required):

**GET /api/problems**:

- Returns all problems sorted by `createdAt` descending
- Excludes `flagAnswer` field via `.select("-flagAnswer")`

**GET /api/problems/:id**:

- Returns single problem by ID
- Excludes `flagAnswer` field
- Returns HTTP 404 if problem not found

**Admin Routes** (protected by auth and adminOnly middleware):

**GET /api/problems/admin/all**:

- Returns complete problem documents including `flagAnswer`
- Sorted by `createdAt` descending

**GET /api/problems/admin/:id**:

- Returns full problem document including `flagAnswer`

**POST /api/problems**:

- Request body: `{ title, description, flagAnswer, difficulty }`

- Creates new problem with `createdBy` set to `req.user.id`
- Returns created problem document

**PUT /api/problems/:id**:

- Request body: `{ title, description, flagAnswer, difficulty }`
- Updates existing problem
- Returns updated problem document or HTTP 404 if not found

**DELETE /api/problems/:id**:

- Deletes problem by ID
- Returns confirmation message or HTTP 404 if not found

## 3.7 Attempt Routes

**File**: `server/src/routes/attemptRoutes.js`

All routes protected by auth middleware.

**GET /api/attempts/mine/solved**:

- Finds attempts for current user where `result: "correct"`
- Selects only `problemId` field
- Returns `{ solvedProblemIds }` as array of stringified problem IDs

**GET /api/attempts/mine/:problemId**:

- Finds all attempts for current user and specified problem
- Sorts by `createdAt` descending
- Returns list of attempts

**POST /api/attempts/:problemId**:

- Request body: `{ answers }` (array or single value)
- Loads Problem by `problemId` (HTTP 404 if not found)
- Checks for existing correct attempt for this user/problem combination
    - Returns HTTP 400 if problem already solved (no more attempts allowed)
- Normalizes `answers` to array of strings
- Determines correctness: `answersArray.includes(problem.flagAnswer)`
- Creates Attempt document with `result` set to "correct" or "incorrect"
- Returns created attempt document

**GET /api/attempts/:problemId**:

- Finds attempts for current user and specified problem
- Sorts by `createdAt` descending
- Returns list of attempts

### 3.8 User Routes

**File**: `server/src/routes/userRoutes.js`

All routes protected by auth and adminOnly middleware.

**GET /api/users**:

- Returns list of all users
- Fields: `name`, `email`, `isAdmin`, `createdAt`

**GET /api/users/:id**:

- Returns single user by ID
- Fields: `name`, `email`, `isAdmin`, `createdAt`
- Returns HTTP 404 if user not found

**GET /api/users/:id/attempts**:

- Returns all attempts for specified user
- Populates `problemId` with `title` and `difficulty` fields
- Sorted by `createdAt` descending

### 3.9 Leaderboard Routes

**File**: `server/src/routes/leaderboardRoutes.js`

**GET /api/leaderboard**:

- Query parameter: `limit` (optional, default: "50")
- Implements MongoDB aggregation pipeline on Attempt collection:
    - First stage: Groups by `{ userId, problemId }` to count attempts per problem and detect correct results
    - Second stage: Groups by `userId` to sum `totalAttempts` and count problems with correct attempts (`solvedCount`)
    - Lookup stage: Joins with users collection
    - Filter stage: Excludes admin users (`user.isAdmin !== true`)
    - Sort stage: Orders by `solvedCount` descending, then `totalAttempts` ascending, then user name
    - Limit stage: Applies specified limit
    - Project stage: Returns `userId`, `name`, `email`, `solvedCount`, `totalAttempts`
- Adds `rank` field (1-based index) to each entry
- Returns ranked array

### 3.10 Stats Routes

**File**: `server/src/routes/statsRoutes.js`

**GET /api/stats/overview** (protected by auth and adminOnly middleware):

- Executes parallel count queries using `Promise.all`:
  - `totalUsers`: Count of all users
  - `totalProblems`: Count of all problems
  - `totalAttempts`: Count of all attempts
  - `totalCorrect`: Count where `result: "correct"`
  - `totalIncorrect`: Count where `result: "incorrect"`
- Returns object containing all computed statistics

---

# 4. Frontend Architecture

## 4.1 Global Layout and Providers

**File**: `client/app/layout.js`

Root layout configuration:

- Imports global styles and fonts
- Wraps all content with:
  - `AuthProvider` from `client/context/AuthContext.jsx`
  - `Navbar` component from `client/components/Navbar.jsx`
  - `Footer` component from `client/components/Footer.jsx`
- Defines static `metadata` including:
  - SEO title and description
  - Keywords
  - Open Graph tags
  - Twitter card metadata

## 4.2 API Utility

**File**: `client/lib/api.js`

Centralized API request handler:

- Defines `API_BASE_URL` from `process.env.NEXT_PUBLIC_API_BASE_URL` or empty string
- Exports `apiRequest(path, options)` function:
  - Calls `fetch(API_BASE_URL + path)` with `credentials: "include"` and JSON `Content-Type`
  - Attempts to parse JSON response (sets `data` to `null` on parse failure)

    - Throws Error with `data.message` or "Request failed" if `res.ok` is false
    - Returns parsed data on success

### 4.3 Authentication Context

**File**: `client/context/AuthContext.jsx`

Global authentication state management:

**State**:

- `user`: Current authenticated user object or null
- `loading`: Boolean indicating initial authentication check

**Initialization**:

- On mount, calls `apiRequest("/api/auth/me")` to fetch current user
- Sets `user` on success or null on failure
- Sets `loading` to false when complete

**Methods**:

- `register(name, email, password)`: Calls `POST /api/auth/register` and updates `user` state
- `login(email, password)`: Calls `POST /api/auth/login` and updates `user` state
- `googleLogin(credential)`: Calls `POST /api/auth/google` with credential and updates `user` state
- `logout()`: Calls `POST /api/auth/logout` and clears `user` state

**Hook**:

- Exposes `useAuth()` hook for consuming components

### 4.4 Routing Guard

**File**: `client/proxy.js`

Client-side JWT verification and route protection:

Uses `NextResponse` and `jose` library's `jwtVerify` to inspect JWT cookies.

**Configuration**:

- Reads auth cookie name from `process.env.COOKIE_NAME` or defaults to "token"

**Helper Function**:

- `getUserFromToken(token)`: Verifies JWT with `JWT_SECRET` and returns payload or null

**Middleware Logic** (`proxy(request)`):

- Extracts pathname from `request.nextUrl`
- Reads and decodes token from cookies
- Defines route type flags:
  - `isAuthPage`: `/login` or `/register`
  - `isAdminRoute`: Paths starting with `/admin`
  - `isHomePage`: Root path `/`

**Redirection Rules**:

- Unauthenticated user accessing `isAdminRoute` redirects to `/login`
- Authenticated non-admin accessing `isAdminRoute` redirects to `/challenges`
- Authenticated user on auth page redirects to `/admin` (if admin) or `/challenges` (if not)
- Authenticated user on homepage redirects to `/admin` (if admin) or `/challenges` (if not)
- Otherwise, calls `NextResponse.next()`

**Matcher Configuration**:

- Applies to paths: `/`, `/admin/:path*`, `/login`, `/register`

## 4.5 Navigation Component

**File**: `client/components/Navbar.jsx`

Dynamic navigation bar:

**Hooks Used**:

- `usePathname`: Current route detection
- `useRouter`: Programmatic navigation
- `useAuth`: Authentication state

**Display Elements**:

- Logo linking to root
- Navigation links:
  - Challenges (hidden for admin when not needed)
  - Admin Panel (admin users only)
  - Leaderboard
- Authentication section:
  - When authenticated:

        - Displays user name and avatar initial
        - Logout button (calls `logout()` and redirects to `/login`)
    - When unauthenticated:
        - Log in button (links to `/login`)
        - Sign up button (links to `/register`)
- Mobile menu with responsive toggle

### 4.6 Key Pages and Flows

**Home Page** (`client/app/page.js`):

- Client-only component with animated sections
- Uses `IntersectionObserver` for scroll animations
- Sections: hero, features, stats, call-to-action
- Provides marketing-style platform overview

**Challenges Listing** (`client/app/challenges/page.jsx`):

- Loads challenges via `apiRequest("/api/problems")`
- Authenticated users: Fetches solved problems via `/api/attempts/mine/solved`
- Stores solved problem IDs in Set for marking
- Implements filter and search query state for local filtering
- Challenge click handling:
    - Solved challenges: Prompts with `window.confirm` before navigation
    - User must confirm to view details of solved challenges
- Renders difficulty-specific colors and icons

**Challenge Detail** (`client/app/challenges/[id]/page.jsx`):

- Reads `id` from `useParams()`
- Fetches challenge details via `apiRequest("/api/problems/:id")`
- Authenticated users: Fetches attempts via `/api/attempts/mine/:id`
- Tracks:
    - List of user's attempts
    - Completion status (`hasCompletedChallenge`)

**Flag Submission Logic**:

- Validates non-empty flag and user authentication
- Prevents submissions when `hasCompletedChallenge` is true
- Calls `POST /api/attempts/:id` with `{ answers: [answer] }`
- Clears input field
- Displays success message with auto-dismiss (using `setTimeout` and event listeners)
- Reloads attempts list

**Display Elements**:

- Difficulty badge
- Prior attempts history

**Login Page** (`client/app/login/page.jsx`):

- Uses `login(email, password)` from AuthContext
- Redirects authenticated users:
  - Admins to `/admin`
  - Non-admins to `/challenges`
- Google Identity Integration:
  - Loads Google Identity script dynamically
  - Initializes with `NEXT_PUBLIC_GOOGLE_CLIENT_ID`
  - Callback invokes `googleLogin(response.credential)`
  - Redirects based on `isAdmin` status

**Register Page** (`client/app/register/page.jsx`):

- Uses `register(name, email, password)` and `googleLogin` methods
- Redirects authenticated users (same logic as login)
- Includes client-side validation:
  - Email format validation
  - Password complexity requirements

**Admin Dashboard** (`client/app/admin/page.jsx`):

- Uses `useAuth` to determine current user
- Displays appropriate messages:
  - Not authenticated: "Authentication Required"
  - Authenticated but not admin: "Access Denied"
- Admin users:
  - Calls `apiRequest("/api/stats/overview")` for statistics
  - Loads: `totalProblems`, `totalUsers`, `totalAttempts`
  - Renders dashboard UI with stats and navigation links

**Admin Problem Management**:

**Problems List** (`client/app/admin/problems/page.jsx`):

- Loads problems via `apiRequest("/api/problems/admin/all")` (includes `flagAnswer`)
- Local filtering by difficulty
- Provides links to create and edit problems
- `handleDelete` function:
  - Confirms deletion in browser
  - Calls `DELETE /api/problems/:id`
  - Removes deleted problem from state
- Displays authentication and access control messages

**Create Problem** (`client/app/admin/problems/new/page.jsx`):

- Form state: `title`, `description`, `flagAnswer`, `difficulty`, `points`
- Note: `points` field exists in frontend form but not in backend model
- On submit: Calls `POST /api/problems` with form data
- Redirects to `/admin/problems` on success
- Implements authentication and authorization checks

**Edit Problem** (`client/app/admin/problems/[id]/edit/page.jsx`):

- Reads `id` from `useParams()`
- On load: Calls `GET /api/problems/admin/:id` and populates form
- Form fields: `title`, `description`, `flagAnswer`, `difficulty`, `category`, `points`
- Note: `category` and `points` fields exist in frontend form but not in backend model
- On submit: Uses `PUT /api/problems/:id` to update
- Handles loading and error states
- Implements admin access checks

**Admin User Management**:

**Users List** (`client/app/admin/users/page.jsx`):

- Fetches `GET /api/users` (admin-only endpoint)
- Displays: user names, emails, admin status, creation dates

**User Detail** (`client/app/admin/users/[id]/page.jsx`):

- User info: `GET /api/users/:id`
- User attempts: `GET /api/users/:id/attempts`
- Renders attempts with problem title and difficulty (from populated `problemId`)

**Leaderboard Page** (`client/app/leaderboard/page.jsx`):

- Calls `apiRequest("/api/leaderboard")`
- Displays ranked entries with:
    - Rank-specific badges for top positions
    - Fields: `rank`, `name`, `email`, `solvedCount`, `totalAttempts`

**Support Page** (`client/app/support/page.jsx`) and Support API:

**Frontend**:

- Form state: `{ name, email, subject, category, message }`
- Uses `react-toastify` for user notifications
- Client-side validation of required fields

- On submit: Sends `POST /api/support` with JSON body
- Displays notifications based on response's `success` and `message` fields

**Backend Route** (`client/app/api/support/route.js`):

- Next.js API route implementation
- Validates:
  - Presence of required fields
  - Email format using regex
- Configures `nodemailer` transporter:
  - Gmail SMTP
  - Environment variables: `EMAIL_USER`, `EMAIL_PASSWORD`, `SUPPORT_EMAIL`
- Sends two emails:
  - Admin/support email: HTML+text format with all details and timestamp
  - User auto-reply: HTML+text format with summary and optional documentation link (uses `NEXT_PUBLIC_APP_URL` or fallback)
- Returns JSON response:
  - Success: `{ success: true, message: ... }` with HTTP 200
  - Error: `{ success: false, message: ... }` with HTTP 500
  - Logs errors to console

**Additional Content Pages**:

- Informational pages exist under `client/app/` including:
  - About
  - Documentation
  - Privacy Policy
  - Terms of Service
- Implemented as Next.js pages providing platform content and documentation

---

# 5. Docker and Deployment Configuration

## Docker Compose Configuration

**File**: `docker-compose.yml`

**Server Service**:

- Build context: `./server`
- Environment file: `./server/.env.docker`
- Environment variables:
  - `PORT=5000`
  - `NODE_ENV=production`
- Port mapping: `5000:5000`

**Client Service**:

- Build context: `./client`
- Environment variables:
  - `BACKEND_URL=http://server:5000`
  - `NODE_ENV=production`
- Dependencies: server
- Port mapping: `3000:3000`

### Client Dockerfile

**File**: `client/Dockerfile`

Build process:

- Base image: `node:20-alpine`
- Install dependencies: `npm install`
- Build application: `npm run build`
- Environment: `NODE_ENV=production`
- Exposed port: `3000`
- Start command: `npm start`

### Server Dockerfile

**File**: `server/Dockerfile`

Build process:

- Base image: `node:20-alpine`
- Install production dependencies: `npm install --omit=dev`
- Environment: `NODE_ENV=production`
- Exposed port: `5000`
- Start command: `node src/server.js`

---

## 6. Summary of Implemented Responsibilities

### Backend Responsibilities

**Authentication and Authorization**:

- User registration with bcrypt password hashing
- Traditional login with credential verification
- Google OAuth authentication and account linkage via Google ID tokens
- JWT issuance with 7-day expiration

- HTTP-only cookie-based session management
- Role-based authorization distinguishing admin and non-admin users

**Problem Management**:

- CRUD operations on CTF challenges
- Public API endpoints exclude flag answers
- Admin API endpoints include complete problem data including flags
- Association of problems with creator users

**Attempt Tracking**:

- Recording of user submission attempts
- Correctness evaluation against flag answers
- Per-user-per-problem uniqueness constraint on correct solutions
- Prevention of multiple correct submissions for the same problem

**Leaderboard System**:

- Aggregation of user performance metrics
- Calculation of solved problem counts and total attempts
- Ranking based on problems solved and attempt efficiency
- Exclusion of admin accounts from leaderboard

**Administrative Statistics**:

- Total user count
- Total problem count
- Total attempt count
- Correct and incorrect attempt counts

## Frontend Responsibilities

**State Management**:

- Centralized authentication state via React Context
- Global access to user data and authentication methods
- Loading state management during initial authentication check

**Navigation and Routing**:

- Dynamic navigation adapting to authentication state and user role
- Protected route implementation with role-based redirection
- Automatic redirection for authenticated users on auth pages
- Mobile-responsive navigation with toggle menu

**User Interface Components**:

**Challenge System**:

- Challenge listing with filtering and search functionality
- Challenge detail pages with flag submission interface
- Visual indicators for solved challenges
- Difficulty-based styling and icons
- Attempt history display

**Admin Interface**:

- Dashboard with platform statistics
- Problem management (create, read, update, delete)
- User listing and detail views
- User attempt history viewing

**Leaderboard Display**:

- Ranked user listing
- Visual indicators for top performers
- Display of performance metrics

**Support System**:

- Contact form with category selection
- Email integration for support ticket submission
- Automated reply email system
- Client-side form validation

**Additional Features**:

- Marketing and informational pages
- Documentation pages
- Terms of service and privacy policy pages

All implementations are based on the code and configuration present in the repository without introducing assumptions beyond what is explicitly implemented in the codebase.