Secure CTF Platform – Project Report

Cybersecurity Project – Web Application (Next.js + Firebase) – 2026

Group members:
Romain Herrenknecht · Stanislas Thibaud · Gaspard Auclair ·
Maël Tellus · Romain Oualid

---

1. Context and Project Objectives

The objective of this project is to design and implement a secure web-based Capture The Flag (CTF) platform allowing multiple users to authenticate, browse challenges, submit solutions, and view a global ranking, while strictly complying with cybersecurity best practices.

The core constraint of the project is the following:

No critical logic must be executed on the client side.
All validation, verification, and sensitive operations must be handled server-side.

This constraint drives the entire architecture and security model of the platform. The project focuses not only on functional completeness but also on demonstrating a correct and defendable security-oriented design.

2. Functional Scope

The platform fulfills the following functional requirements:

- Multi-user support
- Secure registration and authentication
- Full CTF workflow:
    - Challenge listing
    - Challenge detail view
    - Flag submission
    - Automatic verification
    - Scoring
- Global leaderboard
- Public user profiles

Delivered Features

- Secure authentication using Email/Password and OAuth providers (Google, GitHub)
- Mandatory email verification before accessing protected areas
- Challenge list with filtering (category, difficulty, search)
- Challenge detail page with downloadable attachments and hints
- Server-side flag submission and verification
- Scoring based on challenge difficulty
- Prevention of duplicate scoring for the same challenge
- Global leaderboard with clickable public profiles
- “Coming Soon” challenges to indicate higher-level content

- Voluntary “I want to cheat” feature, publicly recorded on user profiles as a pedagogical choice

---

3. Technical Choices

Frontend

- Next.js (App Router)
- TypeScript (strict mode)
- Tailwind CSS for a consistent and responsive UI
- Firebase Client SDK for authentication context

Backend

- Next.js API Routes for all critical logic
- Firebase Admin SDK for secure server-side access to Firestore
- Application-level rate limiting

Important note:
No Firebase Cloud Functions are used. All validation, authentication verification, and scoring logic are handled by the Next.js server layer.

This choice ensures full control over security while avoiding unnecessary complexity and paid services.

4. Global Architecture

The architecture enforces a strict separation between the client and critical logic.

The client:

- Displays UI
- Sends authenticated requests
- Never accesses secrets
- Never validates flags
- Never updates scores

The server:

- Verifies Firebase ID tokens
- Validates all inputs
- Performs all sensitive operations
- Updates Firestore using the Admin SDK

```
Client (Next.js UI)
   |
   | Firebase ID Token
   v
Next.js API Routes (Server-side logic)
   |
   v
Firestore (users / challenges / submissions)
```

Core Security Rule

Never trust the client.
All validation and score updates are performed server-side using authenticated API routes.

---

5. Data Model (Firestore)

Main Collections

users/{uid}

- displayName
- score
- solvedChallenges[]
- cheatedChallenges[]
- isBanned
- createdAt

Role:

- User profile
- Global score
- Progress tracking

challenges/{challengeId}

- title
- category
- difficulty
- points
- flagHash
- isComingSoon

Role:

- Challenge definition
- Flags are never stored in plaintext

submissions/{submissionId}

- userId
- challengeId
- status
- submittedAt
- ipHash

Role:

- Submission traceability
- Minimal audit logging

6. Security Measures

6.1 Authentication

- Firebase Authentication (Email/Password + OAuth)
- Mandatory email verification before accessing protected routes
- Generic error messages to prevent user enumeration
- Systematic Firebase ID token verification on every API request

6.2 HTTPS

- HTTPS enforced in production via Vercel hosting
- Security headers enabled (CSP, HSTS, X-Frame-Options)

Note:
Local development runs over HTTP on localhost, which is acceptable for development purposes.

6.3 Secure Storage

- Flags are never stored in plaintext
- Flags are stored as salted cryptographic hashes
- Secrets (Admin SDK credentials) are stored only as server-side environment variables
- All sensitive Firestore writes are performed via the Admin SDK

6.4 Personal Data Protection

- Public profiles never expose email addresses
- IP addresses are hashed before storage
- Only minimal necessary data is logged

7. Protection Against Web Attacks

7.1 Rate Limiting and Anti-Bruteforce

- Rate limiting applied to sensitive API routes (flag submission, authentication-related actions)
- Fixed request windows sufficient for demonstration purposes
- Security-relevant events are minimally logged

7.2 Access Control

- Protected routes (/challenges, /leaderboard) require authentication and verified email
- Firestore security rules provide defense-in-depth
- Direct client writes to sensitive collections are restricted

7.3 Server-side Validation

- All inputs (challengeId, flags) are validated server-side
- No trust is placed in client-side data
- Token verification is systematic

8. OWASP Top 10 Coverage

Risk	Mitigation
A01 Broken Access Control	API-based access control + Firestore Rules
A02 Cryptographic Failures	Salted hashing of flags, no secret exposure
A03 Injection	Server-side validation, Firestore parameterization
A04 Insecure Design	Critical logic isolated in API Routes
A05 Security Misconfiguration	Security headers, controlled environment variables
A07 Identification & Authentication Failures	Email verification, token validation, generic errors
A09 Security Logging & Monitoring Failures	Minimal security logs with hashed IPs

This table provides a synthesis. Implementation details are visible in the codebase and documentation.

9. Automatic Verification, Scoring, and Ranking

9.1 Flag Verification

When a flag is submitted:

1. The server normalizes the input
2. Computes a salted hash
3. Compares it to the stored hash
4. Updates the database using an atomic Firestore transaction

9.2 Scoring

Example scoring system:

- Easy: 100 points
- Medium: 250 points
- Hard: 500 points
- Expert: 1000 points
- Insane: 2000 points

The scoring system reflects progressive difficulty. “Coming Soon” challenges indicate higher-level content not accessible in the current instance.

9.3 Leaderboard

- Global leaderboard sorted by score
- Public profiles accessible from the leaderboard
- No client-side score computation

---

10. Demonstration Procedure

Prerequisites

- Node.js installed
- Firebase project configured (Auth + Firestore)
- Environment variables configured server-side

Recommended Steps

1. Generate challenge files: `npm run generate-files`
2. Reset and seed Firestore: `npm run reset-challenges`
3. Start the application: `npm run dev`
4. Create an account and verify the email
5. Access `/challenges`
6. Solve a basic challenge and observe score update
7. View the leaderboard and open a public profile

Note:
After reseeding, Firestore document IDs change. A reset script is provided if a clean state is required.

11. Limitations and Perspectives

Limitations (Student Context)

- Rate limiting is implemented in memory; a production system would require Redis or equivalent
- “Coming Soon” challenges are intentionally unsolvable
- The “I want to cheat” feature is a pedagogical choice to make cheating visible and traceable

Perspectives

- Secure admin interface for challenge creation and editing
- More realistic challenge assets (PCAPs, binaries, memory dumps)
- Improved Firestore indexing and caching for scalability

12. Conclusion

This project delivers a complete and security-focused CTF platform. The implementation emphasizes strong authentication, server-side flag verification, secure data storage, access control, rate limiting, and a strict separation of responsibilities.

The platform fulfills all project objectives: multi-user support, challenge browsing, flag submission, automatic verification, scoring, and ranking.
This report documents the design decisions and security mechanisms. The source code serves as the primary proof of implementation.