Audits & Reports

- Security Audit
- Security QA Audit
- Compliance Overview
- Security Architecture
- Secret Rotation Runbook — 2026-04-20

Security Audit

Drop Security Audit (originally FontelePay)

> **Rebrand note:** FontelePay was renamed to Drop. This audit predates the backend implementation and PSD2 pass-through architecture. Many issues identified here (localStorage PIN, client-side auth) have been addressed in the current backend. See `docs/security/SECURITY-ARCHITECTURE.md` for current security posture.

Date: 2026-02-05 **Auditor:** John (AI Director) **Scope:** Full MVP application scan (pre-backend, client-side only)

Executive Summary

Category	Status	Issues
Dependencies	PASS	0 vulnerabilities
Hardcoded Secrets	PASS	None found
XSS Prevention	PASS	No dangerous patterns
Data Storage	⚠ WARNING	PIN in localStorage
Authentication	⚠ WARNING	Client-side only
Input Validation	⚠ WARNING	Minimal validation

Overall: ACCEPTABLE FOR MVP DEMO, NOT PRODUCTION READY

---

1. Dependency Audit

```
$ npm audit
found 0 vulnerabilities
```

 All npm packages are up to date with no known vulnerabilities.

---

2. Secret Management

Checked For:

- Hardcoded API keys
- Hardcoded passwords
- Hardcoded tokens
- .env files in repo

Findings:

 No hardcoded secrets in source code No .env files committed Mock tokens are clearly labeled as mocks

Recommendation:

- Create `.env.example` template for production
- Use environment variables for real API keys
- Add `.env*` to .gitignore (already present)

---

3. XSS Prevention

Checked For:

- `dangerouslySetInnerHTML`
- `eval()`
- Direct `innerHTML` manipulation

Findings:

 No dangerous DOM manipulation patterns found React's built-in XSS protection active

4. Data Storage (CRITICAL)

Issue: PIN Stored in localStorage

```
// AppContext.tsx:87
localStorage.setItem(STORAGE_KEY, JSON.stringify(user));
// user object includes plaintext PIN!
```

Risk Level: **HIGH** for production

Why It Matters:

- localStorage accessible via XSS attacks
- localStorage persists after browser close
- PIN visible in browser DevTools
- No encryption

Fix Required for Production:

```
// Option 1: Never store PIN client-side
// Use server-side session + httpOnly cookies

// Option 2: Store hashed PIN (still not ideal)
import bcrypt from 'bcryptjs';
const hashedPin = await bcrypt.hash(pin, 10);

// Option 3: Use Web Crypto API
const encoder = new TextEncoder();
const data = encoder.encode(pin);
const hash = await crypto.subtle.digest('SHA-256', data);
```

Immediate Mitigation:

- For MVP demo: ACCEPTABLE (no real money)
- For production: MUST implement server-side auth

5. Authentication Analysis

Current State:

- □ PIN required for login
- □ Wrong PIN rejected
- ⚠ No rate limiting on PIN attempts
- ⚠ No session expiry
- ⚠ No logout on inactivity
- □ No server-side verification

OWASP Top 10 Review:

Vulnerability	Status	Notes
A01 Broken Access Control	⚠	No server-side checks
A02 Cryptographic Failures	□	PIN not hashed
A03 Injection	□	React prevents XSS
A04 Insecure Design	⚠	Client-only auth
A05 Security Misconfiguration	□	N/A for static
A06 Vulnerable Components	□	npm audit clean
A07 Auth Failures	⚠	No rate limiting
A08 Data Integrity	□	No deserialization
A09 Logging Failures	⚠	No audit logging
A10 SSRF	□	No server requests

---

6. Input Validation

Checked Areas:

Input	Validation	Status
Phone number	Length >= 6	⚠ Weak
OTP	Length === 6	⚠ No server verify

Input	Validation	Status
PIN	Length === 4	✅ OK
Transfer amount	> 0, <= balance	✅ OK
IBAN	None	⚠ Should validate format
Card number	Length === 16	⚠ No Luhn check

Recommendations:

- Add IBAN format validation (checksum)
- Add Luhn algorithm for card numbers
- Add phone number format validation
- Implement server-side validation

---

7. Production Readiness Checklist

Must Have Before Launch:

- ☐ Server-side authentication (NextAuth.js or similar)
- ☐ Hashed/encrypted PIN storage
- ☐ HTTPS enforcement
- ☐ Rate limiting on auth endpoints
- ☐ Session management with expiry
- ☐ Audit logging
- ☐ Error monitoring (Sentry)
- ☐ CSP headers
- ☐ CORS configuration

Nice to Have:

- ☐ 2FA / MFA
- ☐ Device fingerprinting
- ☐ Anomaly detection
- ☐ PCI-DSS compliance audit

---

8. Recommendations Summary

Immediate (Before any real users):

1. Move authentication to server-side
2. Hash PINs with bcrypt
3. Add rate limiting

Short-term (Before production):

4. Implement proper session management
5. Add audit logging
6. Setup error monitoring
7. Add input validation

Long-term (For compliance):

8. PCI-DSS audit for card handling
9. Penetration testing
10. Security certification

Conclusion

The Drop MVP (originally FontelePay) is **acceptable for demonstration purposes** but requires significant security improvements before handling real user data or money.

Key Risk: Plaintext PIN storage in localStorage

Mitigation: This is a mock/demo environment with no real financial transactions. All "money" is simulated.

Next Steps: Implement server-side auth before any production deployment.

Audit completed by automated security scan + manual code review.

Security QA Audit

Drop App Security & QA Audit Report

Date: 2026-02-11 **Auditor:** John (AI Director) **Scope:** Next.js 16 fintech app with SQLite database, JWT auth, 24 API routes

> **HISTORICAL NOTE (2026-03-03):** This audit was performed against the pre-ADR-014 codebase which used SQLite + dual-driver. The SQLite backend and `better-sqlite3` dependency have since been removed. Current architecture: PostgreSQL 16 (all environments) + Drizzle ORM (ADR-014). SQLite-specific findings and the "SQLite in production" recommendation are resolved. **Total API Code:** 1,102 lines

---

Executive Summary

Drop is a payment application for all residents in Norway/Scandinavia, with remittance and QR payment features. Drop uses a PSD2 pass-through model — it never holds customer money. This audit reviewed all source code against OWASP Top 10 security risks, authentication/authorization implementation, database security, and code quality.

Overall Risk Level: MEDIUM-HIGH **Critical Issues:** 3 **High Priority Issues:** 7 **Medium Priority Issues:** 5 **Low Priority Issues:** 4

The application has solid foundations (bcrypt password hashing, parameterized SQL queries, JWT authentication) but has critical gaps in production readiness including hardcoded secrets, missing CSRF protection, exposed sensitive data, and insufficient rate limiting.

CRITICAL ISSUES (Fix Immediately)

1. JWT Secret Hardcoded with Weak Default

File: `src/lib/auth.ts:5` **Risk:** Authentication bypass, token forgery **Issue:**

```
const jwtSecretRaw = process.env.JWT_SECRET || "drop-dev-secret-DO-NOT-USE-IN-PROD";
```

Default fallback is a publicly visible hardcoded string. While there's a warning logged, the app will still start in production with this weak secret.

Impact: Attacker can forge valid JWT tokens and impersonate any user if JWT_SECRET env var is not set.

Fix:

```
const jwtSecretRaw = process.env.JWT_SECRET;
if (!jwtSecretRaw) {
  throw new Error("FATAL: JWT_SECRET environment variable is required");
}
const JWT_SECRET = new TextEncoder().encode(jwtSecretRaw);
```

Recommendation: NEVER allow fallback secrets. Fail fast and loud.

---

2. Full Card Numbers and CVV Exposed via API

File: `src/app/api/cards/[id]/route.ts:28-31` **Risk:** PCI-DSS violation, credential theft **Issue:**

```
data: {
  cardNumber: card.card_number,  // FULL 16-digit number
  cvv: card.cvv,                 // CVV exposed
}
```

The GET endpoint returns the full card number and CVV in plaintext. This violates PCI-DSS compliance and enables account takeover if tokens are compromised.

Impact:

- Regulatory violation (PCI-DSS Level 1 requirement breach)
- If JWT is stolen, attacker gets full payment credentials
- No legitimate client-side use case requires full card number in GET response

Fix:

```
data: {
  lastFour: card.last_four,     // Only last 4 digits
  expiry: card.expiry,
  // NEVER expose full cardNumber or CVV via GET
}
```

Recommendation:

- Only expose last 4 digits in GET responses
- Only return full details on card creation (POST) via secure channel
- Consider encrypting card_number and cvv at rest in database

---

3. Insufficient Balance Validation (Race Condition Risk)

File: `src/app/api/transactions/remittance/route.ts:68-74` **Risk:** Negative balance, double-spend
Issue:

```
const result = db.prepare(
  "UPDATE users SET balance = balance - ? WHERE id = ? AND balance >= ?"
).run(total, u.id, total);
if (result.changes === 0) {
  throw new Error("insufficient_balance");
}
```

While there's optimistic locking, concurrent requests can still cause race conditions because:

1. Rate limiting is per-IP (10 req/min), not per-user
2. User can send 10 parallel requests before first one completes
3. Each checks balance BEFORE deduction

Impact: User can overdraw account by sending multiple remittances simultaneously.

Fix:

```
// Add per-user transaction lock
const userLocks = new Map<string, Promise<void>>();

export async function POST(request: NextRequest) {
  const { user, error } = await requireAuth();
  if (error) return error;
```

```
  const userId = (user as Record<string, unknown>).id as string;

  // Wait for previous transaction from same user
  if (userLocks.has(userId)) {
    await userLocks.get(userId);
  }

  const txPromise = (async () => {
    // ... transaction logic
  })();

  userLocks.set(userId, txPromise);
  try {
    return await txPromise;
  } finally {
    userLocks.delete(userId);
  }
}
```

Recommendation: Implement per-user transaction serialization or use database-level row locking.

HIGH PRIORITY ISSUES (Fix Before Deployment)

4. No CSRF Protection

Files: All API routes **Risk:** Cross-Site Request Forgery attacks **Issue:**

- JWT stored in httpOnly cookie (good)
- SameSite=strict set (good)
- BUT no CSRF token validation on state-changing operations (POST, PATCH, DELETE)

Impact: Attacker can trick authenticated user into making unwanted transactions via malicious website.

Attack Vector:

```
<!-- Attacker's site -->
<form action="https://drop.app/api/transactions/remittance" method="POST">
  <input name="recipientId" value="attacker_recipient">
  <input name="amount" value="50000">
</form>
<script>document.forms[0].submit()</script>
```

Fix: Implement Next.js CSRF protection:

```
// middleware.ts (create in root)
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(request: NextRequest) {
  // CSRF check for state-changing methods
  if (['POST', 'PUT', 'PATCH', 'DELETE'].includes(request.method)) {
    const origin = request.headers.get('origin');
    const host = request.headers.get('host');

    // Block requests from different origins
    if (origin && !origin.endsWith(host || '')) {
      return new NextResponse('CSRF validation failed', { status: 403 });
    }
  }
  return NextResponse.next();
}

export const config = {
  matcher: '/api/:path*',
};
```

Recommendation: Add CSRF token validation or strict origin checking.

---

5. In-Memory Rate Limiter Resets on Process Restart

File: `src/lib/middleware.ts:5` **Risk:** Rate limit bypass, brute force attacks **Issue:**

```
const rateLimitMap = new Map<string, { count: number; resetAt: number }>();
```

Rate limits are stored in process memory. Resets on every deployment or crash.

Impact:

- Attacker can bypass rate limits by triggering app restart (e.g., via resource exhaustion)
- In serverless/multi-instance deployment, each instance has separate limits
- Login endpoint limited to 10 attempts, but resets frequently

Fix: Use persistent rate limiting:

```
// Option 1: Redis (production)
import Redis from 'ioredis';
const redis = new Redis(process.env.REDIS_URL);

export async function rateLimit(ip: string, limit: number, windowMs: number): Promise<boolean>
{
  const key = `ratelimit:${ip}`;
  const count = await redis.incr(key);
  if (count === 1) {
    await redis.pexpire(key, windowMs);
  }
  return count <= limit;
}

// Option 2: SQLite (MVP fallback)
db.prepare(`
  CREATE TABLE IF NOT EXISTS rate_limits (
    ip TEXT PRIMARY KEY,
    count INTEGER,
    reset_at INTEGER
  )
`);
```

Recommendation: Migrate to Redis or database-backed rate limiting before production.

---

6. Weak Password Hashing for Demo User (Migration Risk)

File: `src/lib/db.ts:141` **Risk:** Account compromise **Issue:**

```
// SHA-256 of "demo1234" — NOT production-safe, just for MVP demo
"0ead2060b65992dca4769af601a1b3a35ef38cfad2c2c465bb160ea764157c5d",
```

Demo user password is SHA-256 (unsalted, fast hash). While login endpoint auto-migrates to bcrypt on successful login, this creates a window where:

1. If demo credentials leak, password is crackable
2. Migration only happens on successful login, so if user never logs in, weak hash remains

Impact: Demo/legacy accounts vulnerable to rainbow table attacks.

Fix:

```
// Remove demo user from seed data
// OR force migration on app startup
function migrateWeakHashes(db: Database.Database) {
  const weak = db.prepare(
    "SELECT id, email FROM users WHERE LENGTH(password_hash) = 64"
  ).all();

  for (const user of weak) {
    console.warn(`User ${user.email} has weak hash - forcing password reset`);
    // Option 1: Delete weak accounts
    // Option 2: Mark as requiring password reset
    db.prepare("UPDATE users SET kyc_status = 'locked' WHERE id = ?").run(user.id);
  }
}
```

Recommendation: Remove demo user from production seed, or pre-hash with bcrypt.

---

7. No Content-Security-Policy Headers

Files: All routes **Risk:** XSS attacks, clickjacking **Issue:** No CSP headers set. While no `dangerouslySetInnerHTML` was found, defense-in-depth requires CSP.

Impact:

- If XSS vulnerability is introduced later, no fallback protection
- No protection against clickjacking
- No restriction on script sources

Fix: Add CSP headers in `next.config.ts`:

```
const nextConfig: NextConfig = {
  async headers() {
    return [
      {
        source: '/:path*',
        headers: [
          {
            key: 'Content-Security-Policy',
            value: [
              "default-src 'self'",
              "script-src 'self' 'unsafe-inline' 'unsafe-eval'", // Next.js requires unsafe-
inline/eval in dev
              "style-src 'self' 'unsafe-inline'",
              "img-src 'self' data: https:",
              "font-src 'self' data:",
              "connect-src 'self'",
              "frame-ancestors 'none'",
            ].join('; '),
          },
          {
            key: 'X-Frame-Options',
            value: 'DENY',
          },
          {
            key: 'X-Content-Type-Options',
            value: 'nosniff',
          },
          {
            key: 'Referrer-Policy',
            value: 'strict-origin-when-cross-origin',
          },
        ],
      },
    ];
  },
};
```

Recommendation: Implement strict CSP before production deployment.

---

8. SQL Injection Risk in Dynamic WHERE Clause

File: `src/app/api/merchants/dashboard/route.ts:22-28` **Risk:** SQL injection (low probability, but present) **Issue:**

```
let dateOffset: string;
if (period === "week") {
  dateOffset = "-7 days";
} else if (period === "month") {
  dateOffset = "-30 days";
} else {
  dateOffset = "start of day";
}

const stats = db.prepare(`
  ... WHERE ... AND created_at >= datetime('now', ?)
`).get(merchant.id, dateOffset);
```

While there's a whitelist check, the `dateOffset` variable is passed directly to SQLite's `datetime()` function. SQLite's datetime accepts arbitrary strings, and malicious input could cause unexpected behavior.

Current State: The whitelist prevents injection (period is checked), but code is fragile.

Fix: Use parameterized datetime calculations:

```
const periodMap = {
  today: 0,
  week: 7,
  month: 30,
} as const;

const days = periodMap[period as keyof typeof periodMap] ?? 0;

const stats = db.prepare(`
  SELECT ...
  FROM transactions
  WHERE merchant_id = ?
    AND type = 'qr_payment'
```

```
    AND status = 'completed'
    AND julianday('now') - julianday(created_at) <= ?
`).get(merchant.id, days);
```

Recommendation: Avoid passing strings to SQL datetime functions. Use numeric calculations.

---

9. Insufficient Input Validation on Amount Fields

Files: Multiple transaction endpoints **Risk:** Business logic bypass, financial loss **Issue:**

```
// remittance: amount must be 100-50,000
// top-up: amount must be 0-100,000
// qr-payment: amount >= 1
```

Issues:

1. No validation for negative amounts (though DB would reject, better to fail fast)
2. No validation for decimal precision (could cause rounding errors)
3. Top-up has NO payment verification — user can add unlimited NOK to balance
4. No daily/monthly transaction limits

Fix:

```
// 1. Validate amount precision
function validateAmount(amount: number): string[] {
  const errors: string[] = [];
  if (amount < 0) errors.push("Amount cannot be negative");
  if (!Number.isFinite(amount)) errors.push("Amount must be a valid number");
  if (Math.round(amount * 100) !== amount * 100) {
    errors.push("Amount can only have 2 decimal places");
  }
  return errors;
}

// 2. Add transaction limits per user
const dailyLimit = db.prepare(`
  SELECT COALESCE(SUM(amount), 0) as total
  FROM transactions
  WHERE user_id = ? AND type = 'remittance' AND created_at >= date('now')
`).get(userId) as { total: number };
```

```
if (dailyLimit.total + amount > 50000) {
  return jsonError("limit_exceeded", "Daily remittance limit (50,000 NOK) exceeded", 429);
}

// 3. Top-up requires payment verification
// CRITICAL: Remove mock top-up, integrate real payment gateway
```

Recommendation: Add comprehensive amount validation and real payment integration for top-up.

---

10. No Audit Logging for Sensitive Operations

Files: All transaction routes **Risk:** Compliance violation, no forensics **Issue:** No audit trail for:

- Login attempts (successful/failed)
- Balance changes
- Transaction creation/modification
- Card creation/freezing
- Recipient changes

Impact:

- Cannot investigate fraud
- Cannot prove compliance (GDPR, PSD2)
- Cannot detect account takeover

Fix:

```
// Create audit log table
CREATE TABLE audit_logs (
  id TEXT PRIMARY KEY,
  user_id TEXT,
  ip TEXT,
  action TEXT,  -- 'login', 'transaction.create', 'card.freeze', etc.
  entity_type TEXT,
  entity_id TEXT,
  old_value TEXT,  -- JSON snapshot before change
  new_value TEXT,  -- JSON snapshot after change
  timestamp TEXT DEFAULT (datetime('now'))
);
```

```
// Log every sensitive operation
function auditLog(userId: string, ip: string, action: string, details: object) {
  db.prepare(`
    INSERT INTO audit_logs (id, user_id, ip, action, new_value)
    VALUES (?, ?, ?, ?, ?)
  `).run(randomId('audit'), userId, ip, action, JSON.stringify(details));
}

// Usage
auditLog(user.id, getClientIp(request), 'transaction.remittance.create', {
  amount, recipientId, txId
});
```

Recommendation: Implement comprehensive audit logging before handling real money.

MEDIUM PRIORITY ISSUES (Fix in Next Sprint)

11. Database File Location Not Configurable

File: `src/lib/db.ts:5` **Risk:** Backup/restore complexity, data loss **Issue:**

```
const DB_PATH = path.join(process.cwd(), "drop.db");
```

Database stored in app directory. In production, this could be ephemeral (e.g., Docker container, serverless).

Fix:

```
const DB_PATH = process.env.DATABASE_PATH || path.join(process.cwd(), "drop.db");
```

Recommendation: Use external volume or managed database in production.

---

12. Mock Data File Still Present

File: `src/lib/mock-data.ts` **Risk:** Confusion, accidental use **Issue:** Mock data file exists but is unused (grep confirms no imports). Should be removed to avoid future mistakes.

Fix: Delete file or move to `tests/fixtures/` if needed for testing.

Recommendation: Remove unused code before deployment.

---

13. Health Check Does Not Test Foreign Keys

File: `src/app/api/health/route.ts:7` **Risk:** False positive health status **Issue:**

```
const result = db.prepare("SELECT 1 as ok").get() as { ok: number };
```

Health check only tests basic connectivity. Does not verify:

- Foreign key constraints enabled
- WAL mode active
- Indexes present
- Schema version

Fix:

```
export async function GET() {
  try {
    const db = getDb();

    // Test basic query
    const basic = db.prepare("SELECT 1 as ok").get() as { ok: number };

    // Test foreign keys enabled
    const fk = db.pragma("foreign_keys", { simple: true });

    // Test table existence
    const tables = db.prepare(
      "SELECT COUNT(*) as c FROM sqlite_master WHERE type='table'"
    ).get() as { c: number };

    return NextResponse.json({
      status: "ok",
      db: basic.ok === 1 ? "connected" : "error",
      foreignKeys: fk === 1 ? "enabled" : "DISABLED",
```

```
      tables: tables.c,
      timestamp: new Date().toISOString(),
      version: "1.0.0",
    });
  } catch (error) {
    return NextResponse.json(
      {
        status: "error",
        db: "disconnected",
        error: (error as Error).message,
        timestamp: new Date().toISOString()
      },
      { status: 503 }
    );
  }
}
```

Recommendation: Enhance health check to verify critical database state.

---

14. No Email Validation Beyond Basic @ Check

File: `src/app/api/auth/register/route.ts:26` **Risk:** Invalid emails in database **Issue:**

```
if (!email || typeof email !== "string" || !email.includes("@"))
  errors.push("Valid email required");
```

Accepts invalid emails like `user@`, `@domain.com`, `user space@domain.com`.

Fix:

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
if (!email || !emailRegex.test(email)) {
  errors.push("Valid email required");
}

// Better: use email-validator package
import validator from 'validator';
if (!validator.isEmail(email)) {
  errors.push("Valid email required");
}
```

Recommendation: Use proper email validation library.

---

15. Card Number Generation Not Luhn-Valid

File: `src/app/api/cards/route.ts:49` **Risk:** Card validation failures **Issue:**

```
const cardNumber = "4532" + Array.from({ length: 12 }, () =>
  Math.floor(Math.random() * 10)
).join("");
```

Generates random digits without Luhn checksum. While this is a virtual card, some payment processors validate Luhn algorithm.

Fix:

```
function generateLuhnValid(prefix: string, length: number): string {
  const digits = prefix.split('').map(Number);

  // Generate random digits (all but last)
  while (digits.length < length - 1) {
    digits.push(Math.floor(Math.random() * 10));
  }

  // Calculate Luhn checksum
  let sum = 0;
  let parity = (length - 1) % 2;
  for (let i = 0; i < length - 1; i++) {
    let digit = digits[i];
    if (i % 2 === parity) {
      digit *= 2;
      if (digit > 9) digit -= 9;
    }
    sum += digit;
  }

  const checksum = (10 - (sum % 10)) % 10;
  digits.push(checksum);

  return digits.join('');
}
```

```
const cardNumber = generateLuhnValid("4532", 16);
```

Recommendation: Generate Luhn-valid card numbers for compatibility.

LOW PRIORITY ISSUES (Nice to Have)

16. Error Messages Leak Implementation Details

Files: Various API routes **Risk:** Information disclosure **Issue:** Error messages like "insufficient_balance", "not_found" help attackers enumerate valid IDs.

Recommendation: Use generic error messages for client, detailed logs server-side.

---

17. No Request ID for Debugging

Risk: Difficult to correlate logs **Recommendation:** Add request ID to all API responses:

```
export function jsonError(error: string, message: string, status: number) {
  const requestId = crypto.randomUUID();
  console.error(`[${requestId}] ${error}: ${message}`);
  return NextResponse.json({
    error,
    message,
    requestId
  }, { status });
}
```

---

18. TypeScript Strict Mode Enabled But Type Assertions Used

File: `tsconfig.json:7` **Risk:** Runtime type errors **Issue:** Strict mode enabled (good), but extensive use of `as Record<string, unknown>` bypasses type safety.

Recommendation: Define proper TypeScript interfaces for database models.

---

19. No Password Complexity Requirements

File: `src/app/api/auth/register/route.ts:27` **Risk:** Weak passwords **Issue:** Only requires 8 characters, no complexity rules.

Recommendation:

```
function validatePassword(password: string): string[] {
  const errors: string[] = [];
  if (password.length < 12) errors.push("Password must be at least 12 characters");
  if (!/[A-Z]/.test(password)) errors.push("Password must contain uppercase letter");
  if (!/[a-z]/.test(password)) errors.push("Password must contain lowercase letter");
  if (!/[0-9]/.test(password)) errors.push("Password must contain number");
  return errors;
}
```

CODE QUALITY OBSERVATIONS

Positive

1. **Parameterized SQL Queries** - All queries use `?` placeholders (GOOD)
2. **bcrypt for Password Hashing** - 12 rounds, industry standard (GOOD)
3. **JWT with jose Library** - Modern, secure JWT implementation (GOOD)
4. **httpOnly Cookies** - Prevents XSS token theft (GOOD)
5. **Foreign Keys Enabled** - Data integrity enforced (GOOD)
6. **WAL Mode** - Better concurrency for SQLite (GOOD)
7. **Database Transactions** - Atomic balance updates (GOOD)
8. **TypeScript Strict Mode** - Type safety enabled (GOOD)
9. **No dangerouslySetInnerHTML** - XSS prevention (GOOD)

Needs Improvement

1. **No TypeScript Interfaces for DB Models** - Excessive type assertions
2. **No Error Boundary** - Unhandled exceptions may leak stack traces

3. **No Logging Framework** - Only console.error in auth.ts
4. **No Monitoring/Metrics** - No instrumentation for performance tracking
5. **No API Documentation** - No OpenAPI/Swagger spec
6. **No Tests** - No unit/integration tests found
7. **Mixed Concerns** - db.ts contains both schema and seed data

---

DEPENDENCY AUDIT

Current Versions (2026-02-11)

- `next`: 16.1.6 (latest: 16.1.6) ✓
- `react`: 19.2.3 (latest: 19.2.4) - minor update available
- `bcryptjs`: 3.0.3 ✓
- `better-sqlite3`: 12.6.2 ✓
- `jose`: 6.1.3 ✓

Outdated Packages (Non-Critical)

- `@types/node`: 20.19.33 → 25.2.3 (major version available)
- `eslint`: 9.39.2 → 10.0.0 (major version available)

Recommendation: Update React to 19.2.4 for bug fixes. Defer @types/node and eslint major updates until tested.

Known Vulnerabilities

Run `npm audit` to check for CVEs. No critical vulnerabilities detected in package.json review.

DEPLOYMENT READINESS CHECKLIST

BLOCKER ISSUES (must fix before production):

- ☐ Remove hardcoded JWT_SECRET fallback
- ☐ Remove CVV exposure from card GET endpoint
- ☐ Implement CSRF protection

- [ ] Migrate rate limiting to persistent storage
- [ ] Remove mock top-up, integrate real payment gateway
- [ ] Add audit logging for financial transactions
- [ ] Configure CSP headers
- [ ] Remove demo user or pre-hash with bcrypt

HIGH PRIORITY (fix before MVP launch):

- [ ] Add per-user transaction serialization
- [ ] Implement comprehensive amount validation
- [ ] Add daily/monthly transaction limits
- [ ] Store database in persistent volume
- [ ] Enhance health check
- [ ] Add proper email validation

RECOMMENDED (fix in next sprint):

- [ ] Generate Luhn-valid card numbers
- [ ] Add request IDs for debugging
- [ ] Define TypeScript interfaces for models
- [ ] Add password complexity requirements
- [ ] Write unit tests for auth and transactions
- [ ] Add API documentation (OpenAPI)

SECURITY SCORE

Category	Score	Notes
Authentication	6/10	Good JWT + bcrypt, but weak secret fallback
Authorization	7/10	Proper user-scoped queries, but no RBAC beyond user/merchant
Injection	9/10	All parameterized, minor dateOffset risk
XSS	8/10	No dangerouslySetInnerHTML, but no CSP
CSRF	2/10	SameSite=strict helps, but no CSRF tokens

Category	Score	Notes
Data Exposure	4/10	CVV exposed, no audit logs
Rate Limiting	5/10	Implemented but in-memory
Cryptography	8/10	Strong bcrypt, secure JWT, but secrets management issues
Configuration	5/10	Hardcoded paths, missing env vars
Logging	3/10	Minimal logging, no audit trail

Overall Security Score: 57/100 (MEDIUM)

RECOMMENDATIONS SUMMARY

Immediate Actions (Before Production)

1. Remove JWT_SECRET fallback, fail if not set
2. Never expose CVV via API
3. Implement CSRF protection (origin checking or tokens)
4. Migrate rate limiting to Redis/database
5. Add audit logging for compliance
6. Set CSP headers in next.config.ts
7. Remove or properly hash demo user password

Next Sprint

1. Add comprehensive input validation
2. Implement transaction limits (daily/monthly)
3. Add per-user transaction locks
4. Write security tests
5. Document all API endpoints
6. Set up error monitoring (Sentry, etc.)

Future Enhancements

1. Implement 2FA for high-value transactions
2. Add fraud detection rules
3. Encrypt sensitive data at rest
4. Set up automated security scanning (Snyk, Dependabot)
5. Conduct penetration testing before launch

CONCLUSION

Drop has a solid technical foundation with proper use of bcrypt, parameterized SQL, and JWT authentication. However, **it is NOT production-ready** due to:

1. **CRITICAL:** Hardcoded secret fallbacks
2. **CRITICAL:** Exposed sensitive payment data (CVV)
3. **CRITICAL:** Missing CSRF protection
4. **HIGH:** In-memory rate limiting (easily bypassed)
5. **HIGH:** No audit logging (compliance risk)

Estimated Remediation Time:

- Critical issues: 8-12 hours
- High priority: 16-24 hours
- Medium priority: 8-12 hours
- **Total: 32-48 hours before production deployment**

Sign-off: This audit was conducted on source code only. A runtime penetration test is recommended before handling real financial transactions.

Audit Report Generated: 2026-02-11 **Next Review:** After critical fixes implemented

Compliance Overview

Drop Compliance Status

Last updated: 2026-02-13 **Source:** `legal/` directory (16 regulatory documents), `security/` directory (5 security documents), `legal/drop-gap-analysis-v2.md`, `legal/drop-regulatory-map-v2.md`

Overall Compliance Readiness: 8/100

Drop is an MVP-stage application using a **PSD2 pass-through model** (AISP reads bank balances, PISP initiates payments — Drop never holds customer money). Regulatory compliance is not expected at this stage, but documentation is being prepared. Cards are a FUTURE feature, gated behind feature flags.

Regulatory Framework

Applicable Regulations

Regulation	Norwegian Law	Relevance
PSD2	Betalingstjenesteloven (LOV-2018-11-23-85)	Core -- payment services regulation
AML/KYC	Hvitvaskingsloven (LOV-2018-06-01-23)	Core -- anti-money laundering
GDPR	Personopplysningsloven (LOV-2018-06-15-38)	Core -- personal data protection
ICT Security	IKT-forskriften / DORA	Required for financial enterprises
Financial Enterprise	Finansforetaksloven (LOV-2015-04-10-17)	Licensing and governance
Currency Registry	Valutaregisterloven	Cross-border payment reporting
Consumer Protection	Finansavtaleloven	Terms and user rights

Source: `legal/drop-regulatory-map-v2.md:1-80`

Compliance Readiness by Area

1. Licensing (0% ready)

Source: `legal/drop-gap-analysis-v2.md:31-50`

Requirement	Status	Gap
Finanstilsynet license	Not applied	FULL GAP
Client fund safeguarding	Not applicable (demo)	FULL GAP
Initial capital (20K-125K EUR)	Not secured	FULL GAP
Business plan	Exists as draft	PARTIAL
Agent arrangement	None	FULL GAP

Recommended path: Agent model under licensed PSP (1-3 months) while preparing full license application (6-12 months).

2. PSD2 / SCA (10% ready)

Source: `legal/drop-gap-analysis-v2.md:53-78`

Requirement	Status	Code Reference
Strong Customer Authentication	NOT IMPLEMENTED	No BankID, email+password only
BankID integration	NOT IMPLEMENTED	Mentioned in architecture, not in code
Dynamic linking	NOT IMPLEMENTED	No amount+payee tied to auth
Open Banking AISP/PISP	NOT IMPLEMENTED	Balance is local, not from bank
Framework agreement	PARTIAL	Landing page has `vilkar.html`
Fee transparency pre-auth	PARTIAL	Fee shown in API after submission
Session management	IMPLEMENTED	`lib/auth.ts`, `lib/middleware.ts`

3. AML/KYC (5% ready)

Requirement	Status	Gap
Customer identification	Mock only	Auto-approve KYC
Transaction monitoring	NOT IMPLEMENTED	No monitoring system

Requirement	Status	Gap
Suspicious activity reporting	NOT IMPLEMENTED	No SAR capability
Risk assessment	Document exists	`legal/risikovurdering-hvitvasking.md`
AML procedures	Document exists	`legal/hvitvaskingsrutiner.md`

4. GDPR (15% ready)

Requirement	Status	Document
Privacy notice	EXISTS (draft)	`legal/personvernerklaering.md`
DPIA	EXISTS (draft)	`legal/dpia-vurdering.md`
Terms of service	EXISTS (draft)	`legal/brukervilkar.md`
Processing register	NOT CREATED	--
DPO appointment	NOT DONE	--
Data retention policy	NOT DEFINED	--
Consent management	NOT IMPLEMENTED	--

5. ICT Security (25% ready)

Source: `security/security-rapport-2026-02-12.md:187-188`

Requirement	Status	Document/Code
Security policy	EXISTS (draft)	`legal/ikt-sikkerhetspolicy.md`
Incident handling	EXISTS (draft)	`legal/hendelseshaandtering.md`
Business continuity	EXISTS (draft)	`legal/beredskapsplan.md`
Outsourcing policy	EXISTS (draft)	`legal/utkontraktering-policy.md`
Security audit	COMPLETED	`security/drop-security-rapport.md`
Penetration testing	NOT DONE	--
Security hardening	IN PROGRESS	`security/security-hardening-implementation.md`

Security Audit Summary

Date: 2026-02-12 **Source:** `security/drop-security-rapport.md`

Before Hardening

- 4 CRITICAL, 5 HIGH, 6 MEDIUM, 4 LOW findings

After Hardening (2026-02-13)

Source: `security/security-hardening-implementation.md`

- 0 CRITICAL (all resolved)
- 0 HIGH (all resolved)
- 2 MEDIUM remaining (CSP tightening, proxy config)
- 4 LOW (acknowledged, out of scope)

Key Remediations Completed

1. **C1** -- Card data: Schema now stores only `last_four` and `token_ref` (no PAN/CVV)
2. **C2** -- Demo credentials: Gated behind `NODE_ENV !== 'production'`
3. **C4** -- SHA-256 passwords: Removed entirely, bcrypt only
4. **C6/H1** -- Session revocation: Implemented and active
5. **H4** -- Input sanitization: Applied to all text fields
6. **M5** -- Notification IDs: Validated (max 100, format check)
7. **M6** -- Settings: Currency/language validated against whitelists

---

Legal Documents Inventory

Location: `~/ALAI/products/Drop/legal/`

Document	File	Status
Privacy notice	`personvernerklaering.md`	Draft
DPIA assessment	`dpia-vurdering.md`	Draft
Terms of service	`brukervilkar.md`	Draft
AML procedures	`hvitvaskingsrutiner.md`	Draft
AML risk assessment	`risikovurdering-hvitvasking.md`	Draft
ICT security policy	`ikt-sikkerhetspolicy.md`	Draft
Incident handling	`hendelseshaandtering.md`	Draft
Business continuity	`beredskapsplan.md`	Draft
Outsourcing policy	`utkontraktering-policy.md`	Draft

Document	File	Status
Internal control	`internkontroll.md`	Draft
Suitability assessment	`egnethetsvurdering.md`	Draft
Complaint handling	`klagebehandling.md`	Draft
Licensing preparation	`konsesjonssoknad-forberedelse.md`	Draft
Business plan	`virksomhetsplan.md`	Draft
Gap analysis v2	`drop-gap-analysis-v2.md`	Complete
Regulatory map v2	`drop-regulatory-map-v2.md`	Complete

Security Documents Inventory

Location: `~/ALAI/products/Drop/security/`

Document	File	Status
Security audit rapport	`drop-security-rapport.md`	Complete (2026-02-12)
Gap analysis	`gap-analysis.md`	Complete (2026-02-12)
Hardening checklist	`hardening-checklist.md`	In progress
Hardening implementation	`security-hardening-implementation.md`	Complete (2026-02-13)
Formal assessment	`security-rapport-2026-02-12.md`	Complete (2026-02-12)

Remediation Phases

Phase 1 -- Current Sprint (in progress)

Security fixes, architecture cleanup, test suite, CI/CD.

Phase 2 -- Banking Integration (pending partner selection)

BankID, Open Banking AISP/PISP, real KYC, PostgreSQL migration.

Phase 3 -- Production Launch (after Phase 2 + follow-up audit)

Audit logging, error handling, monitoring, staging environment, load testing, external penetration test.

Source: `security/security-rapport-2026-02-12.md:196-220`

Security Architecture

Drop Security Architecture

Last updated: 2026-02-14 **Source:** Security audit (`security/drop-security-rapport.md`), hardening implementation (`security/security-hardening-implementation.md`), source code

> **Architecture model:** Drop uses a PSD2 pass-through model. It never holds customer money — AISP reads bank balances via Open Banking, PISP initiates payments from the user's bank account. Cards are a FUTURE feature, gated behind feature flags (all default to false).

Authentication

JWT Token Management

Library: `jose` ^6.1.3 (well-maintained, no known vulnerabilities) **Algorithm:** HS256 with explicit `setProtectedHeader` **Source:** `src/drop-app/src/lib/auth.ts`

Setting	Value	Source
Algorithm	HS256	`auth.ts`
Expiry	24 hours	`auth.ts` cookie `maxAge: 60 * 60 * 24`
`setIssuedAt()`	Yes	Prevents token reuse
Secret (production)	`JWT_SECRET` env var	Fatal error if missing
Secret (development)	`process.cwd()` hash	Dev fallback for uniqueness

Cookie Configuration

Source: `src/drop-app/src/lib/auth.ts:48-54`

Property	Value	Purpose

`httpOnly`	`true`	Prevents JavaScript access (XSS mitigation)
`secure`	`true` (production)	HTTPS-only cookie transport
`sameSite`	`"strict"`	CSRF prevention
`maxAge`	24 hours	Session lifetime
`path`	`"/"`	Cookie scope

Password Hashing

Library: `bcryptjs` ^3.0.3 (pure JS, no native compilation issues) **Source:** `src/drop-app/src/lib/utils-server.ts:8-16`

Setting	Value
Algorithm	bcrypt
Cost factor	12
SHA-256 fallback	**Removed** (security fix C4)

After hardening, `verifyPassword()` only accepts bcrypt hashes. SHA-256 legacy support has been removed entirely.

Session Management

Source: `src/drop-app/src/lib/auth.ts:45-65`, `src/lib/middleware.ts:42-77`

Sessions are tracked in the `sessions` table:

Column	Type	Purpose
`id`	TEXT PK	Session identifier (format: `ses_<hex16>`)
`user_id`	TEXT FK	References `users.id`
`token_hash`	TEXT	SHA-256 hash of JWT token
`expires_at`	TEXT	Expiration timestamp
`revoked`	INTEGER	0 = active, 1 = revoked

Lifecycle:

1. **Login** -- Session created with token hash (`auth.ts:56-65`)
2. **Each request** -- Session checked for revocation (`middleware.ts:66-74`)
3. **Logout** -- All user sessions revoked server-side (`auth/logout/route.ts:5-14`)

Authorization

IDOR Protection

All data access queries include `AND user_id = ?` to scope data to the authenticated user:

- Recipients: scoped to user
- Cards: scoped to user
- Transactions: scoped to user
- Bank accounts: scoped to user
- Notifications: scoped to user
- Settings: scoped to user

Merchant endpoints verify merchant role and ownership.

Role-Based Access

Roles (from `lib/db.ts` schema `CHECK` constraint):

- `user` -- Standard user
- `merchant` -- Merchant with dashboard access

KYC Status (required for financial operations):

- `pending` -- Default on registration
- `approved` -- Required for remittance
- `rejected` -- Blocked from financial operations

Input Validation

Rate Limiting

Source: `src/drop-app/src/lib/middleware.ts:6-31`

Endpoint Type	Limit	Window
Auth routes (login, register)	10 requests	60 seconds
Transaction routes (remittance, qr-payment)	10 requests	60 seconds

Endpoint Type	Limit	Window
Rate endpoints (/api/rates)	120 requests	60 seconds

Implementation: PostgreSQL-backed persistent rate limiting (survives restarts). Per-IP tracking using `X-Forwarded-For` header. (ADR-014: SQLite removed 2026-03-03)

Rate limit table (`rate_limits`):

- `key` -- IP address (TEXT PK)
- `count` -- Request count (INTEGER)
- `reset_at` -- Window expiry (INTEGER, Unix timestamp)

CSRF Protection

Source: `src/drop-app/src/lib/middleware.ts:44-55`

Origin header validation on all authenticated requests:

- Validates `Origin` header against allowed origins list
- Allowed origins: `NEXT_PUBLIC_APP_URL`, `http://localhost:3000`, `http://localhost:3001`
- Combined with `sameSite: "strict"` cookies

Input Sanitization

Source: `src/drop-app/src/lib/middleware/validation.ts:149-203`

- `sanitizeText()` -- Removes HTML tags, control characters, trims whitespace, enforces max length
- `validateName()` -- Rejects XSS payloads, script tags, numbers-only names
- `validateEmail()` -- Regex validation for email format
- `validatePhone()` -- International format validation
- `validateAmount()` -- Positive, finite, max 2 decimal places
- `validateIBAN()` -- Format and checksum validation
- `validatePIN()` -- Exactly 4 digits
- `validateCurrency()` -- Whitelist: EUR, USD, GBP, BAM, CHF, PLN, NOK, RSD, TRY, PKR
- `validateLanguage()` -- Whitelist: nb, en, bs, sq

Applied to: recipients, merchants, settings, notifications.

Amount Validation

Endpoint	Min	Max	Source

Remittance	100 NOK	50,000 NOK	`transactions/remittance/route.ts`
QR Payment	1 NOK	100,000 NOK	`transactions/qr-payment/route.ts`

Additional checks: `Number.isFinite()` to prevent NaN/Infinity injection. Pagination limited to max 50 per page.

SQL Injection Prevention

All 24 API endpoints use **parameterized queries** exclusively (`?` placeholders). No string concatenation in SQL statements.

Example from `transactions/route.ts`:

```
// Dynamic WHERE clauses use parameter arrays
const conditions: string[] = [];
const params: unknown[] = [];
if (type) { conditions.push("type = ?"); params.push(type); }
```

Merchant dashboard uses strict whitelist for `period` parameter.

Security Headers

Source: `src/drop-app/next.config.ts:6-46`

Header	Value
Content-Security-Policy	`default-src 'self'; script-src 'self' 'unsafe-inline' 'unsafe-eval'; ...`
X-Frame-Options	`DENY`
X-Content-Type-Options	`nosniff`
Referrer-Policy	`strict-origin-when-cross-origin`
Permissions-Policy	`camera=(self), microphone=(), geolocation=(self)`
Strict-Transport-Security	`max-age=63072000; includeSubDomains; preload`

Known limitation: CSP includes `unsafe-inline` and `unsafe-eval` (required for Next.js dev mode). Should be tightened with nonce-based CSP for production.

API Response Masking

- **Card numbers:** Masked as `**** **** **** XXXX` in responses (`cards/[id]/route.ts:25-35`)
- **CVV:** Displayed as `***` in responses
- **Bank accounts:** Only last 4 digits visible (`utils-server.ts:23-26`)

Transaction Integrity

Source: `transactions/remittance/route.ts`, `transactions/qr-payment/route.ts`

- Atomic balance deduction using PostgreSQL transactions (Drizzle ORM)
- `WHERE balance >= $1` prevents overdraw
- PostgreSQL MVCC with explicit `FOR UPDATE` row locking
- Fee calculated and included in deduction

Feature Flags

Source: `src/drop-app/src/lib/feature-flags.ts`

Gated features (disabled by default):

Flag	Env Var	Default
`virtualCards`	`NEXT_PUBLIC_FF_VIRTUAL_CARDS`	`false`
`physicalCards`	`NEXT_PUBLIC_FF_PHYSICAL_CARDS`	`false`
`cardDetails`	`NEXT_PUBLIC_FF_CARD_DETAILS`	`false`
`cardFreeze`	`NEXT_PUBLIC_FF_CARD_FREEZE`	`false`
`cardPin`	`NEXT_PUBLIC_FF_CARD_PIN`	`false`
`spendingLimits`	`NEXT_PUBLIC_FF_SPENDING_LIMITS`	`false`
`notifications`	`NEXT_PUBLIC_FF_NOTIFICATIONS`	`true`
`merchantDashboard`	`NEXT_PUBLIC_FF_MERCHANT_DASHBOARD`	`true`

API routes check feature flags and return 404 when disabled.

Dependency Security

Package	Version	Risk Assessment
`jose`	^6.1.3	Low -- Well-maintained JWT library
`bcryptjs`	^3.0.3	Low -- Pure JS bcrypt
`drizzle-orm`	latest	Low -- Type-safe ORM, parameterized queries
`next`	16.1.6	Low -- Recent version
`react`	19.2.3	Low -- Latest major
`radix-ui`	^1.4.3	Low -- UI components only

No known vulnerable dependencies identified (from `security/drop-security-rapport.md:400-411`).

Secret Rotation Runbook — 2026-04-20

Secret Rotation Runbook — 2026-04-20

Incident Summary

Date: 2026-04-20
Scope: 14 leaked credentials (git history exposure in two repositories)
Root Cause: Secrets echoed in CI logs, git-tracked config files, missing pre-commit hooks
Impact: No confirmed breach; rotation completed within 4 hours

Leaked Credentials

1. AWS IAM access keys (2x)
2. Azure Storage Account connection strings (3x)
3. BookStack API token
4. Slack webhook URLs (2x)
5. Cloudflare Access Client credentials (2x)
6. Bitwarden CLI session tokens (4x — expired)

Rotation Order (Critical ? Supporting)

1. **Bitwarden CLI unlock** — immediate session key rotation
2. **AWS IAM keys** — create new key, update config, delete old key
3. **Azure Storage Account keys** — regenerate key2 first (non-breaking), migrate scripts, regenerate key1
4. **Slack webhooks** — regenerate via Slack app settings
5. **Cloudflare Access** — rotate service tokens via Zero Trust dashboard
6. **BookStack API** — create new token, update ~/system/config/.bookstack-cred-cache.json, revoke old

Rotation Commands

```
# AWS IAM
aws iam create-access-key --user-name alai-admin --output json > /tmp/new-key.json
# Extract AccessKeyId and SecretAccessKey, update ~/.aws/credentials
aws iam delete-access-key --access-key-id OLD_KEY_ID --user-name alai-admin

# Azure Storage Account
az storage account keys renew --account-name alaibackups0ebb --key secondary
# Update ~/system/config/azure-backup.env with new key2 connection string
az storage account keys renew --account-name alaibackups0ebb --key primary

# Verification (test old credential returns 401/403)
curl -I https://alaibackups0ebb.blob.core.windows.net/system-db-backups \
  -H "x-ms-version: 2021-08-06" \
  -H "Authorization: Bearer OLD_TOKEN"
```

Lessons Learned

- **NEVER echo AWS IAM credentials:** `aws iam create-access-key` output must pipe to `jq` → Bitwarden immediately
- **Gitignore enforcement:** All files matching `*-cred-*`, `*.env`, `*-secret.*` → global gitignore + pre-commit hook
- **Hourly backup cron:** Add `gitleaks` scan before git bundle creation
- **Bitwarden item naming convention:** `ALAI — <service> — <context>` (Login type) for consistency

New Bitwarden Naming Convention

Format: `ALAI — <Service Name> — <Context>`
Type: Login (not Secure Note)
Examples:

- `ALAI — AWS IAM — alai-admin`
- `ALAI — Azure Storage — alaibackups0ebb`
- `ALAI — BookStack API — docs.basicconsulting.no`

Verification Protocol

```
# Test old credential fails (401/403)
curl -I <endpoint> -H "Authorization: Bearer OLD_TOKEN"

# Test new credential succeeds (200/204)
curl -I <endpoint> -H "Authorization: Bearer NEW_TOKEN"

# Update all config files
grep -r "OLD_TOKEN" ~/system/config/ ~/system/tools/
sed -i '' 's/OLD_TOKEN/NEW_TOKEN/g' ~/system/config/*.json
```

Next Incident Actions

1. Run `gitleaks detect --source ~/system/ --verbose` immediately
2. Prioritize rotation: IAM → Storage → API → Webhooks
3. Update Bitwarden with new credentials during rotation (not after)
4. Verify old credential invalidation with `curl 401` test
5. Document in BookStack within 24h