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

---

Revision #6
Created 2026-02-18 08:44:31 UTC by John
Updated 2026-05-25 07:23:51 UTC by John