Security Architecture

Bilko — Security Architecture

Status: PLANNED (backend not built yet, security measures documented for implementation)

This document defines the security architecture for Bilko, a financial SaaS handling sensitive accounting data.

Security Principles

1. **Defense in Depth** — Multiple layers of security (network, application, database)
2. **Least Privilege** — Users and services get minimum necessary permissions
3. **Zero Trust** — Verify every request, never assume trust
4. **Encryption Everywhere** — Data encrypted in transit and at rest
5. **Immutable Audit Trail** — All actions logged, tamper-proof

Security Layers Diagram

```
graph TD
    CLIENT["Client Browser / PWA"]

    subgraph NETWORK["Network Layer"]
        CF["Cloudflare\nDDoS Protection\nTLS 1.3 termination\nHSTS"]
    end

    subgraph APP_LAYER["Application Layer"]
        HELMET["Helmet.js\nCSP + X-Frame + HSTS\nno X-Powered-By"]
        CORS["CORS Whitelist\nbilko.io only\nno wildcard *"]
        RATE["Rate Limiter\nexpress-rate-limit\n5 req/min auth\n100 req/min general"]
        AUTH_MW["Auth Middleware\nJWT verify\norg-scope injection"]
        RBAC_MW["RBAC Middleware\nrole check\nowner / admin / accountant / viewer"]
        ZOD["Zod Validation\nall request bodies\ntype-safe parsing"]
    end
```

```
    subgraph DATA_LAYER["Data Layer"]
        PRISMA_ORM["Prisma ORM\nparameterized queries\nno raw SQL for user input\norg-scoped
WHERE"]
        PG_ENC["PostgreSQL Railway\nAES-256 disk encryption\nbackup encryption"]
    end

    subgraph AUDIT["Audit Layer"]
        LOG["LoggedAction table\nAPPEND-ONLY\nIP + user + timestamp\nold/new values"]
    end

    CLIENT --> CF --> HELMET --> CORS --> RATE --> AUTH_MW --> RBAC_MW --> ZOD --> PRISMA_ORM
--> PG_ENC
    PRISMA_ORM --> LOG
```

Authentication

Strategy: JWT (JSON Web Tokens)

Why JWT?

- Stateless (scales horizontally)
- Works with mobile PWA
- Industry standard

Token Types

Access Token

- **Lifetime:** 15 minutes
- **Storage:** `Authorization: Bearer <token>` header
- **Contains:** User ID, organization ID, role
- **Refresh:** Automatic via refresh token

Refresh Token

- **Lifetime:** 7 days
- **Storage:** httpOnly cookie (not accessible to JavaScript)
- **Purpose:** Obtain new access token
- **Rotation:** New refresh token issued on each refresh

- **Revocation:** Stored in database, can be invalidated

JWT Payload Example

```
{
  "sub": "user-uuid",
  "org": "org-uuid",
  "role": "admin",
  "iat": 1640000000,
  "exp": 1640000900
}
```

Token Flow

```
1. User logs in → POST /api/v1/auth/login
   ← Access token (header) + Refresh token (httpOnly cookie)

2. User makes request → GET /api/v1/invoices (Authorization: Bearer <access>)
   ← Protected resource

3. Access token expires (15 min) → POST /api/v1/auth/refresh (httpOnly cookie)
   ← New access token + New refresh token

4. User logs out → POST /api/v1/auth/logout
   → Delete refresh token from DB
   ← 204 No Content
```

JWT Auth Flow (Sequence)

```
sequenceDiagram
    actor User
    participant FE as Frontend (bilko.io)
    participant API as Express API (api.bilko.io)
    participant DB as PostgreSQL

    User->>FE: Enter email + password
    FE->>API: POST /api/v1/auth/login
    API->>DB: SELECT user WHERE email = ?
```

```
    DB-->>API: User record (passwordHash)
    API->>API: bcrypt.compare(password, hash)
    alt Password valid
        API->>API: jwt.sign({sub, org, role}, JWT_SECRET, 15m)
        API->>API: jwt.sign({sub}, JWT_REFRESH_SECRET, 7d)
        API->>DB: INSERT refreshToken (hashed, expiresAt)
        API-->>FE: 200 { accessToken } + Set-Cookie: refreshToken (httpOnly)
        FE->>FE: Store accessToken in memory
    else Password invalid
        API-->>FE: 401 Unauthorized
    end

    Note over FE,API: 15 minutes later — access token expires
    FE->>API: POST /api/v1/auth/refresh (Cookie: refreshToken)
    API->>DB: SELECT refreshToken WHERE token = ? AND expiresAt > NOW()
    DB-->>API: Valid token record
    API->>API: Rotate: delete old, issue new refresh token
    API->>DB: DELETE old refreshToken, INSERT new refreshToken
    API-->>FE: 200 { newAccessToken } + Set-Cookie: newRefreshToken

    Note over User,DB: User logs out
    User->>FE: Click logout
    FE->>API: POST /api/v1/auth/logout
    API->>DB: DELETE refreshToken WHERE userId = ?
    API-->>FE: 204 No Content
    FE->>FE: Clear accessToken from memory
```

Implementation (Backend)

```
import jwt from 'jsonwebtoken';
import bcrypt from 'bcrypt';

// Generate access token
const accessToken = jwt.sign(
  { sub: user.id, org: user.organizationId, role: user.role },
  process.env.JWT_SECRET!,
  { expiresIn: '15m' }
);
```

```
// Generate refresh token
const refreshToken = jwt.sign(
  { sub: user.id },
  process.env.JWT_REFRESH_SECRET!,
  { expiresIn: '7d' }
);

// Store refresh token in DB (for revocation)
await prisma.refreshToken.create({
  data: {
    token: refreshToken,
    userId: user.id,
    expiresAt: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000),
  },
});
```

Password Security

Hashing: bcrypt

Algorithm: bcrypt with 12 salt rounds

Why bcrypt?

- Designed for passwords (slow by design, resists brute force)
- Auto-salted (each password has unique salt)
- Adaptive (can increase rounds as hardware improves)

Password Requirements

- **Minimum length:** 8 characters
- **Complexity:** At least one uppercase, one lowercase, one number
- **No common passwords:** Check against list of 10K most common passwords
- **No reuse:** Previous 5 passwords stored (hashed) and blocked

Implementation

```
import bcrypt from 'bcrypt';

// Hash password (registration)
const passwordHash = await bcrypt.hash(password, 12);

// Verify password (login)
const isValid = await bcrypt.compare(password, user.passwordHash);
```

Two-Factor Authentication (2FA)

Strategy: TOTP (Time-based One-Time Password)

Compatible with:

- Google Authenticator
- Authy
- 1Password
- Microsoft Authenticator

Setup Flow

```
1. User enables 2FA → POST /api/v1/auth/2fa/setup
   ← QR code + secret (base32)

2. User scans QR code in authenticator app
   → Generates 6-digit code

3. User verifies code → POST /api/v1/auth/2fa/verify { code }
   ← 200 OK (2FA enabled)
```

Login Flow with 2FA

```
1. User logs in → POST /api/v1/auth/login { email, password }
   ← 200 OK + { requires2FA: true, tempToken }
```

```
2. User enters code → POST /api/v1/auth/2fa/login { tempToken, code }
   ← Access token + Refresh token
```

Backup Codes

Generate 10 single-use backup codes during 2FA setup:

- Stored hashed (bcrypt)
- Used when authenticator unavailable
- Marked as used after redemption

Authorization (RBAC)

RBAC Permission Model

```
classDiagram
    class Owner {
        +createInvoice()
        +editInvoice()
        +deleteInvoice()
        +viewInvoice()
        +approveExpense()
        +generateReport()
        +inviteUser()
        +editOrgSettings()
        +deleteOrg()
    }
    class Admin {
        +createInvoice()
        +editInvoice()
        +viewInvoice()
        +approveExpense()
        +generateReport()
        -deleteInvoice()
        -inviteUser()
        -editOrgSettings()
    }
```

```
    class Accountant {
        +viewInvoice()
        +viewExpense()
        +generateReport()
        -createInvoice()
        -editInvoice()
        -approveExpense()
        -inviteUser()
    }
    class Viewer {
        +viewDashboard()
        +viewReports()
        -createInvoice()
        -editInvoice()
        -generateReport()
        -inviteUser()
    }

    Owner --|> Admin : inherits all Admin permissions
    Admin --|> Accountant : inherits read access
    Accountant --|> Viewer : inherits view access
```

Roles

Role	Permissions
owner	Full access (edit org settings, invite users, delete data)
admin	Manage invoices, expenses, contacts, reports (no org settings)
accountant	Read invoices/expenses, create reports (no edit)
viewer	Read-only access (dashboard, reports)

Permission Matrix

Action	owner	admin	accountant	viewer
Create invoice	□	□	□	□
Edit invoice	□	□	□	□
Delete invoice	□	□	□	□

Action	owner	admin	accountant	viewer
View invoice	✅	✅	✅	✅
Approve expense	✅	✅	✅	❌
Generate report	✅	✅	✅	✅
Invite user	✅	✅	❌	❌
Edit org settings	✅	❌	❌	❌

Implementation (Middleware)

```
import { Request, Response, NextFunction } from 'express';

function requireRole(roles: string[]) {
  return (req: Request, res: Response, next: NextFunction) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ error: 'Forbidden' });
    }
    next();
  };
}

// Usage
app.post('/api/v1/invoices', requireRole(['owner', 'admin']), createInvoice);
```

Encryption

In Transit: TLS 1.3

All traffic encrypted via HTTPS:

- Frontend (Vercel): Automatic HTTPS
- Backend (Railway): Automatic HTTPS
- Certificate: Let's Encrypt (auto-renewed)

TLS Configuration:

- Minimum version: TLS 1.3
- Cipher suites: Modern only (no legacy ciphers)

- HSTS enabled (Strict-Transport-Security header)

At Rest: Database Encryption

PostgreSQL (Railway):

- Disk encryption: AES-256 (Railway default)
- Backup encryption: AES-256
- Column-level encryption: Not needed (disk encryption sufficient for accounting data)

Cloudflare R2 (Files):

- Server-side encryption: AES-256 (default)
- No client-side encryption needed (files are receipts/invoices, not PII)

Secrets Management

NEVER commit secrets to git:

- `.env` files in `.gitignore`
- Use platform-provided secrets (Vercel, Railway)
- Rotate JWT secrets quarterly
- Rotate API keys annually

OWASP Top 10 Mitigations

1. Injection (SQL Injection)

Mitigation: Prisma ORM parameterized queries

```
// SAFE — Prisma auto-escapes
await prisma.invoice.findMany({
  where: { customerId: req.params.id }
});

// UNSAFE — Never use raw SQL for user input
await prisma.$queryRaw`SELECT * FROM invoices WHERE customer_id = ${req.params.id}`;
```

---

2. Broken Authentication

Mitigations:

- bcrypt password hashing (12 rounds)
- JWT with short expiry (15 min)
- Refresh token rotation
- 2FA (TOTP)
- Rate limiting on auth endpoints (5 req/min)

---

3. Sensitive Data Exposure

Mitigations:

- TLS 1.3 in transit
- AES-256 at rest
- No PII in JWTs (only user ID)
- No passwords in logs
- No sensitive data in URLs (use POST body)

---

4. XML External Entities (XXE)

Not applicable — Bilko does not parse XML.

---

5. Broken Access Control

Mitigations:

- RBAC enforced on every endpoint
- Organization-scoped queries (middleware)
- No direct object reference (use UUIDs, not auto-increment IDs)

```
// Organization scoping middleware
app.use('/api/v1/*', (req, res, next) => {
  req.prismaWhere = { organizationId: req.user.organizationId };
  next();
});

// Apply to queries
```

```
await prisma.invoice.findMany({ where: req.prismaWhere });
```

---

6. Security Misconfiguration

Mitigations:

- Helmet.js security headers
- CORS whitelist (no `*` in production)
- Error messages sanitized (no stack traces in production)
- Disable `X-Powered-By` header

```
import helmet from 'helmet';

app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "'unsafe-inline'"], // Next.js requires unsafe-inline
      styleSrc: ["'self'", "'unsafe-inline'"],
      imgSrc: ["'self'", "data:", "https:"],
    },
  },
}));
```

---

7. Cross-Site Scripting (XSS)

Mitigations:

- React auto-escapes output (default safe)
- CSP headers (Content-Security-Policy)
- Sanitize user input (Zod validation)
- No `dangerouslySetInnerHTML` without sanitization

```
// SAFE — React escapes by default
<p>{invoice.description}</p>

// UNSAFE — Only use with sanitized HTML
<div dangerouslySetInnerHTML={{ __html: sanitizedHTML }} />
```

---

8. Insecure Deserialization

Not applicable — Bilko does not deserialize untrusted data.

---

9. Using Components with Known Vulnerabilities

Mitigations:

- Dependabot alerts enabled (GitHub)
- Weekly `npm audit` checks
- Automated security updates (Dependabot PRs)
- Lock file committed (`package-lock.json`)

---

10. Insufficient Logging & Monitoring

Mitigations:

- Audit trail (LoggedAction table)
- Error tracking (Sentry recommended)
- Access logs (Railway built-in)
- Failed login attempts logged
- Anomaly detection (future: alert on 10+ failed logins)

Rate Limiting

Rate Limiting Flow

```
flowchart TD
    REQ["Incoming Request"]
    ENDPOINT{{"Endpoint?"}}

    AUTH_CHECK{{"IP count\n≤5 per 15min?"}}
    REG_CHECK{{"IP count\n≤3 per 60min?"}}
    REFRESH_CHECK{{"IP count\n≤10 per 15min?"}}
    REPORT_CHECK{{"User count\n≤10 per 15min?"}}
    GENERAL_CHECK{{"IP count\n≤100 per 15min?"}}
```

```
    PASS["Pass to route handler"]
    BLOCK["429 Too Many Requests\n'Try again later'"]

    REQ --> ENDPOINT
    ENDPOINT -->|"/auth/login"| AUTH_CHECK
    ENDPOINT -->|"/auth/register"| REG_CHECK
    ENDPOINT -->|"/auth/refresh"| REFRESH_CHECK
    ENDPOINT -->|"/reports/*"| REPORT_CHECK
    ENDPOINT -->|"all other /api/*"| GENERAL_CHECK

    AUTH_CHECK -->|Yes| PASS
    AUTH_CHECK -->|No| BLOCK
    REG_CHECK -->|Yes| PASS
    REG_CHECK -->|No| BLOCK
    REFRESH_CHECK -->|Yes| PASS
    REFRESH_CHECK -->|No| BLOCK
    REPORT_CHECK -->|Yes| PASS
    REPORT_CHECK -->|No| BLOCK
    GENERAL_CHECK -->|Yes| PASS
    GENERAL_CHECK -->|No| BLOCK
```

Prevent brute force and abuse:

Endpoint	Limit	Window
`/api/v1/auth/login`	5 requests	15 minutes
`/api/v1/auth/register`	3 requests	60 minutes
`/api/v1/auth/refresh`	10 requests	15 minutes
`/api/v1/*` (general)	100 requests	15 minutes
`/api/v1/reports/*`	10 requests	15 minutes

Implementation

```
import rateLimit from 'express-rate-limit';

const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5,
  message: 'Too many login attempts, try again later',
```

```
});

app.post('/api/v1/auth/login', authLimiter, loginHandler);
```

Input Validation

All inputs validated with **Zod** schemas:

Example: Invoice Validation

```
import { z } from 'zod';

const createInvoiceSchema = z.object({
  customerId: z.string().uuid(),
  invoiceDate: z.string().regex(/^\d{4}-\d{2}-\d{2}$/),
  dueDate: z.string().regex(/^\d{4}-\d{2}-\d{2}$/),
  currencyCode: z.enum(['EUR', 'RSD', 'BAM', 'HRK']),
  items: z.array(z.object({
    description: z.string().min(1).max(500),
    quantity: z.number().positive(),
    unitPrice: z.number().nonnegative(),
    taxRate: z.number().min(0).max(100),
  })),
});

// Middleware
function validate(schema: z.ZodSchema) {
  return (req, res, next) => {
    try {
      req.body = schema.parse(req.body);
      next();
    } catch (error) {
      res.status(400).json({ error: error.errors });
    }
  };
}

// Usage
```

```
app.post('/api/v1/invoices', validate(createInvoiceSchema), createInvoice);
```

File Upload Security

Allowed File Types

- **Receipts:** JPG, PNG, PDF
- **Max size:** 10MB per file

Validation

```
import multer from 'multer';
import path from 'path';

const upload = multer({
  limits: { fileSize: 10 * 1024 * 1024 }, // 10MB
  fileFilter: (req, file, cb) => {
    const allowedTypes = ['.jpg', '.jpeg', '.png', '.pdf'];
    const ext = path.extname(file.originalname).toLowerCase();
    if (allowedTypes.includes(ext)) {
      cb(null, true);
    } else {
      cb(new Error('Invalid file type'));
    }
  },
});
```

Virus Scanning (Planned)

Phase 2: Integrate ClamAV for virus scanning before upload to R2.

Audit Trail

LoggedAction Table (Immutable)

All mutations logged:

- Table name
- Action (INSERT, UPDATE, DELETE)
- User ID
- Timestamp
- Old values (UPDATE/DELETE)
- New values (INSERT/UPDATE)
- Client IP
- SQL query

Example Audit Log Entry

```
{
  "eventId": 12345,
  "tableName": "invoices",
  "action": "UPDATE",
  "userId": "user-uuid",
  "actionTimestamp": "2026-02-20T10:30:00Z",
  "rowData": { "id": "invoice-uuid", "status": "draft" },
  "changedFields": { "status": { "old": "draft", "new": "sent" } },
  "clientIp": "192.168.1.10"
}
```

Audit Queries

```
// Get user activity
await prisma.loggedAction.findMany({
  where: { userId: 'user-uuid' },
  orderBy: { actionTimestamp: 'desc' },
  take: 100,
});

// Get invoice history
await prisma.loggedAction.findMany({
  where: {
    tableName: 'invoices',
    rowData: { path: ['id'], equals: 'invoice-uuid' },
  },
});
```

Data Retention & Deletion

User Data Deletion (GDPR Right to Erasure)

Process:

1. User requests deletion → POST /api/v1/account/delete
2. Soft delete user record (mark `deletedAt`)
3. Anonymize LoggedAction entries (replace user ID with "deleted-user")
4. Delete PII (email, name)
5. **Keep financial records** (required by law, minimum 5 years)

Soft Delete Implementation:

```
await prisma.user.update({
  where: { id: userId },
  data: {
    email: `deleted-${userId}@example.com`,
    fullName: 'Deleted User',
    passwordHash: '',
    deletedAt: new Date(),
  },
});
```

Security Testing

Static Analysis

- **ESLint:** Security rules enabled (no-eval, no-unsafe-regex)
- **TypeScript:** Strict mode (catches type errors)

Dependency Scanning

- **npm audit:** Weekly checks
- **Dependabot:** Automatic PRs for vulnerabilities

Penetration Testing (Future)

- **Phase 2:** Hire security firm for penetration testing
- **Scope:** Auth, API, file uploads, SQL injection
- **Frequency:** Annually

Incident Response Plan

Detection

- Monitor error rates (Sentry)
- Monitor failed login attempts (>10 in 1 hour = alert)
- Railway metrics (CPU spike, memory leak)

Response

1. **Identify:** What is the breach? (data leak, DDoS, unauthorized access)
2. **Contain:** Block attacker IP, revoke compromised tokens
3. **Eradicate:** Fix vulnerability, patch code
4. **Recover:** Restore from backup if needed
5. **Document:** Write post-mortem, update security docs

Notification

- **Internal:** Slack alert to #security channel
- **External:** Email users if PII compromised (GDPR 72h requirement)

Security Checklist (Pre-Launch)

- ☐ JWT secrets generated (32+ chars)
- ☐ HTTPS enforced (no HTTP allowed)
- ☐ CORS whitelist configured (no `*`)
- ☐ Rate limiting enabled (auth endpoints)
- ☐ Helmet.js security headers configured
- ☐ bcrypt password hashing (12 rounds)

- [ ] Prisma queries parameterized (no raw SQL)
- [ ] Input validation (Zod schemas)
- [ ] File upload restrictions (type, size)
- [ ] Audit trail enabled (LoggedAction)
- [ ] Error messages sanitized (no stack traces)
- [ ] Dependabot alerts enabled
- [ ] Backup strategy tested
- [ ] Incident response plan documented
- [ ] Security review completed

Related Documents

- Compliance: COMPLIANCE.md
- Deployment: ../infrastructure/DEPLOYMENT.md
- Testing: ../testing/TESTING-GUIDE.md

Last Updated: 2026-02-20 **Status:** PLANNED — Backend not built yet, security measures to be implemented **Compliance:** OWASP Top 10, GDPR Article 32 (Security of Processing)

Revision #4
Created 2026-02-23 10:48:11 UTC by John
Updated 2026-05-31 20:02:48 UTC by John