

# Security Fixes Quick Reference

---

This document provides code snippets for immediate implementation of critical security fixes identified in the security audit.

## **CRITICAL: SQL Injection Fix**

---

**File:** `lib/database-query-executor.ts`

### **Add SQL Validation Function**

Add this function before the `executeQuery` function:

```

/**
 * Validates SQL query to prevent injection attacks
 * Only allows SELECT statements and blocks dangerous operations
 */
function validateSQL(sql: string): { valid: boolean; error?: string } {
  if (!sql || sql.trim().length === 0) {
    return { valid: false, error: 'Empty SQL query' };
  }

  // Remove comments and normalize whitespace
  const normalized = sql
    .replace(/--.*$/gm, '') // Remove line comments
    .replace(/\/\*[\s\S]*?\*\/g, '') // Remove block comments
    .replace(/\s+/g, ' ')
    .trim()
    .toUpperCase();

  // Must start with SELECT
  if (!normalized.startsWith('SELECT')) {
    return { valid: false, error: 'Only SELECT queries are allowed' };
  }

  // Dangerous patterns that should never appear
  const dangerousPatterns = [
    { pattern: /\bDROP\s+/i, description: 'DROP statements' },
    { pattern: /\bDELETE\s+FROM\b/i, description: 'DELETE statements' },
    { pattern: /\bUPDATE\s+/i, description: 'UPDATE statements' },
    { pattern: /\bINSERT\s+INTO\b/i, description: 'INSERT statements' },
    { pattern: /\bTRUNCATE\s+/i, description: 'TRUNCATE statements' },
    { pattern: /\bALTER\s+/i, description: 'ALTER statements' },
    { pattern: /\bCREATE\s+/i, description: 'CREATE statements' },
    { pattern: /\bEXEC\s*\(/i, description: 'EXEC function calls' },
    { pattern: /\bEXECUTE\s+/i, description: 'EXECUTE statements' },
    { pattern: /\bGRANT\s+/i, description: 'GRANT statements' },
    { pattern: /\bREVOKE\s+/i, description: 'REVOKE statements' },
    { pattern: /xp_\w+/i, description: 'Extended stored procedures' },
    { pattern: /sp_\w+/i, description: 'System stored procedures' },
    { pattern: /\s*SELECT/i, description: 'Multiple statements' },
    { pattern: /\bUNION\s+(?!ALL\s+SELECT)/i, description: 'UNION without SELECT' },
    { pattern: /\bINTO\s+OUTFILE/i, description: 'File operations' },
    { pattern: /\bINTO\s+DUMPFILE/i, description: 'File operations' },
    { pattern: /\bLOAD_FILE\s*\(/i, description: 'File reading functions' },
  ];

  for (const { pattern, description } of dangerousPatterns) {
    if (pattern.test(sql)) {
      return {
        valid: false,
        error: `SQL query contains dangerous operation: ${description}`
      };
    }
  }

  // Additional security checks

  // Check for excessive statement terminators (potential SQL injection)
  const semicolonCount = (sql.match(/;/g) || []).length;
  if (semicolonCount > 0) {
    return {
      valid: false,
      error: 'Multiple SQL statements are not allowed'
    };
  }
}

```

```

    }

    // Limit query complexity (prevent DoS via complex queries)
    const subqueryCount = (sql.match(/SELECT/gi) || []).length;
    if (subqueryCount > 5) {
      return {
        valid: false,
        error: 'Query too complex (max 5 subqueries allowed)'
      };
    }

    return { valid: true };
  }
}

```

## Update executeQuery Function

Replace line 56 in `executeQuery` :

```

// BEFORE (VULNERABLE):
result = await prisma.$queryRawUnsafe(sql)

// AFTER (SECURE):
// Validate SQL before execution
const validation = validateSQL(sql);
if (!validation.valid) {
  console.error('SQL Validation Failed:', validation.error);
  console.error('Rejected SQL:', sql);
  throw new Error(`Query rejected: ${validation.error}`);
}

// Execute with validation passed
result = await prisma.$queryRawUnsafe(sql)

```



## HIGH: Remove Weak Encryption Fallback

**File:** `lib/encryption.ts`

## Replace lines 9-10:

```
// BEFORE (INSECURE):  
const ENCRYPTION_KEY = process.env.ENCRYPTION_KEY || 'picard-ai-default-encryption-  
key-change-in-production'  
  
// AFTER (SECURE):  
const ENCRYPTION_KEY = process.env.ENCRYPTION_KEY;  
  
if (!ENCRYPTION_KEY) {  
  throw new Error('ENCRYPTION_KEY environment variable is required but not set');  
}  
  
if (ENCRYPTION_KEY.length < 32) {  
  throw new Error('ENCRYPTION_KEY must be at least 32 characters long for security');  
}  
  
// Validate it's not the old default key  
if (ENCRYPTION_KEY.includes('default') || ENCRYPTION_KEY.includes('change-in-produc-  
tion')) {  
  throw new Error('ENCRYPTION_KEY appears to be using a default value. Please set a  
secure key.');
```

---

## HIGH: Password Strength Validation

---

**File:** app/api/signup/route.ts

**Add validation function at the top of the file:**

```

/**
 * Validates password meets security requirements
 */
function validatePasswordStrength(password: string): {
  valid: boolean;
  errors: string[]
} {
  const errors: string[] = [];

  // Length requirement
  if (password.length < 12) {
    errors.push('Password must be at least 12 characters long');
  }

  if (password.length > 128) {
    errors.push('Password must not exceed 128 characters');
  }

  // Complexity requirements
  if (!/[A-Z]/.test(password)) {
    errors.push('Password must contain at least one uppercase letter (A-Z)');
  }

  if (!/[a-z]/.test(password)) {
    errors.push('Password must contain at least one lowercase letter (a-z)');
  }

  if (!/[0-9]/.test(password)) {
    errors.push('Password must contain at least one number (0-9)');
  }

  if (!/^[^A-Za-z0-9]/.test(password)) {
    errors.push('Password must contain at least one special character (!@#$$%^&*...)');
  }

  // Check against common passwords
  const commonPasswords = [
    'password', 'password123', '123456', '12345678', 'qwerty',
    'abc123', 'monkey', '1234567', 'letmein', 'trustno1',
    'dragon', 'baseball', 'iloveyou', 'master', 'sunshine',
    'ashley', 'bailey', 'shadow', 'superman', 'qazwsx',
    'michael', 'football', 'welcome', 'jesus', 'ninja',
    'mustang', 'password1', 'admin', 'admin123', 'picard'
  ];

  if (commonPasswords.includes(password.toLowerCase())) {
    errors.push('Password is too common. Please choose a more unique password');
  }

  // Check for sequential characters
  if (/([?:abc|bcd|cde|def|efg|fgh|ghi|hij|ijk|jkl|klm|lmn|mno|nop|opq|pqr|qrs|rst|stu|
tuv|uvw|vwx|wxy|xyz|012|123|234|345|456|567|678|789)/i.test(password)) {
    errors.push('Password contains sequential characters');
  }

  // Check for repeated characters
  if (/([.])\1{2,}/.test(password)) {
    errors.push('Password contains too many repeated characters');
  }

  return {
    valid: errors.length === 0,

```

```

    errors
  };
}

```

### Update the POST handler (after line 25):

```

// Add after the null check and before existingUser check
const passwordValidation = validatePasswordStrength(password);
if (!passwordValidation.valid) {
  return NextResponse.json(
    {
      message: 'Password does not meet security requirements',
      errors: passwordValidation.errors
    },
    { status: 400 }
  );
}

```

## HIGH: Sanitize Sensitive Logs

**File:** lib/database-query-executor.ts

### Add sanitization function:

```

/**
 * Sanitizes SQL query for logging (removes potential PII)
 */
function sanitizeSQLForLogging(sql: string): string {
  return sql
    // Replace string literals with placeholder
    .replace(/'[^']*'/g, '****')
    .replace(/"[^"]*" /g, '****')
    // Replace numbers that might be IDs or sensitive
    .replace(/\b\d{6,}\b/g, '#####');
}

```

### Update logging statements:

```

// Line 53 - BEFORE:
console.log('Executing SQL:', sql)

// Line 53 - AFTER:
if (process.env.NODE_ENV === 'development') {
  console.log('Executing SQL:', sanitizeSQLForLogging(sql));
}

// Line 81 - BEFORE:
console.error('Failed SQL:', sql)

// Line 81 - AFTER:
console.error('Failed SQL:', sanitizeSQLForLogging(sql))

```

**File:** app/api/query/route.ts

## Update all logging statements:

```
// Line 241 - BEFORE:
console.log('✅ Using cached SQL for query:', query);

// Line 241 - AFTER:
if (process.env.NODE_ENV === 'development') {
  console.log('✅ Using cached SQL (query hash:',
    query.substring(0, 20) + '...' + ')');
}

// Line 271 - BEFORE:
console.log('🔄 Generating new SQL for query:', query);

// Line 271 - AFTER:
console.log('🔄 Generating new SQL translation');
```

---

## 🟡 MEDIUM: Input Length Validation

---

**File:** app/api/query/route.ts

### Add validation constants at the top:

```
// Security limits
const MAX_QUERY_LENGTH = 1000;
const MAX_DATABASE_ID_LENGTH = 100;
const MAX_CONTEXT_LENGTH = 2000;
```

## Update validation section (around line 174):

```
// BEFORE:
if (!query || !databaseId) {
  return NextResponse.json({ error: 'Missing required fields' }, { status: 400 })
}

// AFTER:
if (!query || !databaseId) {
  return NextResponse.json({ error: 'Missing required fields' }, { status: 400 })
}

// Length validation
if (query.length > MAX_QUERY_LENGTH) {
  return NextResponse.json({
    error: `Query too long. Maximum ${MAX_QUERY_LENGTH} characters allowed.`
  }, { status: 400 })
}

if (databaseId.length > MAX_DATABASE_ID_LENGTH) {
  return NextResponse.json({
    error: 'Invalid database ID format'
  }, { status: 400 })
}

if (context && context.length > MAX_CONTEXT_LENGTH) {
  return NextResponse.json({
    error: `Context too long. Maximum ${MAX_CONTEXT_LENGTH} characters allowed.`
  }, { status: 400 })
}

// Sanitize inputs
const sanitizedQuery = query.trim();
const sanitizedDatabaseId = databaseId.trim();

if (sanitizedQuery.length === 0) {
  return NextResponse.json({
    error: 'Query cannot be empty'
  }, { status: 400 })
}
```

## MEDIUM: Session Timeout Configuration

**File:** lib/auth.ts

## Update session configuration (around line 78):

```
// BEFORE:
session: {
  strategy: 'jwt' as const,
},

// AFTER:
session: {
  strategy: 'jwt' as const,
  maxAge: 8 * 60 * 60, // 8 hours in seconds
  updateAge: 60 * 60, // Update session every hour
},
jwt: {
  maxAge: 8 * 60 * 60, // 8 hours
},
```

---

## Testing the Fixes

After implementing these fixes, test with:

### 1. Test SQL Validation

```
// Should be blocked:
const maliciousQueries = [
  "SELECT * FROM users; DROP TABLE users;",
  "SELECT * FROM users WHERE id = 1 OR 1=1",
  "SELECT * FROM users UNION SELECT * FROM passwords",
  "DELETE FROM users WHERE 1=1",
  "UPDATE users SET admin = true",
];

// Should be allowed:
const validQueries = [
  "SELECT * FROM users WHERE status = 'active'",
  "SELECT COUNT(*) FROM orders WHERE date > '2024-01-01'",
  "SELECT u.name, o.total FROM users u JOIN orders o ON u.id = o.user_id",
];
```

## 2. Test Password Validation

```
// Should be rejected:
const weakPasswords = [
  "password",
  "123456",
  "short",
  "NoSpecial123",
  "no-uppercase-123!",
];

// Should be accepted:
const strongPasswords = [
  "MyStr0ng!Passw0rd",
  "C0mpl3x&Secur3!Pass",
  "Un!qu3P@ssw0rd2024",
];
```

## 3. Test Input Validation

```
# Test query length limit
curl -X POST http://localhost:3000/api/query \
  -H "Content-Type: application/json" \
  -d "{\"query\": \"${printf 'a%.0s' {1..1001}}\\\", \"databaseId\": \"sales\\\"}"
# Should return 400 error
```

## Deployment Checklist

Before deploying to production:

- [ ] All critical fixes implemented
- [ ] SQL validation tested with malicious inputs
- [ ] Weak encryption fallback removed
- [ ] Password validation enforced
- [ ] Sensitive data removed from logs
- [ ] Input length validation in place
- [ ] Session timeouts configured
- [ ] Environment variables validated
- [ ] Security tests passing
- [ ] Code review completed

## Emergency Rollback

If issues arise after deployment:

1. **Disable SQL validation temporarily** (if causing false positives):

```
typescript
// In validateSQL function, at the start:
if (process.env.DISABLE_SQL_VALIDATION === 'true') {
```

```
console.warn('⚠️ SQL validation is disabled - this is insecure!');  
return { valid: true };  
}
```

2. **Monitor error logs** for rejected queries
3. **Update validation rules** based on legitimate queries being blocked
4. **Re-enable with improved rules**

**Important:** Never leave SQL validation disabled in production!

---

## Questions or Issues?

---

If you encounter problems implementing these fixes:

1. Check the full security audit report for context
  2. Review error logs for specific validation failures
  3. Test in development environment first
  4. Consider gradual rollout with feature flags
- 

**Last Updated:** November 5, 2025

**Security Audit Version:** 1.0