# Security and Error Handling

## Overview

This document details the security architecture, error handling strategies, and data protection mechanisms in CareIQ Builder.

## Security Architecture

### 1. Defense in Depth

CareIQ Builder implements multiple security layers:

┌─────────────────────────────────────────────┐  
│ Layer 1: Client-Side (Browser) │  
│ - No credentials in code │  
│ - No direct external API calls │  
│ - Input validation │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ Layer 2: ServiceNow Platform │  
│ - Authentication & Authorization │  
│ - Request validation │  
│ - Rate limiting │  
│ - Audit logging │  
│ - Script security restrictions │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ Layer 3: CareIQ Services Script Include │  
│ - Business logic validation │  
│ - Token management │  
│ - Request construction │  
│ - Response sanitization │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ Layer 4: CareIQ Platform (External) │  
│ - API authentication │  
│ - Data validation │  
│ - Access control │  
└─────────────────────────────────────────────┘

### 2. No Direct External API Calls

**Security Principle**: All external API calls routed through ServiceNow

**Implementation**:

// ❌ WRONG - Direct call to CareIQ from client  
fetch('https://careiq.platform.com/api/questions', {  
 headers: {  
 'Authorization': 'Bearer ' + token // Exposes token!  
 }  
});  
  
// ✅ CORRECT - Call through ServiceNow  
dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {  
 requestBody: JSON.stringify({...})  
});

**Benefits**: - Credentials never exposed to client - ServiceNow authenticates requests - Centralized security control - Audit logging at ServiceNow level - Rate limiting possible - Request validation server-side

**Architecture**:

Client (Browser)  
 ↓ HTTPS  
ServiceNow REST API (validates, logs, authenticates)  
 ↓ HTTPS  
CareIQ Services Script Include (adds credentials, constructs request)  
 ↓ HTTPS + OAuth Token  
CareIQ Platform

### 3. ServiceNow Authentication

**Session Management**: - Users authenticate to ServiceNow - ServiceNow manages session - Session timeout enforced by platform - Component inherits ServiceNow authentication

**Authorization**: - ServiceNow ACLs (Access Control Lists) control access - User permissions checked at API level - Role-based access control (RBAC)

**Token Management**:

// Script Include handles token securely  
getCareIQAccessToken: function() {  
 // Token stored securely in ServiceNow  
 // Not accessible from client code  
 var tokenRecord = new GlideRecord('x\_careiq\_tokens');  
 tokenRecord.query();  
 if (tokenRecord.next()) {  
 return tokenRecord.getValue('token');  
 }  
 return null;  
}

### 4. Input Validation

**Client-Side Validation** (UX, not security):

'SAVE\_QUESTION': (coeffects) => {  
 const {action, state, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 const changes = state.questionChanges[questionId];  
  
 // Validate before sending  
 if (!changes || Object.keys(changes).length === 0) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'warning',  
 message: 'No changes to save'  
 });  
 return; // Don't proceed  
 }  
  
 // Validate label length  
 if (changes.label && changes.label.length > 500) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Question label too long (max 500 characters)'  
 });  
 return;  
 }  
  
 // Proceed with API call  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {...});  
}

**Server-Side Validation** (security critical):

(function process(request, response) {  
 var requestData = request.body.data;  
  
 // Validate required fields  
 if (!requestData) {  
 response.setStatus(400);  
 response.setBody({  
 success: false,  
 message: 'Request data is required'  
 });  
 return;  
 }  
  
 if (!requestData.question\_id) {  
 response.setStatus(400);  
 response.setBody({  
 success: false,  
 message: 'question\_id is required'  
 });  
 return;  
 }  
  
 // Validate data types  
 if (typeof requestData.question\_id !== 'string') {  
 response.setStatus(400);  
 response.setBody({  
 success: false,  
 message: 'question\_id must be a string'  
 });  
 return;  
 }  
  
 // Validate UUID format  
 var uuidRegex = /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$/i;  
 if (!uuidRegex.test(requestData.question\_id)) {  
 response.setStatus(400);  
 response.setBody({  
 success: false,  
 message: 'question\_id must be a valid UUID'  
 });  
 return;  
 }  
  
 // Proceed with business logic  
 // ...  
})(request, response);

### 5. ServiceNow Security Restrictions

**Error Object Access Restriction**:

ServiceNow prevents direct access to error object properties for security:

// ❌ NOT ALLOWED  
catch (e) {  
 var msg = e.message; // Security violation  
 var stack = e.stack; // Security violation  
 gs.error(e.message); // Not allowed  
}  
  
// ✅ ALLOWED  
catch (e) {  
 var errorMsg = 'Unexpected server error occurred';  
  
 try {  
 // toString() method is allowed  
 if (e && typeof e.toString === 'function') {  
 errorMsg = e.toString();  
 }  
 } catch (innerE) {  
 // Even toString failed  
 errorMsg = 'Server error occurred';  
 }  
  
 gs.error('Error in API: ' + errorMsg);  
}

**Why This Restriction**: - Prevents exposing sensitive error details - Avoids stack trace exposure to client - Enforces sanitized error messages

### 6. Data Sanitization

**Output Encoding**:

// In view layer, snabbdom handles XSS prevention automatically  
<div>{state.userInput}</div> // Automatically escaped  
  
// For attributes  
<div title={state.userInput}></div> // Automatically escaped  
  
// For innerHTML-like scenarios (rare, avoid if possible)  
<div dangerouslySetInnerHTML={{\_\_html: sanitize(state.htmlContent)}}></div>

**Input Sanitization**:

// Remove potentially dangerous characters  
function sanitizeInput(input) {  
 if (typeof input !== 'string') return input;  
  
 // Remove control characters  
 input = input.replace(/[\x00-\x1F\x7F]/g, '');  
  
 // Trim whitespace  
 input = input.trim();  
  
 // Limit length  
 if (input.length > 10000) {  
 input = input.substring(0, 10000);  
 }  
  
 return input;  
}

### 7. HTTPS Enforcement

**All Communication Over HTTPS**: - ServiceNow enforces HTTPS - CareIQ API uses HTTPS - No sensitive data in HTTP

**Certificate Validation**: - ServiceNow validates SSL certificates - No self-signed certificates in production

### 8. Audit Logging

**ServiceNow Audit Trail**: - All API calls logged by ServiceNow - User, timestamp, action recorded - Can be reviewed by administrators

**Application Logging**:

// Script Include logging  
builderUpdateQuestion: function(requestData) {  
 gs.info('Builder API: updateQuestion called by ' + gs.getUserID());  
 gs.info('Builder API: questionId=' + requestData.question\_id);  
  
 try {  
 var result = this.callCareIQAPI(requestData);  
 gs.info('Builder API: updateQuestion succeeded');  
 return result;  
 } catch (e) {  
 gs.error('Builder API: updateQuestion failed - ' + e.toString());  
 throw e;  
 }  
}

### 9. Rate Limiting

**ServiceNow Level**: - ServiceNow can enforce rate limits - Per-user or per-endpoint limits - Prevents abuse

**Application Level**:

// Client-side throttling (UX, not security)  
'API\_CALL\_WITH\_THROTTLE': (coeffects) => {  
 const {state, updateState, dispatch} = coeffects;  
  
 const now = Date.now();  
 const lastCall = state.lastAPICallTime || 0;  
  
 // Enforce minimum 500ms between calls  
 if (now - lastCall < 500) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'warning',  
 message: 'Please wait before making another request'  
 });  
 return;  
 }  
  
 updateState({lastAPICallTime: now});  
 dispatch('MAKE\_API\_REQUEST', {...});  
}

### 10. Cross-Site Scripting (XSS) Prevention

**Snabbdom Auto-Escaping**: - Virtual DOM automatically escapes content - Text nodes are safe by default

**Dangerous Patterns to Avoid**:

// ❌ DANGEROUS - innerHTML  
element.innerHTML = userInput;  
  
// ❌ DANGEROUS - dangerouslySetInnerHTML  
<div dangerouslySetInnerHTML={{\_\_html: userInput}}></div>  
  
// ✅ SAFE - Text content  
<div>{userInput}</div>  
  
// ✅ SAFE - Attribute  
<div title={userInput}></div>

### 11. SQL Injection Prevention

**Not Applicable**: - No direct database queries from client - ServiceNow handles database interactions - CareIQ API handles its own database security

**If Using ServiceNow GlideRecord**:

// ✅ SAFE - Parameterized query  
var gr = new GlideRecord('table\_name');  
gr.addQuery('field', userInput); // Parameterized, safe  
gr.query();  
  
// ❌ DANGEROUS - String concatenation  
var gr = new GlideRecord('table\_name');  
gr.addEncodedQuery('field=' + userInput); // Vulnerable if userInput not validated

### 12. Cross-Site Request Forgery (CSRF) Prevention

**ServiceNow Built-In Protection**: - ServiceNow automatically includes CSRF tokens - Validated on server side - No additional implementation needed

## Error Handling

### 1. Error Hierarchy

┌─────────────────────────────────────────────┐  
│ User-Facing Errors │  
│ - Clear, actionable messages │  
│ - Displayed in system messages │  
│ - No technical details │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ Application Errors │  
│ - Action handler errors │  
│ - Validation errors │  
│ - Business logic errors │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ API Errors │  
│ - HTTP error codes │  
│ - Structured error responses │  
│ - Error details from backend │  
└─────────────────────────────────────────────┘  
 ↓  
┌─────────────────────────────────────────────┐  
│ System Errors │  
│ - ServiceNow platform errors │  
│ - CareIQ platform errors │  
│ - Network errors │  
└─────────────────────────────────────────────┘

### 2. Client-Side Error Handling

**Action Handler Errors**:

'SAVE\_QUESTION': (coeffects) => {  
 const {action, state, dispatch} = coeffects;  
  
 try {  
 const {questionId} = action.payload;  
 const changes = state.questionChanges[questionId];  
  
 // Validation  
 if (!questionId) {  
 throw new Error('Question ID is required');  
 }  
  
 if (!changes) {  
 throw new Error('No changes found for question');  
 }  
  
 // Proceed with save  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {...});  
  
 } catch (error) {  
 // Handle error gracefully  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: error.message || 'Failed to save question'  
 });  
  
 // Log to console for debugging  
 console.error('SAVE\_QUESTION error:', error);  
 }  
}

**Effect Error Handling**:

'UPDATE\_QUESTION\_ERROR': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 const error = action.payload;  
 const questionId = action.meta?.questionId;  
  
 // Clear loading state  
 if (questionId) {  
 const updatedSaving = {...state.savingQuestions};  
 delete updatedSaving[questionId];  
 updateState({savingQuestions: updatedSaving});  
 }  
  
 // Determine error type and message  
 let errorMessage = 'Failed to save question';  
 let messageType = 'error';  
  
 if (error.status === 0) {  
 errorMessage = 'Network connection lost. Please check your internet connection.';  
 } else if (error.status === 400) {  
 errorMessage = error.message || 'Invalid request. Please check your input.';  
 } else if (error.status === 401 || error.status === 403) {  
 errorMessage = 'You do not have permission to perform this action.';  
 } else if (error.status === 404) {  
 errorMessage = 'Question not found. It may have been deleted.';  
 } else if (error.status === 409) {  
 errorMessage = error.message || 'Conflict detected. Please refresh and try again.';  
 messageType = 'warning';  
 } else if (error.status >= 500) {  
 errorMessage = 'Server error. Please try again later.';  
 } else if (error.message) {  
 errorMessage = error.message;  
 }  
  
 // Show error message  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: messageType,  
 message: errorMessage  
 });  
  
 // Log for debugging  
 console.error('UPDATE\_QUESTION\_ERROR:', {  
 status: error.status,  
 message: error.message,  
 detail: error.detail,  
 questionId: questionId  
 });  
}

### 3. Server-Side Error Handling

**ServiceNow REST API**:

(function process(request, response) {  
 var requestData = request.body.data;  
  
 try {  
 // Validation  
 if (!requestData) {  
 throw {  
 status: 400,  
 message: 'Request data is required'  
 };  
 }  
  
 if (!requestData.question\_id) {  
 throw {  
 status: 400,  
 message: 'question\_id is required'  
 };  
 }  
  
 // Call Script Include  
 var careiqServices = new x\_1628056\_careiq.CareIQServices();  
 var result = careiqServices.builderUpdateQuestion(requestData);  
  
 // Check result  
 if (!result.success) {  
 throw {  
 status: 400,  
 message: result.message || 'Operation failed'  
 };  
 }  
  
 // Return success  
 response.setStatus(200);  
 response.setBody({  
 success: true,  
 message: result.message || 'Question updated successfully',  
 data: result.data  
 });  
  
 } catch (e) {  
 // Determine status code  
 var status = e.status || 500;  
 var message = 'Unexpected server error occurred';  
  
 // Extract message safely  
 try {  
 if (e.message) {  
 message = e.message;  
 } else if (e && typeof e.toString === 'function') {  
 message = e.toString();  
 }  
 } catch (innerE) {  
 // Use default message  
 }  
  
 // Log error (server-side only)  
 gs.error('REST API Error [update-question]: ' + message);  
  
 // Return error response  
 response.setStatus(status);  
 response.setBody({  
 success: false,  
 message: message,  
 detail: e.detail || null  
 });  
 }  
})(request, response);

**Script Include Error Handling**:

builderUpdateQuestion: function(requestData) {  
 try {  
 var questionId = requestData.question\_id;  
 var label = requestData.label;  
  
 // Validate  
 if (!questionId || !label) {  
 return {  
 success: false,  
 message: 'Missing required fields'  
 };  
 }  
  
 // Prepare CareIQ API request  
 var careiqUrl = this.getCareIQBaseUrl() + '/builder/question/' + questionId;  
 var careiqPayload = {  
 label: label,  
 type: requestData.type,  
 voice: requestData.voice  
 };  
  
 // Make request  
 var request = new sn\_ws.RESTMessageV2();  
 request.setHttpMethod('PUT');  
 request.setEndpoint(careiqUrl);  
 request.setRequestHeader('Content-Type', 'application/json');  
 request.setRequestHeader('Authorization', 'Bearer ' + this.getAccessToken());  
 request.setRequestBody(JSON.stringify(careiqPayload));  
  
 var response = request.execute();  
 var httpStatus = response.getStatusCode();  
 var responseBody = response.getBody();  
  
 // Handle response  
 if (httpStatus >= 200 && httpStatus < 300) {  
 var parsedResponse = JSON.parse(responseBody);  
 return {  
 success: true,  
 message: 'Question updated successfully',  
 data: parsedResponse  
 };  
 } else {  
 // CareIQ API error  
 gs.error('CareIQ API error: ' + httpStatus + ' - ' + responseBody);  
 return {  
 success: false,  
 message: 'Failed to update question: ' + responseBody  
 };  
 }  
  
 } catch (e) {  
 // Safe error handling  
 var errorMsg = 'Unexpected error in builderUpdateQuestion';  
 try {  
 if (e && typeof e.toString === 'function') {  
 errorMsg = e.toString();  
 }  
 } catch (innerE) {  
 // Use default  
 }  
  
 gs.error('Script Include Error: ' + errorMsg);  
  
 return {  
 success: false,  
 message: errorMsg  
 };  
 }  
}

### 4. Error Response Format

**Standard Error Response**:

{  
 success: false,  
 message: 'User-friendly error message',  
 detail: 'Optional detailed error information',  
 code: 'ERROR\_CODE', // Optional error code  
 field: 'fieldName' // Optional field that caused error  
}

**HTTP Status Codes Used**: - 400 - Bad Request (validation errors) - 401 - Unauthorized - 403 - Forbidden (insufficient permissions) - 404 - Not Found - 409 - Conflict (duplicate, constraint violation) - 500 - Internal Server Error - 503 - Service Unavailable

### 5. User Feedback on Errors

**System Messages**:

// Success (green)  
dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'success',  
 message: 'Question saved successfully!'  
});  
  
// Error (red)  
dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Failed to save question. Please try again.'  
});  
  
// Warning (yellow)  
dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'warning',  
 message: 'Question saved, but duplicate answer detected.'  
});  
  
// Info (blue)  
dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'info',  
 message: 'Changes will be published after approval.'  
});

**Loading States**: - Clear on success - Clear on error - Prevents stuck loading indicators

**Retry Mechanisms**:

'API\_ERROR\_WITH\_RETRY': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Store failed action for retry  
 updateState({  
 lastFailedAction: action.meta?.originalAction,  
 retryCount: (state.retryCount || 0) + 1  
 });  
  
 // Show error with retry button (if not exceeded retry limit)  
 if (state.retryCount < 3) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Request failed. Click to retry.',  
 action: 'RETRY\_LAST\_ACTION'  
 });  
 } else {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Request failed after multiple attempts. Please contact support.'  
 });  
 }  
}

### 6. Logging and Debugging

**Client-Side Logging**:

// Development logging  
if (process.env.NODE\_ENV === 'development') {  
 console.log('State:', state);  
 console.log('Action:', action);  
}  
  
// Error logging (always)  
console.error('Error in action handler:', error);  
console.error('Stack trace:', error.stack);  
  
// Performance logging  
console.time('API\_CALL');  
dispatch('MAKE\_API\_REQUEST', {...});  
// ... later ...  
console.timeEnd('API\_CALL');

**Server-Side Logging**:

// Info logging  
gs.info('Builder API: Operation started by user ' + gs.getUserID());  
  
// Error logging  
gs.error('Builder API: Operation failed - ' + errorMessage);  
  
// Debug logging (if debug enabled)  
if (gs.getProperty('careiq.debug.enabled') === 'true') {  
 gs.debug('Builder API: Request data: ' + JSON.stringify(requestData));  
}

## Best Practices

### Security

**DO**: ✅ Route all external API calls through ServiceNow ✅ Validate input on server side ✅ Use HTTPS for all communication ✅ Use safe error handling (no e.message in ServiceNow) ✅ Sanitize output ✅ Log security-relevant events ✅ Implement rate limiting ✅ Use authentication and authorization

**DON’T**: ❌ Expose credentials in client code ❌ Make direct API calls to external services ❌ Trust client-side validation alone ❌ Return detailed error messages to client ❌ Log sensitive data ❌ Access e.message or e.stack in ServiceNow catch blocks

### Error Handling

**DO**: ✅ Handle errors at every layer ✅ Provide user-friendly messages ✅ Clear loading states on error ✅ Log errors for debugging ✅ Return structured error responses ✅ Use appropriate HTTP status codes ✅ Offer retry for transient errors

**DON’T**: ❌ Expose stack traces to users ❌ Leave error states unhandled ❌ Show technical jargon to users ❌ Forget to clear loading indicators ❌ Swallow errors silently

## Summary

CareIQ Builder’s security and error handling: - **Defense in Depth**: Multiple security layers - **No Direct External Calls**: All routed through ServiceNow - **Safe Error Handling**: ServiceNow security restrictions followed - **Input Validation**: Both client and server side - **Structured Errors**: Consistent error response format - **User Feedback**: Clear, actionable error messages - **Audit Logging**: All operations logged - **Rate Limiting**: Prevents abuse

This architecture provides secure, resilient, and user-friendly error handling.