PECH Prayer Diary - Security Recommendations

Based on my review of the PECH Prayer Diary codebase, I've identified several areas where security could be enhanced. Below are specific recommendations with implementation details.

1. Enhance Content Security Policies

The application doesn't currently implement Content Security Policies (CSP), which would help prevent XSS attacks and other code injection vulnerabilities.

Implementation:

Add the following CSP meta tag to the `<head>` section of `index.html`:

```
html

<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' https
```

For even better protection, configure CSP at the server level. If you're using GitHub Pages, this isn't possible, but you can implement this if you move to a custom web server.

2. Implement Supabase Row-Level Security (RLS)

Although the app uses client-side permission checks, these could be bypassed. Supabase's Row-Level Security (RLS) provides server-side enforcement.

Implementation:

```sql
-- Example RLS policy for the prayer_updates table

-- First, create authentication validation function
CREATE OR REPLACE FUNCTION auth.is_admin()
RETURNS BOOLEAN AS $$
  SELECT EXISTS (
    SELECT 1 FROM profiles
    WHERE id = auth.uid() AND user_role = 'Administrator'
  );
$$ LANGUAGE sql SECURITY DEFINER;

CREATE OR REPLACE FUNCTION auth.can_edit_updates()
RETURNS BOOLEAN AS $$
  SELECT EXISTS (
    SELECT 1 FROM profiles
    WHERE id = auth.uid() AND (user_role = 'Administrator' OR prayer_update_editor = true)
  );
$$ LANGUAGE sql SECURITY DEFINER;

-- Enable RLS on prayer_updates table
ALTER TABLE prayer_updates ENABLE ROW LEVEL SECURITY;

-- Create policies
-- Read policy - anyone who is authenticated can read updates
CREATE POLICY "Anyone can read prayer updates"
ON prayer_updates FOR SELECT
USING (auth.role() = 'authenticated');

-- Insert policy - only admins and update editors can create
CREATE POLICY "Editors can create prayer updates"
ON prayer_updates FOR INSERT
WITH CHECK (auth.can_edit_updates());

-- Update policy - only admins and update editors can update
CREATE POLICY "Editors can update prayer updates"
ON prayer_updates FOR UPDATE
USING (auth.can_edit_updates());

-- Delete policy - only admins can delete
CREATE POLICY "Admin can delete prayer updates"
ON prayer_updates FOR DELETE
USING (auth.is_admin());
```

Apply similar policies to all tables in your database.

3. Secure Local Storage Usage

The app stores sensitive information in localStorage which could be vulnerable to XSS attacks.

Implementation:

Create a secure storage utility that encrypts sensitive data:

javascript

```
// Add to a new file called secure-storage.js
const secureStorage = {
    // Generate a device-specific key
    async getEncryptionKey() {
        let key = localStorage.getItem('app_encryption_key');

        if (!key) {
            // Generate a random key
            const buffer = new Uint8Array(32);
            window.crypto.getRandomValues(buffer);
            key = Array.from(buffer).map(b => b.toString(16).padStart(2, '0')).join('');
            localStorage.setItem('app_encryption_key', key);
        }

        // Convert to CryptoKey
        const keyData = new TextEncoder().encode(key);
        return window.crypto.subtle.importKey(
            'raw',
            keyData,
            { name: 'AES-GCM' },
            false,
            ['encrypt', 'decrypt']
        );
    },

    // Encrypt a value
    async encrypt(value) {
        const key = await this.getEncryptionKey();

        // Create initialization vector
        const iv = window.crypto.getRandomValues(new Uint8Array(12));

        // Encrypt
        const encodedValue = new TextEncoder().encode(JSON.stringify(value));
        const encryptedBuffer = await window.crypto.subtle.encrypt(
            { name: 'AES-GCM', iv },
            key,
            encodedValue
        );

        // Convert to base64
        const encryptedArray = Array.from(new Uint8Array(encryptedBuffer));
        const encryptedBase64 = btoa(String.fromCharCode.apply(null, encryptedArray));
        const ivBase64 = btoa(String.fromCharCode.apply(null, Array.from(iv)));

        return `${ivBase64}.${encryptedBase64}`;
```

```
},

// Decrypt a value
async decrypt(encryptedValue) {
    try {
        const key = await this.getEncryptionKey();

        // Split IV and data
        const [ivBase64, encryptedBase64] = encryptedValue.split('.');

        // Convert from base64
        const iv = new Uint8Array(
            atob(ivBase64).split('').map(c => c.charCodeAt(0))
        );
        const encryptedArray = new Uint8Array(
            atob(encryptedBase64).split('').map(c => c.charCodeAt(0))
        );

        // Decrypt
        const decryptedBuffer = await window.crypto.subtle.decrypt(
            { name: 'AES-GCM', iv },
            key,
            encryptedArray
        );

        // Decode
        const decodedValue = new TextDecoder().decode(decryptedBuffer);
        return JSON.parse(decodedValue);
    } catch (error) {
        console.error('Decryption error:', error);
        return null;
    }
},

// Secure set item
async setItem(key, value) {
    const encryptedValue = await this.encrypt(value);
    localStorage.setItem(key, encryptedValue);
},

// Secure get item
async getItem(key) {
    const encryptedValue = localStorage.getItem(key);
    if (!encryptedValue) return null;
    return await this.decrypt(encryptedValue);
},
```

```
    // Remove item
    removeItem(key) {
        localStorage.removeItem(key);
    }
};

// Usage example:
// Instead of localStorage.setItem('sensitive_data', JSON.stringify(data))
// Use: await secureStorage.setItem('sensitive_data', data)
```

4. Implement Token Refresh and Proper Token Handling

The current token handling could be improved to reduce the risk of token theft.

Implementation:

javascript

```
// Add to auth.js
async function refreshAuthToken() {
    try {
        // Only refresh if we're within 5 minutes of expiration
        const { data: { session } } = await supabase.auth.getSession();

        if (!session) return null;

        // Check if token is about to expire (within 5 minutes)
        const expiresAt = session.expires_at * 1000; // Convert to milliseconds
        const fiveMinutesFromNow = Date.now() + 5 * 60 * 1000;

        if (expiresAt < fiveMinutesFromNow) {
            console.log('Token about to expire, refreshing...');
            const { data, error } = await supabase.auth.refreshSession();

            if (error) throw error;

            return data.session;
        }

        return session;
    } catch (error) {
        console.error('Error refreshing token:', error);
        return null;
    }
}

// Add a timer to check token expiration periodically
function startTokenRefreshTimer() {
    // Check every 4 minutes
    const intervalId = setInterval(async () => {
        if (isLoggedIn()) {
            try {
                await refreshAuthToken();
            } catch (error) {
                console.error('Error in token refresh timer:', error);
            }
        } else {
            // User is logged out, stop checking
            clearInterval(intervalId);
        }
    }, 4 * 60 * 1000);
}

// Call startTokenRefreshTimer when the user logs in
```

```
document.addEventListener('login-state-changed', function(event) {
    if (event.detail && event.detail.loggedIn) {
        startTokenRefreshTimer();
    }
});
```

5. Input Validation and Content Sanitization

The app should validate all user inputs and sanitize content to prevent XSS attacks.

Implementation:

javascript

```
// Add to a new file called validation.js
const validation = {
    // Sanitize HTML content to prevent XSS
    sanitizeHtml(html) {
        if (!html) return '';

        // Create a DOMParser to parse the HTML
        const parser = new DOMParser();
        const doc = parser.parseFromString(html, 'text/html');

        // Remove potentially dangerous elements and attributes
        const sanitize = (node) => {
            // If this is a text node, return as is
            if (node.nodeType === Node.TEXT_NODE) {
                return;
            }

            // Remove script tags completely
            if (node.tagName === 'SCRIPT') {
                node.parentNode.removeChild(node);
                return;
            }

            // Remove dangerous attributes
            const dangerousAttrs = ['onclick', 'onload', 'onerror', 'onmouseover', 'onmouseout'
            dangerousAttrs.forEach(attr => {
                if (node.hasAttribute && node.hasAttribute(attr)) {
                    node.removeAttribute(attr);
                }
            });

            // For style attribute, only allow safe properties
            if (node.hasAttribute && node.hasAttribute('style')) {
                const style = node.getAttribute('style');
                // Remove any url or expression
                if (style.includes('url(') || style.includes('expression(')) {
                    node.removeAttribute('style');
                }
            }

            // Recursively sanitize children
            if (node.childNodes) {
                Array.from(node.childNodes).forEach(sanitize);
            }
        };
```

```
        // Start sanitizing from the body
        sanitize(doc.body);

        // Return the sanitized HTML
        return doc.body.innerHTML;
    },

    // Validate email format
    validateEmail(email) {
        const emailRegex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
        return emailRegex.test(email);
    },

    // Validate date format (YYYY-MM-DD)
    validateDate(date) {
        const dateRegex = /^\d{4}-\d{2}-\d{2}$/;
        if (!dateRegex.test(date)) return false;

        // Check if it's a valid date
        const parsedDate = new Date(date);
        return !isNaN(parsedDate.getTime());
    },

    // Validate phone number (UK format)
    validatePhoneUK(phone) {
        const phoneRegex = /^((\+44)|(0))[0-9]{10}$/;
        return phoneRegex.test(phone);
    }
};

// Then use this for validation and sanitization:
// const sanitizedContent = validation.sanitizeHtml(updateEditor.root.innerHTML);
// if (!validation.validateEmail(email)) {
//     throw new Error('Invalid email format');
// }
```

6. Implement CSRF Protection

Although Supabase handles CSRF protection internally, add an extra layer of security for sensitive operations.

Implementation:

```javascript
// Add to app.js
function generateCsrfToken() {
    const token = Math.random().toString(36).substring(2, 15) +
                  Math.random().toString(36).substring(2, 15);

    // Store in sessionStorage
    sessionStorage.setItem('csrfToken', token);
    return token;
}

function validateCsrfToken(token) {
    const storedToken = sessionStorage.getItem('csrfToken');

    // Remove the token after checking (one-time use)
    sessionStorage.removeItem('csrfToken');

    return token === storedToken;
}

// Then use for sensitive operations:
// In the form submission handler:
// const csrfToken = document.getElementById('csrf-token').value;
// if (!validateCsrfToken(csrfToken)) {
//     throw new Error('Invalid security token');
// }

// Add a hidden input to sensitive forms:
// <input type="hidden" id="csrf-token" name="csrf-token" value="${generateCsrfToken()}">
```

7. Implement Rate Limiting

Protect against brute force attacks by implementing rate limiting.

Implementation:

javascript

```
// Add to auth.js
const loginAttempts = {
    count: 0,
    resetTime: 0,
    maxAttempts: 5,
    lockoutDuration: 15 * 60 * 1000 // 15 minutes
};

// Modify handleAuth function to include rate limiting
async function handleAuth(e) {
    e.preventDefault();

    // Check for rate limiting
    const now = Date.now();

    // If we're in a lockout period
    if (loginAttempts.resetTime > now) {
        const remainingMinutes = Math.ceil((loginAttempts.resetTime - now) / 60000);

        const errorElem = document.getElementById('auth-error');
        errorElem.querySelector('p').textContent =
            `Too many login attempts. Please try again in ${remainingMinutes} minutes.`;
        errorElem.classList.remove('d-none');

        return;
    }

    // Reset attempts if the lockout period has passed
    if (loginAttempts.resetTime > 0 && loginAttempts.resetTime <= now) {
        loginAttempts.count = 0;
        loginAttempts.resetTime = 0;
    }

    // Original login logic...
    try {
        // Login logic here...

        // On successful login, reset login attempts
        loginAttempts.count = 0;
        loginAttempts.resetTime = 0;

    } catch (error) {
        // Increment failed attempts
        loginAttempts.count++;

        // If exceeded max attempts, set lockout
```

```
            if (loginAttempts.count >= loginAttempts.maxAttempts) {
                loginAttempts.resetTime = now + loginAttempts.lockoutDuration;

                const errorElem = document.getElementById('auth-error');
                errorElem.querySelector('p').textContent =
                    `Too many failed login attempts. Please try again in 15 minutes.`;
                errorElem.classList.remove('d-none');

                return;
            }

            // Normal error handling...
        }
    }
```

8. Secure Logout Process

Improve the logout process to ensure all tokens are properly invalidated.

Implementation:

javascript

```
// Modify the logout function in auth.js
async function logout() {
    // Prevent multiple calls
    if (logoutInProgress) {
        console.log("Logout already in progress, ignoring duplicate call");
        return;
    }

    logoutInProgress = true;
    console.log("Attempting to logout user...");

    try {
        // 1. Sign out from Supabase (invalidate server-side tokens)
        await supabase.auth.signOut({ scope: 'global' });

        // 2. Clear all authentication data from storage
        localStorage.removeItem('supabase.auth.token');
        localStorage.removeItem('supabase.auth.expires_at');
        sessionStorage.removeItem('supabase.auth.token');

        // 3. Remove token from all cookie storage
        document.cookie.split(';').forEach(cookie => {
            const [name] = cookie.trim().split('=');
            if (name.includes('supabase') || name.includes('sb-')) {
                document.cookie = `${name}=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;`;
            }
        });

        // 4. Clear global variables
        currentUser = null;
        userProfile = null;

        // 5. Update UI
        showLoggedOutState();

        // 6. Notify service worker to clear any cached authenticated responses
        if (navigator.serviceWorker && navigator.serviceWorker.controller) {
            navigator.serviceWorker.controller.postMessage({
                type: 'LOGOUT',
                timestamp: Date.now()
            });
        }

        // Success message
        showNotification('Success', 'You have been securely logged out.', 'success');
    } catch (error) {
```

```
        console.error("Error during logout:", error);

        // Try a fallback approach
        localStorage.clear();
        sessionStorage.clear();
        window.location.reload();
    } finally {
        logoutInProgress = false;
    }
}
```

9. Secure WebSocket Handling

The Supabase realtime connection needs proper handling to prevent security issues.

Implementation:

javascript

```
// Add to app.js
function setupSecureRealtimeChannel() {
    // Only set up realtime subscriptions after authentication
    document.addEventListener('login-state-changed', async function(event) {
        if (!event.detail || !event.detail.loggedIn) {
            // Unsubscribe from all channels when logged out
            supabase.removeAllChannels();
            return;
        }

        try {
            // Subscribe to profile changes - limit to just the user's own profile
            const userId = getUserId();

            const profileChannel = supabase
                .channel('profile-changes')
                .on(
                    'postgres_changes',
                    {
                        event: '*',
                        schema: 'public',
                        table: 'profiles',
                        filter: `id=eq.${userId}`
                    },
                    (payload) => {
                        console.log('Profile changed:', payload);
                        // Update cached profile data
                        if (userProfile && userProfile.id === payload.new.id) {
                            userProfile = payload.new;
                        }
                    }
                )
                .subscribe();

            // Subscribe to prayer updates - all users can see updates
            const updatesChannel = supabase
                .channel('updates-changes')
                .on(
                    'postgres_changes',
                    {
                        event: '*',
                        schema: 'public',
                        table: 'prayer_updates'
                    },
                    (payload) => {
                        console.log('Update changed:', payload);
```

```
                        // Refresh data if on the updates view
                        if (document.getElementById('updates-view') &&
                            !document.getElementById('updates-view').classList.contains('d-none
                            loadPrayerUpdates();
                        }
                    }
                )
                .subscribe();

            // Store channel references for later cleanup
            window.activeChannels = {
                profile: profileChannel,
                updates: updatesChannel
            };
        } catch (error) {
            console.error('Error setting up realtime channels:', error);
        }
    });
}

// Clean up channels on logout
function cleanupRealtimeChannels() {
    if (window.activeChannels) {
        for (const key in window.activeChannels) {
            if (window.activeChannels[key]) {
                supabase.removeChannel(window.activeChannels[key]);
            }
        }
        window.activeChannels = {};
    }
}
```

10. Implement Secure File Uploads for Profile Images

The profile image upload functionality should include validation and scanning.

Implementation:

javascript

```
// Add to profile.js
async function handleProfileImageUpload(file) {
    // 1. Validate the file
    if (!file) {
        throw new Error('No file selected');
    }

    // Check file type
    const allowedTypes = ['image/jpeg', 'image/png', 'image/gif'];
    if (!allowedTypes.includes(file.type)) {
        throw new Error('Please select a valid image file (JPEG, PNG, or GIF)');
    }

    // Check file size (max 2MB)
    const maxSize = 2 * 1024 * 1024; // 2MB
    if (file.size > maxSize) {
        throw new Error('Image size must be less than 2MB');
    }

    // 2. Resize and compress the image
    try {
        const resizedFile = await resizeImage(file, 800, 800);

        // 3. Create a unique filename with user ID
        const userId = getUserId();
        const timestamp = Date.now();
        const fileExt = file.name.split('.').pop();
        const filename = `profile_${userId}_${timestamp}.${fileExt}`;

        // 4. Upload to Supabase Storage
        const { data, error } = await supabase.storage
            .from('profiles')
            .upload(filename, resizedFile, {
                cacheControl: '3600',
                upsert: true
            });

        if (error) throw error;

        // 5. Get the public URL
        const { publicURL, error: urlError } = supabase.storage
            .from('profiles')
            .getPublicUrl(filename);

        if (urlError) throw urlError;
```

```
        return publicURL;
    } catch (error) {
        console.error('Error uploading profile image:', error);
        throw error;
    }
}

// Image resize helper
function resizeImage(file, maxWidth, maxHeight) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.src = URL.createObjectURL(file);

        img.onload = () => {
            let width = img.width;
            let height = img.height;

            // Calculate new dimensions
            if (width > maxWidth) {
                height = Math.round(height * (maxWidth / width));
                width = maxWidth;
            }

            if (height > maxHeight) {
                width = Math.round(width * (maxHeight / height));
                height = maxHeight;
            }

            // Create canvas for resizing
            const canvas = document.createElement('canvas');
            canvas.width = width;
            canvas.height = height;

            // Draw resized image
            const ctx = canvas.getContext('2d');
            ctx.drawImage(img, 0, 0, width, height);

            // Convert to Blob
            canvas.toBlob(
                (blob) => resolve(blob),
                file.type,
                0.9 // Quality
            );

            // Clean up
            URL.revokeObjectURL(img.src);
        };
```

```
        img.onerror = () => {
            URL.revokeObjectURL(img.src);
            reject(new Error('Failed to load image'));
        };
    });
}
```

By implementing these security enhancements, you'll significantly improve the security posture of your PECH Prayer Diary application, protecting both your users' data and the integrity of your system.