Security Implementation Guide

Jupiter Swap DApp

Security Guide

Comprehensive Security Implementation

Wallet Security: Multi-provider Support
Transaction Security: Validation & Simulation
Input Sanitization: XSS & Injection Prevention
API Security: Rate Limiting & Authentication
DeFi Security: MEV & Slippage Protection
Error Handling: Secure Error Management
Monitoring: Real-time Security Alerts
Compliance: Security Best Practices

Security Highlights

Multi-layer Wallet Security
Transaction Validation & Simulation
Comprehensive Input Sanitization
Advanced Error Handling
MEV Protection Strategies
Rate Limiting & DDoS Protection
Real-time Security Monitoring
OWASP Compliance

Author: Kamel (@treizeb__)
Company: DeAura.io
Updated: July 14, 2025

Contents

1 Wallet Security Implementation

1.1 Multi-Provider Wallet Integration

```typescript
/**
 * Secure Wallet Provider - Multi-provider Support
 * Implements comprehensive wallet security patterns
 */
export class SecureWalletProvider extends Component<WalletProviderProps> {
  private readonly logger: Logger;
  private readonly securityMonitor: SecurityMonitor;
  private connectionAttempts = new Map<string, number>();
  private readonly maxConnectionAttempts = 3;
  private readonly connectionCooldown = 30000; // 30 seconds

  constructor(props: WalletProviderProps) {
    super(props);
    this.logger = new Logger('SecureWalletProvider');
    this.securityMonitor = new SecurityMonitor();
  }

  /**
   * Secure wallet connection with validation
   * @param walletName - Name of wallet to connect
   * @returns Promise<void>
   */
  async connectWallet(walletName: string): Promise<void> {
    try {
      // Check connection attempts
      this.validateConnectionAttempts(walletName);

      // Validate wallet availability
      await this.validateWalletAvailability(walletName);

      // Perform secure connection
      const wallet = await this.performSecureConnection(walletName);

      // Validate connected wallet
      await this.validateConnectedWallet(wallet);

      // Setup security monitoring
      this.setupWalletMonitoring(wallet);

      this.logger.info('Wallet connected securely', {
        walletName,
        publicKey: wallet.publicKey?.toString()
      });

    } catch (error) {
      this.handleConnectionError(walletName, error);
      throw error;
    }
  }

  /**
   * Validate connection attempts to prevent brute force
   */
  private validateConnectionAttempts(walletName: string): void {
    const attempts = this.connectionAttempts.get(walletName) || 0;

    if (attempts >= this.maxConnectionAttempts) {
      const error = new SecurityError(
        'Too many connection attempts. Please wait before trying again.',
```

```
        'RATE_LIMITED'
      );

      this.securityMonitor.reportSecurityEvent({
        type: 'WALLET_CONNECTION_RATE_LIMITED',
        walletName,
        attempts,
        timestamp: Date.now(),
      });

      throw error;
    }
  }

  /**
   * Validate wallet availability and integrity
   */
  private async validateWalletAvailability(walletName: string): Promise<void> {
    const wallet = this.getWalletByName(walletName);

    if (!wallet) {
      throw new SecurityError(
        `Wallet ${walletName} not found or not supported`,
        'WALLET_NOT_FOUND'
      );
    }

    // Check if wallet is properly installed
    if (!wallet.readyState || wallet.readyState === WalletReadyState.NotDetected) {
      throw new SecurityError(
        `Wallet ${walletName} is not installed or not ready`,
        'WALLET_NOT_READY'
      );
    }

    // Validate wallet integrity (check for known malicious modifications)
    await this.validateWalletIntegrity(wallet);
  }

  /**
   * Perform secure wallet connection
   */
  private async performSecureConnection(walletName: string): Promise<Wallet> {
    const wallet = this.getWalletByName(walletName);

    // Set connection timeout
    const connectionPromise = wallet.connect();
    const timeoutPromise = new Promise((_, reject) =>
      setTimeout(() => reject(new Error('Connection timeout')), 15000)
    );

    try {
      await Promise.race([connectionPromise, timeoutPromise]);
      return wallet;
    } catch (error) {
      // Increment connection attempts
      const attempts = (this.connectionAttempts.get(walletName) || 0) + 1;
      this.connectionAttempts.set(walletName, attempts);

      // Set cooldown timer
      setTimeout(() => {
        this.connectionAttempts.delete(walletName);
      }, this.connectionCooldown);
```

```
123
124       throw error;
125     }
126   }
127
128   /**
129    * Validate connected wallet state
130    */
131   private async validateConnectedWallet(wallet: Wallet): Promise<void> {
132     if (!wallet.connected) {
133       throw new SecurityError('Wallet connection failed', 'CONNECTION_FAILED');
134     }
135
136     if (!wallet.publicKey) {
137       throw new SecurityError('Wallet public key not available', 'NO_PUBLIC_KEY');
138     }
139
140     // Validate public key format
141     try {
142       new PublicKey(wallet.publicKey.toString());
143     } catch {
144       throw new SecurityError('Invalid public key format', 'INVALID_PUBLIC_KEY');
145     }
146
147     // Check for known malicious addresses
148     await this.validatePublicKeyReputation(wallet.publicKey);
149   }
150
151   /**
152    * Setup wallet monitoring for security events
153    */
154   private setupWalletMonitoring(wallet: Wallet): void {
155     // Monitor for unexpected disconnections
156     wallet.on('disconnect', () => {
157       this.securityMonitor.reportSecurityEvent({
158         type: 'WALLET_UNEXPECTED_DISCONNECT',
159         publicKey: wallet.publicKey?.toString(),
160         timestamp: Date.now(),
161       });
162     });
163
164     // Monitor for account changes
165     wallet.on('accountChanged', (publicKey) => {
166       this.securityMonitor.reportSecurityEvent({
167         type: 'WALLET_ACCOUNT_CHANGED',
168         oldPublicKey: wallet.publicKey?.toString(),
169         newPublicKey: publicKey?.toString(),
170         timestamp: Date.now(),
171       });
172     });
173   }
174
175   /**
176    * Validate wallet integrity against known threats
177    */
178   private async validateWalletIntegrity(wallet: Wallet): Promise<void> {
179     // Check wallet version and known vulnerabilities
180     const walletInfo = await this.getWalletInfo(wallet);
181
182     if (walletInfo.hasKnownVulnerabilities) {
183       this.logger.warn('Wallet has known vulnerabilities', {
184         walletName: wallet.adapter.name,
185         version: walletInfo.version,
```

```
        vulnerabilities: walletInfo.vulnerabilities,
      });

      // Don't block but warn user
      this.securityMonitor.reportSecurityEvent({
        type: 'WALLET_VULNERABILITY_DETECTED',
        walletName: wallet.adapter.name,
        vulnerabilities: walletInfo.vulnerabilities,
        timestamp: Date.now(),
      });
    }
  }

  /**
   * Validate public key reputation
   */
  private async validatePublicKeyReputation(publicKey: PublicKey): Promise<void> {
    const reputation = await this.checkPublicKeyReputation(publicKey);

    if (reputation.isBlacklisted) {
      throw new SecurityError(
        'This wallet address has been flagged for suspicious activity',
        'BLACKLISTED_ADDRESS'
      );
    }

    if (reputation.riskScore > 0.8) {
      this.logger.warn('High-risk wallet detected', {
        publicKey: publicKey.toString(),
        riskScore: reputation.riskScore,
        reasons: reputation.riskFactors,
      });

      this.securityMonitor.reportSecurityEvent({
        type: 'HIGH_RISK_WALLET_DETECTED',
        publicKey: publicKey.toString(),
        riskScore: reputation.riskScore,
        riskFactors: reputation.riskFactors,
        timestamp: Date.now(),
      });
    }
  }

  /**
   * Handle connection errors securely
   */
  private handleConnectionError(walletName: string, error: any): void {
    // Log error without exposing sensitive information
    this.logger.error('Wallet connection failed', {
      walletName,
      errorType: error.constructor.name,
      errorCode: error.code,
      // Don't log full error message as it might contain sensitive info
    });

    // Report security event
    this.securityMonitor.reportSecurityEvent({
      type: 'WALLET_CONNECTION_FAILED',
      walletName,
      errorType: error.constructor.name,
      timestamp: Date.now(),
    });
  }
```

```
249
250   render() {
251     return (
252       <WalletProvider
253         wallets={this.getSecureWallets()}
254         onError={this.handleWalletError}
255         autoConnect={false} // Never auto-connect for security
256       >
257         <WalletModalProvider>
258           {this.props.children}
259         </WalletModalProvider>
260       </WalletProvider>
261     );
262   }
263
264   /**
265    * Get list of secure, validated wallets
266    */
267   private getSecureWallets(): Wallet[] {
268     const supportedWallets = [
269       new PhantomWalletAdapter(),
270       new SolflareWalletAdapter(),
271       new BackpackWalletAdapter(),
272       new GlowWalletAdapter(),
273       new LedgerWalletAdapter(),
274     ];
275
276     // Filter out wallets with known security issues
277     return supportedWallets.filter(wallet =>
278       this.isWalletSecure(wallet)
279     );
280   }
281
282   /**
283    * Check if wallet meets security requirements
284    */
285   private isWalletSecure(wallet: Wallet): boolean {
286     // Check against known insecure wallets
287     const insecureWallets = ['FakeWallet', 'TestWallet'];
288
289     if (insecureWallets.includes(wallet.adapter.name)) {
290       return false;
291     }
292
293     // Check wallet adapter version
294     if (wallet.adapter.version && this.isVersionVulnerable(wallet.adapter.version)) {
295       return false;
296     }
297
298     return true;
299   }
300 }
```

Listing 1: Secure Wallet Provider Implementation

2 Transaction Security

2.1 Transaction Validation and Simulation

```
1 /**
2  * Transaction Security Service
```

```
 * Implements comprehensive transaction validation and security checks
 */
export class TransactionSecurityService extends BaseService {
  private readonly validator: TransactionValidator;
  private readonly simulator: TransactionSimulator;
  private readonly riskAnalyzer: RiskAnalyzer;
  private readonly securityMonitor: SecurityMonitor;

  constructor(
    logger: Logger,
    errorHandler: ErrorHandler,
    connection: Connection,
    securityMonitor: SecurityMonitor
  ) {
    super(logger, errorHandler);
    this.validator = new TransactionValidator(logger);
    this.simulator = new TransactionSimulator(connection, logger);
    this.riskAnalyzer = new RiskAnalyzer(logger);
    this.securityMonitor = securityMonitor;
  }

  /**
   * Validate and secure transaction before signing
   * @param transaction - Transaction to validate
   * @param userPublicKey - User's public key
   * @returns Promise<SecurityValidationResult>
   */
  async validateTransaction(
    transaction: VersionedTransaction,
    userPublicKey: PublicKey
  ): Promise<SecurityValidationResult> {
    this.logger.info('Starting transaction security validation', {
      userPublicKey: userPublicKey.toString(),
    });

    const validationResult: SecurityValidationResult = {
      isValid: false,
      riskLevel: 'unknown',
      warnings: [],
      errors: [],
      recommendations: [],
      securityScore: 0,
    };

    try {
      // 1. Basic transaction structure validation
      await this.validateTransactionStructure(transaction, validationResult);

      // 2. Validate transaction instructions
      await this.validateInstructions(transaction, userPublicKey, validationResult);

      // 3. Simulate transaction execution
      await this.simulateTransaction(transaction, validationResult);

      // 4. Analyze transaction risk
      await this.analyzeTransactionRisk(transaction, userPublicKey, validationResult)
    ;

      // 5. Check for known attack patterns
      await this.checkAttackPatterns(transaction, validationResult);

      // 6. Validate transaction limits
```

```
64        await this.validateTransactionLimits(transaction, userPublicKey,
      validationResult);
65
66        // 7. Calculate final security score
67        this.calculateSecurityScore(validationResult);
68
69        // 8. Report security metrics
70        this.reportSecurityMetrics(validationResult);
71
72        return validationResult;
73
74      } catch (error) {
75        this.logger.error('Transaction validation failed', error);
76        validationResult.errors.push('Transaction validation failed');
77        validationResult.isValid = false;
78        return validationResult;
79      }
80    }
81
82    /**
83     * Validate basic transaction structure
84     */
85    private async validateTransactionStructure(
86      transaction: VersionedTransaction,
87      result: SecurityValidationResult
88    ): Promise<void> {
89      // Check transaction version
90      if (transaction.version !== 0) {
91        result.warnings.push('Using versioned transaction - ensure compatibility');
92      }
93
94      // Validate message structure
95      const message = transaction.message;
96      if (!message) {
97        result.errors.push('Transaction message is missing');
98        return;
99      }
100
101     // Check account keys
102     if (!message.staticAccountKeys || message.staticAccountKeys.length === 0) {
103       result.errors.push('No account keys found in transaction');
104       return;
105     }
106
107     // Check instructions
108     if (!message.compiledInstructions || message.compiledInstructions.length === 0) {
109       result.errors.push('No instructions found in transaction');
110       return;
111     }
112
113     // Validate instruction count
114     if (message.compiledInstructions.length > 50) {
115       result.warnings.push('Transaction has many instructions - may fail due to
      compute limits');
116     }
117
118     // Check for duplicate account keys
119     const accountKeyStrings = message.staticAccountKeys.map(key => key.toString());
120     const uniqueKeys = new Set(accountKeyStrings);
121     if (uniqueKeys.size !== accountKeyStrings.length) {
122       result.warnings.push('Transaction contains duplicate account keys');
123     }
124   }
```

```
125
126   /**
127    * Validate transaction instructions for security
128    */
129   private async validateInstructions(
130     transaction: VersionedTransaction,
131     userPublicKey: PublicKey,
132     result: SecurityValidationResult
133   ): Promise<void> {
134     const message = transaction.message;
135     const instructions = message.compiledInstructions;
136
137     for (let i = 0; i < instructions.length; i++) {
138       const instruction = instructions[i];
139
140       try {
141         // Get program ID
142         const programId = message.staticAccountKeys[instruction.programIdIndex];
143
144         // Validate against known programs
145         const programValidation = await this.validateProgram(programId);
146         if (!programValidation.isValid) {
147           result.errors.push(`Instruction ${i}: ${programValidation.reason}`);
148           continue;
149         }
150
151         if (programValidation.riskLevel === 'high') {
152           result.warnings.push(`Instruction ${i}: High-risk program detected`);
153         }
154
155         // Validate instruction data
156         await this.validateInstructionData(instruction, programId, result, i);
157
158         // Check for privilege escalation
159         await this.checkPrivilegeEscalation(instruction, userPublicKey, message,
    result, i);
160
161       } catch (error) {
162         result.warnings.push(`Instruction ${i}: Validation failed - ${error.message
    }`);
163       }
164     }
165   }
166
167   /**
168    * Simulate transaction to check for failures
169    */
170   private async simulateTransaction(
171     transaction: VersionedTransaction,
172     result: SecurityValidationResult
173   ): Promise<void> {
174     try {
175       const simulationResult = await this.simulator.simulateVersionedTransaction(
    transaction);
176
177       if (simulationResult.value.err) {
178         result.errors.push(`Transaction simulation failed: ${JSON.stringify(
    simulationResult.value.err)}`);
179         return;
180       }
181
182       // Check compute units consumed
183       const computeUnits = simulationResult.value.unitsConsumed || 0;
```

```
      if (computeUnits > 1000000) { // 1M compute units
        result.warnings.push('Transaction consumes high compute units - may be expensive');
      }

      // Check logs for warnings
      const logs = simulationResult.value.logs || [];
      const warningLogs = logs.filter(log =>
        log.toLowerCase().includes('warning') ||
        log.toLowerCase().includes('error')
      );

      if (warningLogs.length > 0) {
        result.warnings.push('Transaction simulation produced warnings');
      }

      // Analyze balance changes
      await this.analyzeBalanceChanges(simulationResult, result);

    } catch (error) {
      result.warnings.push(`Transaction simulation failed: ${error.message}`);
    }
  }

  /**
   * Analyze transaction risk factors
   */
  private async analyzeTransactionRisk(
    transaction: VersionedTransaction,
    userPublicKey: PublicKey,
    result: SecurityValidationResult
  ): Promise<void> {
    const riskFactors = await this.riskAnalyzer.analyzeTransaction(
      transaction,
      userPublicKey
    );

    result.riskLevel = riskFactors.overallRisk;

    // Add specific risk warnings
    if (riskFactors.hasUnknownPrograms) {
      result.warnings.push('Transaction interacts with unknown programs');
    }

    if (riskFactors.hasHighValueTransfer) {
      result.warnings.push('Transaction involves high-value transfers');
    }

    if (riskFactors.hasComplexInstructions) {
      result.warnings.push('Transaction has complex instruction patterns');
    }

    if (riskFactors.interactsWithSuspiciousAccounts) {
      result.errors.push('Transaction interacts with flagged accounts');
    }

    // Add recommendations based on risk
    if (riskFactors.overallRisk === 'high') {
      result.recommendations.push('Consider reviewing transaction details carefully');
      result.recommendations.push('Verify all recipient addresses');
    }
  }
```

```
245
246   /**
247    * Check for known attack patterns
248    */
249   private async checkAttackPatterns(
250     transaction: VersionedTransaction,
251     result: SecurityValidationResult
252   ): Promise<void> {
253     const patterns = [
254       this.checkDrainAttack(transaction),
255       this.checkPhishingPattern(transaction),
256       this.checkMEVAttack(transaction),
257       this.checkReentrancyPattern(transaction),
258     ];
259
260     const detectedPatterns = await Promise.all(patterns);
261
262     for (const pattern of detectedPatterns) {
263       if (pattern.detected) {
264         if (pattern.severity === 'critical') {
265           result.errors.push(`Critical security threat detected: ${pattern.
    description}`);
266         } else {
267           result.warnings.push(`Potential security issue: ${pattern.description}`);
268         }
269       }
270     }
271   }
272
273   /**
274    * Validate transaction limits
275    */
276   private async validateTransactionLimits(
277     transaction: VersionedTransaction,
278     userPublicKey: PublicKey,
279     result: SecurityValidationResult
280   ): Promise<void> {
281     // Check transaction size
282     const serializedSize = transaction.serialize().length;
283     if (serializedSize > 1232) { // Solana transaction size limit
284       result.errors.push('Transaction exceeds maximum size limit');
285     }
286
287     // Check daily transaction limits (if implemented)
288     const dailyLimits = await this.checkDailyLimits(userPublicKey);
289     if (dailyLimits.exceeded) {
290       result.warnings.push('Daily transaction limits may be exceeded');
291     }
292
293     // Check for rapid transaction patterns
294     const rapidPattern = await this.checkRapidTransactions(userPublicKey);
295     if (rapidPattern.detected) {
296       result.warnings.push('Rapid transaction pattern detected - possible automation
    ');
297     }
298   }
299
300   /**
301    * Calculate overall security score
302    */
303   private calculateSecurityScore(result: SecurityValidationResult): void {
304     let score = 100;
305
```

```
    // Deduct points for errors and warnings
    score -= result.errors.length * 25;
    score -= result.warnings.length * 10;

    // Adjust based on risk level
    switch (result.riskLevel) {
      case 'critical':
        score -= 50;
        break;
      case 'high':
        score -= 30;
        break;
      case 'medium':
        score -= 15;
        break;
      case 'low':
        score -= 5;
        break;
    }

    result.securityScore = Math.max(0, score);
    result.isValid = result.errors.length === 0 && score >= 50;
  }

  /**
   * Report security metrics
   */
  private reportSecurityMetrics(result: SecurityValidationResult): void {
    this.securityMonitor.reportSecurityEvent({
      type: 'TRANSACTION_VALIDATION_COMPLETED',
      isValid: result.isValid,
      riskLevel: result.riskLevel,
      securityScore: result.securityScore,
      errorCount: result.errors.length,
      warningCount: result.warnings.length,
      timestamp: Date.now(),
    });
  }

  /**
   * Validate program against known programs
   */
  private async validateProgram(programId: PublicKey): Promise<ProgramValidation> {
    const programIdString = programId.toString();

    // Known safe programs
    const safePrograms = new Set([
      '11111111111111111111111111111111', // System Program
      'TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA', // Token Program
      'ATokenGPvbdGVxr1b2hvZbsiqW5xWH25efTNsLJA8knL', // Associated Token Program
      'JUP6LkbZbjS1jKKwapdHNy74zcZ3tLUZoi5QNyVTaV4', // Jupiter V6
      'JUP4Fb2cqiRUcaTHdrPC8h2gNsA2ETXiPDD33WcGuJB', // Jupiter V4
    ]);

    if (safePrograms.has(programIdString)) {
      return {
        isValid: true,
        riskLevel: 'low',
        reason: 'Known safe program',
      };
    }

    // Known risky programs (example)
```

```
    const riskyPrograms = new Set([
      // Add known risky program IDs here
    ]);

    if (riskyPrograms.has(programIdString)) {
      return {
        isValid: false,
        riskLevel: 'high',
        reason: 'Known risky program',
      };
    }

    // Unknown program - medium risk
    return {
      isValid: true,
      riskLevel: 'medium',
      reason: 'Unknown program - proceed with caution',
    };
  }

  /**
   * Check for drain attack patterns
   */
  private async checkDrainAttack(transaction: VersionedTransaction): Promise<
    AttackPattern> {
    // Look for patterns that drain all tokens from an account
    const message = transaction.message;
    const instructions = message.compiledInstructions;

    let suspiciousTransferCount = 0;
    let totalValueTransferred = 0;

    for (const instruction of instructions) {
      const programId = message.staticAccountKeys[instruction.programIdIndex];

      // Check for token transfer instructions
      if (programId.toString() === 'TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA') {
        // Parse token transfer instruction
        const transferData = this.parseTokenTransferInstruction(instruction);
        if (transferData) {
          suspiciousTransferCount++;
          totalValueTransferred += transferData.amount;
        }
      }
    }

    const detected = suspiciousTransferCount > 10 || totalValueTransferred > 1000000;
     // 1M tokens

    return {
      detected,
      severity: detected ? 'critical' : 'low',
      description: detected ? 'Potential drain attack detected' : 'No drain attack
    pattern',
      confidence: detected ? 0.8 : 0.1,
    };
  }

  /**
   * Parse token transfer instruction data
   */
  private parseTokenTransferInstruction(instruction: any): { amount: number } | null
    {
```

```
    try {
      // This is a simplified parser - in reality, you'd use proper instruction
    parsing
      const data = instruction.data;
      if (data && data.length >= 9) {
        // Token transfer instruction has specific format
        const amount = data.readBigUInt64LE(1);
        return { amount: Number(amount) };
      }
    } catch (error) {
      // Ignore parsing errors
    }
    return null;
  }
}
```

Listing 2: Comprehensive Transaction Security

3 Input Sanitization

3.1 XSS and Injection Prevention

```
/**
 * Input Sanitization Service
 * Prevents XSS, injection attacks, and validates all user inputs
 */
export class InputSanitizationService {
  private readonly logger: Logger;
  private readonly validator: Validator;

  constructor(logger: Logger) {
    this.logger = logger;
    this.validator = new Validator();
  }

  /**
   * Sanitize and validate token amount input
   * @param input - Raw input string
   * @returns SanitizedInput<number>
   */
  sanitizeTokenAmount(input: string): SanitizedInput<number> {
    const result: SanitizedInput<number> = {
      isValid: false,
      sanitizedValue: 0,
      originalValue: input,
      errors: [],
      warnings: [],
    };

    try {
      // Remove any HTML tags
      const htmlCleaned = this.stripHtmlTags(input);

      // Remove any script-like content
      const scriptCleaned = this.removeScriptContent(htmlCleaned);

      // Validate numeric format
      const numericValidation = this.validateNumericInput(scriptCleaned);
      if (!numericValidation.isValid) {
        result.errors.push(...numericValidation.errors);
        return result;
```

```
      }

      const numericValue = parseFloat(scriptCleaned);

      // Validate range
      if (numericValue < 0) {
        result.errors.push('Amount cannot be negative');
        return result;
      }

      if (numericValue > Number.MAX_SAFE_INTEGER) {
        result.errors.push('Amount exceeds maximum safe value');
        return result;
      }

      // Check for reasonable decimal places
      const decimalPlaces = this.countDecimalPlaces(scriptCleaned);
      if (decimalPlaces > 18) {
        result.warnings.push('Amount has many decimal places - may cause precision
    issues');
      }

      result.isValid = true;
      result.sanitizedValue = numericValue;

    } catch (error) {
      result.errors.push('Invalid amount format');
      this.logger.warn('Token amount sanitization failed', { input, error });
    }

    return result;
  }

  /**
   * Sanitize and validate public key input
   * @param input - Raw input string
   * @returns SanitizedInput<PublicKey>
   */
  sanitizePublicKey(input: string): SanitizedInput<PublicKey> {
    const result: SanitizedInput<PublicKey> = {
      isValid: false,
      sanitizedValue: null,
      originalValue: input,
      errors: [],
      warnings: [],
    };

    try {
      // Remove whitespace and control characters
      const cleaned = input.trim().replace(/[\x00-\x1F\x7F]/g, '');

      // Remove HTML tags
      const htmlCleaned = this.stripHtmlTags(cleaned);

      // Validate base58 format
      if (!this.isValidBase58(htmlCleaned)) {
        result.errors.push('Invalid public key format');
        return result;
      }

      // Validate length
      if (htmlCleaned.length !== 44) {
        result.errors.push('Public key must be 44 characters long');
```

```
        return result;
      }

      // Try to create PublicKey object
      const publicKey = new PublicKey(htmlCleaned);

      // Additional validation
      if (publicKey.toString() !== htmlCleaned) {
        result.errors.push('Public key validation failed');
        return result;
      }

      // Check against known invalid keys
      if (this.isKnownInvalidKey(publicKey)) {
        result.errors.push('Invalid or blacklisted public key');
        return result;
      }

      result.isValid = true;
      result.sanitizedValue = publicKey;

    } catch (error) {
      result.errors.push('Invalid public key format');
      this.logger.warn('Public key sanitization failed', { input, error });
    }

    return result;
  }

  /**
   * Sanitize search query input
   * @param input - Raw search query
   * @returns SanitizedInput<string>
   */
  sanitizeSearchQuery(input: string): SanitizedInput<string> {
    const result: SanitizedInput<string> = {
      isValid: false,
      sanitizedValue: '',
      originalValue: input,
      errors: [],
      warnings: [],
    };

    try {
      // Remove control characters
      let cleaned = input.replace(/[\x00-\x1F\x7F]/g, '');

      // Remove HTML tags
      cleaned = this.stripHtmlTags(cleaned);

      // Remove script content
      cleaned = this.removeScriptContent(cleaned);

      // Remove SQL injection patterns
      cleaned = this.removeSqlInjectionPatterns(cleaned);

      // Limit length
      if (cleaned.length > 100) {
        cleaned = cleaned.substring(0, 100);
        result.warnings.push('Search query truncated to 100 characters');
      }

      // Validate minimum length
```

```
      if (cleaned.trim().length < 1) {
        result.errors.push('Search query cannot be empty');
        return result;
      }

      // Check for suspicious patterns
      if (this.containsSuspiciousPatterns(cleaned)) {
        result.warnings.push('Search query contains suspicious patterns');
      }

      result.isValid = true;
      result.sanitizedValue = cleaned.trim();

    } catch (error) {
      result.errors.push('Search query validation failed');
      this.logger.warn('Search query sanitization failed', { input, error });
    }

    return result;
  }

  /**
   * Sanitize URL input
   * @param input - Raw URL string
   * @returns SanitizedInput<string>
   */
  sanitizeUrl(input: string): SanitizedInput<string> {
    const result: SanitizedInput<string> = {
      isValid: false,
      sanitizedValue: '',
      originalValue: input,
      errors: [],
      warnings: [],
    };

    try {
      // Remove control characters
      let cleaned = input.replace(/[\x00-\x1F\x7F]/g, '');

      // Remove HTML tags
      cleaned = this.stripHtmlTags(cleaned);

      // Validate URL format
      const url = new URL(cleaned);

      // Check protocol
      if (!['http:', 'https:'].includes(url.protocol)) {
        result.errors.push('Only HTTP and HTTPS URLs are allowed');
        return result;
      }

      // Check for suspicious domains
      if (this.isSuspiciousDomain(url.hostname)) {
        result.errors.push('Suspicious or blacklisted domain');
        return result;
      }

      // Check for URL shorteners (potential security risk)
      if (this.isUrlShortener(url.hostname)) {
        result.warnings.push('URL shortener detected - verify destination');
      }

      result.isValid = true;
```

```
      result.sanitizedValue = url.toString();

    } catch (error) {
      result.errors.push('Invalid URL format');
      this.logger.warn('URL sanitization failed', { input, error });
    }

    return result;
  }

  /**
   * Strip HTML tags from input
   */
  private stripHtmlTags(input: string): string {
    return input.replace(/<[^>]*>/g, '');
  }

  /**
   * Remove script content and dangerous patterns
   */
  private removeScriptContent(input: string): string {
    // Remove script tags and content
    let cleaned = input.replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi
    , '');

    // Remove javascript: URLs
    cleaned = cleaned.replace(/javascript:/gi, '');

    // Remove on* event handlers
    cleaned = cleaned.replace(/\bon\w+\s*=/gi, '');

    // Remove data: URLs (can contain scripts)
    cleaned = cleaned.replace(/data:/gi, '');

    return cleaned;
  }

  /**
   * Remove SQL injection patterns
   */
  private removeSqlInjectionPatterns(input: string): string {
    const sqlPatterns = [
      /(\b(SELECT|INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|EXEC|UNION)\b)/gi,
      /(--|\/\*|\*\/)/g,
      /(\b(OR|AND)\s+\d+\s*=\s*\d+)/gi,
      /('|(\\')|(;))/g,
    ];

    let cleaned = input;
    for (const pattern of sqlPatterns) {
      cleaned = cleaned.replace(pattern, '');
    }

    return cleaned;
  }

  /**
   * Validate numeric input format
   */
  private validateNumericInput(input: string): ValidationResult {
    const result: ValidationResult = {
      isValid: true,
      errors: [],
```

```
    };

    // Check for valid numeric format
    if (!/^-?\d*\.?\d+([eE][+-]?\d+)?$/.test(input)) {
      result.isValid = false;
      result.errors.push('Invalid numeric format');
    }

    // Check for scientific notation abuse
    if (/[eE]/.test(input)) {
      const parts = input.split(/[eE]/);
      if (parts.length === 2) {
        const exponent = parseInt(parts[1]);
        if (Math.abs(exponent) > 20) {
          result.isValid = false;
          result.errors.push('Exponent too large');
        }
      }
    }

    return result;
  }

  /**
   * Count decimal places in numeric string
   */
  private countDecimalPlaces(input: string): number {
    const decimalIndex = input.indexOf('.');
    if (decimalIndex === -1) return 0;

    const afterDecimal = input.substring(decimalIndex + 1);
    return afterDecimal.replace(/[eE].*$/, '').length;
  }

  /**
   * Validate base58 format
   */
  private isValidBase58(input: string): boolean {
    const base58Regex = /^[1-9A-HJ-NP-Za-km-z]+$/;
    return base58Regex.test(input);
  }

  /**
   * Check if public key is known to be invalid
   */
  private isKnownInvalidKey(publicKey: PublicKey): boolean {
    const invalidKeys = new Set([
      '11111111111111111111111111111111', // System Program (not a wallet)
      '00000000000000000000000000000000', // Invalid key
    ]);

    return invalidKeys.has(publicKey.toString());
  }

  /**
   * Check for suspicious patterns in text
   */
  private containsSuspiciousPatterns(input: string): boolean {
    const suspiciousPatterns = [
      /<script/i,
      /javascript:/i,
      /vbscript:/i,
      /onload=/i,
```

```
      /onerror=/i,
      /eval\(/i,
      /document\.cookie/i,
      /window\.location/i,
    ];

    return suspiciousPatterns.some(pattern => pattern.test(input));
  }

  /**
   * Check if domain is suspicious
   */
  private isSuspiciousDomain(hostname: string): boolean {
    const suspiciousDomains = new Set([
      'localhost',
      '127.0.0.1',
      '0.0.0.0',
      // Add known malicious domains
    ]);

    // Check for suspicious TLDs
    const suspiciousTlds = ['.tk', '.ml', '.ga', '.cf'];
    const hasSuspiciousTld = suspiciousTlds.some(tld => hostname.endsWith(tld));

    return suspiciousDomains.has(hostname) || hasSuspiciousTld;
  }

  /**
   * Check if domain is a URL shortener
   */
  private isUrlShortener(hostname: string): boolean {
    const shorteners = new Set([
      'bit.ly',
      'tinyurl.com',
      't.co',
      'goo.gl',
      'ow.ly',
      'short.link',
    ]);

    return shorteners.has(hostname);
  }
}
```

Listing 3: Comprehensive Input Sanitization

4 Security Monitoring

4.1 Real-time Security Event Monitoring

```
/**
 * Security Monitor - Real-time Security Event Tracking
 * Monitors and responds to security events in real-time
 */
export class SecurityMonitor {
  private readonly logger: Logger;
  private readonly eventStore: SecurityEventStore;
  private readonly alertManager: AlertManager;
  private readonly metrics: SecurityMetrics;
  private readonly riskThresholds: RiskThresholds;

```

```
  constructor(
    logger: Logger,
    eventStore: SecurityEventStore,
    alertManager: AlertManager
  ) {
    this.logger = logger;
    this.eventStore = eventStore;
    this.alertManager = alertManager;
    this.metrics = new SecurityMetrics();
    this.riskThresholds = this.loadRiskThresholds();
  }

  /**
   * Report a security event
   * @param event - Security event to report
   */
  async reportSecurityEvent(event: SecurityEvent): Promise<void> {
    try {
      // Enrich event with additional context
      const enrichedEvent = await this.enrichSecurityEvent(event);

      // Store event
      await this.eventStore.store(enrichedEvent);

      // Update metrics
      this.metrics.recordEvent(enrichedEvent);

      // Check for immediate threats
      const threatLevel = await this.assessThreatLevel(enrichedEvent);

      if (threatLevel >= this.riskThresholds.immediateResponse) {
        await this.handleImmediateThreat(enrichedEvent);
      }

      // Check for patterns
      await this.checkSecurityPatterns(enrichedEvent);

      this.logger.info('Security event reported', {
        type: event.type,
        threatLevel,
        timestamp: event.timestamp,
      });

    } catch (error) {
      this.logger.error('Failed to report security event', error);
    }
  }

  /**
   * Get security dashboard data
   * @param timeRange - Time range for data
   * @returns Promise<SecurityDashboard>
   */
  async getSecurityDashboard(timeRange: TimeRange): Promise<SecurityDashboard> {
    const events = await this.eventStore.getEvents(timeRange);

    return {
      totalEvents: events.length,
      eventsByType: this.groupEventsByType(events),
      threatLevelDistribution: this.analyzeThreatLevels(events),
      topRisks: await this.identifyTopRisks(events),
      securityScore: this.calculateSecurityScore(events),
      recommendations: await this.generateRecommendations(events),
```

```
      alerts: await this.getActiveAlerts(),
      trends: this.analyzeTrends(events),
    };
  }

  /**
   * Check for security anomalies
   * @param userPublicKey - User to check
   * @returns Promise<AnomalyReport>
   */
  async checkUserAnomalies(userPublicKey: PublicKey): Promise<AnomalyReport> {
    const userEvents = await this.eventStore.getUserEvents(
      userPublicKey.toString(),
      { hours: 24 } // Last 24 hours
    );

    const anomalies: SecurityAnomaly[] = [];

    // Check for unusual activity patterns
    const activityAnomaly = this.detectActivityAnomaly(userEvents);
    if (activityAnomaly) {
      anomalies.push(activityAnomaly);
    }

    // Check for suspicious transaction patterns
    const transactionAnomaly = this.detectTransactionAnomaly(userEvents);
    if (transactionAnomaly) {
      anomalies.push(transactionAnomaly);
    }

    // Check for failed authentication attempts
    const authAnomaly = this.detectAuthenticationAnomaly(userEvents);
    if (authAnomaly) {
      anomalies.push(authAnomaly);
    }

    return {
      userPublicKey: userPublicKey.toString(),
      anomaliesDetected: anomalies.length,
      anomalies,
      riskScore: this.calculateUserRiskScore(anomalies),
      recommendations: this.generateUserRecommendations(anomalies),
    };
  }

  /**
   * Enrich security event with additional context
   */
  private async enrichSecurityEvent(event: SecurityEvent): Promise<
    EnrichedSecurityEvent> {
    const enriched: EnrichedSecurityEvent = {
      ...event,
      id: this.generateEventId(),
      ipAddress: await this.getClientIpAddress(),
      userAgent: await this.getUserAgent(),
      geolocation: await this.getGeolocation(),
      sessionId: await this.getSessionId(),
      enrichedAt: Date.now(),
    };

    // Add threat intelligence
    if (event.publicKey) {
```

```
      enriched.threatIntelligence = await this.getThreatIntelligence(event.publicKey);
    }

    return enriched;
  }

  /**
   * Assess threat level of security event
   */
  private async assessThreatLevel(event: EnrichedSecurityEvent): Promise<number> {
    let threatLevel = 0;

    // Base threat level by event type
    const baseThreatLevels: Record<string, number> = {
      'WALLET_CONNECTION_FAILED': 2,
      'TRANSACTION_VALIDATION_FAILED': 5,
      'HIGH_RISK_WALLET_DETECTED': 7,
      'BLACKLISTED_ADDRESS': 10,
      'DRAIN_ATTACK_DETECTED': 10,
      'SUSPICIOUS_PATTERN_DETECTED': 6,
      'RATE_LIMITED': 3,
    };

    threatLevel = baseThreatLevels[event.type] || 1;

    // Increase threat level based on frequency
    const recentSimilarEvents = await this.eventStore.getRecentSimilarEvents(
      event,
      { minutes: 10 }
    );

    if (recentSimilarEvents.length > 5) {
      threatLevel += 3;
    }

    // Increase threat level for known bad actors
    if (event.threatIntelligence?.isKnownBadActor) {
      threatLevel += 5;
    }

    return Math.min(threatLevel, 10); // Cap at 10
  }

  /**
   * Handle immediate security threats
   */
  private async handleImmediateThreat(event: EnrichedSecurityEvent): Promise<void> {
    this.logger.warn('Immediate security threat detected', {
      eventType: event.type,
      eventId: event.id,
    });

    // Send immediate alert
    await this.alertManager.sendImmediateAlert({
      severity: 'critical',
      title: 'Immediate Security Threat Detected',
      description: `Security event: ${event.type}`,
      event,
      timestamp: Date.now(),
    });

    // Take protective actions
```

```
    if (event.publicKey) {
      await this.implementProtectiveActions(event.publicKey);
    }
  }

  /**
   * Check for security patterns and correlations
   */
  private async checkSecurityPatterns(event: EnrichedSecurityEvent): Promise<void> {
    // Check for coordinated attacks
    const coordinatedAttack = await this.detectCoordinatedAttack(event);
    if (coordinatedAttack) {
      await this.reportSecurityEvent({
        type: 'COORDINATED_ATTACK_DETECTED',
        description: 'Multiple related security events detected',
        relatedEvents: coordinatedAttack.relatedEvents,
        timestamp: Date.now(),
      });
    }

    // Check for escalating threats
    const escalatingThreat = await this.detectEscalatingThreat(event);
    if (escalatingThreat) {
      await this.reportSecurityEvent({
        type: 'ESCALATING_THREAT_DETECTED',
        description: 'Security threat is escalating',
        escalationPattern: escalatingThreat.pattern,
        timestamp: Date.now(),
      });
    }
  }

  /**
   * Detect activity anomalies
   */
  private detectActivityAnomaly(events: SecurityEvent[]): SecurityAnomaly | null {
    const activityEvents = events.filter(e =>
      ['WALLET_CONNECTION', 'TRANSACTION_SUBMITTED'].includes(e.type)
    );

    // Check for unusual activity volume
    const hourlyActivity = this.groupEventsByHour(activityEvents);
    const avgHourlyActivity = Object.values(hourlyActivity).reduce((a, b) => a + b,
    0) / 24;

    const maxHourlyActivity = Math.max(...Object.values(hourlyActivity));

    if (maxHourlyActivity > avgHourlyActivity * 5) {
      return {
        type: 'UNUSUAL_ACTIVITY_VOLUME',
        severity: 'medium',
        description: 'Unusual spike in activity detected',
        confidence: 0.8,
        details: {
          maxHourlyActivity,
          avgHourlyActivity,
          threshold: avgHourlyActivity * 5,
        },
      };
    }

    return null;
  }
```

```
260
261   /**
262    * Generate security recommendations
263    */
264   private async generateRecommendations(events: SecurityEvent[]): Promise<
        SecurityRecommendation[]> {
265     const recommendations: SecurityRecommendation[] = [];
266
267     // Analyze event patterns
268     const eventTypes = this.groupEventsByType(events);
269
270     // High number of failed connections
271     if (eventTypes['WALLET_CONNECTION_FAILED'] > 10) {
272       recommendations.push({
273         type: 'SECURITY_IMPROVEMENT',
274         priority: 'high',
275         title: 'Review Wallet Connection Security',
276         description: 'High number of failed wallet connections detected',
277         action: 'Implement additional connection validation',
278       });
279     }
280
281     // High-risk transactions
282     if (eventTypes['HIGH_RISK_TRANSACTION'] > 5) {
283       recommendations.push({
284         type: 'USER_EDUCATION',
285         priority: 'medium',
286         title: 'User Education on Transaction Security',
287         description: 'Users are submitting high-risk transactions',
288         action: 'Provide better transaction security guidance',
289       });
290     }
291
292     return recommendations;
293   }
294
295   /**
296    * Calculate overall security score
297    */
298   private calculateSecurityScore(events: SecurityEvent[]): number {
299     let score = 100;
300
301     const eventTypes = this.groupEventsByType(events);
302
303     // Deduct points for security events
304     score -= (eventTypes['SECURITY_VIOLATION'] || 0) * 10;
305     score -= (eventTypes['HIGH_RISK_TRANSACTION'] || 0) * 5;
306     score -= (eventTypes['FAILED_VALIDATION'] || 0) * 3;
307
308     // Bonus points for good security practices
309     score += (eventTypes['SUCCESSFUL_VALIDATION'] || 0) * 0.1;
310
311     return Math.max(0, Math.min(100, score));
312   }
313
314   /**
315    * Load risk thresholds configuration
316    */
317   private loadRiskThresholds(): RiskThresholds {
318     return {
319       immediateResponse: 8,
320       highAlert: 6,
321       mediumAlert: 4,
```

```
322         lowAlert: 2,
323       };
324     }
325 }
```

Listing 4: Security Monitoring System

5 Conclusion

This comprehensive security implementation guide provides detailed patterns and practices for securing the Jupiter Swap DApp against common threats and vulnerabilities.

5.1 Security Implementation Summary

Security Features Implemented:

- **Multi-layer Wallet Security:** Connection validation and monitoring
- **Transaction Security:** Comprehensive validation and simulation
- **Input Sanitization:** XSS and injection prevention
- **Real-time Monitoring:** Security event tracking and alerting
- **Risk Assessment:** Automated threat level calculation
- **Attack Prevention:** Pattern detection and response
- **Compliance:** OWASP security standards
- **Incident Response:** Automated protective actions

Security implementation designed and implemented by Kamel (@treizeb__)
DeAura.io - July 2025