Partner Integration Framework

Overview

Establish strategic partnerships with key Web3 infrastructure providers, security platforms, and development tools to create a comprehensive ecosystem around Audityzer. This framework will enable seamless integrations, expand market reach, and provide enhanced value to users.

Strategic Partnership Categories

1. Infrastructure Partners

- **Chainlink**: Oracle security testing and price feed validation
- **Alchemy**: Enhanced RPC services and developer tools integration
- **Infura**: Reliable blockchain connectivity and analytics
- **QuickNode**: Multi-chain RPC optimization and monitoring

2. Security Platform Partners

- **Immunefi**: Bug bounty platform integration and vulnerability reporting
- **Code4rena**: Audit contest platform and community engagement
- **HackenProof**: Enterprise security services and compliance
- **OpenZeppelin**: Security standards and best practices integration

3. Development Tool Partners

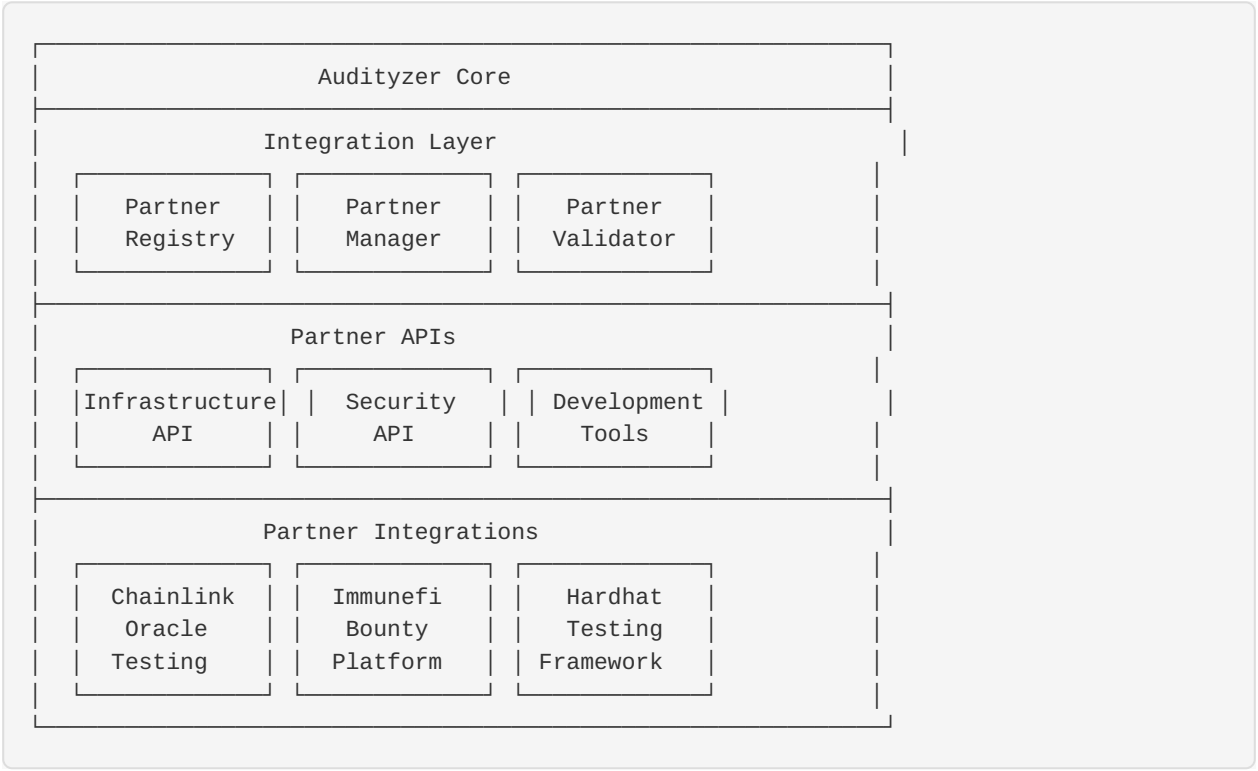
- **Hardhat**: Native testing framework integration
- **Foundry**: Advanced testing and fuzzing capabilities
- **Remix**: Browser-based development environment
- **Tenderly**: Simulation and debugging platform

4. Wallet & AA Partners

- **MetaMask Snaps**: Browser extension integration
- **Pimlico**: Account Abstraction infrastructure
- **Stackup**: ERC-4337 bundler services
- **Safe**: Multi-signature wallet security testing

Integration Architecture

Partner Integration Framework



Base Partner Integration Interface

```
// src/core/partners/PartnerIntegration.js
class PartnerIntegration {
  constructor(config = {}) {
    this.config = config;
    this.name = 'BasePartnerIntegration';
    this.version = '1.0.0';
    this.category = 'general';
    this.status = 'inactive';
  }

  /**
   * Initialize the partner integration
   * @param {Object} context - Audityzer integration context
   */
  async initialize(context) {
    this.context = context;
    this.logger = context.logger;
    this.utils = context.utils;
    this.status = 'initializing';

    try {
      await this.setup();
      this.status = 'active';
      this.logger.info(`  ${this.name} integration initialized`);
    } catch (error) {
      this.status = 'error';
      this.logger.error(`  ${this.name} integration failed: ${error.message}`);
      throw error;
    }
  }

  /**
   * Setup partner-specific configuration
   */
  async setup() {
    // Override in subclass
  }

  /**
   * Test the integration connection
   * @returns {Promise<Object>} Health check result
   */
  async healthCheck() {
    return {
      status: this.status,
      timestamp: new Date().toISOString(),
      details: {}
    };
  }

  /**
   * Get integration metadata
   * @returns {Object} Integration information
   */
  getMetadata() {
    return {
      name: this.name,
      version: this.version,
```

```
      category: this.category,
      status: this.status,
      description: this.description,
      capabilities: this.capabilities || []
    };
  }

  /**
   * Cleanup resources
   */
  async cleanup() {
    this.status = 'inactive';
  }
}

module.exports = PartnerIntegration;
```

Specific Partner Integrations

1. Chainlink Oracle Integration

```
// src/core/partners/chainlink/ChainlinkIntegration.js
const { PartnerIntegration } = require('../PartnerIntegration');
const { ethers } = require('ethers');

class ChainlinkIntegration extends PartnerIntegration {
  constructor(config) {
    super(config);
    this.name = 'ChainlinkIntegration';
    this.version = '1.0.0';
    this.category = 'infrastructure';
    this.description = 'Chainlink oracle security testing and price feed validation';
    this.capabilities = [
      'price-feed-validation',
      'oracle-manipulation-testing',
      'data-freshness-checks',
      'aggregator-security-analysis'
    ];
  }

  async setup() {
    // Initialize Chainlink price feed registry
    this.priceFeeds = await this.loadPriceFeedRegistry();

    // Setup oracle manipulation detection
    this.oracleDetector = new ChainlinkOracleDetector(this.config);

    // Initialize price deviation monitoring
    this.priceMonitor = new PriceDeviationMonitor(this.config);
  }

  /**
   * Validate Chainlink price feeds used in a contract
   * @param {Object} contract - Contract to analyze
   * @returns {Promise<Array>} Validation results
   */
  async validatePriceFeeds(contract) {
    const findings = [];

    try {
      // Extract price feed addresses from contract
      const priceFeedAddresses = await this.extractPriceFeedAddresses(contract);

      for (const address of priceFeedAddresses) {
        const validation = await this.validateSinglePriceFeed(address, contract.chain);
        findings.push(...validation);
      }

    } catch (error) {
      this.logger.error(`Price feed validation failed: ${error.message}`);
    }

    return findings;
  }

  async validateSinglePriceFeed(feedAddress, chain) {
    const findings = [];
    const feedInfo = await this.getPriceFeedInfo(feedAddress, chain);
```

```
    // Check if feed is deprecated
    if (feedInfo.deprecated) {
      findings.push({
        type: 'oracle-deprecated',
        severity: 'high',
        title: 'Deprecated Price Feed',
        description: `Price feed ${feedAddress} is deprecated and may stop updating`,
        recommendation: 'Migrate to the recommended replacement feed',
        metadata: {
          feedAddress,
          replacementFeed: feedInfo.replacement,
          deprecationDate: feedInfo.deprecationDate
        }
      });
    }

    // Check heartbeat and deviation threshold
    if (feedInfo.heartbeat > 86400) { // 24 hours
      findings.push({
        type: 'oracle-stale-data',
        severity: 'medium',
        title: 'Long Heartbeat Interval',
        description: `Price feed has a ${feedInfo.heartbeat / 3600}h heartbeat, data
may be stale`,
        recommendation: 'Implement additional staleness checks in your contract',
        metadata: {
          feedAddress,
          heartbeat: feedInfo.heartbeat,
          lastUpdate: feedInfo.lastUpdate
        }
      });
    }

    // Check for price manipulation vulnerability
    const manipulationRisk = await this.assessManipulationRisk(feedAddress, chain);
    if (manipulationRisk.risk > 0.7) {
      findings.push({
        type: 'oracle-manipulation-risk',
        severity: 'high',
        title: 'High Price Manipulation Risk',
        description: 'Price feed may be vulnerable to manipulation attacks',
        recommendation: 'Use multiple price sources and implement circuit breakers',
        metadata: {
          feedAddress,
          riskScore: manipulationRisk.risk,
          factors: manipulationRisk.factors
        }
      });
    }

    return findings;
  }

  async assessManipulationRisk(feedAddress, chain) {
    const factors = [];
    let riskScore = 0;

    // Check liquidity of underlying markets
    const liquidity = await this.checkUnderlyingLiquidity(feedAddress, chain);
```

```
    if (liquidity < 1000000) { // Less than $1M liquidity
      factors.push('Low underlying market liquidity');
      riskScore += 0.3;
    }

    // Check number of data sources
    const dataSources = await this.getDataSourceCount(feedAddress, chain);
    if (dataSources < 5) {
      factors.push('Few data sources');
      riskScore += 0.2;
    }

    // Check for recent price anomalies
    const anomalies = await this.detectPriceAnomalies(feedAddress, chain);
    if (anomalies.length > 0) {
      factors.push('Recent price anomalies detected');
      riskScore += 0.3;
    }

    return {
      risk: Math.min(riskScore, 1.0),
      factors
    };
  }

  /**
   * Generate oracle security testing suite
   * @param {Object} contract - Contract using oracles
   * @returns {Promise<Object>} Test suite configuration
   */
  async generateOracleTestSuite(contract) {
    const testSuite = {
      name: 'Chainlink Oracle Security Tests',
      tests: []
    };

    const priceFeedAddresses = await this.extractPriceFeedAddresses(contract);

    for (const feedAddress of priceFeedAddresses) {
      // Staleness test
      testSuite.tests.push({
        name: `Staleness Test - ${feedAddress}`,
        type: 'oracle-staleness',
        config: {
          feedAddress,
          maxStaleness: 3600, // 1 hour
          testScenarios: ['normal', 'stale', 'very_stale']
        }
      });

      // Price manipulation test
      testSuite.tests.push({
        name: `Price Manipulation Test - ${feedAddress}`,
        type: 'oracle-manipulation',
        config: {
          feedAddress,
          manipulationScenarios: [
            'sudden_spike',
            'gradual_drift',
```

```
            'flash_crash',
            'circuit_breaker_test'
          ]
        }
      });

      // Aggregator upgrade test
      testSuite.tests.push({
        name: `Aggregator Upgrade Test - ${feedAddress}`,
        type: 'oracle-upgrade',
        config: {
          feedAddress,
          testUpgradeScenarios: true
        }
      });
    }

    return testSuite;
  }
}

module.exports = ChainlinkIntegration;
```

2. Immunefi Bug Bounty Integration

```
// src/core/partners/immunefi/ImmunefiIntegration.js
const { PartnerIntegration } = require('../PartnerIntegration');

class ImmunefiIntegration extends PartnerIntegration {
  constructor(config) {
    super(config);
    this.name = 'ImmunefiIntegration';
    this.version = '1.0.0';
    this.category = 'security';
    this.description = 'Immunefi bug bounty platform integration';
    this.capabilities = [
      'vulnerability-submission',
      'bounty-program-management',
      'researcher-collaboration',
      'payout-automation'
    ];
  }

  async setup() {
    if (!this.config.apiKey) {
      throw new Error('Immunefi API key required');
    }

    this.apiClient = new ImmunefiAPIClient({
      apiKey: this.config.apiKey,
      baseUrl: 'https://api.immunefi.com/v1'
    });

    // Verify API connection
    await this.apiClient.authenticate();
  }

  /**
   * Submit vulnerability findings to Immunefi
   * @param {Array} vulnerabilities - Vulnerability findings
   * @param {Object} options - Submission options
   * @returns {Promise<Object>} Submission result
   */
  async submitVulnerabilities(vulnerabilities, options = {}) {
    const submissions = [];

    for (const vuln of vulnerabilities) {
      // Only submit high/critical vulnerabilities
      if (!['high', 'critical'].includes(vuln.severity)) {
        continue;
      }

      try {
        const submission = await this.submitSingleVulnerability(vuln, options);
        submissions.push(submission);
      } catch (error) {
        this.logger.error(`Failed to submit vulnerability: ${error.message}`);
      }
    }

    return {
      submitted: submissions.length,
      submissions,
```

```
      totalVulnerabilities: vulnerabilities.length
    };
  }

  async submitSingleVulnerability(vulnerability, options) {
    const submissionData = {
      title: vulnerability.title,
      description: this.formatVulnerabilityDescription(vulnerability),
      severity: this.mapSeverityToImmunefi(vulnerability.severity),
      category: this.mapCategoryToImmunefi(vulnerability.type),
      proof_of_concept: vulnerability.proofOfConcept || this.generate-
POC(vulnerability),
      impact: vulnerability.impact || this.generateImpactDescription(vulnerability),
      recommendation: vulnerability.recommendation,
      metadata: {
        detector: vulnerability.metadata?.detector,
        audityzer_version: this.context.version,
        analysis_timestamp: vulnerability.metadata?.timestamp,
        confidence: vulnerability.confidence
      }
    };

    if (options.programId) {
      submissionData.program_id = options.programId;
    }

    const response = await this.apiClient.submitVulnerability(submissionData);

    return {
      id: response.id,
      status: response.status,
      url: response.url,
      estimatedReward: response.estimated_reward,
      vulnerability: vulnerability
    };
  }

  /**
   * Create or update bug bounty program
   * @param {Object} programConfig - Program configuration
   * @returns {Promise<Object>} Program details
   */
  async createBountyProgram(programConfig) {
    const programData = {
      name: programConfig.name,
      description: programConfig.description,
      project_type: programConfig.projectType || 'smart_contract',
      blockchain: programConfig.blockchain || 'ethereum',
      rewards: {
        critical: programConfig.rewards?.critical || 50000,
        high: programConfig.rewards?.high || 25000,
        medium: programConfig.rewards?.medium || 5000,
        low: programConfig.rewards?.low || 1000
      },
      scope: {
        in_scope: programConfig.scope?.inScope || [],
        out_of_scope: programConfig.scope?.outOfScope || []
      },
      rules: programConfig.rules || this.getDefaultRules(),
```

```
      contact: programConfig.contact
    };

    const response = await this.apiClient.createProgram(programData);

    return {
      id: response.id,
      url: response.url,
      status: response.status,
      created_at: response.created_at
    };
  }

  /**
   * Get bounty program analytics
   * @param {string} programId - Program ID
   * @returns {Promise<Object>} Analytics data
   */
  async getProgramAnalytics(programId) {
    const analytics = await this.apiClient.getProgramAnalytics(programId);

    return {
      totalSubmissions: analytics.total_submissions,
      validSubmissions: analytics.valid_submissions,
      totalPayout: analytics.total_payout,
      averageResponseTime: analytics.avg_response_time,
      topResearchers: analytics.top_researchers,
      vulnerabilityBreakdown: analytics.vulnerability_breakdown,
      monthlyTrends: analytics.monthly_trends
    };
  }

  /**
   * Setup automated vulnerability monitoring
   * @param {Object} config - Monitoring configuration
   * @returns {Promise<Object>} Monitor setup result
   */
  async setupVulnerabilityMonitoring(config) {
    const monitor = {
      id: this.generateMonitorId(),
      name: config.name || 'Auditzer Auto-Monitor',
      triggers: {
        severity_threshold: config.severityThreshold || 'high',
        auto_submit: config.autoSubmit || false,
        notification_channels: config.notifications || []
      },
      filters: {
        vulnerability_types: config.vulnerabilityTypes || [],
        confidence_threshold: config.confidenceThreshold || 0.8
      }
    };

    // Register webhook for real-time notifications
    if (config.webhookUrl) {
      await this.apiClient.registerWebhook({
        url: config.webhookUrl,
        events: ['vulnerability_submitted', 'bounty_awarded', 'status_updated']
      });
    }
```

```
    return monitor;
  }

  formatVulnerabilityDescription(vulnerability) {
    return `
## Vulnerability Description

${vulnerability.description}

## Technical Details

**Type:** ${vulnerability.type}
**Severity:** ${vulnerability.severity}
**Confidence:** ${vulnerability.confidence || 'N/A'}

## Location

${vulnerability.location ? `
**File:** ${vulnerability.location.file}
**Line:** ${vulnerability.location.line}
**Function:** ${vulnerability.location.function}
` : 'Location information not available'}

## Detection Method

This vulnerability was detected using Auditzer's automated security analysis engine.

**Detector:** ${vulnerability.metadata?.detector || 'Unknown'}
**Analysis Time:** ${vulnerability.metadata?.timestamp || 'Unknown'}

## Additional Information

${vulnerability.additionalInfo || 'No additional information provided'}
    `.trim();
  }

  generatePOC(vulnerability) {
    // Generate basic proof of concept based on vulnerability type
    const pocTemplates = {
      'reentrancy': `
// Proof of Concept for Reentrancy Attack
contract ReentrancyAttack {
    VulnerableContract target;

    constructor(address _target) {
        target = VulnerableContract(_target);
    }

    function attack() external payable {
        target.deposit{value: msg.value}();
        target.withdraw(msg.value);
    }

    receive() external payable {
        if (address(target).balance >= msg.value) {
            target.withdraw(msg.value);
        }
    }
```

```
}
      `,
      'oracle-manipulation': `
// Proof of Concept for Oracle Manipulation
// 1. Flash loan large amount of tokens
// 2. Manipulate price in DEX
// 3. Trigger vulnerable contract with manipulated price
// 4. Profit from price difference
// 5. Repay flash loan
      `,
      'access-control': `
// Proof of Concept for Access Control Bypass
// Call the vulnerable function directly without proper authorization
contract AccessControlExploit {
    function exploit(address target) external {
        VulnerableContract(target).privilegedFunction();
    }
}
      `
    };

    return pocTemplates[vulnerability.type] || 'Proof of concept to be developed based
on specific vulnerability details.';
  }
}

module.exports = ImmunefiIntegration;
```

3. MetaMask Snaps Integration

```
// src/core/partners/metamask/MetaMaskSnapsIntegration.js
const { PartnerIntegration } = require('../PartnerIntegration');

class MetaMaskSnapsIntegration extends PartnerIntegration {
  constructor(config) {
    super(config);
    this.name = 'MetaMaskSnapsIntegration';
    this.version = '1.0.0';
    this.category = 'wallet';
    this.description = 'MetaMask Snaps integration for real-time security monitoring';
    this.capabilities = [
      'transaction-analysis',
      'real-time-warnings',
      'contract-verification',
      'security-insights'
    ];
  }

  async setup() {
    // Initialize Snap development environment
    this.snapConfig = await this.generateSnapConfiguration();

    // Setup security rule engine
    this.securityEngine = new SnapSecurityEngine(this.config);
  }

  /**
   * Generate MetaMask Snap for security monitoring
   * @param {Object} config - Snap configuration
   * @returns {Promise<Object>} Generated Snap package
   */
  async generateSecuritySnap(config) {
    const snapManifest = {
      version: '1.0.0',
      description: 'Audityzer Security Monitor - Real-time transaction security analys-
is',
      proposedName: 'Audityzer Security Monitor',
      repository: {
        type: 'git',
        url: 'https://github.com/cyfrin/audityzer-snap.git'
      },
      source: {
        shasum: await this.calculateSnapShasum(),
        location: {
          npm: {
            filePath: 'dist/bundle.js',
            iconPath: 'images/icon.svg',
            packageName: '@audityzer/metamask-snap',
            registry: 'https://registry.npmjs.org/'
          }
        }
      },
      initialPermissions: {
        'endowment:rpc': {
          dapps: true,
          snaps: false
        },
        'endowment:network-access': {},
```

```
        'snap_dialog': {},
        'snap_notify': {},
        'snap_getBip44Entropy': [
          {
            coinType: 60 // Ethereum
          }
        ]
      },
      manifestVersion: '0.1'
    };

    const snapCode = await this.generateSnapCode(config);

    return {
      manifest: snapManifest,
      code: snapCode,
      packageJson: await this.generateSnapPackageJson(config)
    };
  }

  async generateSnapCode(config) {
    return `
import { OnRpcRequestHandler, OnTransactionHandler } from '@metamask/snaps-types';
import { panel, text, heading, divider } from '@metamask/snaps-ui';

// Security analysis engine
class AudityzerSecurityEngine {
  async analyzeTransaction(transaction) {
    const risks = [];

    // Check for known malicious contracts
    if (await this.isKnownMaliciousContract(transaction.to)) {
      risks.push({
        level: 'critical',
        message: 'Interacting with known malicious contract',
        recommendation: 'Do not proceed with this transaction'
      });
    }

    // Check for suspicious function calls
    const suspiciousFunctions = await this.detectSuspiciousFunctions(transaction.data);
    if (suspiciousFunctions.length > 0) {
      risks.push({
        level: 'high',
        message: \`Suspicious function calls detected: \${suspiciousFunctions.join(',
')}\`,
        recommendation: 'Review transaction details carefully'
      });
    }

    // Check transaction value and gas
    if (this.isUnusualValue(transaction.value) || this.isUnusualGas(transaction.gas)) {
      risks.push({
        level: 'medium',
        message: 'Unusual transaction value or gas limit',
        recommendation: 'Verify transaction parameters'
      });
    }
```

```
    return risks;
  }

  async isKnownMaliciousContract(address) {
    // Check against Audityzer's malicious contract database
    const response = await fetch(\`https://api.audityzer.com/v1/contracts/\${address}/
reputation\`);
    const data = await response.json();
    return data.malicious || false;
  }

  async detectSuspiciousFunctions(data) {
    const suspicious = [];

    // Check for common attack patterns
    const attackPatterns = [
      { signature: '0xa9059cbb', name: 'transfer', risk: 'token_drain' },
      { signature: '0x23b872dd', name: 'transferFrom', risk: 'unauthorized_transfer' },
      { signature: '0x095ea7b3', name: 'approve', risk: 'unlimited_approval' }
    ];

    for (const pattern of attackPatterns) {
      if (data.startsWith(pattern.signature)) {
        suspicious.push(pattern.name);
      }
    }

    return suspicious;
  }
}

const securityEngine = new AudityzerSecurityEngine();

export const onRpcRequest: OnRpcRequestHandler = async ({ origin, request }) => {
  switch (request.method) {
    case 'audityzer_analyzeContract':
      return await analyzeContract(request.params.address);

    case 'audityzer_getSecurityScore':
      return await getSecurityScore(request.params.address);

    case 'audityzer_reportVulnerability':
      return await reportVulnerability(request.params);

    default:
      throw new Error('Method not found.');
  }
};

export const onTransaction: OnTransactionHandler = async ({ transaction }) => {
  const risks = await securityEngine.analyzeTransaction(transaction);

  if (risks.length === 0) {
    return {
      content: panel([
        heading('  Transaction Security Check'),
        text('No security risks detected. Transaction appears safe to proceed.')
      ])
    };
```

```
  }

  const criticalRisks = risks.filter(r => r.level === 'critical');
  const highRisks = risks.filter(r => r.level === 'high');

  if (criticalRisks.length > 0) {
    return {
      content: panel([
        heading('  CRITICAL SECURITY WARNING'),
        divider(),
        ...criticalRisks.map(risk => text(\`  \${risk.message}\`)),
        divider(),
        text('  Recommendation: DO NOT PROCEED with this transaction'),
        text('This transaction has been flagged as potentially malicious by Audityz-
er.')
      ])
    };
  }

  if (highRisks.length > 0) {
    return {
      content: panel([
        heading('⚠ Security Warning'),
        divider(),
        ...highRisks.map(risk => text(\`⚠ \${risk.message}\`)),
        divider(),
        text('  Please review transaction details carefully before proceeding.')
      ])
    };
  }

  return {
    content: panel([
      heading('⚠ Security Notice'),
      divider(),
      ...risks.map(risk => text(\`ℹ \${risk.message}\`)),
      divider(),
      text('  Please verify transaction details.')
    ])
  };
};

async function analyzeContract(address) {
  try {
    const response = await fetch(\`https://api.audityzer.com/v1/contracts/\${address}/
analyze\`);
    const analysis = await response.json();

    return {
      address,
      securityScore: analysis.securityScore,
      vulnerabilities: analysis.vulnerabilities,
      recommendations: analysis.recommendations,
      lastAnalyzed: analysis.timestamp
    };
  } catch (error) {
    return { error: 'Failed to analyze contract' };
  }
}
```

```
async function getSecurityScore(address) {
  try {
    const response = await fetch(\`https://api.audityzer.com/v1/contracts/\${address}/
score\`);
    const data = await response.json();

    return {
      score: data.score,
      grade: data.grade,
      factors: data.factors
    };
  } catch (error) {
    return { error: 'Failed to get security score' };
  }
}

async function reportVulnerability(params) {
  try {
    const response = await fetch('https://api.audityzer.com/v1/vulnerabilities/report',
{
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(params)
    });

    const result = await response.json();

    return {
      success: true,
      reportId: result.id,
      message: 'Vulnerability reported successfully'
    };
  } catch (error) {
    return { error: 'Failed to report vulnerability' };
  }
}
    `;
  }

  /**
   * Deploy Snap to MetaMask Snaps registry
   * @param {Object} snapPackage - Snap package to deploy
   * @returns {Promise<Object>} Deployment result
   */
  async deploySnap(snapPackage) {
    // Build Snap package
    const builtPackage = await this.buildSnapPackage(snapPackage);

    // Validate Snap
    const validation = await this.validateSnap(builtPackage);
    if (!validation.valid) {
      throw new Error(`Snap validation failed: ${validation.errors.join(', ')}`);
    }

    // Submit to MetaMask Snaps registry
    const submission = await this.submitToRegistry(builtPackage);

    return {
```

```
      snapId: submission.id,
      version: submission.version,
      status: submission.status,
      registryUrl: submission.url,
      installCommand: `npm install ${submission.packageName}`
    };
  }
}

module.exports = MetaMaskSnapsIntegration;
```

Partnership Program Structure

Partner Onboarding Process

```
// src/core/partners/PartnerOnboarding.js
class PartnerOnboarding {
  constructor() {
    this.stages = [
      'application',
      'technical_review',
      'integration_development',
      'testing',
      'certification',
      'launch'
    ];
  }

  async initiatePartnership(application) {
    const partnership = {
      id: this.generatePartnershipId(),
      partner: application.partner,
      category: application.category,
      stage: 'application',
      submittedAt: new Date(),
      status: 'pending_review'
    };

    // Validate application
    const validation = await this.validateApplication(application);
    if (!validation.valid) {
      partnership.status = 'rejected';
      partnership.rejectionReason = validation.errors.join(', ');
      return partnership;
    }

    // Create partnership record
    await this.createPartnershipRecord(partnership);

    // Notify partnership team
    await this.notifyPartnershipTeam(partnership);

    return partnership;
  }

  async validateApplication(application) {
    const errors = [];

    // Required fields
    const required = ['partner', 'category', 'description', 'technicalContact', 'busi-
nessContact'];
    for (const field of required) {
      if (!application[field]) {
        errors.push(`Missing required field: ${field}`);
      }
    }

    // Category validation
    const validCategories = ['infrastructure', 'security', 'development', 'wallet'];
    if (!validCategories.includes(application.category)) {
      errors.push(`Invalid category: ${application.category}`);
    }
```

```
    // Technical requirements
    if (application.category === 'infrastructure' && !application.apiDocumentation) {
      errors.push('API documentation required for infrastructure partners');
    }

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

  async progressPartnership(partnershipId, stage, data = {}) {
    const partnership = await this.getPartnership(partnershipId);

    if (!partnership) {
      throw new Error('Partnership not found');
    }

    const currentStageIndex = this.stages.indexOf(partnership.stage);
    const newStageIndex = this.stages.indexOf(stage);

    if (newStageIndex !== currentStageIndex + 1) {
      throw new Error('Invalid stage progression');
    }

    partnership.stage = stage;
    partnership.updatedAt = new Date();

    // Stage-specific actions
    switch (stage) {
      case 'technical_review':
        await this.conductTechnicalReview(partnership, data);
        break;
      case 'integration_development':
        await this.setupIntegrationEnvironment(partnership, data);
        break;
      case 'testing':
        await this.initiateIntegrationTesting(partnership, data);
        break;
      case 'certification':
        await this.certifyIntegration(partnership, data);
        break;
      case 'launch':
        await this.launchPartnership(partnership, data);
        break;
    }

    await this.updatePartnership(partnership);
    return partnership;
  }
}
```

Partner Certification Program

```
// src/core/partners/PartnerCertification.js
class PartnerCertification {
  constructor() {
    this.certificationLevels = ['bronze', 'silver', 'gold', 'platinum'];
    this.requirements = {
      bronze: {
        integration_tests: 10,
        uptime_requirement: 0.95,
        response_time: 5000,
        documentation_score: 0.7
      },
      silver: {
        integration_tests: 25,
        uptime_requirement: 0.98,
        response_time: 2000,
        documentation_score: 0.8,
        security_audit: true
      },
      gold: {
        integration_tests: 50,
        uptime_requirement: 0.99,
        response_time: 1000,
        documentation_score: 0.9,
        security_audit: true,
        performance_benchmarks: true
      },
      platinum: {
        integration_tests: 100,
        uptime_requirement: 0.995,
        response_time: 500,
        documentation_score: 0.95,
        security_audit: true,
        performance_benchmarks: true,
        community_contribution: true
      }
    };
  }

  async evaluatePartner(partnerId) {
    const partner = await this.getPartner(partnerId);
    const metrics = await this.collectPartnerMetrics(partnerId);

    let certificationLevel = null;

    // Check each level from highest to lowest
    for (const level of this.certificationLevels.reverse()) {
      if (await this.meetsRequirements(metrics, level)) {
        certificationLevel = level;
        break;
      }
    }

    const certification = {
      partnerId,
      level: certificationLevel,
      metrics,
      evaluatedAt: new Date(),
      validUntil: new Date(Date.now() + 365 * 24 * 60 * 60 * 1000), // 1 year
```

```
      requirements: this.requirements[certificationLevel]
    };

    await this.issueCertification(certification);
    return certification;
  }

  async meetsRequirements(metrics, level) {
    const requirements = this.requirements[level];

    // Check each requirement
    for (const [requirement, threshold] of Object.entries(requirements)) {
      if (!this.checkRequirement(metrics, requirement, threshold)) {
        return false;
      }
    }

    return true;
  }

  checkRequirement(metrics, requirement, threshold) {
    switch (requirement) {
      case 'integration_tests':
        return metrics.testsPassed >= threshold;
      case 'uptime_requirement':
        return metrics.uptime >= threshold;
      case 'response_time':
        return metrics.averageResponseTime <= threshold;
      case 'documentation_score':
        return metrics.documentationScore >= threshold;
      case 'security_audit':
        return metrics.securityAuditPassed === true;
      case 'performance_benchmarks':
        return metrics.performanceBenchmarksPassed === true;
      case 'community_contribution':
        return metrics.communityContributions >= 5;
      default:
        return false;
    }
  }
}
```

Success Metrics & KPIs

Partnership Success Metrics

- **Partner Count**: Target 20+ strategic partners in first year
- **Integration Quality**: 95% uptime across all partner integrations
- **User Adoption**: 60% of users utilize at least one partner integration
- **Revenue Impact**: 30% revenue increase through partner channels

Technical Performance

- **API Response Time**: < 500ms average across all partner APIs
- **Error Rate**: < 1% integration failure rate

- **Uptime**: 99.9% availability for partner services
- **Data Accuracy**: 99.5% accuracy in partner data synchronization

Community Engagement

- **Developer Adoption**: 1000+ developers using partner integrations
- **Documentation Quality**: 4.5/5 average rating for integration docs
- **Support Response**: < 4 hours average response time for partner issues
- **Certification Rate**: 80% of partners achieve Bronze certification or higher

---

Implementation Timeline

Phase 1: Foundation (Months 1-2)

- [ ] Develop partner integration framework
- [ ] Create base integration interfaces
- [ ] Implement partner registry system
- [ ] Design certification program

Phase 2: Core Partners (Months 3-4)

- [ ] Integrate Chainlink oracle testing
- [ ] Implement Immunefi bug bounty integration
- [ ] Develop MetaMask Snaps integration
- [ ] Launch Pimlico AA partnership

Phase 3: Expansion (Months 5-6)

- [ ] Add Hardhat/Foundry integrations
- [ ] Implement additional security platform partnerships
- [ ] Launch partner marketplace
- [ ] Begin certification program

Phase 4: Optimization (Months 7-8)

- [ ] Performance optimization across all integrations
- [ ] Advanced analytics and monitoring
- [ ] Partner success program launch
- [ ] Community feedback integration

This comprehensive partner integration framework will create a robust ecosystem around Audityzer, providing enhanced value to users while expanding market reach through strategic partnerships.