Plugin System Architecture

Overview

Design and implement a comprehensive plugin system that allows the community to extend Audityzer's capabilities through custom vulnerability detectors, report generators, CI/CD integrations, and specialized testing modules. This system will foster ecosystem growth and enable rapid adaptation to new security challenges.

Architecture Design

Core Plugin System Components

```
┌─────────────────────────────────────────────────────────────┐
│                      Audityzer Core                         │
├─────────────────────────────────────────────────────────────┤
│                    Plugin Manager                           │
│  ┌─────────────┐ ┌─────────────┐ ┌─────────────┐           │
│  │   Plugin    │ │   Plugin    │ │   Plugin    │           │
│  │  Registry   │ │   Loader    │ │  Validator  │           │
│  └─────────────┘ └─────────────┘ └─────────────┘           │
├─────────────────────────────────────────────────────────────┤
│                      Plugin API                             │
│  ┌─────────────┐ ┌─────────────┐ ┌─────────────┐           │
│  │Vulnerability│ │   Report    │ │   CI/CD     │           │
│  │  Detection  │ │ Generation  │ │ Integration │           │
│  │    API      │ │    API      │ │    API      │           │
│  └─────────────┘ └─────────────┘ └─────────────┘           │
├─────────────────────────────────────────────────────────────┤
│                    Plugin Types                             │
│  ┌─────────────┐ ┌─────────────┐ ┌─────────────┐           │
│  │ Detectors   │ │ Reporters   │ │ Integrations│           │
│  │             │ │             │ │             │           │
│  │ • Custom    │ │ • PDF       │ │ • GitHub    │           │
│  │   Rules     │ │ • Slack     │ │ • GitLab    │           │
│  │ • AI Models │ │ • Email     │ │ • Jenkins   │           │
│  │ • Patterns  │ │ • Dashboard │ │ • CircleCI  │           │
│  └─────────────┘ └─────────────┘ └─────────────┘           │
└─────────────────────────────────────────────────────────────┘
```

Plugin Lifecycle Management

```
// src/core/plugins/PluginManager.js
class PluginManager {
  constructor() {
    this.plugins = new Map();
    this.registry = new PluginRegistry();
    this.loader = new PluginLoader();
    this.validator = new PluginValidator();
    this.hooks = new HookSystem();
  }

  async loadPlugin(pluginPath, options = {}) {
    try {
      // Step 1: Validate plugin structure and security
      const validation = await this.validator.validate(pluginPath);
      if (!validation.isValid) {
        throw new Error(`Plugin validation failed: ${validation.errors.join(', ')}`);
      }

      // Step 2: Load plugin manifest
      const manifest = await this.loader.loadManifest(pluginPath);

      // Step 3: Check compatibility
      if (!this.isCompatible(manifest)) {
        throw new Error(`Plugin incompatible with Audityzer ${this.getVersion()}`);
      }

      // Step 4: Load plugin code
      const plugin = await this.loader.loadPlugin(pluginPath, manifest);

      // Step 5: Initialize plugin
      await plugin.initialize(this.createPluginContext(manifest));

      // Step 6: Register plugin
      this.plugins.set(manifest.id, {
        manifest,
        instance: plugin,
        status: 'active',
        loadedAt: new Date()
      });

      // Step 7: Register hooks and APIs
      this.registerPluginHooks(plugin, manifest);

      console.log(`  Plugin loaded: ${manifest.name} v${manifest.version}`);
      return plugin;

    } catch (error) {
      console.error(`  Failed to load plugin: ${error.message}`);
      throw error;
    }
  }

  async unloadPlugin(pluginId) {
    const pluginInfo = this.plugins.get(pluginId);
    if (!pluginInfo) {
      throw new Error(`Plugin not found: ${pluginId}`);
    }
    try {
      // Cleanup plugin resources
      if (pluginInfo.instance.cleanup) {
        await pluginInfo.instance.cleanup();
      }

      // Unregister hooks
      this.unregisterPluginHooks(pluginId);

      // Remove from registry
      this.plugins.delete(pluginId);

      console.log(`  Plugin unloaded: ${pluginInfo.manifest.name}`);
    } catch (error) {
      console.error(`  Failed to unload plugin: ${error.message}`);
      throw error;
    }
  }

  getPlugin(pluginId) {
    const pluginInfo = this.plugins.get(pluginId);
    return pluginInfo ? pluginInfo.instance : null;
  }

  listPlugins() {
    return Array.from(this.plugins.values()).map(info => ({
      id: info.manifest.id,
      name: info.manifest.name,
      version: info.manifest.version,
      status: info.status,
      loadedAt: info.loadedAt
    }));
  }
}
```

Plugin Types & Interfaces

1. Vulnerability Detection Plugins

Base Detector Interface

```
// src/core/plugins/interfaces/VulnerabilityDetector.js
class VulnerabilityDetector {
  constructor(config = {}) {
    this.config = config;
    this.name = 'BaseDetector';
    this.version = '1.0.0';
    this.supportedLanguages = ['solidity'];
    this.severity = 'medium';
  }

  /**
   * Initialize the detector with Audityzer context
   * @param {Object} context - Audityzer plugin context
   */
  async initialize(context) {
    this.context = context;
    this.logger = context.logger;
    this.utils = context.utils;
  }

  /**
   * Analyze code for vulnerabilities
   * @param {Object} analysisContext - Code analysis context
   * @returns {Promise<Array>} Array of vulnerability findings
   */
  async analyze(analysisContext) {
    throw new Error('analyze() method must be implemented by subclass');
  }

  /**
   * Get detector metadata
   * @returns {Object} Detector information
   */
  getMetadata() {
    return {
      name: this.name,
      version: this.version,
      description: this.description,
      supportedLanguages: this.supportedLanguages,
      severity: this.severity,
      tags: this.tags || []
    };
  }

  /**
   * Validate detector configuration
   * @param {Object} config - Configuration to validate
   * @returns {Object} Validation result
   */
  validateConfig(config) {
    return { isValid: true, errors: [] };
  }

  /**
   * Cleanup resources
   */
  async cleanup() {
    // Override in subclass if needed
  }
}

module.exports = VulnerabilityDetector;
```

Example Custom Detector

```javascript
// plugins/detectors/custom-reentrancy/index.js
const { VulnerabilityDetector } = require('audityzer-plugin-api');

class CustomReentrancyDetector extends VulnerabilityDetector {
  constructor(config) {
    super(config);
    this.name = 'CustomReentrancyDetector';
    this.version = '1.0.0';
    this.description = 'Advanced reentrancy detection with ML-based pattern recogni-
tion';
    this.severity = 'high';
    this.tags = ['reentrancy', 'ml', 'advanced'];
  }

  async initialize(context) {
    await super.initialize(context);

    // Load ML model
    this.model = await this.loadMLModel();

    // Initialize pattern database
    this.patterns = await this.loadPatterns();
  }

  async analyze(analysisContext) {
    const { sourceCode, ast, contractName } = analysisContext;
    const findings = [];

    try {
      // Step 1: Static pattern analysis
      const staticFindings = await this.analyzeStaticPatterns(ast);

      // Step 2: ML-based analysis
      const mlFindings = await this.analyzeMachineLearning(sourceCode);

      // Step 3: Cross-reference and validate
      const validatedFindings = this.validateFindings([...staticFindings, ...mlFind-
ings]);

      // Step 4: Format findings
      for (const finding of validatedFindings) {
        findings.push({
          type: 'reentrancy',
          severity: this.calculateSeverity(finding),
          title: finding.title,
          description: finding.description,
          location: finding.location,
          recommendation: finding.recommendation,
          confidence: finding.confidence,
          metadata: {
            detector: this.name,
            version: this.version,
            analysisMethod: finding.method,
            timestamp: new Date().toISOString()
          }
        });
      }
    } catch (error) {
      this.logger.error(`Analysis failed: ${error.message}`);
    }

    return findings;
  }

  async analyzeStaticPatterns(ast) {
    const findings = [];

    // Traverse AST looking for reentrancy patterns
    this.utils.traverseAST(ast, (node) => {
      if (this.isVulnerablePattern(node)) {
        findings.push({
          method: 'static',
          title: 'Potential Reentrancy Vulnerability',
          description: this.generateDescription(node),
          location: this.getLocation(node),
          confidence: 0.8
        });
      }
    });

    return findings;
  }

  async analyzeMachineLearning(sourceCode) {
    // Use ML model to detect subtle reentrancy patterns
    const features = this.extractFeatures(sourceCode);
    const prediction = await this.model.predict(features);

    if (prediction.probability > 0.7) {
      return [{
        method: 'ml',
        title: 'ML-Detected Reentrancy Risk',
        description: `Machine learning model detected potential reentrancy with ${(pre-
diction.probability * 100).toFixed(1)}% confidence`,
        location: prediction.location,
        confidence: prediction.probability
      }];
    }

    return [];
  }
}

module.exports = CustomReentrancyDetector;
```

2. Report Generation Plugins

Base Reporter Interface

```javascript
// src/core/plugins/interfaces/ReportGenerator.js
class ReportGenerator {
  constructor(config = {}) {
    this.config = config;
    this.name = 'BaseReporter';
    this.version = '1.0.0';
    this.outputFormats = ['json'];
  }

  async initialize(context) {
    this.context = context;
    this.logger = context.logger;
    this.utils = context.utils;
  }

  /**
   * Generate report from analysis results
   * @param {Object} results - Analysis results
   * @param {Object} options - Generation options
   * @returns {Promise<Object>} Generated report
   */
  async generateReport(results, options = {}) {
    throw new Error('generateReport() method must be implemented by subclass');
  }

  /**
   * Get supported output formats
   * @returns {Array} Supported formats
   */
  getSupportedFormats() {
    return this.outputFormats;
  }

  /**
   * Validate report options
   * @param {Object} options - Options to validate
   * @returns {Object} Validation result
   */
  validateOptions(options) {
    return { isValid: true, errors: [] };
  }
}

module.exports = ReportGenerator;
```

Example Slack Reporter

```
// plugins/reporters/slack-reporter/index.js
const { ReportGenerator } = require('audityzer-plugin-api');
const { WebClient } = require('@slack/web-api');

class SlackReporter extends ReportGenerator {
  constructor(config) {
    super(config);
    this.name = 'SlackReporter';
    this.version = '1.0.0';
    this.description = 'Send security analysis results to Slack channels';
    this.outputFormats = ['slack'];
  }

  async initialize(context) {
    await super.initialize(context);

    if (!this.config.token) {
      throw new Error('Slack token required for SlackReporter');
    }

    this.slack = new WebClient(this.config.token);
  }

  async generateReport(results, options = {}) {
    const { channel = '#security', username = 'Audityzer' } = options;

    try {
      // Create summary message
      const summary = this.createSummary(results);

      // Create detailed blocks
      const blocks = this.createMessageBlocks(results);

      // Send to Slack
      const response = await this.slack.chat.postMessage({
        channel,
        username,
        icon_emoji: ':shield:',
        text: summary,
        blocks
      });

      return {
        success: true,
        messageId: response.ts,
        channel: response.channel,
        url: `https://slack.com/app_redirect?channel=${response.channel}&message_ts=${r
esponse.ts}`
      };

    } catch (error) {
      this.logger.error(`Slack report failed: ${error.message}`);
      throw error;
    }
  }

  createSummary(results) {
    const { vulnerabilities, summary } = results;
    const criticalCount = vulnerabilities.filter(v => v.severity === 'critical').length
;
    const highCount = vulnerabilities.filter(v => v.severity === 'high').length;

    if (criticalCount > 0) {
      return `  Critical security issues found! ${criticalCount} critical, $
{highCount} high severity vulnerabilities detected.`;
    } else if (highCount > 0) {
      return `⚠ Security issues detected: ${highCount} high severity vulnerabilities
found.`;
    } else {
      return `  Security analysis completed: $
{summary.total} tests passed, no critical issues found.`;
    }
  }

  createMessageBlocks(results) {
    const blocks = [
      {
        type: 'header',
        text: {
          type: 'plain_text',
          text: '  Audityzer Security Report'
        }
      },
      {
        type: 'section',
        fields: [
          {
            type: 'mrkdwn',
            text: `*Project:* ${results.project || 'Unknown'}`
          },
          {
            type: 'mrkdwn',
            text: `*Timestamp:* ${new Date().toISOString()}`
          },
          {
            type: 'mrkdwn',
            text: `*Total Tests:* ${results.summary.total}`
          },
          {
            type: 'mrkdwn',
            text: `*Vulnerabilities:* ${results.vulnerabilities.length}`
          }
        ]
      }
    ];

    // Add vulnerability details
    if (results.vulnerabilities.length > 0) {
      blocks.push({
        type: 'divider'
      });

      const criticalVulns = results.vulnerabilities.filter(v => v.severity === 'critic-
al');
      const highVulns = results.vulnerabilities.filter(v => v.severity === 'high');

      if (criticalVulns.length > 0) {
        blocks.push({
          type: 'section',
          text: {
            type: 'mrkdwn',
            text: `*  Critical Vulnerabilities (${criticalVulns.length}):*\n${criticalVulns.map(v => `• ${v.title}`).join('\n')}`
          }
        });
      }

      if (highVulns.length > 0) {
        blocks.push({
          type: 'section',
          text: {
            type: 'mrkdwn',
            text: `*⚠ High Severity Vulnerabilities (${highVulns.length}):*\n${highVulns.map(v => `• ${v.title}`).join('\n')}`
          }
        });
      }
    }

    return blocks;
  }
}

module.exports = SlackReporter;
```

3. CI/CD Integration Plugins

Base Integration Interface

```javascript
// src/core/plugins/interfaces/CIIntegration.js
class CIIntegration {
  constructor(config = {}) {
    this.config = config;
    this.name = 'BaseCIIntegration';
    this.version = '1.0.0';
    this.supportedPlatforms = [];
  }

  async initialize(context) {
    this.context = context;
    this.logger = context.logger;
    this.utils = context.utils;
  }

  /**
   * Setup CI/CD integration
   * @param {Object} projectConfig - Project configuration
   * @returns {Promise<Object>} Setup result
   */
  async setup(projectConfig) {
    throw new Error('setup() method must be implemented by subclass');
  }

  /**
   * Create CI/CD configuration files
   * @param {Object} config - Integration configuration
   * @returns {Promise<Array>} Created files
   */
  async createConfigFiles(config) {
    throw new Error('createConfigFiles() method must be implemented by subclass');
  }

  /**
   * Validate CI/CD configuration
   * @param {Object} config - Configuration to validate
   * @returns {Object} Validation result
   */
  validateConfig(config) {
    return { isValid: true, errors: [] };
  }
}

module.exports = CIIntegration;
```

Example GitHub Actions Integration

```javascript
// plugins/integrations/github-actions/index.js
const { CIIntegration } = require('audityzer-plugin-api');
const fs = require('fs-extra');
const path = require('path');

class GitHubActionsIntegration extends CIIntegration {
  constructor(config) {
    super(config);
    this.name = 'GitHubActionsIntegration';
    this.version = '1.0.0';
    this.description = 'GitHub Actions integration for automated security testing';
    this.supportedPlatforms = ['github'];
  }

  async setup(projectConfig) {
    const workflowsDir = path.join(projectConfig.projectPath, '.github', 'workflows');
    await fs.ensureDir(workflowsDir);

    const files = await this.createConfigFiles({
      ...projectConfig,
      workflowsDir
    });

    return {
      success: true,
      files,
      instructions: this.getSetupInstructions()
    };
  }

  async createConfigFiles(config) {
    const files = [];

    // Main security workflow
    const securityWorkflow = this.generateSecurityWorkflow(config);
    const securityPath = path.join(config.workflowsDir, 'security-tests.yml');
    await fs.writeFile(securityPath, securityWorkflow);
    files.push(securityPath);

    // PR workflow
    const prWorkflow = this.generatePRWorkflow(config);
    const prPath = path.join(config.workflowsDir, 'pr-security-check.yml');
    await fs.writeFile(prPath, prWorkflow);
    files.push(prPath);

    // Scheduled audit workflow
    const scheduledWorkflow = this.generateScheduledWorkflow(config);
    const scheduledPath = path.join(config.workflowsDir, 'scheduled-audit.yml');
    await fs.writeFile(scheduledPath, scheduledWorkflow);
    files.push(scheduledPath);

    return files;
  }

  generateSecurityWorkflow(config) {
    return `name: Security Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  security-audit:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Install Audityzer
      run: npm install -g audityzer@latest

    - name: Run security tests
      run: audityzer run ${config.target || 'contracts'} --format json --output secur-
ity-report.json
      env:
        ETHEREUM_RPC_URL: \${{ secrets.ETHEREUM_RPC_URL }}
        ${config.aa?.enabled ? 'PIMLICO_API_KEY: ${{ secrets.PIMLICO_API_KEY }}' : ''}

    - name: Upload security report
      uses: actions/upload-artifact@v3
      if: always()
      with:
        name: security-report
        path: security-report.json

    - name: Comment PR with results
      if: github.event_name == 'pull_request'
      uses: actions/github-script@v6
      with:
        script: |
          const fs = require('fs');
          const report = JSON.parse(fs.readFileSync('security-report.json', 'utf8'));

          const vulnerabilities = report.vulnerabilities || [];
          const critical = vulnerabilities.filter(v => v.severity === 'critic-
al').length;
          const high = vulnerabilities.filter(v => v.severity === 'high').length;

          let comment = '##   Security Analysis Results\\n\\n';

          if (critical > 0) {
            comment += \`  **\${critical} Critical** vulnerabilities found!\\n\`;
          }
          if (high > 0) {
            comment += \`⚠ **\${high} High** severity vulnerabilities found!\\n\`;
          }
          if (critical === 0 && high === 0) {
            comment += '  No critical or high severity vulnerabilities found!\\n';
          }

          comment += \`\\n**Total vulnerabilities:** \${vulnerabilities.length}\\n\`;
          comment += \`**Tests run:** \${report.summary?.total || 0}\\n\`;

          github.rest.issues.createComment({
            issue_number: context.issue.number,
            owner: context.repo.owner,
            repo: context.repo.repo,
            body: comment
          });

    - name: Fail on critical vulnerabilities
      run: |
        CRITICAL_COUNT=\$(jq '.vulnerabilities | map(select(.severity == "critical")) |
length' security-report.json)
        if [ "\$CRITICAL_COUNT" -gt 0 ]; then
          echo "  Critical vulnerabilities found. Failing build."
          exit 1
        fi
`;
  }

  generatePRWorkflow(config) {
    return `name: PR Security Check

on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  quick-security-check:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3
      with:
        fetch-depth: 0

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Install Audityzer
      run: npm install -g audityzer@latest

    - name: Get changed files
      id: changed-files
      run: |
        echo "files=\$(git diff --name-only \${{ github.event.pull_request.base.sha }}
\${{ github.sha }} | grep -E '\\.(sol|js|ts)\$' | tr '\\n' ' ')" >> \$GITHUB_OUTPUT

    - name: Run targeted security analysis
      if: steps.changed-files.outputs.files != ''
      run: |
        echo "Analyzing changed files: \${{ steps.changed-files.outputs.files }}"
        audityzer run \${{ steps.changed-files.outputs.files }} --format json --output
pr-security-report.json --quick

    - name: Post security summary
      if: always()
      uses: actions/github-script@v6
      with:
        script: |
          const fs = require('fs');

          if (!fs.existsSync('pr-security-report.json')) {
            console.log('No security report generated');
            return;
          }

          const report = JSON.parse(fs.readFileSync('pr-security-report.json',
'utf8'));
          const vulnerabilities = report.vulnerabilities || [];

          let status = '  Clean';
          if (vulnerabilities.some(v => v.severity === 'critical')) {
            status = '  Critical Issues';
          } else if (vulnerabilities.some(v => v.severity === 'high')) {
            status = '⚠ Issues Found';
          }

          const body = \`## Security Check: \${status}

**Files analyzed:** \${{ steps.changed-files.outputs.files || 'None' }}
**Vulnerabilities found:** \${vulnerabilities.length}
**Analysis time:** \${new Date().toISOString()}

\${vulnerabilities.length > 0 ?
  '### Issues Found:\\n' + vulnerabilities.slice(0, 5).map(v =>
    \`- **\${v.severity.toUpperCase()}**: \${v.title}\`
  ).join('\\n') +
  (vulnerabilities.length > 5 ? \`\\n... and \${vulnerabilities.length - 5} more\` :
'')
  : ''}
          \`;

          github.rest.issues.createComment({
            issue_number: context.issue.number,
            owner: context.repo.owner,
            repo: context.repo.repo,
            body: body
          });
`;
  }

  getSetupInstructions() {
    return [
      '1. Add the following secrets to your GitHub repository:',
      '   - ETHEREUM_RPC_URL: Your Ethereum RPC endpoint',
      '   - PIMLICO_API_KEY: Your Pimlico API key (if using AA features)',
      '2. Commit and push the generated workflow files',
      '3. GitHub Actions will automatically run security tests on pushes and PRs',
      '4. Check the Actions tab in your repository to monitor test results'
    ];
  }
}

module.exports = GitHubActionsIntegration;
```

Plugin Marketplace

Plugin Registry System

```javascript
// src/core/plugins/PluginRegistry.js
class PluginRegistry {
  constructor() {
    this.registryUrl = 'https://registry.audityzer.com';
    this.localCache = new Map();
    this.cacheExpiry = 24 * 60 * 60 * 1000; // 24 hours
  }

  async searchPlugins(query, options = {}) {
    const { category, tags, author, verified } = options;

    const searchParams = new URLSearchParams({
      q: query,
      ...(category && { category }),
      ...(tags && { tags: tags.join(',') }),
      ...(author && { author }),
      ...(verified !== undefined && { verified })
    });

    const response = await fetch(`${this.registryUrl}/search?${searchParams}`);
    const results = await response.json();

    return results.plugins.map(plugin => ({
      id: plugin.id,
      name: plugin.name,
      description: plugin.description,
      version: plugin.version,
      author: plugin.author,
      category: plugin.category,
      tags: plugin.tags,
      downloads: plugin.downloads,
      rating: plugin.rating,
      verified: plugin.verified,
      lastUpdated: plugin.lastUpdated
    }));
  }

  async getPlugin(pluginId) {
    // Check local cache first
    const cached = this.getCachedPlugin(pluginId);
    if (cached && !this.isCacheExpired(cached)) {
      return cached.data;
    }

    // Fetch from registry
    const response = await fetch(`${this.registryUrl}/plugins/${pluginId}`);
    if (!response.ok) {
      throw new Error(`Plugin not found: ${pluginId}`);
    }

    const plugin = await response.json();

    // Cache the result
    this.localCache.set(pluginId, {
      data: plugin,
      timestamp: Date.now()
    });
    return plugin;
  }

  async installPlugin(pluginId, version = 'latest') {
    const plugin = await this.getPlugin(pluginId);

    if (version !== 'latest' && !plugin.versions.includes(version)) {
      throw new Error(`Version ${version} not available for ${pluginId}`);
    }

    const downloadUrl = `${this.registryUrl}/plugins/${pluginId}/download/${version}`;
    const response = await fetch(downloadUrl);

    if (!response.ok) {
      throw new Error(`Failed to download plugin: ${response.statusText}`);
    }

    return response.arrayBuffer();
  }

  async publishPlugin(pluginPackage, metadata) {
    const formData = new FormData();
    formData.append('package', pluginPackage);
    formData.append('metadata', JSON.stringify(metadata));

    const response = await fetch(`${this.registryUrl}/publish`, {
      method: 'POST',
      body: formData,
      headers: {
        'Authorization': `Bearer ${this.getAuthToken()}`
      }
    });

    if (!response.ok) {
      throw new Error(`Failed to publish plugin: ${response.statusText}`);
    }

    return response.json();
  }
}
```

Plugin CLI Commands

```
// src/cli/plugin-commands.js
const { program } = require('commander');
const PluginManager = require('../core/plugins/PluginManager');
const PluginRegistry = require('../core/plugins/PluginRegistry');

// Plugin management commands
program
  .command('plugin')
  .description('Manage Audityzer plugins')
  .addCommand(
    program.createCommand('search')
      .description('Search for plugins in the registry')
      .argument('<query>', 'Search query')
      .option('-c, --category <category>', 'Filter by category')
      .option('-t, --tags <tags>', 'Filter by tags (comma-separated)')
      .option('-a, --author <author>', 'Filter by author')
      .option('--verified', 'Show only verified plugins')
      .action(async (query, options) => {
        const registry = new PluginRegistry();
        const results = await registry.searchPlugins(query, options);

        console.log(`Found ${results.length} plugins:\n`);

        results.forEach(plugin => {
          console.log(`  ${plugin.name} v${plugin.version}`);
          console.log(`   ${plugin.description}`);
          console.log(`   Author: ${plugin.author} | Downloads: ${plugin.downloads} |
Rating: ${plugin.rating}/5`);
          if (plugin.verified) console.log(`     Verified`);
          console.log(`   Install: audityzer plugin install ${plugin.id}\n`);
        });
      })
  )
  .addCommand(
    program.createCommand('install')
      .description('Install a plugin')
      .argument('<plugin-id>', 'Plugin ID to install')
      .option('-v, --version <version>', 'Specific version to install', 'latest')
      .option('-g, --global', 'Install globally')
      .action(async (pluginId, options) => {
        const manager = new PluginManager();

        console.log(`Installing plugin: ${pluginId}@${options.version}...`);

        try {
          await manager.installFromRegistry(pluginId, options.version, {
            global: options.global
          });

          console.log(`  Plugin ${pluginId} installed successfully!`);
        } catch (error) {
          console.error(`  Installation failed: ${error.message}`);
          process.exit(1);
        }
      })
  )
  .addCommand(
    program.createCommand('list')
```

```
        .description('List installed plugins')
        .option('-a, --all', 'Show all plugins (including disabled)')
        .action(async (options) => {
          const manager = new PluginManager();
          const plugins = await manager.listPlugins(options.all);

          if (plugins.length === 0) {
            console.log('No plugins installed.');
            return;
          }

          console.log('Installed plugins:\n');

          plugins.forEach(plugin => {
            const status = plugin.status === 'active' ? ' ' : ' ';
            console.log(`${status} ${plugin.name} v${plugin.version}`);
            console.log(`   ID: ${plugin.id}`);
            console.log(`   Status: ${plugin.status}`);
            console.log(`   Loaded: ${plugin.loadedAt}\n`);
          });
        })
    )
    .addCommand(
      program.createCommand('uninstall')
        .description('Uninstall a plugin')
        .argument('<plugin-id>', 'Plugin ID to uninstall')
        .action(async (pluginId) => {
          const manager = new PluginManager();

          try {
            await manager.uninstallPlugin(pluginId);
            console.log(`  Plugin ${pluginId} uninstalled successfully!`);
          } catch (error) {
            console.error(`  Uninstallation failed: ${error.message}`);
            process.exit(1);
          }
        })
    )
    .addCommand(
      program.createCommand('enable')
        .description('Enable a plugin')
        .argument('<plugin-id>', 'Plugin ID to enable')
        .action(async (pluginId) => {
          const manager = new PluginManager();

          try {
            await manager.enablePlugin(pluginId);
            console.log(`  Plugin ${pluginId} enabled!`);
          } catch (error) {
            console.error(`  Enable failed: ${error.message}`);
            process.exit(1);
          }
        })
    )
    .addCommand(
      program.createCommand('disable')
        .description('Disable a plugin')
        .argument('<plugin-id>', 'Plugin ID to disable')
        .action(async (pluginId) => {
```

```
      const manager = new PluginManager();

      try {
        await manager.disablePlugin(pluginId);
        console.log(`  Plugin ${pluginId} disabled!`);
      } catch (error) {
        console.error(`  Disable failed: ${error.message}`);
        process.exit(1);
      }
    })
);
```

Security & Validation

Plugin Security Framework

```
// src/core/plugins/PluginValidator.js
class PluginValidator {
  constructor() {
    this.securityRules = [
      this.validateManifest,
      this.checkCodeSafety,
      this.validateDependencies,
      this.checkPermissions,
      this.scanForMalware
    ];
  }

  async validate(pluginPath) {
    const results = {
      isValid: true,
      errors: [],
      warnings: [],
      securityScore: 100
    };

    for (const rule of this.securityRules) {
      try {
        const ruleResult = await rule.call(this, pluginPath);

        if (!ruleResult.passed) {
          results.isValid = false;
          results.errors.push(...ruleResult.errors);
        }

        results.warnings.push(...ruleResult.warnings);
        results.securityScore -= ruleResult.penalty || 0;

      } catch (error) {
        results.isValid = false;
        results.errors.push(`Validation rule failed: ${error.message}`);
      }
    }

    return results;
  }

  async validateManifest(pluginPath) {
    const manifestPath = path.join(pluginPath, 'plugin.json');

    if (!fs.existsSync(manifestPath)) {
      return {
        passed: false,
        errors: ['Missing plugin.json manifest file']
      };
    }

    const manifest = JSON.parse(fs.readFileSync(manifestPath, 'utf8'));
    const errors = [];
    const warnings = [];

    // Required fields
    const requiredFields = ['id', 'name', 'version', 'description', 'author', 'main'];
    for (const field of requiredFields) {
      if (!manifest[field]) {
        errors.push(`Missing required field: ${field}`);
      }
    }

    // Version format
    if (manifest.version && !semver.valid(manifest.version)) {
      errors.push('Invalid version format (must be semver)');
    }

    // Permissions validation
    if (manifest.permissions) {
      const allowedPermissions = ['filesystem', 'network', 'process', 'env'];
      const invalidPermissions = manifest.permissions.filter(p => !allowedPermis-
sions.includes(p));

      if (invalidPermissions.length > 0) {
        errors.push(`Invalid permissions: ${invalidPermissions.join(', ')}`);
      }

      // Warn about dangerous permissions
      const dangerousPermissions = manifest.permissions.filter(p => ['process', 'env'].
includes(p));
      if (dangerousPermissions.length > 0) {
        warnings.push(`Plugin requests dangerous permissions: ${dangerousPermis-
sions.join(', ')}`);
      }
    }

    return {
      passed: errors.length === 0,
      errors,
      warnings,
      penalty: warnings.length * 5
    };
  }

  async checkCodeSafety(pluginPath) {
    const errors = [];
    const warnings = [];

    // Scan for dangerous patterns
    const dangerousPatterns = [
      /eval\s*\(/g,
      /Function\s*\(/g,
      /require\s*\(\s*['"`]child_process['"`]\s*\)/g,
      /require\s*\(\s*['"`]fs['"`]\s*\)/g,
      /process\.exit/g,
      /process\.kill/g
    ];

    const jsFiles = glob.sync('**/*.js', { cwd: pluginPath });

    for (const file of jsFiles) {
      const content = fs.readFileSync(path.join(pluginPath, file), 'utf8');

      for (const pattern of dangerousPatterns) {
        if (pattern.test(content)) {
          warnings.push(`Potentially dangerous code in ${file}: ${pattern.source}`);
        }
      }
    }

    return {
      passed: errors.length === 0,
      errors,
      warnings,
      penalty: warnings.length * 10
    };
  }

  async validateDependencies(pluginPath) {
    const packageJsonPath = path.join(pluginPath, 'package.json');

    if (!fs.existsSync(packageJsonPath)) {
      return { passed: true, errors: [], warnings: [] };
    }

    const packageJson = JSON.parse(fs.readFileSync(packageJsonPath, 'utf8'));
    const errors = [];
    const warnings = [];

    // Check for known vulnerable packages
    const vulnerablePackages = await this.getVulnerablePackages();
    const dependencies =
{ ...packageJson.dependencies, ...packageJson.devDependencies };

    for (const [pkg, version] of Object.entries(dependencies)) {
      if (vulnerablePackages.includes(pkg)) {
        errors.push(`Vulnerable dependency detected: ${pkg}@${version}`);
      }
    }

    return {
      passed: errors.length === 0,
      errors,
      warnings,
      penalty: errors.length * 20
    };
  }
}
```

Success Metrics

Plugin Ecosystem Health

- **Plugin Count**: Target 100+ plugins in first year
- **Active Developers**: Target 50+ plugin developers
- **Download Volume**: Target 10K+ plugin downloads monthly
- **Quality Score**: Maintain 4.0+ average plugin rating

Community Engagement

- **Contribution Rate**: 30% of users contribute plugins or improvements

- **Plugin Diversity**: Cover all major vulnerability types and integrations
- **Update Frequency**: 80% of plugins updated within 6 months
- **Security Compliance**: 100% of verified plugins pass security validation

Technical Performance

- **Plugin Load Time**: < 500ms average plugin initialization
- **Memory Overhead**: < 50MB per active plugin
- **API Stability**: 99.9% backward compatibility maintained
- **Error Rate**: < 1% plugin execution failures

This comprehensive plugin system will enable rapid ecosystem growth and allow the community to extend Audityzer's capabilities far beyond what the core team could build alone.