

CyberSwarm Agents Documentation

Table of Contents

- [1. Agent System Overview](#)
- [2. Agent Types](#)
- [3. Agent Architecture](#)
- [4. Agent Communication](#)
- [5. Task Management](#)
- [6. Chain of Thought](#)
- [7. Agent Implementation Details](#)

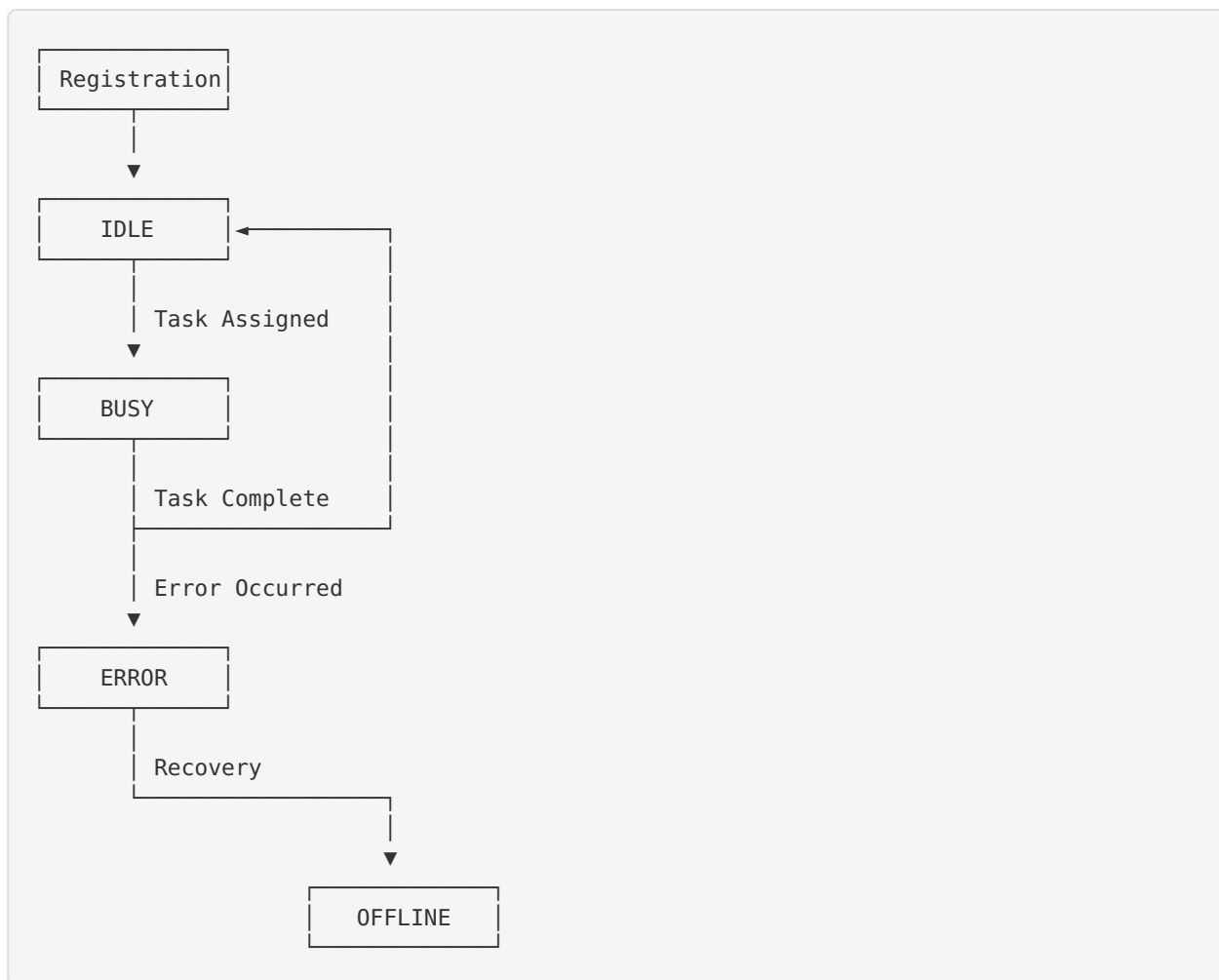
Agent System Overview

The CyberSwarm platform employs a multi-agent architecture where specialized autonomous agents collaborate to simulate realistic cybersecurity scenarios. Each agent type has specific capabilities, responsibilities, and decision-making processes.

Core Principles

- Autonomy:** Agents operate independently with their own decision-making logic
- Specialization:** Each agent type focuses on specific cybersecurity domains
- Collaboration:** Agents communicate and coordinate through events and tasks
- Transparency:** All agent reasoning is logged via Chain of Thought
- Adaptability:** Agents learn and adjust strategies based on outcomes

Agent Lifecycle



Agent Types

1. Reconnaissance Agent

Purpose: Gather intelligence about target systems and networks

Capabilities:

- Port scanning
- Service detection
- OS fingerprinting
- Network mapping
- Vulnerability scanning
- OSINT gathering

Supported Tasks:

```
const reconTasks = [
  'port_scan',
  'service_detection',
  'os_fingerprinting',
  'network_mapping',
  'vulnerability_scan',
  'subdomain_enumeration',
  'dns_enumeration',
  'whois_lookup'
];
```

Decision-Making Process:

1. Analyze target scope
2. Select appropriate scanning techniques
3. Determine scan intensity (stealth vs. speed)
4. Execute reconnaissance
5. Parse and categorize results
6. Emit findings as events

Example Chain of Thought:

Step 1: Target Analysis

- Reasoning: "Target is 192.168.1.0/24 network. Need to identify live hosts first."
- Confidence: 0.95

Step 2: Technique Selection

- Reasoning: "Using TCP SYN scan for speed and stealth. Scanning common ports first."
- Confidence: 0.90

Step 3: Execution

- Reasoning: "Discovered 15 live hosts. Proceeding with service detection on open ports."
- Confidence: 0.88

Step 4: Results Analysis

- Reasoning: "Found web server on 192.168.1.100:80 running Apache 2.4.41. Potential vulnerability."
- Confidence: 0.92

2. Exploitation Agent

Purpose: Attempt to exploit identified vulnerabilities

Capabilities:

- CVE exploitation
- Privilege escalation
- Lateral movement
- Payload delivery
- Persistence establishment
- Data exfiltration

Supported Tasks:

```
const exploitTasks = [
  'exploit_vulnerability',
  'privilege_escalation',
  'lateral_movement',
  'establish_persistence',
  'exfiltrate_data',
  'deploy_payload',
  'crack_credentials',
  'bypass_authentication'
];
```

Decision-Making Process:

1. Evaluate vulnerability severity and exploitability
2. Select appropriate exploit technique
3. Assess detection risk
4. Execute exploitation attempt
5. Verify success
6. Establish foothold if successful

Example Chain of Thought:

Step 1: Vulnerability Assessment

- Reasoning: "CVE-2024-1234 identified on target. CVSS score 9.8. High priority."
- Confidence: 0.93

Step 2: Exploit Selection

- Reasoning: "Public exploit available. Requires no authentication. Low complexity."
- Confidence: 0.91

Step 3: Risk Analysis

- Reasoning: "Target has basic IDS. Using obfuscated payload to reduce detection."
- Confidence: 0.85

Step 4: Execution

- Reasoning: "Exploit successful. Gained shell access as www-data user."
- Confidence: 0.97

Step 5: Post-Exploitation

- Reasoning: "Attempting privilege escalation via kernel exploit."
- Confidence: 0.82

3. Defense Agent

Purpose: Detect, respond to, and mitigate attacks

Capabilities:

- Intrusion detection
- Anomaly detection
- Incident response
- Patch management
- Access control enforcement
- Threat hunting

Supported Tasks:

```
const defenseTasks = [
  'detect_intrusion',
  'analyze_anomaly',
  'respond_to_incident',
  'apply_patch',
  'block_ip',
  'isolate_system',
  'collect_forensics',
  'threat_hunt'
];
```

Decision-Making Process:

1. Monitor for suspicious activity
2. Analyze detected anomalies
3. Determine threat severity
4. Select appropriate response
5. Execute defensive action
6. Verify effectiveness

Example Chain of Thought:

```
Step 1: Detection
- Reasoning: "Unusual network traffic detected from 192.168.1.100. Multiple failed login attempts."
- Confidence: 0.89

Step 2: Analysis
- Reasoning: "Pattern matches brute force attack. Source IP not in whitelist."
- Confidence: 0.94

Step 3: Threat Assessment
- Reasoning: "High severity. Active attack in progress. Immediate action required."
- Confidence: 0.96

Step 4: Response Selection
- Reasoning: "Blocking source IP and enabling account lockout policy."
- Confidence: 0.92

Step 5: Execution
- Reasoning: "Firewall rule applied. Attack traffic dropped. Monitoring for evasion attempts."
- Confidence: 0.90
```

4. Strategy Adaptation Agent

Purpose: Analyze defensive capabilities and adapt attack strategies

Capabilities:

- Defense analysis
- Strategy formulation
- Tactic recommendation
- Pattern recognition
- Countermeasure identification
- Risk assessment

Supported Tasks:

```
const strategyTasks = [
  'analyze_defense',
  'adapt_strategy',
  'recommend_tactics',
  'assess_risk',
  'identify_countermeasures',
  'evaluate_effectiveness',
  'plan_campaign',
  'optimize_approach'
];
```

Decision-Making Process:

1. Observe defensive actions
2. Analyze defensive patterns
3. Identify weaknesses
4. Formulate alternative strategies
5. Recommend tactical adjustments
6. Evaluate strategy effectiveness

Example Chain of Thought:

Step 1: Defense Observation

- Reasoning: "Blue team deployed IDS with signature-based detection. Fast patch deployment observed."
- Confidence: 0.91

Step 2: Pattern Analysis

- Reasoning: "Defense focuses on known threats. Limited behavioral analysis. Patch window ~24 hours."
- Confidence: 0.88

Step 3: Weakness Identification

- Reasoning: "Zero-day vulnerabilities likely undetected. Living-off-the-land techniques may bypass IDS."
- Confidence: 0.85

Step 4: Strategy Formulation

- Reasoning: "Recommend fileless malware and legitimate tool abuse. Target unpatched systems."
- Confidence: 0.87

Step 5: Tactical Recommendation

- Reasoning: "Use PowerShell for execution. Avoid known malicious signatures. Exploit patch delay."
- Confidence: 0.89

5. Orchestrator Agent

Purpose: Coordinate multi-agent operations and manage workflows

Capabilities:

- Task distribution
- Agent coordination
- Workflow management
- Resource allocation

- Conflict resolution
- Performance monitoring

Supported Tasks:

```
const orchestratorTasks = [  
  'distribute_tasks',  
  'coordinate_agents',  
  'manage_workflow',  
  'allocate_resources',  
  'resolve_conflicts',  
  'monitor_performance',  
  'optimize_operations',  
  'handle_failures'  
];
```

Decision-Making Process:

1. Receive high-level objectives
2. Break down into subtasks
3. Assign tasks to appropriate agents
4. Monitor execution progress
5. Handle failures and conflicts
6. Aggregate results

Agent Architecture

Base Agent Class

```
abstract class BaseAgent {
  protected agentId: string;
  protected agentName: string;
  protected agentType: string;
  protected status: AgentStatus;
  protected supportedTasks: string[];

  constructor(config: AgentConfig) {
    this.agentId = config.agentId;
    this.agentName = config.agentName;
    this.agentType = config.agentType;
    this.status = 'IDLE';
    this.supportedTasks = config.supportedTasks;
  }

  abstract async executeTask(task: Task): Promise<TaskResult>;

  protected async logChainOfThought(
    stepNumber: number,
    stepType: string,
    description: string,
    reasoning: string,
    data?: any,
    confidence?: number
  ): Promise<void> {
    // Log reasoning step
  }

  protected async emitEvent(
    eventType: string,
    payload: any,
    severity?: Severity
  ): Promise<void> {
    // Emit event to system
  }

  protected async updateStatus(status: AgentStatus): Promise<void> {
    this.status = status;
    // Notify orchestrator
  }
}
```

Agent Implementation Example

```
// lib/agents/reconnaissance-agent.ts
export class ReconnaissanceAgent extends BaseAgent {
  constructor() {
    super({
      agentId: `recon-${uuid()}`,
      agentName: 'Reconnaissance Agent',
      agentType: 'reconnaissance',
      supportedTasks: [
        'port_scan',
        'service_detection',
        'os_fingerprinting',
        'network_mapping'
      ]
    });
  }

  async executeTask(task: Task): Promise<TaskResult> {
    this.updateStatus('BUSY');

    try {
      switch (task.taskName) {
        case 'port_scan':
          return await this.performPortScan(task);
        case 'service_detection':
          return await this.detectServices(task);
        case 'os_fingerprinting':
          return await this.fingerprintOS(task);
        default:
          throw new Error(`Unsupported task: ${task.taskName}`);
      }
    } catch (error) {
      this.updateStatus('ERROR');
      throw error;
    } finally {
      this.updateStatus('IDLE');
    }
  }

  private async performPortScan(task: Task): Promise<TaskResult> {
    // Step 1: Analyze target
    await this.logChainOfThought(
      1,
      'analysis',
      'Target analysis',
      `Analyzing target ${task.target}. Determining scan strategy.`,
      { target: task.target },
      0.95
    );

    // Step 2: Execute scan
    await this.logChainOfThought(
      2,
      'execution',
      'Port scanning',
      `Executing TCP SYN scan on common ports.`,
      { ports: [80, 443, 22, 21, 3389] },
      0.90
    );

    // Simulate scan
    await this.simulateNetworkOperation('port_scan', task.target, 3);
  }
}
```

```

const openPorts = this.generateOpenPorts();

// Step 3: Results
await this.logChainOfThought(
  3,
  'results',
  'Scan complete',
  `Discovered ${openPorts.length} open ports.`,
  { openPorts },
  0.92
);

// Emit event
await this.emitEvent(
  'ports_discovered',
  {
    target: task.target,
    openPorts,
    scanType: 'TCP_SYN'
  },
  'Medium'
);

return {
  success: true,
  data: { openPorts }
};
}

private generateOpenPorts(): number[] {
  const commonPorts = [21, 22, 23, 25, 80, 443, 3306, 3389, 8080];
  return commonPorts.filter(() => Math.random() > 0.6);
}

private async simulateNetworkOperation(
  operation: string,
  target: string,
  duration: number
): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, duration * 1000));
}
}

```

Agent Communication

Event-Based Communication

Agents communicate through events:

```

interface AgentEvent {
  eventType: string;
  sourceAgentId: string;
  targetAgentId?: string;
  payload: any;
  severity?: Severity;
  timestamp: Date;
}

// Example: Recon agent discovers vulnerability
await this.emitEvent(
  'vulnerability_discovered',
  {
    target: '192.168.1.100',
    cve: 'CVE-2024-1234',
    severity: 'Critical',
    exploitable: true
  },
  'Critical'
);

// Exploitation agent receives event and acts
eventEmitter.on('vulnerability_discovered', async (event) => {
  if (event.payload.exploitable) {
    await this.queueTask({
      taskName: 'exploit_vulnerability',
      target: event.payload.target,
      details: { cve: event.payload.cve }
    });
  }
});

```

Task-Based Communication

Orchestrator assigns tasks to agents:

```

interface Task {
  id: string;
  taskId: string;
  agentType: string;
  taskName: string;
  target?: string;
  details?: any;
  status: TaskStatus;
  priority: number;
  createdAt: Date;
}

// Orchestrator creates task
const task: Task = {
  id: uuid(),
  taskId: `task-${Date.now()}`,
  agentType: 'exploitation',
  taskName: 'exploit_vulnerability',
  target: '192.168.1.100',
  details: {
    cve: 'CVE-2024-1234',
    payload: 'reverse_shell'
  },
  status: 'PENDING',
  priority: 9,
  createdAt: new Date()
};

// Agent receives and executes task
const result = await agent.executeTask(task);

```

Task Management

Task Queue

```

class TaskQueue {
  private queue: Task[] = [];

  enqueue(task: Task): void {
    this.queue.push(task);
    this.queue.sort((a, b) => b.priority - a.priority);
  }

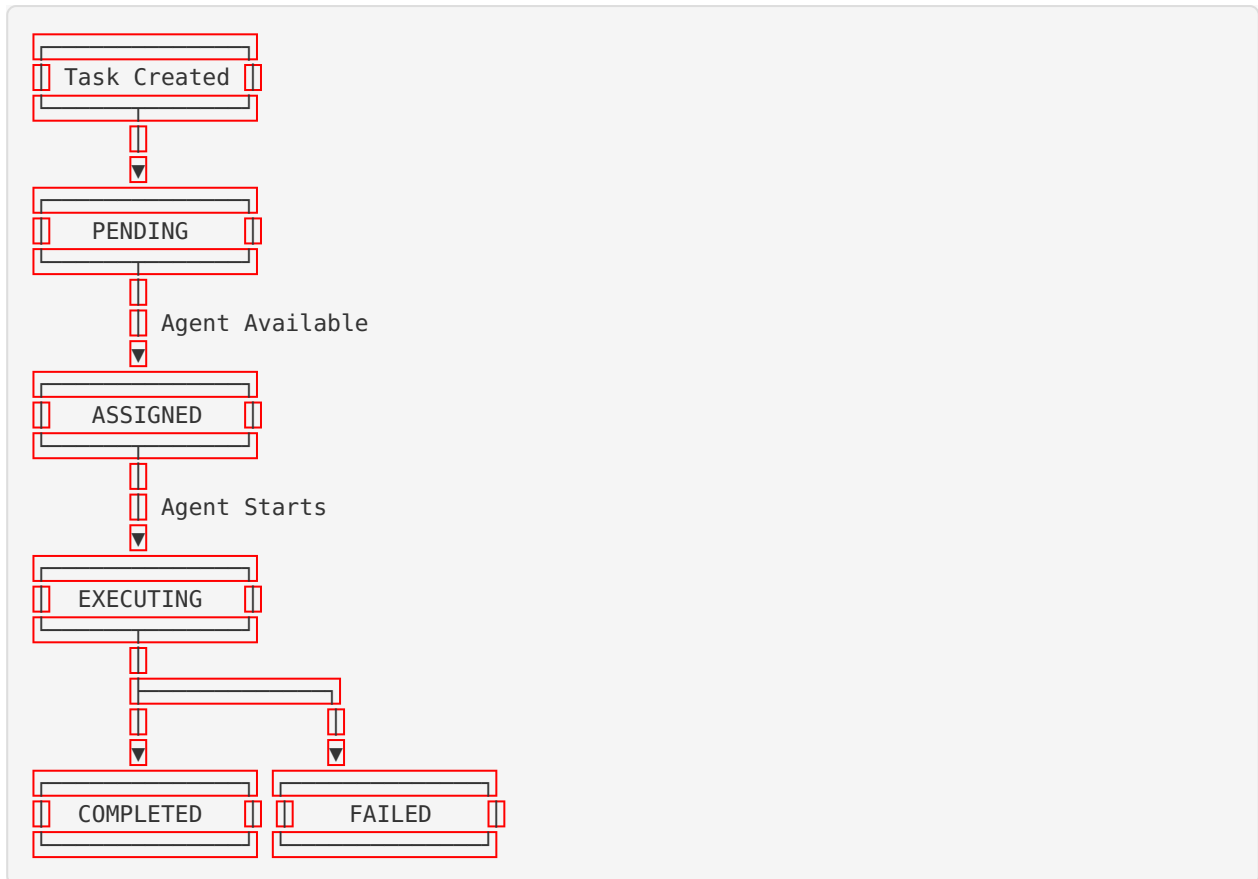
  dequeue(): Task | undefined {
    return this.queue.shift();
  }

  getTasksForAgent(agentType: string): Task[] {
    return this.queue.filter(t => t.agentType === agentType);
  }

  updateTaskStatus(taskId: string, status: TaskStatus): void {
    const task = this.queue.find(t => t.taskId === taskId);
    if (task) {
      task.status = status;
    }
  }
}

```

Task Execution Flow



Chain of Thought

Purpose

Chain of Thought provides transparency into agent decision-making:

1. **Debugging:** Understand why agents made specific decisions
2. **Learning:** Study agent reasoning for educational purposes
3. **Auditing:** Review agent actions for compliance
4. **Optimization:** Identify inefficiencies in agent logic

Structure

```

interface ChainOfThought {
  id: string;
  stepNumber: number;
  stepType: string;           // 'analysis', 'planning', 'execution', 'evaluation'
  description: string;        // Brief description
  reasoning: string;          // Detailed reasoning
  data?: any;                 // Supporting data
  confidence?: number;        // 0.0 to 1.0
  timestamp: Date;
  agentId: string;
  taskId?: string;
}
  
```

Example: Complete Chain of Thought

```
// Exploitation Task: Exploit CVE-2024-1234

// Step 1: Vulnerability Assessment
{
  stepNumber: 1,
  stepType: 'analysis',
  description: 'Vulnerability assessment',
  reasoning: 'Analyzing CVE-2024-1234. CVSS score 9.8 (Critical). Remote code execution vulnerability in Apache Struts. Public exploit available. Target confirmed running vulnerable version 2.5.30.',
  data: {
    cve: 'CVE-2024-1234',
    cvss: 9.8,
    exploitAvailable: true,
    targetVersion: '2.5.30',
    vulnerableVersions: ['2.5.0-2.5.30']
  },
  confidence: 0.95
}

// Step 2: Exploit Selection
{
  stepNumber: 2,
  stepType: 'planning',
  description: 'Exploit selection',
  reasoning: 'Selected public exploit from Exploit-DB. Requires no authentication. Low complexity. High reliability. Will use reverse shell payload for post-exploitation access.',
  data: {
    exploitSource: 'Exploit-DB',
    exploitId: 'EDB-50123',
    authentication: false,
    complexity: 'low',
    reliability: 'high',
    payload: 'reverse_shell'
  },
  confidence: 0.92
}

// Step 3: Detection Risk Analysis
{
  stepNumber: 3,
  stepType: 'analysis',
  description: 'Detection risk assessment',
  reasoning: 'Target has basic IDS (Snort). Signature-based detection likely. Using obfuscated payload and encrypted communication to reduce detection probability. Estimated detection risk: 30%.',
  data: {
    idsPresent: true,
    idsType: 'Snort',
    detectionMethod: 'signature-based',
    obfuscation: true,
    encryption: true,
    estimatedDetectionRisk: 0.30
  },
  confidence: 0.87
}

// Step 4: Exploit Execution
{
  stepNumber: 4,
  stepType: 'execution',
```

```

    description: 'Exploit execution',
    reasoning: 'Sending crafted HTTP request to vulnerable endpoint. Payload delivered
successfully. Waiting for callback on port 4444.',
    data: {
      method: 'POST',
      endpoint: '/struts2-showcase/actionChain1.action',
      payloadSize: 2048,
      callbackPort: 4444
    },
    confidence: 0.90
  }

// Step 5: Success Verification
{
  stepNumber: 5,
  stepType: 'evaluation',
  description: 'Exploitation result',
  reasoning: 'Received callback from target. Shell access established as www-data
user. Exploitation successful. Proceeding with post-exploitation activities.',
  data: {
    success: true,
    shellAccess: true,
    user: 'www-data',
    privileges: 'limited',
    nextSteps: ['privilege_escalation', 'lateral_movement']
  },
  confidence: 0.98
}

```

Agent Implementation Details

Reconnaissance Agent Implementation

File: lib/agents/reconnaissance-agent.ts

Key Methods:

- performPortScan() : Scans target ports
- detectServices() : Identifies running services
- fingerprintOS() : Determines operating system
- mapNetwork() : Creates network topology

Simulation Techniques:

- Random port generation
- Service version simulation
- OS detection simulation
- Network delay simulation

Exploitation Agent Implementation

File: lib/agents/exploitation-agent.ts

Key Methods:

- exploitVulnerability() : Attempts exploitation
- escalatePrivileges() : Gains higher privileges
- establishPersistence() : Maintains access
- exfiltrateData() : Extracts sensitive data

Simulation Techniques:

- CVE database lookup
- Exploit success probability
- Privilege escalation paths
- Detection evasion simulation

Defense Agent Implementation

File: `lib/agents/defense-agent.ts`

Key Methods:

- `detectIntrusion()` : Identifies attacks
- `analyzeAnomaly()` : Investigates suspicious activity
- `respondToIncident()` : Takes defensive action
- `applyPatch()` : Remediates vulnerabilities

Simulation Techniques:

- Signature matching
- Behavioral analysis
- Threat intelligence correlation
- Response effectiveness simulation

Strategy Adaptation Agent Implementation

File: `lib/agents/strategy-adaptation-agent.ts`

Key Methods:

- `analyzeDefense()` : Studies defensive capabilities
- `adaptStrategy()` : Modifies attack approach
- `recommendTactics()` : Suggests new techniques
- `assessRisk()` : Evaluates operation risk

Simulation Techniques:

- Defense pattern recognition
- Strategy effectiveness scoring
- Tactic recommendation engine
- Risk calculation algorithms

This comprehensive agent documentation provides the foundation for understanding, implementing, and extending the CyberSwarm multi-agent system.