OPENCLAW SECURITY HARDENING GUIDE

Post-Deployment Security Lockdown

With Copy-Paste Agent Prompts at Every Step

ScaleUP Media • 2026 Edition

How to Use This Guide

This guide is designed to be worked through section by section. Each section contains:

- An explanation of WHY this security control matters
- The technical details of WHAT to configure
- A ready-to-use AGENT PROMPT you can copy and paste directly into your AI coding agent (Claude, Cursor, Windsurf, etc.) to have it implement the hardening step for you

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
The purple boxes like this one contain your agent prompts.
Copy the entire contents and paste into your AI agent.
Each prompt is self-contained and tells the agent exactly
what to do, what to check, and what to output.
```

Work through them in order. Each section builds on the previous one. After completing all sections, use the master checklist in Section 11 to verify everything is locked down.

✅ **PRO TIP:** For the training video: walk through each section, explain the concept, then show yourself pasting the agent prompt and reviewing the output. That's the workflow your SPRINT members will replicate.

1. Pre-Hardening Assessment

Before you touch a single setting, you need to understand your current attack surface. This section walks you through a full security audit of your deployed OpenClaw instance. Do NOT skip this — most security breaches happen because people patch random things without understanding what's actually exposed.

1.1 Document Your Current Configuration

Pull up your OpenClaw dashboard and document every single integration, API connection, and webhook that's currently active. You need a full inventory before you start locking things down.

Configuration Audit Checklist

- List all active API keys and their permission scopes
- Document every webhook URL currently registered
- Identify all third-party integrations (CRMs, payment processors, email providers)
- Map all user accounts and their access levels
- Record which models are accessible and their routing configurations
- Note any custom endpoints or proxy configurations
- Screenshot your current environment variable setup (redact sensitive values)

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Audit my OpenClaw deployment for security vulnerabilities. Do the
following:

1. SCAN THE ENTIRE CODEBASE for hardcoded API keys, secrets, passwords,
   or tokens. Check all files including .env.example, docker-compose.yml,
   CI/CD configs, and README files. Report every instance found.

2. LIST every environment variable that contains a secret or API key.
   For each one, tell me:
   - Variable name
   - What service it authenticates to
   - Whether it has an expiration date
   - The file(s) where it is referenced

3. FIND all API endpoints in the application. For each endpoint, report:
   - Route path and HTTP method
   - Whether authentication is required
   - What authorization checks exist
   - Whether rate limiting is applied
```

```
4. CHECK for common security misconfigurations:
   - CORS set to wildcard (*)
   - Debug mode enabled in production
   - Stack traces exposed in error responses
   - Swagger/API docs publicly accessible
   - Default credentials still active

5. OUTPUT a security audit report as a markdown file with:
   - Critical findings (fix immediately)
   - High priority findings (fix within 24 hours)
   - Medium priority findings (fix within 1 week)
   - Low priority findings (fix in next maintenance cycle)

Do NOT modify any code yet. This is audit only.
```

🛑 **CRITICAL:** If your agent finds API keys hardcoded anywhere in your frontend code, STOP. That is your number one priority to fix before anything else in this guide.

1.2 Threat Model for OpenClaw Deployments

OpenClaw deployments face a specific set of threats that differ from traditional web apps. Understanding these will help you prioritize which hardening steps matter most for your specific setup.

Threat Category	Risk Level	Description	Impact
API Key Theft	CRITICAL	Exposed keys allow unlimited model access	Runaway costs, data exfiltration
Prompt Injection	HIGH	Malicious inputs manipulate model behavior	Data leaks, unauthorized actions
Model Abuse	HIGH	Unauthorized users consuming expensive models	Cost explosion ($1K+/day possible)
Webhook Hijacking	MEDIUM	Intercepted webhooks expose data flows	Data breach, workflow manipulation
Rate Limit Bypass	MEDIUM	Attackers overwhelm your instance	Service disruption, inflated costs
Session Hijacking	MEDIUM	Stolen session tokens grant full access	Account takeover
Data Exfiltration	HIGH	Context/memory data extracted via prompts	Customer data breach

1.3 Security Baseline Score

Before proceeding, score your current deployment against this baseline. Be honest — this is for your own protection. You'll re-score after completing the hardening process to measure improvement.

Security Control	Status	Points
API keys stored in environment variables (not code)	Yes / No	10
HTTPS enforced on all endpoints	Yes / No	10
Rate limiting configured	Yes / No	8
Authentication required for all API routes	Yes / No	10
Model access restricted by user role	Yes / No	8
Webhook signatures validated	Yes / No	7
Logging and monitoring active	Yes / No	7
Input validation on all user-facing endpoints	Yes / No	8
CORS properly configured (not wildcard)	Yes / No	7
Secrets rotated in last 90 days	Yes / No	5
Backup and recovery plan documented	Yes / No	5
Error messages sanitized (no stack traces exposed)	Yes / No	5

✅ **PRO TIP:** Score of 70+ means you have a solid foundation. Below 50 means you have critical gaps. Below 30 — treat this as an emergency.

2. API Key & Secrets Management

This is the single most important section in this entire guide. Your API keys are the keys to the kingdom. If someone gets your Anthropic API key, they can rack up thousands of dollars in charges in hours. If they get your OpenClaw admin key, they own your entire deployment.

2.1 Key Rotation Protocol

If you've been running with the same API keys since deployment, rotate them NOW. Here's the zero-downtime rotation process:

1. Generate a new API key in your provider's dashboard (Anthropic, OpenAI, etc.)
2. Add the new key as a secondary key in your OpenClaw environment configuration
3. Update your OpenClaw deployment to use the new key
4. Verify all model calls are succeeding with the new key (check logs for 5 minutes minimum)

5. Revoke the old key in your provider's dashboard
6. Update any backup configurations or documentation with the new key reference
7. Set a calendar reminder to rotate again in 90 days maximum

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Harden API key and secrets management in my OpenClaw deployment:

1. FIND every location where API keys or secrets are stored or referenced.
   Search for patterns: "sk-ant-", "sk-", "api_key", "secret", "token",
   "password", "ANTHROPIC", "OPENAI" across the entire codebase.

2. MOVE all hardcoded secrets to environment variables:
   - Create or update the .env file with all required secrets
   - Replace every hardcoded secret in source code with
process.env.VARIABLE_NAME
   - Add .env to .gitignore if not already present
   - Create a .env.example file with placeholder values (never real keys)

3. IMPLEMENT a secrets validation module that checks on startup:
   - All required environment variables are present
   - No environment variable is set to a placeholder/example value
   - API keys match expected format patterns
   - Log a clear error and refuse to start if any check fails

4. ADD a .gitignore entry for: .env, .env.local, .env.production,
   *.pem, *.key, and any other sensitive file patterns.

5. CREATE a key rotation script that:
   - Accepts new key values as arguments
   - Updates the environment configuration
   - Triggers a graceful restart of the application
   - Verifies the new key works with a test API call
   - Logs the rotation event with timestamp (not the key value)

6. VERIFY no secrets exist in git history. Run:
   git log --all -p | grep -i "sk-ant-\|sk-\|api_key.*=.*[A-Za-z0-9]"
   Report any findings.

Output all changes as a clear diff showing before/after for each file.
```

⚠️ **WARNING:** If you're using a platform like Railway, Vercel, or Render, use their built-in secrets management. Never paste keys directly into Dockerfiles, docker-compose files, or CI/CD pipeline configs.

2.2 Key Scope Restriction

Most people deploy with full-access API keys. That's like giving every employee the master key to the building. Here's how to implement least-privilege access:

Key Type	Scope	Access Level	Rotation Frequency
Admin Key	Full system configuration	Owner only	60 days
API Key (Production)	Model routing, completions	Application server	90 days
API Key (Development)	Testing, sandbox models only	Development team	30 days
Webhook Secret	Webhook signature validation	Integration endpoints	90 days
Read-Only Key	Dashboard, logs, monitoring	Support/monitoring team	120 days

2.3 Secrets Scanning

Set up automated scanning to catch accidentally committed secrets. This has saved countless deployments from catastrophic breaches.

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Set up automated secrets scanning for my OpenClaw repository:

1. CREATE a pre-commit hook script (.husky/pre-commit or
   .git/hooks/pre-commit) that scans staged files for:
   - API key patterns (sk-ant-*, sk-*, Bearer *, etc.)
   - Common secret variable names with values assigned
   - Private key file contents (BEGIN RSA PRIVATE KEY, etc.)
   - High-entropy strings that look like tokens (40+ char alphanumeric)
   Block the commit if any are found.

2. CREATE a .secrets-patterns file with regex patterns to scan for:
   sk-ant-[a-zA-Z0-9]{20,}
   sk-[a-zA-Z0-9]{20,}
   AKIA[0-9A-Z]{16}
   ghp_[a-zA-Z0-9]{36}
   password\s*=\s*['"][^'"]+['"]
   secret\s*=\s*['"][^'"]+['"]

3. ADD a CI/CD step that runs the same scan on every pull request.
   If running GitHub Actions, create .github/workflows/secrets-scan.yml
   If running another CI, provide equivalent config.

4. SCAN the entire git history for leaked secrets:
   - Search all commits for the patterns above
   - Output a report of any findings with commit hash, file, and line
   - Provide instructions for cleaning git history if secrets are found

5. Ensure the scanning does NOT flag .env.example files or test fixtures
   that contain obviously fake placeholder values.
```

```
Output the complete hook script, CI config, and pattern file.
```

✅ **PRO TIP:** Run the secrets scan against your entire repo history right now. You may have committed keys months ago that are still in your git history even if you deleted them from the current code.

3. Authentication & Access Control

A deployed OpenClaw instance needs proper authentication on every single endpoint. If you deployed with default settings, there's a good chance some of your routes are publicly accessible. Let's fix that.

3.1 Authentication Layer Setup

Every request to your OpenClaw instance should be authenticated. No exceptions. Here's the recommended authentication stack:

JWT Token Configuration

```
JWT_SECRET=<64-char-cryptographically-random-string>
JWT_EXPIRY=15m          # Short-lived access tokens
JWT_REFRESH_EXPIRY=7d   # Refresh tokens with longer life
JWT_ALGORITHM=RS256     # Use asymmetric signing for production
JWT_ISSUER=your-openclaw-instance.com
```

🛑 **CRITICAL:** Never use HS256 in production if multiple services need to verify tokens. HS256 uses a shared secret — if any verifying service is compromised, the attacker can forge tokens. Use RS256 (asymmetric) so only your auth service holds the signing key.

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement authentication hardening on my OpenClaw deployment:

1. AUDIT all API routes and identify which ones currently lack
   authentication. List every unprotected route.

2. CREATE an authentication middleware that:
   - Validates JWT tokens on every request (except health check)
   - Checks token expiration and rejects expired tokens
   - Validates the token issuer and audience claims
   - Extracts user role from the token for authorization checks
   - Returns 401 for missing/invalid tokens (no detailed error info)
   - Logs failed authentication attempts with IP, timestamp, and reason
```

```
3. IMPLEMENT token generation with:
   - RS256 algorithm (generate RSA key pair if not present)
   - 15-minute access token expiry
   - 7-day refresh token expiry
   - Include user ID, role, and permissions in the payload
   - Never include sensitive data (email, name) in the token

4. ADD a refresh token endpoint that:
   - Accepts a valid refresh token
   - Issues a new access token
   - Rotates the refresh token (one-time use)
   - Invalidates the old refresh token immediately
   - Rejects refresh if the user account is disabled

5. APPLY the auth middleware to ALL routes except:
   - GET /health (returns minimal status only)
   - POST /auth/login
   - POST /auth/refresh

6. GENERATE secure RSA keys for JWT signing:
   - 2048-bit minimum key length
   - Store private key in environment variable or secrets manager
   - Output the public key for token verification services

Test that unauthenticated requests to protected routes return 401.
```

3.2 Role-Based Access Control (RBAC)

Implement tiered access so different users can only reach what they need. This is especially critical for the model routing layer where cost implications are massive.

Role	Model Access	Admin Panel	API Keys	Billing
Owner	All models (Opus, Sonnet, Haiku)	Full access	Create/Rotate/Delete	Full access
Admin	All models	Read + Config	View/Rotate	View only
Developer	Sonnet + Haiku only	Read only	View own keys	No access
API Consumer	As configured per key	No access	Use assigned key	No access
Viewer	No direct access	Read only	No access	No access

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement Role-Based Access Control (RBAC) for my OpenClaw deployment:

1. CREATE a roles and permissions system with these roles:
```

```
   - owner: Full access to everything
   - admin: All models, config access, can rotate keys
   - developer: Sonnet + Haiku only, read-only admin, own keys only
   - api_consumer: Only models assigned to their API key
   - viewer: Read-only dashboard, no model access

2. CREATE an authorization middleware that:
   - Extracts the user role from the JWT token
   - Checks if the role has permission for the requested action
   - Returns 403 Forbidden for unauthorized access attempts
   - Logs all authorization failures with user ID, role, and action

3. IMPLEMENT model-tier access control:
   - Define which models each role can access
   - Block requests to unauthorized models BEFORE they reach the API
   - Return a clear error: "Your role does not have access to this model"
   - Log model access attempts by role for cost tracking

4. ADD per-role rate limits:
   - owner: 100 req/min, no daily cost cap
   - admin: 60 req/min, $500/day cost cap
   - developer: 30 req/min, $100/day cost cap
   - api_consumer: 20 req/min, $50/day cost cap (configurable per key)
   - viewer: 10 req/min (dashboard only, no model calls)

5. CREATE an admin endpoint to manage user roles:
   - POST /admin/users/:id/role - Update user role (owner only)
   - GET /admin/users - List all users and their roles
   - Require owner role for all role management operations

Ensure the RBAC checks happen AFTER authentication but BEFORE
any business logic or external API calls.
```

3.3 Session Management

Proper session handling prevents token replay attacks and unauthorized session persistence.

- Implement absolute session timeouts (maximum 24 hours regardless of activity)
- Use sliding window expiration for active sessions (15-minute refresh cycle)
- Invalidate all sessions on password change or key rotation
- Store session tokens in httpOnly, secure, SameSite=Strict cookies (never localStorage)
- Implement session binding to IP address with tolerance for mobile network switching
- Log all session creation and destruction events for audit trail
- Limit concurrent sessions per user (recommend maximum of 3)

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Harden session management for my OpenClaw deployment:

1. CONFIGURE cookie-based token storage:
```

```
   - Set httpOnly: true (prevents JavaScript access)
   - Set secure: true (HTTPS only)
   - Set sameSite: 'strict' (prevents CSRF)
   - Set path: '/' and appropriate domain

2. IMPLEMENT session controls:
   - Maximum 3 concurrent sessions per user
   - Absolute timeout of 24 hours
   - Sliding window of 15 minutes for active sessions
   - On new session beyond limit, invalidate the oldest session
   - Invalidate ALL sessions when password changes or key rotates

3. ADD session tracking that stores:
   - Session ID, user ID, creation time, last activity time
   - IP address and user agent at session creation
   - Alert if session IP changes dramatically (different country)

4. CREATE a session management endpoint:
   - GET /auth/sessions - List active sessions for current user
   - DELETE /auth/sessions/:id - Revoke a specific session
   - DELETE /auth/sessions - Revoke all sessions (force re-login)

5. REMOVE any tokens stored in localStorage or sessionStorage.
   Migrate to httpOnly cookies exclusively.

Test: Verify tokens are not accessible via document.cookie in browser
console (httpOnly prevents this).
```

4. Network Security & Transport

Your OpenClaw instance is communicating sensitive data — API keys, user prompts, model responses, potentially PII. Every bit of that traffic needs to be encrypted and controlled.

4.1 TLS/HTTPS Enforcement

This should already be in place, but verify it's configured correctly. Common mistakes include allowing HTTP fallback or using outdated TLS versions.

- Enforce TLS 1.2 minimum (TLS 1.3 preferred)
- Disable TLS 1.0 and 1.1 completely
- Use strong cipher suites only (ECDHE + AES-GCM)
- Enable HSTS with a minimum max-age of 31536000 (1 year)
- Include includeSubDomains in HSTS header
- Set up automatic certificate renewal (Let's Encrypt with certbot)
- Redirect ALL HTTP traffic to HTTPS with 301 permanent redirects

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Harden TLS and HTTPS configuration for my OpenClaw deployment:

1. CHECK current TLS configuration:
   - What TLS versions are enabled?
   - What cipher suites are in use?
   - Is HTTP-to-HTTPS redirect configured?
   - Is HSTS header present and correctly configured?

2. If using NGINX, CREATE or UPDATE the SSL config:
   ssl_protocols TLSv1.2 TLSv1.3;
   ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:
               ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384;
   ssl_prefer_server_ciphers off;
   ssl_session_timeout 1d;
   ssl_session_cache shared:SSL:10m;
   ssl_session_tickets off;
   add_header Strict-Transport-Security
     "max-age=31536000; includeSubDomains; preload" always;

3. If using Node.js directly, configure TLS options:
   - Set minVersion: 'TLSv1.2'
   - Reject unauthorized certificates on outbound calls
   - Enable certificate pinning for model provider APIs

4. REDIRECT all HTTP (port 80) to HTTPS (port 443) with 301 status.

5. SET UP automatic certificate renewal with certbot or equivalent.
   Test renewal with --dry-run before going live.

6. VERIFY the configuration by testing with:
   curl -I http://yourdomain.com (should get 301 to https)
   curl -I https://yourdomain.com (should see HSTS header)

Report current state and all changes made.
```

4.2 CORS Configuration

A wildcard CORS policy is one of the most common security mistakes. It effectively tells browsers that ANY website can make requests to your OpenClaw instance.

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Lock down CORS configuration on my OpenClaw deployment:

1. FIND the current CORS configuration. Check for:
   - Access-Control-Allow-Origin: * (this is the problem)
   - Any CORS middleware configuration
   - Custom headers that bypass CORS

2. REPLACE wildcard CORS with a strict whitelist:
```

```
   - Only allow specific origins that need access
   - Configure allowed methods: GET, POST, OPTIONS only
   - Configure allowed headers: Authorization, Content-Type only
   - Set Access-Control-Max-Age: 86400 (cache preflight for 24 hours)
   - Do NOT allow Access-Control-Allow-Credentials with wildcard origin

3. IMPLEMENT dynamic origin validation:
   - Store allowed origins in environment variable as comma-separated list
   - Validate the Origin header against the whitelist
   - Return the matched origin (not wildcard) in the response
   - Return no CORS headers for unmatched origins
   - Log rejected CORS requests for monitoring

4. ADD the following configuration:
   ALLOWED_ORIGINS=https://yourdomain.com,https://app.yourdomain.com
   ALLOWED_METHODS=GET,POST,OPTIONS
   ALLOWED_HEADERS=Authorization,Content-Type

5. TEST by making a fetch request from an unauthorized origin.
   Verify it is blocked by the browser's CORS policy.

Output the CORS middleware code and configuration.
```

4.3 Firewall & IP Restrictions

If your OpenClaw admin panel is accessible from any IP address, you're one leaked password away from a full compromise. Lock it down.

Endpoint Category	Access Policy	Implementation
Admin Panel (/admin/*)	Whitelist IP only	Firewall rule + application-level check
API Endpoints (/api/v1/*)	Authenticated + rate limited	API key + IP-based rate limiting
Webhook Receivers (/webhooks/*)	Source IP + signature	Provider IP ranges + HMAC validation
Health Check (/health)	Public	No authentication required (returns minimal info)
Documentation (/docs)	Internal only or disabled	Remove or restrict in production

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement IP restrictions and firewall rules for my OpenClaw deployment:

1. CREATE IP-based access control middleware:
   - Accept a whitelist of IP addresses/CIDR ranges from env vars
   - Apply IP whitelist to all /admin/* routes
   - Log all blocked access attempts with IP, path, and timestamp
```

```
   - Support both IPv4 and IPv6
   - Handle X-Forwarded-For header for proxied deployments

2. CONFIGURE the following access policies:
   ADMIN_ALLOWED_IPS=<your-ip>,<team-vpn-ip>
   Apply to: /admin/*, /api/v1/admin/*

3. DISABLE or restrict these endpoints in production:
   - /docs, /swagger, /api-docs (API documentation)
   - /debug/*, /test/* (debug endpoints)
   - Any endpoint that exposes system information

4. IMPLEMENT webhook source validation:
   - Verify webhook requests come from expected IP ranges
   - Validate webhook signatures (HMAC-SHA256)
   - Reject webhooks that fail either check

5. ENSURE the health check endpoint (/health) returns ONLY:
   { "status": "ok", "timestamp": "<iso-date>" }
   No version numbers, no uptime, no internal details.

6. ADD security headers to all responses:
   X-Content-Type-Options: nosniff
   X-Frame-Options: DENY
   X-XSS-Protection: 0
   Content-Security-Policy: default-src 'self'
   Referrer-Policy: strict-origin-when-cross-origin
   Permissions-Policy: camera=(), microphone=(), geolocation=()
   Cache-Control: no-store, no-cache, must-revalidate

Output all middleware code, config changes, and verification steps.
```

🛑 **CRITICAL:** Disable or remove any Swagger/API documentation endpoints in production. These are a roadmap for attackers showing every available endpoint, parameter, and data type.

5. Rate Limiting & Abuse Prevention

Without rate limiting, a single bad actor (or a bug in your own code) can drain your entire monthly budget in hours. This is especially critical with the three-tier model strategy where Opus calls are expensive.

5.1 Multi-Layer Rate Limiting

Implement rate limiting at multiple levels for defense in depth. A single rate limiter at the application level is not enough.

Layer	Tool	Limit	Window
CDN/Edge	Cloudflare / AWS WAF	1000 req/min per IP	1 minute sliding
Reverse Proxy	Nginx rate limiting	100 req/min per IP	1 minute fixed
Application	OpenClaw built-in / custom	60 req/min per API key	1 minute sliding
Model-Specific	Custom middleware	Varies by tier	1 hour rolling
Cost-Based	Custom middleware	$50/day per key default	24 hour rolling

🤖 AGENT PROMPT — Copy & paste this to your AI agent:

```
Implement comprehensive rate limiting for my OpenClaw deployment:

1. CREATE a multi-layer rate limiting system:

   Layer 1 - IP-based (all requests):
   - 100 requests per minute per IP address
   - Use sliding window algorithm
   - Return 429 Too Many Requests with Retry-After header

   Layer 2 - API key-based (authenticated requests):
   - 60 requests per minute per API key
   - Use token bucket algorithm for burst tolerance
   - Include rate limit headers: X-RateLimit-Limit,
     X-RateLimit-Remaining, X-RateLimit-Reset

   Layer 3 - Model-specific (model API calls):
   - Haiku: 60 req/min, 100K tokens/min
   - Sonnet: 30 req/min, 50K tokens/min
   - Opus: 10 req/min, 20K tokens/min

2. USE Redis for rate limit state (distributed, fast):
   - Store counters with TTL matching the window
   - Handle Redis connection failures gracefully (fail open with logging)
   - Include key prefix to avoid collisions

3. IMPLEMENT progressive responses:
   - First limit hit: Return 429 with Retry-After
   - 10+ limit hits in 5 min: Extend block to 5 minutes
   - 50+ limit hits in 15 min: Block IP for 1 hour, alert admin

4. ADD rate limit bypass for:
   - Health check endpoint
   - Internal monitoring requests (by IP or header)
   - DO NOT bypass for admin routes

5. LOG all rate limit events with: IP, API key (last 4 chars only),
   endpoint, current count, limit, and timestamp.
```

```
Output the complete rate limiting middleware and Redis configuration.
```

5.2 Cost Circuit Breakers

This is the feature that will save your business. Implement automatic shutoffs when spending exceeds thresholds. Think of it like a financial fuse box.

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement cost circuit breakers for my OpenClaw deployment:

1. CREATE a cost tracking module that:
   - Calculates cost per request based on model and token count
   - Uses these rates (update if your rates differ):
     Haiku: $0.25 input / $1.25 output per million tokens
     Sonnet: $3 input / $15 output per million tokens
     Opus: $15 input / $75 output per million tokens
   - Tracks costs per API key, per user, and globally
   - Stores running totals in Redis with 24-hour TTL

2. IMPLEMENT circuit breaker thresholds:
   WARNING ($100/day):
   - Send alert to configured notification channel
   - Log the warning with current spend breakdown by model
   - Continue processing requests normally

   SOFT LIMIT ($250/day):
   - Automatically downgrade: Opus requests -> use Sonnet instead
   - Send urgent alert with model-by-model breakdown
   - Log all downgraded requests

   HARD LIMIT ($500/day):
   - Block ALL non-Haiku requests
   - Send critical alert
   - Return 503 with message: "Cost limit reached, only economy
     model available"

   EMERGENCY SHUTOFF ($1000/day):
   - Block ALL model API calls
   - Send emergency alert (email + SMS if configured)
   - Return 503 with message: "Service temporarily suspended"
   - Require manual reset by owner role

3. ADD notification configuration:
   CIRCUIT_BREAKER_WEBHOOK=https://hooks.slack.com/services/xxx
   CIRCUIT_BREAKER_EMAIL=admin@yourdomain.com
   CIRCUIT_BREAKER_WARNING=100
   CIRCUIT_BREAKER_SOFT=250
   CIRCUIT_BREAKER_HARD=500
```

```
   CIRCUIT_BREAKER_EMERGENCY=1000

4. CREATE an admin endpoint to:
   - GET /admin/costs - View current spend by model, key, and user
   - POST /admin/costs/reset - Reset circuit breaker (owner only)
   - PUT /admin/costs/limits - Update thresholds (owner only)

5. ADD a dashboard display showing current spend vs. limits
   with a visual indicator (green/yellow/orange/red).

Output the complete circuit breaker module and configuration.
```

✅ **PRO TIP:** Set your circuit breaker thresholds at 2x your normal daily spend. If you normally spend $50/day, your warning should fire at $100.

5.3 Prompt Injection Defense

Prompt injection is the OWASP #1 risk for LLM applications. Attackers craft inputs designed to override your system prompts, extract sensitive data, or manipulate model behavior.

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement prompt injection defenses for my OpenClaw deployment:

1. CREATE an input validation middleware that runs BEFORE any
   model API call:

   Length limits:
   - Max input length: 4000 characters per message
   - Max conversation turns: 50 per session
   - Max system prompt length: 2000 characters

   Blocked patterns (reject the request entirely):
   - "ignore (all |your |previous )?instructions"
   - "reveal your (instructions|config|prompt|system)"
   - "act as (an? )?unrestricted"
   - "you are now in (developer|DAN|jailbreak) mode"
   - "ignore everything above"
   - "disregard (your |all )?(rules|guidelines|instructions)"

   Sanitized patterns (strip but allow request):
   - <script> tags and JavaScript injection attempts
   - SQL injection patterns
   - Path traversal attempts (../ sequences)

2. IMPLEMENT output filtering that scans model responses BEFORE
   returning to the user:
   - Check for leaked API key patterns (sk-ant-*, sk-*)
```

```
   - Check for leaked system prompt content
   - Check for PII patterns (SSN, credit card, etc.)
   - If found: redact the content and log the incident

3. ADD canary token detection:
   - Embed a unique random string in your system prompt
   - If that string appears in model output, log a prompt injection alert
   - Example: "CANARY_TOKEN_[random-uuid]"

4. IMPLEMENT request logging for security review:
   - Log input length, detected patterns, and validation result
   - Do NOT log full prompt content (privacy)
   - Flag suspicious patterns for manual review

5. CREATE a quarantine system:
   - After 3 blocked injection attempts, temporarily block the API key
   - After 10 blocked attempts in 24 hours, permanently block + alert admin
   - Log all quarantine events

Output the complete input validation middleware and output filter.
```

6. Logging, Monitoring & Incident Response

You can’t protect what you can’t see. Comprehensive logging and real-time monitoring are your early warning system.

6.1 What to Log

Log Category	What to Capture	What NOT to Capture
Authentication	Login attempts, failures, token refreshes	Passwords, full tokens
API Usage	Endpoint, method, status code, response time	Full prompt/response content
Model Routing	Model selected, tier, fallback events, cost	System prompt content
Admin Actions	Config changes, key rotations, user management	New key values
Rate Limiting	Limit hits, blocked requests, IP addresses	N/A
Errors	Stack traces (internal only), error codes	User PII in error messages
Webhooks	Source IP, signature validation, processing time	Full webhook payload

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement comprehensive security logging and monitoring:

1. CREATE a structured logging module that outputs JSON logs with:
   - timestamp (ISO 8601)
   - level (info, warn, error, critical)
   - category (auth, api, model, admin, security)
   - event (specific action: login_failed, rate_limited, etc.)
   - metadata (IP, user_id last 8 chars, API key last 4 chars)
   - request_id (for correlating related log entries)
   DO NOT log: passwords, full tokens, full API keys, prompt content

2. IMPLEMENT security-specific logging for:
   - Every failed login attempt (IP, attempted username, timestamp)
   - Every rate limit hit (IP, key, endpoint, current count)
   - Every authorization failure (user, role, attempted action)
   - Every circuit breaker event (threshold, current spend, action taken)
   - Every prompt injection detection (pattern matched, action taken)
   - Every admin action (who, what, when, from where)

3. CREATE alert rules that trigger notifications:
   CRITICAL (immediate notification):
   - 10+ failed logins from same IP in 5 minutes
   - Circuit breaker emergency shutoff triggered
   - New API key detected that wasn't created through admin panel
   - Admin action from unrecognized IP

   HIGH (notify within 15 minutes):
   - 5+ prompt injection attempts from same key
   - Cost spike > 3x normal hourly rate
   - Error rate > 10% for 5+ minutes

   MEDIUM (daily digest):
   - Rate limit hits by key/IP summary
   - Failed auth attempts summary
   - Cost summary by model and key

4. ADD a log rotation policy:
   - Rotate logs daily
   - Compress logs older than 1 day
   - Retain security logs for 1 year
   - Retain general logs for 30 days
   - Send logs to external service if configured (ELK, Datadog, etc.)

5. CREATE a /admin/logs endpoint for real-time log viewing:
   - Filter by category, level, timerange
   - Search by IP, user ID, or request ID
   - Require admin role

Output the logging module, alert configuration, and log rotation setup.
```

6.2 Incident Response Playbook

When (not if) a security incident occurs, you need a documented response plan.

Severity	Examples	Response Time	Actions
P0 - Critical	Key compromise, active data breach	< 15 min	Rotate all keys, enable kill switch
P1 - High	Cost spike, injection detected	< 1 hour	Block source, investigate logs
P2 - Medium	Rate limit breaches, unusual traffic	< 4 hours	Monitor, adjust limits
P3 - Low	Single failed auth, config drift	< 24 hours	Log and review

🤖 AGENT PROMPT — Copy & paste this to your AI agent:

```
Create an automated incident response system for my OpenClaw deployment:

1. CREATE an incident detection module that automatically detects:
   - API key compromise (key used from unusual IP/geography)
   - Brute force attacks (10+ failed logins in 5 min from same IP)
   - Cost anomalies (spend > 3x rolling 7-day average)
   - Data exfiltration attempts (unusually large responses)
   - Service degradation (error rate > 10%, latency > 5s avg)

2. IMPLEMENT automatic response actions:

   For detected key compromise:
   - Immediately disable the suspected key
   - Block the source IP
   - Send critical alert with all details
   - Log the full incident timeline

   For brute force:
   - Block the attacking IP for 24 hours
   - If distributed (5+ IPs), enable CAPTCHA on login
   - Alert admin with attack details

   For cost anomaly:
   - Activate soft circuit breaker
   - Alert with spend breakdown
   - Auto-downgrade model tier if needed

3. CREATE a /admin/incidents endpoint:
   - GET /admin/incidents - List all incidents
   - GET /admin/incidents/:id - Full incident details with timeline
   - POST /admin/incidents/:id/resolve - Mark as resolved with notes
   - Require admin role
```

```
4. ADD a kill switch endpoint:
   - POST /admin/killswitch - Immediately block all API calls
   - Require owner role + confirmation parameter
   - Log who activated it and when
   - POST /admin/killswitch/release - Re-enable service

Output the incident detection module, response actions, and API endpoints.
```

7. Data Protection & Privacy

Your OpenClaw instance processes user prompts, model responses, and potentially sensitive business data. Protect it accordingly.

7.1 Data Encryption

Encryption at Rest

- Enable full-disk encryption on all servers hosting OpenClaw
- Use AES-256 encryption for database fields containing secrets
- Encrypt backup files before storing (GPG or age encryption)
- Use encrypted environment variable stores (Vault, AWS Secrets Manager, Doppler)

Encryption in Transit

- TLS 1.2+ on all connections (covered in Section 4)
- Use SSL/TLS for database connections (sslmode=require)
- Encrypt Redis connections with TLS (use rediss:// protocol)
- Verify certificate chains on all outbound API calls

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Implement data protection and encryption for my OpenClaw deployment:

1. AUDIT what data is currently stored and where:
   - Database tables/collections with sensitive data
   - Log files that may contain sensitive content
   - Cache entries (Redis) with sensitive data
   - Temporary files or uploads

2. IMPLEMENT field-level encryption for sensitive database fields:
   - API key storage: Store only bcrypt/argon2 hashes, never plaintext
   - User tokens: Encrypt with AES-256-GCM before storing
   - Webhook secrets: Encrypt at rest
   - Create an encryption utility module with encrypt/decrypt functions
   - Store the encryption key in environment variable, NOT in code

3. CONFIGURE encrypted database connections:
   - Add ?sslmode=require to PostgreSQL connection string
```

```
   - Use rediss:// (with double s) for Redis connections
   - Verify certificates on connection (reject self-signed in production)

4. IMPLEMENT data retention and auto-purge:
   - Delete prompt/response logs older than 7 days automatically
   - Purge expired session data daily
   - Anonymize usage analytics after 90 days
   - Create a cron job or scheduled task for automatic cleanup

5. ADD PII detection on inputs AND outputs:
   - Scan for SSN patterns: \b\d{3}-\d{2}-\d{4}\b
   - Scan for credit card patterns: \b\d{4}[- ]?\d{4}[- ]?\d{4}[- ]?\d{4}\b
   - Scan for email addresses
   - Scan for phone numbers
   - Action: Redact in logs, optionally warn user

6. VERIFY all backup processes encrypt data before storage:
   - Database dumps: pipe through gpg or age before saving
   - Config backups: encrypt and store separately from data backups
   - Test decryption of a recent backup to verify it works

Output all encryption utilities, migration scripts, and cron job configs.
```

8. Deployment & Infrastructure Hardening

Your OpenClaw instance is only as secure as the infrastructure it runs on.

8.1 Container Security (Docker)

If you're running OpenClaw in Docker (most deployments), follow these container hardening best practices:

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Harden the Docker/container configuration for my OpenClaw deployment:

1. UPDATE the Dockerfile with security best practices:
   - Use a specific image tag (e.g., node:20.11-alpine), NEVER 'latest'
   - Create and use a non-root user:
     RUN addgroup -g 1001 openclaw && \
         adduser -u 1001 -G openclaw -s /bin/sh -D openclaw
     USER openclaw
   - Use multi-stage build to minimize final image size
   - Copy only production dependencies (npm ci --only=production)
   - Remove unnecessary tools (curl, wget, etc.) from final image

2. UPDATE docker-compose.yml with security settings:
   security_opt:
```

```
      - no-new-privileges:true
    read_only: true
    tmpfs:
      - /tmp
    cap_drop:
      - ALL
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '1.0'

3. ENSURE no secrets in Docker configuration:
   - Remove any ENV lines with actual secret values
   - Use docker secrets or env_file pointing to .env (not committed)
   - Remove any COPY commands that include .env files
   - Check docker-compose.yml for hardcoded passwords

4. ADD a .dockerignore file that excludes:
   .env, .env.*, .git, node_modules, *.pem, *.key,
   .github, tests, docs, *.md, .vscode

5. SCAN the Docker image for vulnerabilities:
   - Run: docker scout cves <image-name> (or trivy, grype)
   - Report any HIGH or CRITICAL vulnerabilities
   - Provide remediation steps for each

6. CONFIGURE container health checks:
   healthcheck:
     test: ["CMD", "wget", "--spider", "-q",
"http://localhost:3000/health"]
     interval: 30s
     timeout: 10s
     retries: 3
     start_period: 40s

Output the hardened Dockerfile, docker-compose.yml, and .dockerignore.
```

8.2 CI/CD Pipeline Security

Your deployment pipeline is a high-value target. If an attacker compromises your CI/CD, they can inject malicious code into every deployment.

- Require code review approval before any merge to production branches
- Run automated security scans (SAST, dependency audit) on every pull request
- Use short-lived deployment tokens, never long-lived credentials
- Pin all CI/CD action versions to specific commit SHAs (supply chain protection)
- Enable branch protection: no force pushes, no deletions, required status checks
- Audit CI/CD access quarterly — remove anyone who doesn't need deployment access
- Log all deployments with who triggered them, what changed, and when

8.3 Dependency Management

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Harden dependency management for my OpenClaw deployment:

1. RUN a full dependency audit:
   npm audit
   Report all vulnerabilities by severity (critical, high, medium, low).

2. FIX what can be auto-fixed:
   npm audit fix
   For remaining vulnerabilities, provide manual remediation steps.

3. VERIFY lockfile integrity:
   - Ensure package-lock.json exists and is committed
   - Verify npm ci works without errors
   - Check for any packages not in the lockfile

4. CHECK for outdated packages:
   npx npm-check-updates
   Flag any packages more than 2 major versions behind.

5. CREATE a GitHub Actions workflow (or equivalent CI config) that:
   - Runs npm audit on every pull request
   - Fails the build if HIGH or CRITICAL vulnerabilities found
   - Runs weekly scheduled scans of dependencies
   - Creates issues for new vulnerabilities found

6. ADD to package.json scripts:
   "security:audit": "npm audit --audit-level=high",
   "security:check": "npx npm-check-updates",
   "preinstall": "npx npm-force-resolutions"

7. PIN all GitHub Actions to commit SHAs instead of tags:
   WRONG: uses: actions/checkout@v4
   RIGHT: uses: actions/checkout@<full-commit-sha>

Output the audit results, fixed packages, and CI workflow file.
```

⚠️ **WARNING:** Never run npm install in production. Always use npm ci which installs exactly what's in your lockfile. npm install can resolve to different versions and introduce untested code.

9. Backup & Disaster Recovery

Security hardening isn't complete without a recovery plan. If everything goes wrong — ransomware, data corruption, total infrastructure failure — how fast can you get back online?

9.1 Backup Strategy

Component	Frequency	Retention	Storage
Database (full)	Daily	30 days	Encrypted offsite (S3/GCS)
Database (incremental)	Every 6 hours	7 days	Same as full backup
Configuration files	On every change	90 days	Version control (encrypted)
Environment variables	On every change	Indefinite	Encrypted vault backup
Security logs	Real-time stream	1 year	SIEM or log aggregation
Docker images/configs	On every build	Last 10 builds	Container registry

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Set up automated backup and disaster recovery for my OpenClaw deployment:

1. CREATE an automated backup script that:

   Database backup (daily + every 6 hours incremental):
   - Full pg_dump (or mongodump) with compression
   - Encrypt the backup with gpg/age before storage
   - Upload to remote storage (S3, GCS, or equivalent)
   - Verify the upload succeeded
   - Delete local backup file after upload
   - Retain 30 days of daily, 7 days of incremental

   Configuration backup (on change):
   - Export all environment variables (redact actual values, keep
structure)
   - Backup nginx/proxy configs
   - Backup Docker configs
   - Store in encrypted git repository

2. CREATE a recovery script that:
   - Downloads the most recent backup from remote storage
   - Decrypts the backup
   - Restores to a specified database
   - Verifies data integrity (row counts, checksums)
   - Runs a health check suite against the restored instance
   - Reports recovery time and any issues

3. CREATE a backup verification cron job that runs daily:
```

```
    - Restore latest backup to a test database
    - Run basic data integrity checks
    - Run API health checks against test instance
    - Send success/failure notification
    - Tear down test instance after verification

4. SET UP the cron schedule:
    # Full backup daily at 2 AM
    0 2 * * * /opt/openclaw/scripts/backup.sh full
    # Incremental every 6 hours
    0 */6 * * * /opt/openclaw/scripts/backup.sh incremental
    # Verify backup daily at 4 AM
    0 4 * * * /opt/openclaw/scripts/verify-backup.sh

5. DOCUMENT the recovery procedure as a runbook:
    - Step-by-step instructions anyone on the team can follow
    - Include estimated recovery time (target: under 4 hours)
    - List all required credentials and where to find them
    - Include rollback procedures if recovery fails

Output the backup script, recovery script, verification script,
cron configuration, and recovery runbook.
```

✅ **PRO TIP:** A backup you've never tested is not a backup. It's a hope. Schedule recovery tests quarterly and document the results.

10. Ongoing Security Maintenance

Security is not a one-time project. It's an ongoing discipline. Here's your recurring maintenance schedule.

10.1 Security Maintenance Calendar

Frequency	Task	Owner
Daily	Review monitoring dashboard and alerts	On-call / automated
Weekly	Review failed auth logs and blocked requests	Security lead
Weekly	Check dependency vulnerability reports	Developer
Monthly	Rotate API keys approaching 90-day age	Admin
Monthly	Review and update rate limiting thresholds	Admin
Monthly	Audit user accounts and access levels	Admin
Quarterly	Full security assessment (Section 1 checklist)	Security lead

Frequency	Task	Owner
Quarterly	Disaster recovery test	Ops team
Quarterly	Review incident response playbook	Security lead
Annually	Full penetration test / security audit	External consultant
Annually	Review entire security configuration	Admin

🤖 AGENT PROMPT — Copy & paste this to your AI agent:

```
Create automated security maintenance tasks for my OpenClaw deployment:

1. CREATE a weekly security report script that generates:
   - Failed authentication attempts (count, top IPs, trends)
   - Rate limit violations (count, top offenders, patterns)
   - API cost summary (by model, by key, vs. previous week)
   - Circuit breaker events (any triggers, current thresholds)
   - Dependency vulnerability status (new CVEs since last week)
   - Key age report (days until rotation due for each key)
   Output as formatted email or Slack message.

2. CREATE a monthly maintenance checklist script that checks:
   - Which API keys are older than 80 days (approaching 90-day rotation)
   - Which user accounts have been inactive for 30+ days
   - Whether rate limit thresholds match current usage patterns
   - Whether backup verification has passed in the last 7 days
   - Whether any security patches are available for dependencies
   Flag anything that needs attention.

3. CREATE a key rotation reminder system:
   - Track creation date of all API keys
   - Send warning at 80 days: "Key rotation due in 10 days"
   - Send urgent alert at 90 days: "Key rotation overdue"
   - Send critical alert at 100 days: "OVERDUE - rotate immediately"

4. ADD a /admin/security-status endpoint that returns:
   - Overall security score (based on Section 1.3 checklist)
   - Key rotation status (all keys with age)
   - Rate limit configuration summary
   - Circuit breaker status
   - Last backup date and verification status
   - Dependency vulnerability count
   Require admin role.

Output all scripts, cron configurations, and the status endpoint.
```

11. Master Security Hardening Checklist

Work through this after completing the guide. Check off each item as you implement it. This is your at-a-glance reference for the entire hardening process.

#	Security Control	Section	Status
1	Complete pre-hardening security audit (agent prompt)	1.1	☐
2	Score baseline and document starting point	1.3	☐
3	Rotate all API keys and move to env vars	2.1	☐
4	Implement key scope restrictions	2.2	☐
5	Set up automated secrets scanning	2.3	☐
6	Configure JWT authentication on all endpoints	3.1	☐
7	Implement RBAC with model tier restrictions	3.2	☐
8	Harden session management	3.3	☐
9	Enforce TLS 1.2+ and HSTS	4.1	☐
10	Lock down CORS (no wildcards)	4.2	☐
11	Apply IP restrictions to admin panel	4.3	☐
12	Add all security headers	4.3	☐
13	Implement multi-layer rate limiting	5.1	☐
14	Configure cost circuit breakers	5.2	☐
15	Deploy prompt injection defenses	5.3	☐
16	Set up comprehensive security logging	6.1	☐
17	Build monitoring dashboards and alerts	6.1	☐
18	Create incident response automation	6.2	☐
19	Implement data encryption (rest + transit)	7.1	☐
20	Deploy PII detection and auto-purge	7.1	☐
21	Harden Docker/container configuration	8.1	☐
22	Secure CI/CD pipeline	8.2	☐
23	Audit and pin all dependencies	8.3	☐
24	Set up automated encrypted backups	9.1	☐
25	Complete first recovery test	9.1	☐
26	Establish automated maintenance tasks	10.1	☐
27	Re-score using Section 1.3 baseline	1.3	☐
28	Document all changes made during hardening	All	☐

🤖 **AGENT PROMPT — Copy & paste this to your AI agent:**

```
Run a final security verification on my OpenClaw deployment:

1. CHECK all the hardening steps have been applied:
   - Scan for any remaining hardcoded secrets
   - Verify all endpoints require authentication
   - Test CORS by making a request from an unauthorized origin
   - Verify rate limiting is active (make rapid requests, confirm 429)
   - Confirm circuit breakers are configured (check /admin/costs)
   - Test TLS configuration (attempt HTTP, verify redirect)
   - Verify security headers are present on all responses
   - Confirm Docker is running as non-root user
   - Verify backups are running and latest verification passed

2. RUN a simulated attack test:
   - Send 20 rapid requests to test rate limiting
   - Send a known injection pattern to test input validation
   - Make an unauthenticated request to a protected endpoint
   - Attempt to access /admin from a non-whitelisted IP
   - Send a request with an expired/invalid JWT token
   All should be properly blocked and logged.

3. GENERATE a final security report with:
   - Security score (Section 1.3 re-assessment)
   - Summary of all changes made
   - Any remaining vulnerabilities or risks
   - Recommendations for ongoing monitoring
   - Date of next scheduled security review

Output the complete verification results and final security report.
```

Built for the SPRINT Community by ScaleUP Media
Questions? Drop them in the OpenClaw Mastery Slack channel.