Mend.io

PRACTITIONER GUIDE

A PRACTICAL FRAMEWORK

Securing AI agents, MCP servers & LLM apps

How to see, fix, and protect the agentic AI attack surface — from discovery to runtime — with the checklists and templates to start today.

SEE

Discover the agentic attack surface

FIX

Triage AI finding volumes

PROTECT

Guardrails and policy at runtime

INTRODUCTION

The layer governance alone can't reach

Agents, MCP integrations, and LLM-powered applications are entering codebases faster than most security programs can track them. Their behavior isn't fully defined by code — it emerges from models, prompts, retrieved context, and the tools they're permitted to call. This guide is about securing those systems in development and in production.

WHAT THIS GUIDE COVERS

1. How to discover the agents, MCP servers, and AI frameworks already running in your environment — including the ones nobody registered
2. A 12-point agent and MCP misconfiguration checklist
3. How to triage AI finding volumes without drowning your team: what to automate, what to keep human
4. Runtime protection for AI applications: guardrail architectures, prompt hardening, and policy enforcement
5. An agentic AI maturity roadmap aligned to NIST AI RMF, OWASP AIMA, ISO/IEC 42001, and the EU AI Act

WHO THIS GUIDE IS FOR

AI/ML engineers, platform engineers, AppSec teams, and the leaders who sign off on agentic AI going to production. If you're building or approving AI agents, MCP integrations, or LLM-powered applications, this guide is written for you.

COMPANION GUIDE

This guide is the practitioner companion to *AI Security Governance: A Practical Framework*. That guide covers building an AI governance program — inventory, risk classification, policy, and ownership. This one goes deeper on the layer governance alone can't reach: securing agentic AI systems in development and in production.

CONTENTS

Three moves, one framework

01

See what matters

Discover and inventory your agentic attack surface

02

Fix what matters faster

Prioritize and triage AI finding volumes

03

Protect AI in production

Runtime guardrails, prompt hardening, policy enforcement

ARTIFACTS — TAKE THESE WITH YOU

SECTION 1

Why AI apps broke traditional AppSec

Application security was built on a stable assumption: applications do what their code says. Scan the code, scan the dependencies, test the running app, and you've covered the attack surface. Agentic AI breaks that assumption. An AI agent's behavior isn't fully defined by its code — it emerges from the interaction of a model, a system prompt, retrieved context, user input, and the tools the agent is permitted to call. Two identical deployments can behave differently. The same deployment can behave differently tomorrow.

And the components involved — agents, MCP servers, AI frameworks, models, prompts, along with AI-generated code — are entering codebases faster than most security programs can track them, creating new risks across development *and* runtime.

The failure modes are new, too. Prompt injection arrives through data, not code. An over-permissioned agent can take harmful actions without any vulnerability being exploited in the traditional sense. A deprecated model keeps serving predictions long after its maintainer stopped patching it. A poisoned tool description on an MCP server can redirect an agent's behavior without touching your application at all. None of these appear in a CVE feed, and none are caught by scanning source code alone.

The new AppSec mandate

Shift left

Prevent what you can early in the SDLC

+

Protect right

Continuously govern and protect AI applications after deployment — they keep interacting, changing, and making decisions long after they ship

The strategic case in one line: for the executive-level argument, see our CISO's guide, *Securing AI: A CISO's Guide to Modern AppSec.* This guide assumes you're convinced and ready to build.

ARTIFACT 1.1

The agentic AI attack surface map

Five layers where agentic AI risk lives — each with components traditional AppSec doesn't inventory, and failure modes no CVE feed will announce.

LAYER	COMPONENTS	EXAMPLE FAILURE MODES
Interaction	User inputs, retrieved documents, inter-agent messages	Prompt injection, context poisoning, data exfiltration via outputs
Agent	System prompts, agent configs, memory, autonomy settings	Over-permissioned tools, unsafe defaults, goal hijacking
Integration	MCP servers, tool/function definitions, plugins, APIs	Malicious or poisoned tool descriptions, unscoped credentials, shadow MCP servers
Model	Foundation and fine-tuned model, embeddings	EOL/deprecated models, model supply chain risk, unsafe generations
Code	AI-generated code, AI frameworks, SDKs	Vulnerable generated code, framework CVEs, malicious packages

SECTION 2 · SEE WHAT MATTERS

You have more AI integration points than you think

Agents don't arrive through procurement. They arrive when a developer wires an LLM to a tool API to automate a workflow, when a team adopts an agent framework inside an existing service, or when someone stands up an MCP server so their assistant can reach an internal system. Each of these is a real attack surface: an identity that can act, credentials that can be abused, and an integration point that accepts instructions from a model. Three categories to hunt for:

- **Shadow agents** — LLM-driven automation running without security review — often a script or service wrapping a model API with tool access. Signals: outbound calls to model API endpoints from services not in your AI inventory; agent framework imports appearing in repositories; service accounts whose activity patterns look autonomous.

- **Unregistered MCP servers** — MCP servers expose tools, data, and prompts to AI clients. They are deliberately easy to stand up — which means they appear wherever a developer wants their assistant to reach something. Every MCP server is a bridge between a model and a real system, and each one needs an owner, an access scope, and a review.

- **Embedded AI frameworks** — Agent and orchestration frameworks — and their transitive dependencies — ship inside applications like any other open source: with CVEs, abandoned maintainers, and occasionally malicious lookalike packages. Your SCA program should treat them as a first-class category, not generic dependencies.

How to find them

1. **Scan repositories for agentic signatures:** agent framework dependencies, MCP server manifests, model API client libraries, and system-prompt files checked into code.
2. **Watch network egress** for calls to known model API endpoints from services not in the inventory.
3. **Audit service accounts and API keys** for credentials consumed by autonomous processes rather than humans.
4. **Make declaration cheap:** a lightweight registration step for agents and MCP servers at creation time beats forensic discovery later — and non-punitive disclosure norms beat both. Developers who fear being punished for shadow AI simply won't disclose it.
5. **Automate continuously:** point-in-time discovery goes stale in weeks; discovery should run as part of your pipeline, not as an annual audit.

Agent configuration risk

Once discovered, most agent risk lives in configuration — and configuration is checkable. The highest-value review questions are permission questions:

> What tools can this agent call?
>
> What credentials does it hold?
>
> What can it write to?
>
> Who reviews changes to its system prompt?

Model lifecycle risk

Models age out. Providers deprecate versions, stop patching, and eventually stop serving them — and self-hosted models can quietly linger for years.

An EOL model is unpatched software with a public deprecation notice: check every discovered agent and application for the model versions it depends on, and treat end-of-life dates like you treat framework end-of-support dates.

ARTIFACT 2.1

Agent extension for your AI-BOM

Add these columns to the AI-BOM you built with your governance program — one row per agent or MCP server.

FIELD	WHAT TO CAPTURE
Agent/server identity	Name, repo, owning team, business purpose
Model dependency	Model + version, provider, EOL/deprecation status
Autonomy level	Advisory / human-approved actions / autonomous
Tool permissions	Every tool or function the agent can invoke
Credential scope	Keys and service accounts held; what they unlock
Data reach	Sensitivity level of data readable and writable
MCP endpoints	Servers exposed or consumed; owners of each
System prompt location	Where it lives; who can change it; change review process
Last security review	Date, reviewer, findings status

ARTIFACT 2.2

Agent & MCP misconfiguration checklist

Twelve checks to run against every discovered agent and MCP server. Most agent risk lives in configuration — and configuration is checkable.

1. Agent holds credentials scoped to specific resources, not broad service-level access
2. Write access granted only where the agent's function explicitly requires it
3. No shared credentials between agents, or between agents and human users
4. Tool inventory documented: every callable tool is listed and justified
5. High-impact tools (payments, deletion, external communication, code execution) require human approval or are excluded
6. System prompt stored in version control with change review — not editable in production
7. MCP servers authenticate their clients; no anonymous tool access to internal systems
8. MCP tool descriptions reviewed for injection-bearing content before adoption (tool poisoning)
9. Third-party MCP servers vetted like any third-party dependency: provenance, maintenance, permissions
10. Agent memory/context stores reviewed for sensitive data accumulation and retention limits
11. Model version pinned, with EOL/deprecation monitoring and an upgrade owner
12. Agent activity is logged with enough detail to reconstruct what it did and why

The bottleneck moved

AI didn't just expand the attack surface — it expanded the *finding* surface. AI-assisted development ships more code, more dependencies, and more findings per week than security teams were staffed to review. Meanwhile, AI-specific detections add whole new finding categories. The bottleneck is no longer finding vulnerabilities. It is knowing which ones matter and fixing them fast.

Enrich › **Prioritize** › **Triage**

Automation does the volume work; humans make the judgment calls. Teams that treat every finding as equally urgent end up with alert fatigue, developer distrust, and a backlog that grows faster than it burns down.

Prioritize on risk, not severity scores

A critical-severity CVE in a dependency your application never loads matters less than a medium-severity flaw on an internet-facing request path. Prioritization signals worth engineering into your pipeline, roughly in order of value:

Reachability	Is the vulnerable code actually invoked by your application?
Exploitability context	Is the component exposed to untrusted input? Internet-facing? Behind auth?
Business context	A finding in a payments service outranks the same finding in an internal dashboard — contextual project classification automates this
Agentic amplification	A modest flaw an autonomous agent can trigger repeatedly is not a modest risk
Fix availability	Is there a clean upgrade path, and what will it break?

ARTIFACT 3.1

Triage decision matrix

Agentic triage systems investigate findings the way an analyst would — trace the data flow, check reachability, examine the deployment context — and produce a true-positive/false-positive assessment with evidence. Used well, they collapse the noise floor so human attention lands where it should. Used carelessly, they become an unaccountable filter that silently discards real risk. This is the line:

DECISION TYPE	DISPOSITION	RATIONALE
Reachability/dataflow analysis for well-understood vulnerability classes	**Automate**	Deterministic evidence; high volume; machine does it faster and more consistently
FP/TP assessment with clear evidence trails	**Automate, with sampling**	Let agentic triage close FPs, but audit a sample of closures weekly to measure error rate
Findings in Tier-3/high-risk applications (per your governance risk tiers)	**AI-assist, human decides**	AI gathers evidence and drafts assessment; a human owns the disposition
Novel finding classes, AI-specific behaviors, anything without an evidence trail	**Human only**	No established ground truth for the automation to stand on
Accepting risk / deferring a fix	**Human only, documented**	Risk acceptance is an accountability decision, not a technical one

Operating rule 1 — evidence, always

Every automated closure carries evidence. If the triage system can't show *why* something is a false positive, it goes to a human.

Operating rule 2 — measure it like an analyst

Sampled review, error rates tracked over time, and thresholds that trigger retraining or re- scoping when quality drifts.

SECTION 3 · FIX WHAT MATTERS FASTER

Remediation at AI speed

Prioritized, triaged findings still need fixing. The same AI leverage applies downstream: automated remediation for the mechanical share of the backlog — dependency upgrades, config corrections, generated fix PRs — while engineers handle fixes requiring design judgment.

Same gate as AI-generated code

AI-generated fixes get the same review-and-test gate as AI-generated code — because that's what they are.

Track risk retired, not tickets

Remediation metrics track MTTR on the findings that mattered — not raw ticket counts.

ZERO-DAY RESPONSE, AGENT-AWARE

When the next headline vulnerability lands, the question is "where are we exposed, and are we exploitable?" — and your answer now has to include the agent layer. The teams that answer fastest have the Section 2 inventory already built: which applications carry the component, whether it's reachable, and which *agents* can touch the affected systems.

Add one step to your zero-day playbook: alongside "which services are affected," ask "which agents hold permissions on affected systems, and should their autonomy be reduced while we patch?"

An AI app's riskiest hours happen after deployment

Sections 2 and 3 secure what you build. This section secures what's *running* — because an AI application's riskiest hours happen after deployment, when it starts encountering inputs, contexts, and adversaries nobody tested.

OFFENSE AND DEFENSE

This section is the defensive counterpart to our *Practical Guide to AI Red Teaming*. Red teaming tells you where your AI system can be manipulated; runtime protection is how you hold the line in production. Run them as a loop: red team findings tune your guardrails; guardrail logs feed your next red team scope.

The red team ⟷ runtime loop

Red teaming

Tells you where your AI system can be manipulated. Findings tune your guardrails.

→

←

Runtime protection

Holds the line in production. Guardrail logs feed your next red team scope.

Runtime protection has four working parts, covered over the next pages: **guardrails** at the enforcement point, **prompt hardening** at the boundary, **policy enforcement** that makes governance real, and **monitoring** that watches agents work.

Runtime guardrails: deployment options

Guardrails are the enforcement point between your AI application and the world — inspecting inputs before the model sees them and outputs before anyone else does. There are two ways to deploy them:

In-app guardrails

Live inside the application, running via a Python SDK embedded directly into your code, with requests validated before and after the LLM responds. They run in two modes: **Online**, which connects to a central platform for configuration and monitoring, or **Offline**, which runs fully isolated with no telemetry sent, suited to air-gapped environments.

API Server (Docker)

Deploys guardrails as a standalone service your application sends requests to — no code changes required, and no Python dependency. This suits non-Python environments or teams that prefer a service-based integration over an embedded SDK.

CHOOSE	WHEN
In-app	Your application is Python-based and you want the SDK embedded directly in your code — including fully offline, air-gapped deployment if needed.
API Server (Docker)	Your application isn't Python, or you'd rather call a standalone service than embed a library.

What guardrails should check — minimum viable set

INBOUND

Prompt injection patterns · malicious or out-of-policy requests · inputs targeting known jailbreak families

OUTBOUND

Sensitive data patterns (credentials, PII, proprietary code) · unsafe content · responses violating business policy

System prompt hardening

The system prompt is load-bearing security boundary and public attack target at once. Hardening patterns that hold up:

Assume disclosure	Write every system prompt as if the user will eventually read it. Secrets, internal URLs, and credentials do not belong in prompts — ever.
Separate instructions from data	Delimit untrusted content — user input, retrieved documents, tool outputs — explicitly, and instruct the model to treat it as information, not instructions. This doesn't defeat injection alone; it raises the cost.
Constrain the blast radius	The durable defense against goal hijacking is Section 2's permission model — an agent that *can't* call a dangerous tool doesn't need a prompt that begs it not to.
Version and review	Prompts change behavior like code changes behavior: version control, change review, and a named owner.
Test adversarially	Every hardening claim should survive contact with your red team.

Policy enforcement & continuous governance

Your organization has AI policies — data handling rules, approved-model lists, output requirements. Runtime is where they become real: the guardrail layer blocks regulated data from flowing to unapproved models *because every request is*, not *inspected against policy before it happens* because a document asked nicely. Continuous governance re-checks posture as things change:
— a new tool grant, a model swap, a prompt edit — not at annual review time. In regulated contexts, this enforcement-plus-audit-trail is also your evidence base: the EU AI Act and NIST AI RMF both expect demonstrated operational control, not just documented intent.

Runtime monitoring: watching agents work

The Runtime Protection dashboard shows what's actually been inspected and flagged: total protected entities, events by action type (Alert/Block/Obfuscate), guardrail activity over time by weakness type, and a breakdown of inbound vs. outbound events. Every event logs a timestamp, the affected entity, the guardrail type that triggered, direction, detected-activity detail, severity, and the model involved, and is exportable to CSV or PDF for audit purposes.

ARTIFACT 4.1

Prompt-hardening checklist

Seven checks for every system prompt in production. Run them at creation, and again on every material prompt or permission change.

1. No secrets, credentials, or sensitive internal detail in any system prompt
2. Untrusted content explicitly delimited and marked as data, not instructions
3. High-impact instructions enforced by your agent's own permission model, not prompt language alone. System prompts in version control with required review
4. Prompt changes tested against a regression suite of known attack patterns
5. Injection attempts logged with periodic log reviews to inform manual guardrail configuration changes
6. Agent's refusal behavior verified for the tools it must never call
7. Red team validation on every material prompt or permission change

ARTIFACT 4.2

Runtime guardrail reference architecture

Guardrails inspect every input before the LLM sees it and every output before the application returns it — with the dashboard and events log providing visibility.

User — Sends a request · receives the response

↓ request response ↑

AI application — Passes every model call through guardrails

↓ ↑

Guardrails inspect input — BEFORE THE MODEL SEES IT

- **In-app guardrails (SDK)** — Python SDK embedded in the code · Online or Offline mode

OR

- **API Server (Docker)** — Standalone service · no code changes, no Python dependency

↓ ↑

LLM — Generates the response

↓

Guardrails inspect output — Sensitive data · unsafe content · policy violations — beforethe application returns it

RUNTIME PROTECTION DASHBOARD

Protected entities · events by action type (Alert/Block/Obfuscate) · activity by weakness type · inbound vs. outbound

EVENTS LOG

Timestamp · entity · guardrail type · direction · severity · model — exportable to CSV or PDF for audit

SECTION 5

The maturity roadmap

You don't build all of this at once. The four-stage **AI Security Maturity Model** introduced in our governance guide — aligned to NIST AI RMF, OWASP AIMA, ISO/IEC 42001, and the EU AI Act — applies directly to agentic AI.

STAGE 1
Emerging
Ad hoc / awareness

Agents and MCP servers exist in the org; security doesn't know where. No agent inventory, no config review, no runtime controls.

Priority: run Section 2 discovery once, assign an owner, baseline what you find.

STAGE 2
Developing
Defined / reactive

Partial agent inventory; configuration checklist applied to known agents; findings triaged manually; prompt hygiene exists but isn't enforced.

Priority: standardize guardrails, put agent configs and prompts under version control and review.

STAGE 3
Controlling
Managed / proactive

Continuous discovery in the pipeline; risk-based prioritization with reachability and business context; agentic triage operating with human oversight; runtime guardrails deployed on production AI traffic (in-app or via API Server); agent-aware zero-day playbook.

Priority: close the loop — red team findings tune guardrails, guardrail telemetry feeds monitoring baselines.

STAGE 4
Leading
Optimized / adaptive

AI assurance embedded across the SDLC: every agent registered at creation, policy enforced at runtime with audit-grade logging, automated remediation burning down the mechanical backlog, evidence on demand for boards and regulators.

Security is enabling agent adoption velocity, not blocking it.

ARTIFACT 5.1

Agentic AI maturity self-assessment

Score one point per "yes."

SEE

1 Could you list your production agents and MCP servers today?

2 Does each have a named owner?

3 Is discovery continuous rather than annual?

4 Do you track model EOL status per agent?

5 Are agent configs reviewed against a checklist?

FIX

6 Do findings carry reachability/business context?

7 Is FP triage automated with evidence?

8 Are triage error rates measured?

9 Is remediation MTTR tracked on prioritized findings?

10 Does your zero-day playbook include the agent layer?

PROTECT

11 Do guardrails inspect production AI traffic?

12 Are your production AI applications running guardrails at all — in-app or via API Server?

13 Are system prompts versioned and reviewed?

14 Are your AI application's outputs checked for harmful content and sensitive data before a user sees them?

15 Can you produce runtime policy evidence for an auditor?

Interpreting your score

0–5	6–10	11–13	14–15
Stage 1 · Emerging	Stage 2 · Developing	Stage 3 · Controlling	Stage 4 · Leading

SECTION 6

How Mend.io implements this framework

Everything in Sections 2–5 is implementable with disciplined engineering. What a platform changes is the cost — and whether coverage stays continuous when your teams are shipping daily.

SECTION 2 · AUTOMATED

Intelligent Discovery

Powered by Mend AI: discovers agents, MCP servers, and AI frameworks across your codebase, scans agent configurations for risky settings, detects EOL models, and maintains a continuous AI-BOM. Zero-day discovery answers "where are we exposed" with the agent layer included.

SECTION 3 · AT SCALE

Intelligent Prioritization

AI-based detection in Mend SAST, reachability analysis, contextual project classification, and agentic triage separate true from false positives with evidence trails and human-in-the- loop controls.

SECTION 3 · FIX SIDE

Agentic Remediation

Through Mend Intelligent Agent: AI-powered remediation and zero-day process improvements that shorten headline- vulnerability response.

SECTION 4 · AS INFRASTRUCTURE

Continuous Runtime Protection

Mend AI Guardrails: runtime protection via embedded SDK or standalone API Server, runtime AI policy enforcement, system prompt hardening support, and event-level monitoring of guardrail activity.

Mend.io is built for every risk, across AI and AppSec. By securing the code layer and the AI layer—and the interactions between them, where modern application risk now lives—**Mend.io** extends proven AppSec workflows to the models, prompts, and agents inside today's applications, delivering continuous protection across the entire AI application lifecycle.

Get a demo