AI SECURITY ENGINEERING HANDBOOK · 2026

Chapter 06 · Agentic Permissions

Standalone study module for LMS delivery and required reading.

FORMAT	USE	SCOPE	AUDIENCE
Standalone PDF	Study module	Single chapter	Learners

CHAPTER 06

Agentic Permissions

HANDBOOK STUDY COMPANION: STUDY FRAME

Use this chapter to build vocabulary, judgment, and role-readiness. Pair it with the Field Guide when you need applied actions, checklists, and control execution.

STUDY FOCUS

STUDY FOCUS	WHY IT MATTERS
Delegated action security: tool scope, runtime authorization, approvals, action logs, rollback, and blast radius.	Agent security begins when model-mediated output can trigger actions in real systems.

Study Outcomes

- Classify tool permissions and side effects.
- Explain why approvals require context and runtime enforcement.
- Reason about action chains, identity, auditability, and rollback.

DOMAIN MAPPING

RELATED AIPSA DOMAINS	APPLIED NEXT STEP	WORKBENCH INSTRUMENTS	RELATED SERVICES
Agent Security	Agent security	Authority Graph, Adversarial Range	AI Product Security Assessment

CERTIFICATION AND ASSESSMENT BOUNDARY

This chapter supports training, diagnostic preparation, scorecards, interviews, and role-readiness evaluation. It does not guarantee credential outcomes.

The security model for agents breaks down quickly when you follow one question to its conclusion: what is the maximum blast radius of one confused or compromised model call? For a text assistant, the answer may be a bad output. For an agent with write access to email, source code, cloud resources, issue trackers, calendars, and customer records, the answer can be an organization-wide incident triggered by a single injected instruction in a retrieved document. The gap between those two answers is the entire scope of agent security.

What is the maximum blast radius of one confused or compromised model call? For an agent with write access to email, source code, cloud resources, and customer records, the answer can be an organization-wide incident triggered by a single injected instruction in a retrieved document.

HANDBOOK

Every agent interaction follows a delegated action chain. A user prompt becomes model reasoning; model reasoning produces tool arguments; tool execution changes real-world state. The security review must trace this full path — from prompt to side effect — not stop at the model response.

FIGURE 1: FIGURE 10: DELEGATED ACTION CHAIN — USER PROMPT, MODEL REASONING, TOOL ARGUMENTS, RUNTIME AUTHORIZATION GATE, TOOL EXECUTION, REAL-WORLD SIDE EFFECT — WITH THE AUTHORIZATION GATE AS THE CRITICAL CONTROL POINT THAT OPERATES INDEPENDENTLY OF MODEL REASONING

The difference between an AI assistant and an AI agent is blast radius. An assistant's worst outcome is a bad answer, contained within the user interface. An agent with write access to email, cloud infrastructure, and production data can cause organization-wide damage from a single misled model call.

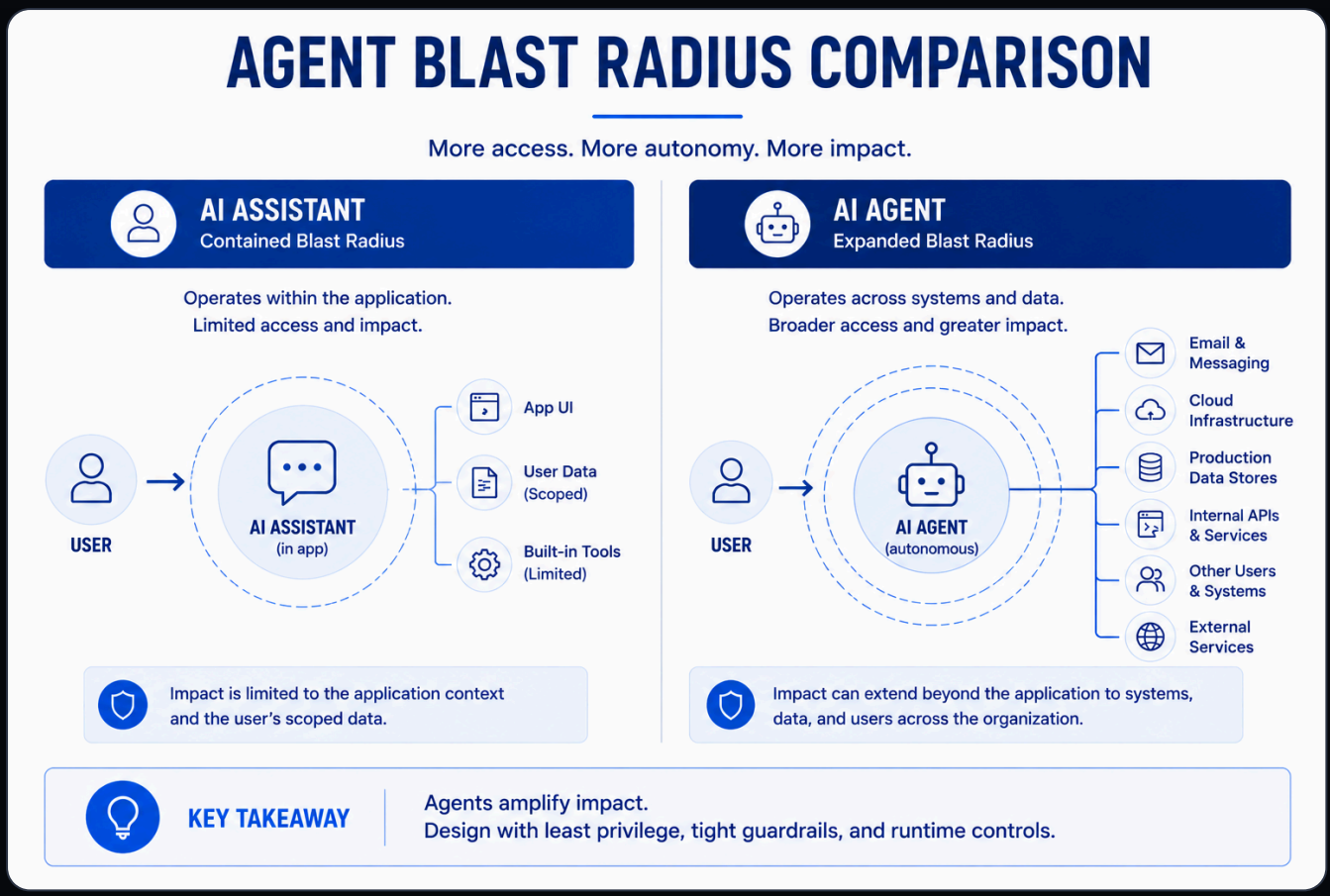

FIGURE 2: FIGURE 11: BLAST RADIUS COMPARISON — AI ASSISTANT CONTAINED WITHIN THE USER INTERFACE VS. AI AGENT EXTENDING INTO EMAIL, CLOUD INFRASTRUCTURE, AND PRODUCTION DATA, ILLUSTRATING THE AUTHORITY GAP THAT BLAST RADIUS DESIGN MUST ADDRESS

CORE CONCEPTS

DELEGATED ACTION MODEL

Agent security starts with the delegated action chain: user request becomes model reasoning, model reasoning becomes tool arguments, tool execution changes state, and the result may influence another model call. Each transition changes the risk. A generated answer can be wrong without changing the world; a tool call can send email, modify records, create cloud resources, or delete data. The security review should trace the full path from prompt to side effect, not just inspect the model response.

TOOL PERMISSION DESIGN

Tool permissions should be scoped by resource target, action type, tenant boundary, user role, time window, quota, and reversibility. A tool called "send_message" is not one permission; sending a draft to the current user, sending an email to a customer, posting in a public channel, and notifying every administrator are different risk classes. Least privilege means the credential and policy wrapper enforce the narrowest action needed for the workflow. Good tool design makes dangerous action impossible by default rather than relying on the model to avoid it.

RUNTIME AUTHORIZATION

Tool labels and descriptions are not enforcement. If a tool is described as read-only but the underlying credential can write, the system is write-capable. Runtime authorization checks the acting user, agent identity, tenant, resource, action, arguments, current context, and policy before execution. The policy should live outside the model so an injected instruction cannot redefine what is allowed. The model can propose an action; the runtime decides whether the action is permitted.

APPROVAL GATE DESIGN

Human approval is valuable when it is rare enough to receive attention, informative enough to support judgment, and placed before actions that are irreversible, externally visible, high-volume, destructive, or privileged. Approval becomes ceremony when every trivial action prompts a click, when the approver lacks context, or when the prompt hides the true target and arguments. A useful approval request shows what will happen, why the agent proposes it, which evidence supports it, what resources are affected, whether it can be undone, and what policy triggered approval. Approval is not a magic shield; it is a control that needs design.

BLAST RADIUS AS ARCHITECTURE CONSTRAINT

Blast radius is the maximum damage a compromised, confused, or misled agent can cause before another control stops it. It must be designed before implementation because after an incident the system has already exercised its available authority. The blast radius of a tool depends on credentials, resource scope, action scope, quotas, environment access, network access, and action chaining. Prompt patches do not reduce the authority already granted to a tool. Architecture does.

THE PRACTITIONER'S CHALLENGE

The political challenge is that agents are often sold internally as productivity accelerators. Teams want tools connected quickly because the demo value is immediate: the agent files tickets, updates documents, searches systems, drafts messages, and completes workflows. Security friction can sound like resistance to automation. The practitioner has to reframe controls as what makes automation deployable, not what makes it slower.

The structural challenge is ownership. The model team may own orchestration, platform engineering may own the runtime, product engineering may own user experience, IT may own SaaS connectors, security may own policy, and business teams may own the workflows. An unsafe tool chain can emerge because every team owns a piece and no one owns the end-to-end authority model. Agent security requires a single view of what the agent can do across systems.

The technical challenge is composition. A single read operation may be low risk, but a sequence of reads can collect enough context for disclosure. A draft action may be low risk until paired with a send action. A code generation tool may be manageable until paired with repository write access and CI triggers. The practitioner must analyze action chains rather than individual tool calls in isolation.

HOW TO APPROACH IT

- Start with a tool inventory. List every tool, connector, API, execution environment, and sub-agent the system can use. For each one, record the underlying credential, action class, resource scope, tenant scope, reversibility, external visibility, data classification, rate limit, and owner. Do not accept the tool's friendly name or manifest description as the security description. Inspect what the credential can actually do.

- Next, classify action risk. Separate read-only, write, destructive, irreversible, external communication, privilege-changing, financial, production-modifying, and code-executing actions. Assign different baseline requirements to each class. Read-only actions may require logging and scope limits. External messages may require approval. Destructive actions may require stricter authorization, delay, dual approval, or prohibition. Code execution may require sandboxing and egress controls.

- Then design runtime authorization around the user and workflow. Decide whether the agent acts as the user, as itself, or as a service account with delegated authority. For each tool call, enforce policy using user identity, tenant, resource target, action type, arguments, and workflow state. Avoid broad static credentials when possible. If the agent acts through a service account, the policy wrapper must reintroduce user-level and tenant-level constraints.

- Design approval gates only where they change outcomes. Identify irreversible or externally visible actions, broad writes, destructive changes, privilege changes, financial transactions, production changes, and sensitive disclosures. For those actions, build approval screens that show the proposed operation, target resources, source evidence, risk reason, reversibility, and alternatives. If approvers cannot understand what they are approving, the gate is theater.

- Analyze action chains and delegation paths. Walk through multi-step workflows and ask what a malicious document, tool output, or user prompt could steer the agent to do. Identify combinations that create higher risk than any individual tool. If one agent can call another, define whether authority transfers, whether the child agent inherits context, what logs link the chain, and which policy engine makes decisions.

- End by designing auditability and rollback. Define required log fields before launch: user, tenant, agent identity, model version, prompt/context references, tool name, arguments, authorization decision, approval decision, result, side effect, reversibility flag, and parent trace ID. For each action class, decide whether rollback is possible and how it is executed. If an action is irreversible, require stronger prevention before it runs.

OUTPUTS AND DELIVERABLES

- The core design deliverables are the **agent tool inventory**, **tool permission matrix**, and **blast-radius worksheet**. The inventory names every connector, API, code runner, browser action, sub-agent, and workflow integration available to the agent. The permission matrix classifies each tool by action type, credential, resource scope, tenant boundary, data classification, rate limit, and owner. The blast-radius worksheet translates those details into a practical question: if this tool is misused once, what is the worst plausible outcome?

- The enforcement deliverables are the **runtime authorization policy**, **approval gate design**, and **sandboxing profile**. The runtime policy defines which identity the agent acts under, which checks occur before execution, what arguments are allowed, and what conditions fail closed. The approval design specifies which actions require approval, what context the approver sees, and what evidence the decision creates. The sandboxing profile defines filesystem access, network egress, credential exposure, execution limits, package installation rules, and isolation boundaries for code-executing or browser-driving agents.

- The operational deliverables are the **agent audit schema**, **rollback plan**, and **agent abuse test plan**. The audit schema ensures every action chain can be reconstructed from user request to model call to tool execution to side effect. The rollback plan distinguishes reversible actions, compensating actions, and irreversible actions that require prevention rather than recovery. The abuse test plan covers prompt injection through retrieved content, unexpected tool arguments, confused-deputy paths, approval bypass, chained low-risk actions, and delegation drift.

COMMON FAILURE MODES

- **Manifest Trust:** The team trusts tool names, descriptions, or manifest labels as if they enforce permissions. This happens when engineering treats the LLM tool interface as the security boundary. Recover by inspecting the underlying credential and placing runtime policy outside the model. A read-only description attached to a write-capable token is not read-only.

- **Approval Fatigue:** The system asks humans to approve too many low-context actions. Approvers learn to click through because the requests are frequent and uninformative. Avoid this by reserving approval for meaningful risk thresholds and showing enough context to make a real decision. A good approval gate should be rare, specific, and evidence-rich.

- **Action Chain Blindness:** The team reviews tools individually and misses the risk created by combining them. Reading a record, summarizing it, drafting a message, and sending it may become a disclosure path. Recover by threat modeling workflows end to end and testing sequences, not just single calls. Tool composition is where agent risk often becomes serious.

- **Rollback Assumption:** The team assumes harmful actions can be undone later. Some actions cannot be fully reversed: external emails, data disclosures, financial transactions, privilege changes, and customer-visible updates may leave permanent effects. Recover by classifying reversibility before launch and applying stronger approval or prohibition to irreversible actions. Rollback is not a substitute for prevention.

IMPLEMENTATION CHECKLIST

- [ ] Inventory every tool, connector, API, code runner, browser action, and sub-agent available to the agent.
- [ ] Classify each tool by read, write, destructive, irreversible, external, privilege-changing, code-executing, or production-modifying action.
- [ ] Verify the underlying credential and API permissions instead of trusting tool labels or descriptions.
- [ ] Define runtime authorization checks for user, tenant, resource, action, arguments, and workflow state.
- [ ] Design approval gates for irreversible, external, destructive, broad-scope, or privileged actions.
- [ ] Analyze action chains for compound risk across multiple low-risk tools.
- [ ] Define sandbox limits for code execution, filesystem access, network egress, and credential exposure.
- [ ] Implement audit logs that reconstruct user request, model call, tool arguments, policy decision, approval, result, and side effect.

RELATED READING

- Handbook chapters: Chapter 3, Threat Modeling; Chapter 4, Prompt Injection; Chapter 13, Evaluation and Regression Testing; Appendix, Field Kit and Templates.
- Field Guide: Agent Security; Prompt Injection and Context Security; Secure AI Architecture Design; Incident Response and AI Observability; LLM Application Security.