Beyond the Security Organization

From Specialist Silos to a Closed-Loop Security Learning System

Large software companies do not primarily have a shortage of security activity. They have a problem converting security activity into organizational learning.

Their specialist teams can discover, investigate, contain, and remediate thousands of security problems while the systems beneath them continue to reproduce similar conditions.

A mature security organization should get structurally better every time reality proves it wrong.

Meeting that standard does not require another specialty. It requires a learning system.

ABSTRACT

Large software organizations have built increasingly capable security specialties, but the connections between them remain comparatively weak. Incidents become action items, red-team discoveries become findings, product-security work becomes tickets, and valuable context is lost at each handoff.

This paper argues for a different operating model: a closed-loop security learning system in which evidence from products, engineering, operations, adversaries, and assurance changes the appropriate layer of the system and is then revalidated. The model places the product inside the loop, defines scope through a product trust surface rather than organizational boundaries, treats shared security context as foundational infrastructure, and separates capability, domain, system, and problem ownership.

These ideas have substantial precedent but are not yet a single established model. AI makes the transition more urgent by making discovery and analysis cheaper while increasing the cost of organizations that cannot convert findings into durable change.

This is a proposed operating model, not a description of how any one company organizes security; it draws on public examples and my own synthesis of patterns across security engineering, assurance, and operations.

SECTION 1

The Security Organization Has a Systems Problem

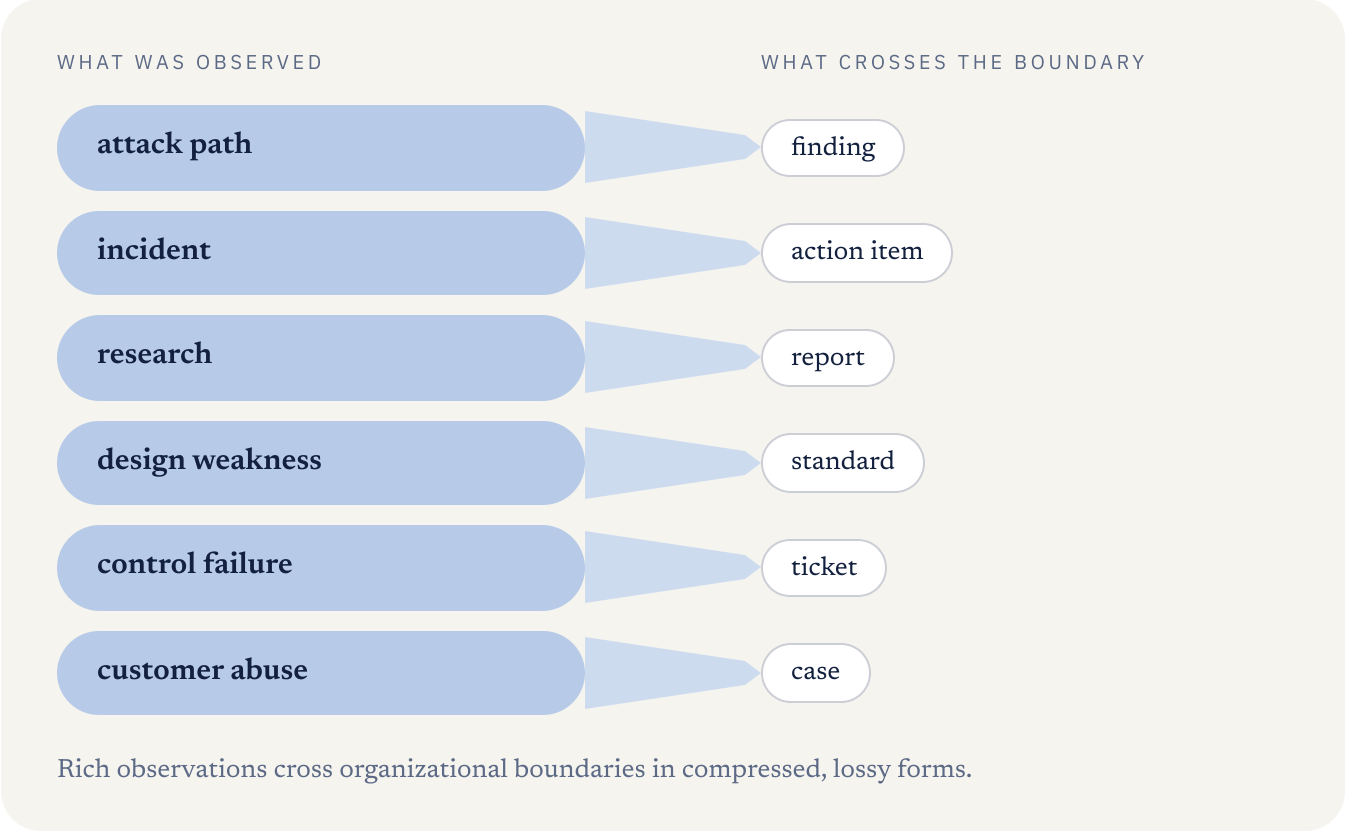
The familiar security organization is built from specialties: product security, detection and response, incident response, red team, threat intelligence, identity, cloud security, privacy, vulnerability management, architecture, risk, and compliance.

That specialization is necessary. The problem is not that specialized teams exist. It is that large organizations are often designed around the specialties rather than around the movement of knowledge between them.

A red team demonstrates an attack path and produces a finding. Incident response discovers an architectural weakness and produces action items. Product Security finds the same authorization mistake in fifteen services and produces fifteen tickets. Threat Intelligence identifies a changing adversary technique and produces an intelligence product.

Every function may be doing good work. Yet the organization can still learn poorly.

Rich observations about system behavior repeatedly cross organizational boundaries in compressed forms:



Rich observations cross organizational boundaries in compressed, lossy forms.

These artifacts are useful. They are also lossy. By the time evidence reaches the people capable of changing the underlying system, the broader meaning may have disappeared.

This creates a familiar outcome: an organization successfully fixes the affected instance while retaining the conditions that made the failure possible. Six months later another team rediscovers substantially the same problem.

The scaling problem is therefore not simply how to perform more security work. It is **how to preserve, connect, and act on what security work teaches the organization**.

This problem becomes more consequential with organizational size. At a few thousand employees, strong practitioners can still carry substantial context through personal relationships. At tens of thousands, that cannot be the primary integration mechanism. Teams, platforms, products, acquisitions, and reporting structures create too much organizational distance.

NIST's systems-security engineering work offers an important foundation for thinking about the problem. It explicitly treats security as an emergent property of a system and argues against leaving security in a traditional stovepipe. Secure behavior depends on the system that interacting components form, not simply on whether each component independently satisfies a checklist.[1]

The same principle applies organizationally. If security emerges from interactions between systems, a security organization that reasons mainly through separate functional queues will eventually struggle to understand the property it is responsible for.

SECTION 2

From Remediation to Learning

A more useful operating question is:

How does evidence about real system behavior become durable improvement?

The resulting model is circular rather than linear.

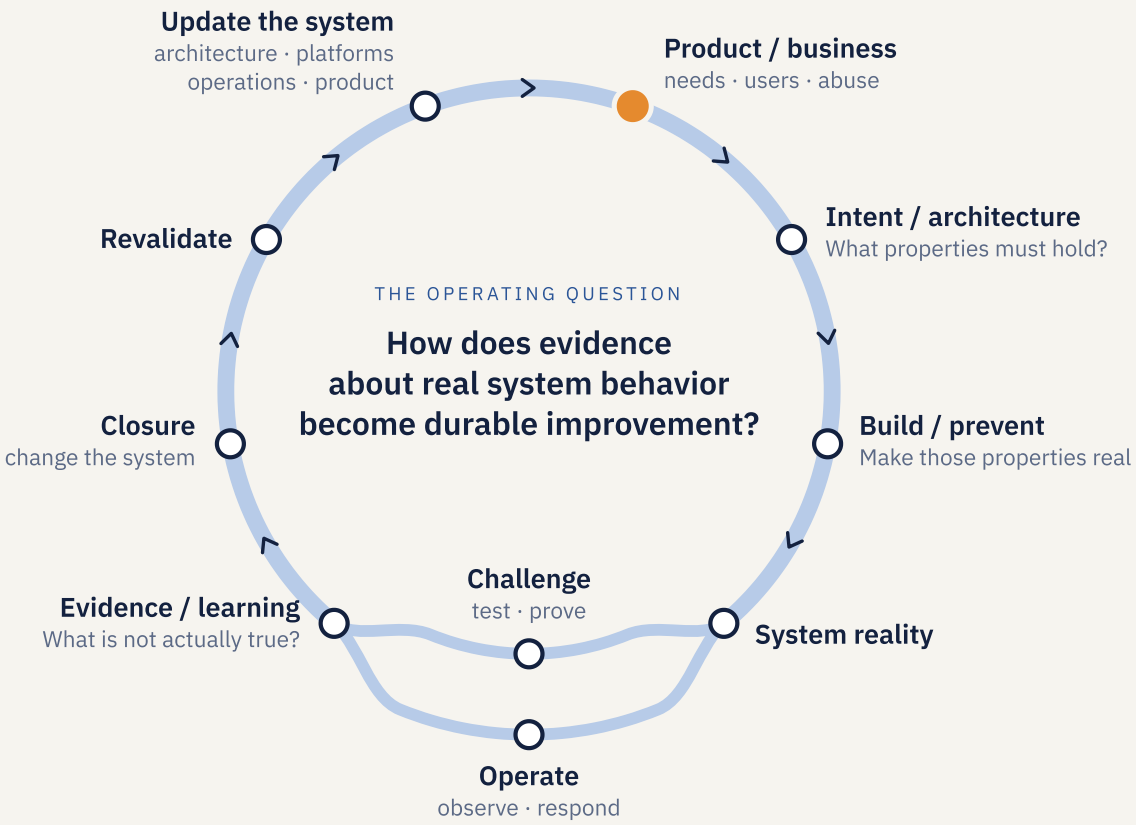


FIGURE 1 **The security learning system**

A vulnerability is not necessarily closed when an endpoint is patched. An incident is not fully learned from when its action items are completed. A red-team exercise has not created maximum value when the report is delivered.

The loop closes when the organization has absorbed the lesson at the appropriate level of abstraction.

Sometimes that is one implementation. Sometimes it is a shared library, identity primitive, deployment platform, detection, secure default, architecture pattern, or product capability. Sometimes the problem genuinely is local and should remain local.

The important capability is knowing the difference.

A worked example

Suppose adversarial testing discovers an alternate API path that allows a user in one tenant to retrieve a resource belonging to another tenant.

The ordinary remediation is straightforward: repair the authorization check, add a test, close the finding.

That may be correct. But investigation shows that tenant authorization is independently implemented by twelve services. Several use similar patterns, and there is no reliable way to identify every path where the invariant should hold.

A learning-system response continues.

The required property is made explicit: a tenant-controlled identity cannot exercise authority over another tenant's resources through any supported access path. Security and platform engineers determine that enforcement belongs in a shared authorization primitive rather than in twelve independent implementations. Security Foundations identifies the services using the old pattern. Service owners migrate. Detection gains telemetry capable of identifying attempted cross-tenant access. Assurance exercises the original attack and equivalent paths against migrated services. The attack becomes a regression test, and architecture guidance stops recommending the obsolete approach.

REMEDIATED

The endpoint was **remediated** when the original exploit stopped working.

LEARNED

The organization **learned** when it became structurally less able to produce the same failure class.

This idea has established predecessors. MITRE's threat-informed defense model, for example, describes a continuous cycle connecting threat intelligence, defensive measures, and testing and evaluation.[2] Microsoft's Secure Future Initiative goes further organizationally: Microsoft says lessons from incidents are fed into security standards and operationalized as paved paths for engineering at scale.[3]

The synthesis proposed here extends the loop through product, architecture, shared engineering systems, operations, and explicit closure.

SECTION 3

The Product Trust Surface

For a software company, the product cannot sit outside this learning system as merely the thing Security protects.

The product is a source of requirements, an object being protected, a source of runtime and customer evidence, a place where abuse becomes visible, and sometimes a destination for what the security organization learns.

Operational attacks can generate customer-facing detections. Incident lessons can change product architecture. Customer abuse can expose missing controls. Internal defensive research can become new product capability.

This creates a boundary problem: what counts as part of the product?

Application code is too narrow. “Everything in the company” is too broad.

A useful abstraction is the **product trust surface**:

DEFINITION

A system, identity, person, or dependency belongs to the product trust surface when control or failure of it can plausibly change an important product security property without first requiring compromise of an unrelated independent security boundary.

The practical test is causal. Can compromise of this thing directly enable someone to alter trusted product code or configuration, exercise privileged product authority, access or modify protected customer information, undermine a critical security control, or materially

impair the ability to detect or recover from those actions?

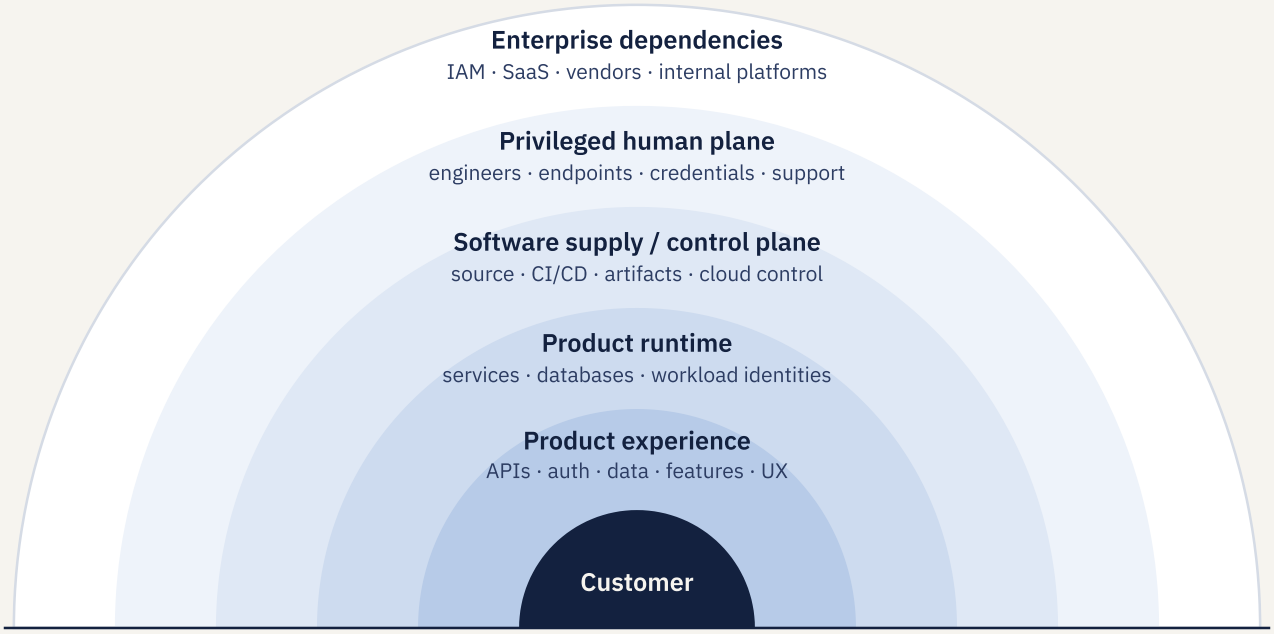


FIGURE 2 **The product trust surface**

A general corporate workstation might not meet this test. A developer workstation holding credentials capable of changing trusted source or production state does. A corporate identity provider may be organizationally "IT" while functioning as a central part of the production trust architecture. A support console may be a business SaaS tool and simultaneously a powerful customer-data control plane.

Okta's public Secure Identity Commitment illustrates the collapsing boundary. The company states that it treats internal technology, people, and processes according to the same cyber threat profile as its customer-facing environment and specifically calls out investment in production-adjacent systems.[4]

Okta also provides a signal in the opposite direction: its July 30, 2026 announcement of the Permiso acquisition said P0 Labs research, combined with Okta Threat Intelligence, would strengthen detections, hunting, and the Okta product roadmap. The acquisition closed on August 26.[5]

These examples do not establish that Okta operates the complete model proposed here. They demonstrate something narrower but important: the line between defending the company, understanding adversaries, and improving the product can be productive rather than rigid.

This leads to a core design principle:

Move information farther than responsibility.

Expanding the trust surface does not mean Security should own everything inside it. Engineering and IT continue to own the systems they build and operate. What changes is how far evidence and learning can travel.

SECTION 4

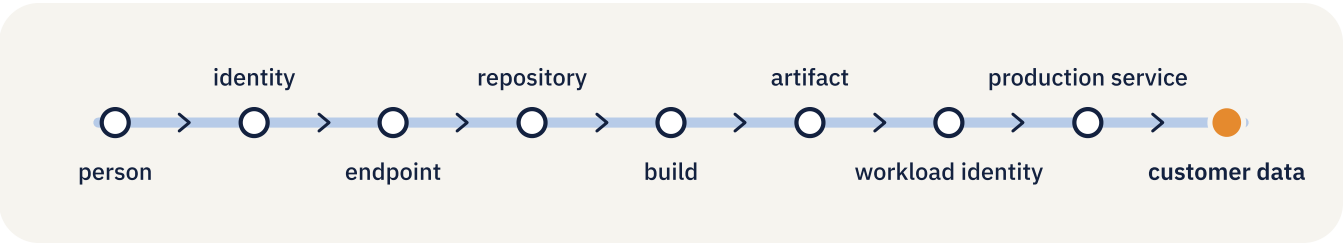
Shared Context as Security Infrastructure

A learning system cannot depend on people manually reconstructing the company for every investigation.

Most mature organizations already possess enormous quantities of security-relevant information. Identity systems know people, workloads, credentials, and permissions. Endpoint systems know devices. Source systems know repositories and owners. CI/CD platforms know builds and deployments. Cloud platforms know resources. Product-security tools know findings. Incident systems know investigations. Detection systems know observed behaviors.

The harder problem is connecting what those datasets mean.

A mature security-foundations capability should make relationships like this progressively easier to reason about:



This allows qualitatively different questions.

Which customer-facing systems depend on this identity primitive? Which changed materially after their last assurance exercise? Which services inherit this platform component? Which privileged developers can affect them? Which lack telemetry for an attack path seen during a recent incident? Which security claims rely on evidence collected before the architecture changed?

At many large organizations, answering questions like these still requires multiple teams and substantial manual reconstruction.

That is not simply a logging problem. It is a shared-context problem.

The metaphor of a **security nervous system** is useful here, but the implementation should be less grandiose than the metaphor. The goal is not a perfect digital twin of the enterprise or a multi-year graph project that promises to know everything.

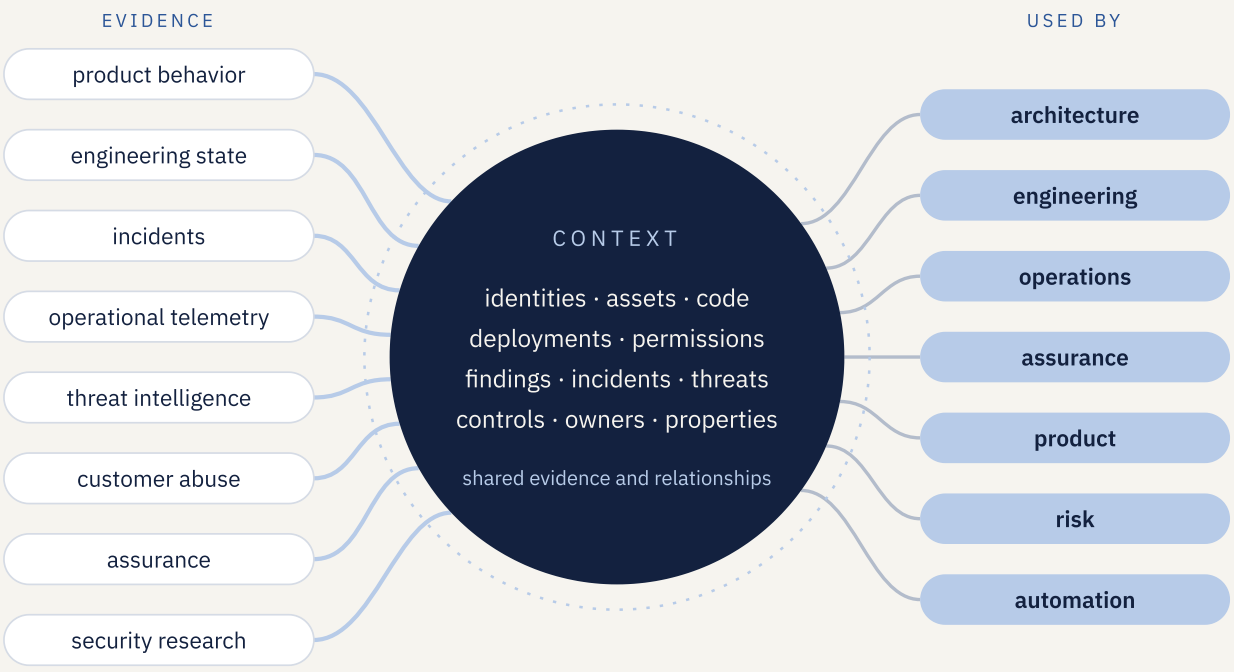


FIGURE 3 **A shared security context layer**

The objective is **composable context**: shared semantics, durable identifiers, relationships, evidence provenance, useful APIs, and progressively better ways to join information across security domains.

There is visible precedent for parts of this. Adobe built Project Caspian after concluding that existing tooling could not capture, manage, and analyze security data at the scale and performance it needed.[6] Its 2026 Security Workbench similarly brings vulnerability, compliance, ticket, and SLA information into a common logical model and shared view rather than forcing teams to reconcile separate tools.[7]

A current Netflix job posting offers another signal. Its Consumer Security Foundations organization describes building shared data, tools, infrastructure, common pipelines, a central source of truth, and a unified risk view for security teams, leadership, and product partners.[8] Because this evidence comes from a job posting rather than durable architectural documentation, it should be treated as an indication of current organizational direction, not a definitive description of Netflix's overall security model.

For a smaller organization, this capability does not require a separate “Foundations” department. The minimum viable version may be common service ownership data, a few durable identifiers, good APIs, and enough shared telemetry to answer recurring cross-system questions. The principle matters more than the org chart.

SECTION 5

Engineering, Assurance, Operations, and Closure

A learning system still depends on specialist capabilities. Their boundaries become clearer when defined by the questions for which each is accountable.

Architecture and intent ask what must be true. **Engineering and prevention** make those properties true. **Operations and resilience** observe real system behavior and respond when properties fail. **Assurance and challenge** ask how the organization knows its claims are true. **Integration and closure** make sure consequential discoveries actually change reality. Security Foundations supplies shared context across all of them, while risk and governance provide constraints, business context, escalation, and legitimate risk acceptance.

These are not sequential stages. An incident may go directly to architecture because it disproves a design assumption. An assurance exercise may change a product because it reveals a missing customer control. Architecture may demand new telemetry because a critical property cannot currently be observed.

Technical assurance deserves particular precision.

ACTIVITY	“Perform a penetration test” is an activity.
OUTPUT	“Deliver a report” is an output.
ASSURANCE CLAIM	**“Cross-tenant access remains prevented across supported access paths, with current adversarial evidence” is an assurance claim.**

Google's 2025 security-assurance paper uses a related technical framing. It focuses on Red Teaming, Vulnerability Management, Detection & Response, and Threat Intelligence working together through remediation to build confidence that software is built securely and continues to operate securely.[9]

That is a useful assurance subsystem. The wider learning-system question is what happens after remediation: whether the discovery changes architecture, engineering platforms, related systems, product behavior, and future testing.

Assurance therefore needs sufficient separation of judgment to say, "this is still not fixed," without becoming the permanent owner of the systems it challenges. Nor should technical assurance be confused with independent Internal Audit; the latter has different organizational independence and governance responsibilities.[10]

Four kinds of ownership

The organization becomes easier to manage when "ownership" is decomposed.

01

Capability owner

Maintains excellence in a discipline such as Red Team, Detection Engineering, Identity Security, or Security Foundations.

02

Domain owner

Understands a product or technical area such as Identity, Developer Platform, Consumer Product, or AI deeply enough to apply specialist capability in context.

03

System owner

The engineering or operational organization accountable for the actual service or platform.

04

Problem or closure owner

Carries a consequential issue across those boundaries until the security outcome is resolved and revalidated.

The fourth is the least explicit in many organizations.

A developer compromise that moves through corporate identity, source control, CI credentials, artifact infrastructure, and cloud permissions has many system and capability owners. It still needs one person maintaining continuity of the security problem.

That person may be a principal security engineer, domain security lead, or explicitly appointed problem owner. They do not need management authority over every participating team. They do need defined escalation rights and accountability for whether the problem reaches a real decision.

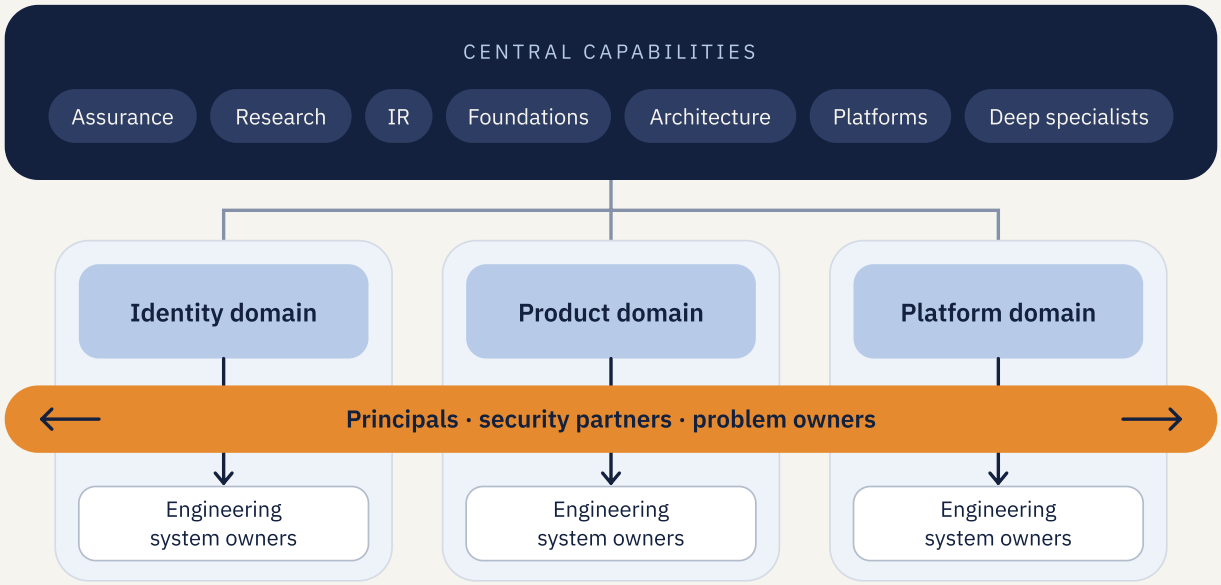


FIGURE 4 **Central capability, federated context, horizontal closure**

The rule is simple: **centralize scarce expertise and shared infrastructure; federate domain context and delivery; matrix consequential cross-boundary problems; leave system ownership with engineering.**

SECTION 6

How the Model Fails

A learning system can easily become more process rather than better security. The test is whether it makes important problems easier to understand and permanently resolve—not whether it creates more coordination.

Everything becomes “systemic”

A serious finding can tempt teams to turn every defect into an architecture program.

Most problems should still be fixed locally. Escalate the layer of the response only when the evidence supports it: the failure recurs, comes from a shared dependency or design assumption, affects multiple systems, or carries enough consequence to justify broader change.

> **Fix at the lowest layer that prevents the failure from recurring where it matters.**

Problem owners become meeting owners

Cross-system problems need continuity, but assigning a Principal does little if participating teams have no reason to prioritize the work.

The problem owner should own the property, dependencies, closure criteria, and escalation —not everybody else's backlog. When normal coordination fails, there must be a short path to the engineering, product, or risk leader who can make the priority decision.

If the role produces meetings without decisions, the operating model is not working.

Foundations becomes an infrastructure project looking for a customer

A “security knowledge plane” can quickly turn into a multi-year graph, lake, or platform effort that promises universal visibility and delivers little operational value.

Start with repeated questions the organization cannot answer: Which systems depend on this primitive? Who can change production? Where else does this attack path apply?

Build only enough shared context to make those questions materially cheaper. New infrastructure should have identifiable consumers and decisions it improves.

Security expands its remit without improving ownership

The product trust surface deliberately crosses Product, IT, identity, development infrastructure, and third parties. That does not make Security responsible for operating all of them.

Use the trust surface to determine where evidence and attention must travel, not where reporting lines should move.

> **Move information farther than responsibility.**

System owners should leave the process with clearer accountability, not less.

Assurance becomes either a gate or an extension of engineering

If assurance can stop anything for any reason, it becomes a source of friction. If it simply accepts the builder's definition of “fixed,” it loses its purpose.

Use narrow blocking conditions for exceptional consequences. Otherwise, assurance should define the claim being tested and the evidence required for closure while engineering retains freedom over implementation.

Close collaboration is useful. Independent judgment is still necessary.

Security partners become ticket routers

A partner function adds value only when it contributes enough technical and domain context to frame problems correctly and connect them to the right capabilities.

If the role mainly receives requests, translates terminology, and forwards work to central teams, it has added another handoff rather than removed one.

A useful partner should reduce the amount of coordination required to reach a good decision.

The loop never gets cheaper

This is the most important failure mode.

If every recurring issue still requires the same investigation, the same set of meetings, the same manual system mapping, and the same one-off remediation, the organization is processing problems rather than learning from them.

A healthy learning system should gradually convert repeated human effort into shared context, secure defaults, platform capabilities, automated tests, detections, and clearer ownership.

The practical test is not whether every finding completes the full model.

> It is whether **important classes of problems become easier to understand, harder to reproduce, and cheaper to resolve the next time they appear.**

SECTION 7

AI Changes the Economics

AI is not the foundation of this model. It changes its economics.

Security reasoning is expensive. Software engineering is expensive. Integrating fragmented systems is expensive. Deeply reviewing thousands of codebases, identity configurations, cloud environments, and runtime states is expensive. Small internal tools frequently cost

more to build than their narrow use cases can justify.

Those constraints are weakening.

Microsoft's July 2026 SFI report describes a multi-agent assessment system that examines source code, identity configuration, network topology, and runtime state together to identify composite vulnerabilities that a single-layer review can miss. Microsoft reports that more than 90 percent of its findings were confirmed by security engineers.[11]

The interesting implication is not simply that AI can find vulnerabilities.

It is that **the amount of the technical system that can be continuously reasoned about is increasing**.

Security practitioners can also build the long tail of internal tooling more cheaply: one-off analyses, integrations, data enrichment, investigative workflows, test harnesses, dashboards, and narrow automation that historically would not have justified a traditional engineering project.

That creates significant leverage for a connected organization.

It creates a different problem for a disconnected one.

Suppose the operating model remains:

A linear queue: nothing returns to the system that produced the finding.

Better automated discovery produces more findings. Better architecture review produces more findings. Better attack simulation produces more findings. The bottleneck moves to context, prioritization, architecture, engineering capacity, ownership, and closure.

As discovery gets cheaper, **discovery becomes less valuable as the primary unit of security work**.

The high-value questions move upward. Is this a new failure class or another instance of one we already know? Which other systems inherit it? Should the platform change? Can the attack become a regression test? Does the product need a new capability? What evidence will establish closure?

> **AI magnifies the quality of the operating model underneath it.**

A connected learning system can turn cheaper reasoning into faster improvement. A collection of queues can turn it into faster production of security work.

SECTION 8

The Strategic Advantage Is Learning Rate

The elements of this model are not inventions.

Systems-security engineering supplies the idea of security as an emergent system property. Threat-informed defense connects adversary knowledge, defenses, and testing. Netflix's paved-road work popularized the idea that security scales by embedding expertise into supported engineering paths rather than placing a gate in front of every developer.[12] Google has articulated a modern technical-assurance model. Adobe and Netflix provide public signals of investment in shared security-data and foundations capabilities. Okta provides visible examples of internal security and research feeding product. Microsoft's SFI explicitly connects incident lessons, engineering standards, paved paths, organizational accountability, and increasingly continuous validation.

The claim is not that these organizations have implemented the exact model proposed here.

The claim is that previously separate ideas are converging on the same underlying problem: **how to turn security evidence into organizational change**.

The advantage of doing so is larger than reducing vulnerability counts. The company becomes better at learning about its technology under hostile conditions.

That suggests a different class of measurement.

CANDIDATE METRIC

Systemic rediscovery rate

One candidate is **systemic rediscovery rate**: the proportion of material incidents or assurance findings attributable to a failure class for which the organization had previously claimed systemic closure.

The metric requires care. Better detection can temporarily increase it, classification is subjective, and "systemic closure" must have a meaningful definition. But it exposes an important distinction. If a company repeatedly discovers high-impact instances of a class it believed it had eliminated, the earlier effort likely achieved remediation without learning.

Other useful questions follow naturally: how quickly can the organization identify every system affected by a newly understood condition? How often does a significant incident change a shared platform or architecture pattern? How much of a novel adversarial technique becomes repeatable assurance? How stale is the evidence supporting important security claims?

Traditional metrics such as remediation time, detection time, coverage, and control adoption remain useful. Learning rate complements them by asking whether the security system is becoming structurally better as a result of the work it performs.

That is the deeper transition.

The modern software company does not need fewer specialists. It needs those specialists to behave as parts of an adaptive system.

The relevant executive question is no longer only how many security activities were performed or how many vulnerabilities were fixed.

It is:

> **When reality disproves something we believed about our security, how reliably does that evidence travel through the organization, change the right layer of the system, and become something we can demonstrate remains true thereafter?**

That is the shift beyond the security organization.

It is the transition from managing security work to managing security learning.

APPENDIX

A

Applying the Security Learning System

Use the model first to diagnose, not to reorganize.

A security leader can see how well their organization learns by following a few serious security problems from the moment they were found to the moment they were closed. A Principal can use the same model to keep a problem that spans many systems on track, without taking ownership away from the engineering teams who run those systems.

A.1 For a Security Assurance Leader: Trace the Learning Loop

Pick three to five serious events from the last year or two. Good candidates are a major incident, an important adversarial finding, a product-security bug that keeps coming back, a pattern of customer abuse, or a control that failed across teams.

For each event, answer seven questions.

Seven things to trace for each consequential event.

1. Property

What did we believe was true?

Describe how the system was supposed to behave, not the name of the bug. For example:

> A compromised developer identity is not sufficient to introduce unauthorized code into trusted production execution.

or:

> A tenant-controlled identity cannot exercise authority over another tenant's resources.

Everything else in the trace hangs off this statement.

2. Evidence

What proved it wrong?

Write down the strongest evidence you have: what happened in the incident, the attack path that worked, the abuse, the exploit, the telemetry, the architecture analysis, or the pattern of repeated failures.

Then check what happened to that evidence as it moved between teams. Did the full story survive, or did it get squeezed into a finding, an action item, or a ticket?

3. Trust surface

What could affect whether the property holds?

List the systems, identities, people, and dependencies involved. Follow the path an attacker or an access right would take, not the org chart.

If it took several teams and a lot of manual work just to build this list, note that. It tells you something about the organization.

4. Closure ownership

Who was responsible for confirming the property was actually restored?

This is a different question from who owned each fix.

If the answer is “every team owned its piece,” ask who owned the whole result.

5. Layer changed

What did we actually change?

Did the fix stop at the one broken instance? Or did it reach something shared: a library, a platform, the architecture, a detection, a way of working, a standard, or the product itself?

Not every problem needs a systemic fix. The question is whether the fix went as high as the problem deserved, given how often it recurs, how much harm it could cause, and how much one change could prevent.

6. Revalidation

How did we know it was fixed?

Did anyone re-run the original attack after the fix? Try similar paths? Confirm every affected system was migrated? Watch the new control work in production?

"We finished the work" and "the property holds again" are different claims.

7. Retention

Where does the lesson live now?

A lesson lasts when it becomes part of something the organization runs or builds on: a regression test, a platform change, an architecture rule, a detection, a secure default, an attack harness, or a product requirement.

If it lives only in a ticket or a postmortem, expect to learn it again.

Reading the Results

After tracing a few events, you will usually see the same breakdowns repeat.

Queue termination	Strong evidence from Red Team, Product Security, or Incident Response becomes someone else's backlog item, and stops there.
Local-remediation trap	Each instance gets fixed, but the same kind of failure keeps coming back.

Context reconstruction tax	Every cross-system problem means pulling people and data together by hand.
Closure ambiguity	Every task is done, but no one can say whether the original property now holds.
Stale assurance	The organization once proved a security claim, but the system has changed so much that the proof no longer counts.

The goal is not to grade the organization. It is to find where learning stops.

From there, a leader can make small, targeted changes without a reorganization: name a closure owner for a few systemic problems, require a retest for serious findings, route incident lessons to platform and architecture owners, or fix one information gap that keeps slowing down work across domains.

Broad-scope **Adversarial Engineering** helps most when a property crosses the usual boundaries of an assessment. Its scope follows the property wherever it goes: endpoint, identity, source code, CI/CD, cloud control plane, runtime, and the product if needed. Its job is to produce strong evidence about how the property can be broken, not to stop at the edge of whichever team asked for the test.

This diagnostic works alongside formal risk governance and independent audit. It does not replace them.

Use the model first to understand how your organization works today. Treat it as a case for organizational change only second.

A.2 For a Principal or Cross-System Problem Owner: The Security Problem Charter

Security problems at the Principal level often arrive in the wrong shape: a single vulnerability, an incident action item, a finding that keeps coming back, or a vague request to “fix security” somewhere.

The Principal’s first job is to restate the problem as a security property that needs to become true.

A good Problem Charter fits on one page.

Security Problem Charter — ONE PAGE

SECURITY PROPERTY

What must be true?

Describe the behavior you need, separate from how it is built today.

> **A compromised developer identity must not be sufficient to introduce unauthorized code into trusted production execution.**

EVIDENCE

Why do we think it is not true now?

Record the strongest evidence. Keep enough technical detail that the problem does not shrink into a generic fix-it ticket.

TRUST SURFACE

What can affect it?

Map the systems, identities, people, and dependencies that matter, based on how the technology actually connects rather than who reports to whom.

OWNERS AND DECISION RIGHTS

Who can change each part, and who decides when priorities conflict?

System owners still own the implementation.

The problem owner keeps track of the overall security outcome.

If they disagree about priority or about how much risk is acceptable, there should be a clear path to escalate to the right engineering, product, or risk decision-maker.

LOCAL OR SYSTEMIC

Is this one bug, or a sign of a wider pattern?

Start with a hypothesis, not a conclusion.

Look for the same logic written in many places, shared building blocks, common design assumptions, or shared dependencies. If the evidence says the bug really is isolated, keep the fix small.

TARGET LAYER

Where should the lasting fix go?

Options include the specific code, a library, a platform building block, the identity model, a deployment control, an architecture pattern, an operating procedure, a detection, or a customer-facing feature.

Choose based on impact, how often the problem recurs, and how much one change would prevent. Do not choose based on a preference for big transformations.

CLOSURE EVIDENCE

What would convince us it is fixed?

Decide this before calling the work done.

Examples include a new adversarial test, negative tests across similar paths, proof that migration is complete, automated enforcement, production telemetry, or watching the system for a reasonable period.

RETAINED LEARNING

What should outlast this problem?

Where it helps, keep the lesson as a regression test, attack harness, secure default, architecture rule, telemetry, detection, engineering pattern, or product feature.

What the Problem Owner Owns

The Principal does not manage the teams involved and should not act as a shadow engineering manager.

What they own is the **continuity of the security problem**: keeping it whole from start to finish.

In practice, that means not letting this:

> **"This security property does not hold"**

quietly turn into this:

> "Team A closed its ticket, Team B shipped its change, and Team C accepted an exception."

with no one checking whether the original problem was actually solved.

Each role stays distinct:

Problem owner	keeps the problem whole and confirms it is really closed.
Adversarial Engineering	produces strong evidence about how the property could fail.
Capability teams	bring specialist expertise and shared tools.
System owners	build and run the systems.
Risk owners	decide what to do about risk that remains.

When these roles are clear, a Principal can drive change across many systems without managing any of them.

APPENDIX

B

The Minimum Viable Security Learning System

You do not need a large new security organization to run this model.

In a smaller company, one team or even one person may cover several capabilities. What matters is whether four basic mechanisms are in place.

1. Challenge important security claims

Someone needs to test whether your most important security assumptions hold up against a realistic attacker.

That could be a small Red Team, adversarial engineers, senior Product Security staff, or outside specialists brought in when needed.

Team size is not the point. What matters is the ability to follow a security claim wherever the attack leads: application, identity, infrastructure, development systems, and any other boundary the attack crosses.

2. Keep system ownership with engineering, and give cross-system problems an owner

Engineering and operations teams stay responsible for the systems they build and run.

Security needs enough authority to define the properties that matter, bring evidence and expertise, and escalate serious problems that stall. It should not take over other teams' systems.

When a serious problem spans several teams, one person should follow it all the way through the fix and the retest.

At moderate scale, this can simply be part of the job of a Principal Security Engineer, architect, or domain security lead. It does not need its own team.

3. Share enough context to reason across boundaries

You do not need a security data lake.

You do need to be able to answer common questions like these:

- Who owns this system?
- Which identities can materially affect it?
- Which repositories and deployment paths produce it?
- What sensitive information does it handle?
- Which shared platforms does it depend on?
- What material incidents, findings, and assurance results apply to it?

When the same question keeps forcing people to piece the answer together by hand, invest in shared identifiers, telemetry, APIs, and links between data sources.

Build this context in response to real needs, not as an attempt to model the whole company.

4. Turn lessons into lasting system change

The key mechanism is getting each lesson to the place where it can do the most good.

When an incident, an adversarial exercise, or a finding reveals a bigger problem, the lesson needs a path to the right layer: the specific code, a shared library, a platform building block, the architecture, a detection, an engineering standard, or the product.

Then test the original condition again.

The minimum viable loop is:

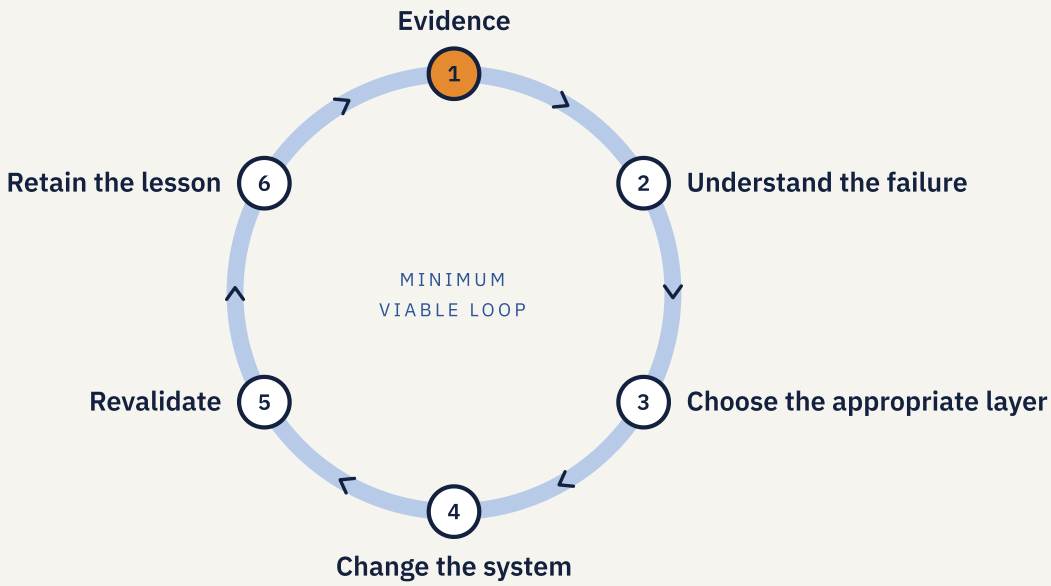


FIGURE B.1 **The minimum viable loop**

What You Do Not Need

A minimum viable learning system does **not** require:

- ~~a separate organization for every capability~~
- ~~a large dedicated Red Team~~
- ~~a company-wide security knowledge graph~~
- ~~permanent security partners in every business unit~~
- ~~an elaborate maturity model~~
- ~~an AI agent layer~~

These may help as you grow. They are not requirements for getting started.

A software company with a few thousand employees could run this model with a mostly central security team, good relationships with engineering, a small adversarial capability, a few senior people who can own cross-system problems, and enough shared context to answer common questions.

Add specialized structure only when scale calls for it.

A Practical Starting Point

Do not start by reorganizing the security team or building a new data platform.

Pick one failure class that is serious or keeps coming back.

1. Write down the security property that should hold.
2. Map everything that can affect it.
3. Name one person to own closure across teams.
4. Decide whether the lasting fix belongs in one place or in something shared.
5. Retest the original condition.
6. Capture the lesson worth keeping.
7. Improve shared data or tools only where this exercise exposed a gap you will hit again.

Then repeat.

If this gets easier each time, because context is easier to find, common failures are harder to reproduce, and lessons keep landing in shared systems, the organization is building a security learning system.

The minimum viable test is simple:

When the organization learns something important about the security of its technology, can it get that knowledge to the people and systems capable of acting on it, verify that reality changed, and avoid learning the same lesson from scratch next time?

Endnotes

1. NIST, “NIST Releases Revised Guidance on Engineering Trustworthy Secure Systems,” November 16, 2022; SP 800-160 Vol. 1 Rev. 1. NIST describes security as an emergent system property and explicitly argues for bringing systems security engineering out of a traditional stovepipe.

2. MITRE Center for Threat-Informed Defense, “Our Mission” and “From Insight to Impact: INFORM Your Defense.” MITRE defines threat-informed defense as a continuous process across cyber threat intelligence, defensive measures, and testing and evaluation.

3. Charlie Bell, “Security Above All Else—Expanding Microsoft’s Secure Future Initiative,” Microsoft Security Blog, May 3, 2024. Microsoft states that lessons from security incidents are fed back into standards and operationalized as paved paths; it also describes Deputy CISO partnerships with engineering and recurring cross-organizational operating mechanisms.

4. Okta, “Secure Identity Commitment.” Okta states that internal technology, people, and processes are treated with the same cyber threat profile as its customer-facing environment and calls out production-adjacent systems.

5. Okta, “Okta Acquires Permiso Security,” announced July 30, 2026; acquisition closed August 26, 2026. Okta states that P0 Labs and Okta Threat Intelligence will strengthen detections, hunting capabilities, and the product roadmap.

6. Jericho Cain et al., “Learnings from Project Caspian, Our Purpose-Built Security Data Platform,” Adobe Security Blog, November 1, 2023.

7. Adobe Security Team, “Building a Security Workbench for Unified Visibility and Prioritization,” April 27, 2026. Adobe describes a common model spanning vulnerability, compliance, ticket, and SLA data and a phased rollout across more than 50 teams.

8. Netflix, current Security Analytics Engineer / Consumer Security Foundations job posting, retrieved September 2026. The posting describes shared data, tooling, infrastructure, pipelines, a central source of truth, and a unified risk view used by security teams, leadership, and product partners. As a job posting, it is a directional organizational signal rather than permanent architectural documentation.

9. Tom Grzelak, Kara Olive, and Moni Pande, “Security Assurance in the Age of Generative AI,” Google, 2025.

10. The Institute of Internal Auditors, Three Lines Model. The IIA distinguishes management and specialist support/challenge functions from Internal Audit’s independent third-line assurance role.

11. Microsoft, “Securing Our Future: July 2026 Progress Report on Microsoft’s Secure Future Initiative,” July 10, 2026. The report describes cross-layer multi-agent assessment and reports that more than 90 percent of resulting findings were confirmed by Microsoft security engineers.

12. Jason Chan, then VP Security at Netflix, discussed the “paved road” model publicly in 2019 as an approach in which supported security mechanisms are made easy enough that engineers naturally adopt them, reducing dependence on gates and repeated specialist intervention.