Least Sharing

Limiting Common Resources and the Paths of Influence They Create

Dr. Ron Ross
CEO, RONROSSECURE, LLC
ISTS FELLOW, DARTMOUTH COLLEGE

Abstract

Least Sharing is a foundational design principle for trustworthy secure systems. NIST SP 800-160, Volume 1, Revision 1 states that system resources are shared among system elements only when necessary and among as few elements as possible. Shared functions, interfaces, state, services, management mechanisms, and recovery dependencies can create paths through which one system element influences others. This article connects justified sharing to stakeholder needs, loss analysis, architecture, the system life cycle, engineering for compromise, complementary principles, and assurance evidence. Every consequential sharing relationship requires a purpose, a bounded set of system elements, and an analysis of the influence and dependence it creates. Separate names, instances, or accounts do not establish independence when the underlying state or authority remains common.

1. The Engineering Problem

Modern systems routinely consolidate resources to reduce cost and simplify operation. Several applications use one database. Multiple mission functions depend on the same identity service. Operational and safety functions share an engineering workstation. Independent recovery copies are administered through a common management service. An AI assistant uses a shared memory store to support several users and tasks. Each arrangement may appear efficient when considered in isolation. The engineering problem is determining whether the resulting paths of influence and common dependencies are acceptable for the system's purpose.

Sharing changes the consequences of a local error or compromise. A corrupted variable can affect every system element that consumes it. A saturated queue can delay unrelated work. A common administrator can change several domains through one management interface. A shared recovery repository can allow an adversary to damage the operational system and the means of restoring it. A system resource becomes a connection among system elements, even when they have no direct communication interface with one another.

> ***"Every shared resource deserves a question: whose compromise can become whose loss?"***

The relevant systems include information technology, operational technology, Internet-of-Things devices, and cyber-physical systems. Their common resources may be physical, computational, informational, or administrative. A shared processor, network link, configuration store, cryptographic service, maintenance mechanism, or supplier update service can create a different kind of exposure. An inventory limited to shared business data will miss many of the relationships that determine system behavior under adversity.

Architecture diagrams can conceal this problem. Two boxes labeled independent may use the same privileged account, configuration authority, storage platform, or restoration source. Two replicated services may depend on the same mutable state. Separate networks may converge at one maintenance laptop. The claimed boundary exists only to the extent that the realized design controls these relationships and the mechanisms that govern them.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

The concern is especially consequential when resources connect functions with different protection needs. A reporting application may tolerate delay that a protective control function cannot. A development environment may permit experimentation that an operational system cannot accept. A remote support activity may need limited diagnostic information without requiring the ability to modify every controller. Sharing across these differences can transfer exposure from a less consequential activity to a more consequential one.

Frontier AI adds practical forms of common state and authority. Agents may share conversation history, retrieval collections, tool services, caches, and task coordinators. A compromised agent can attempt to influence another through information deposited in those resources, even without access to the other agent's execution environment. The engineering question concerns the authority and behavior the shared information can affect, including whether it can reach consequential actions through a trusted intermediary.

Least Sharing makes these connections explicit. Engineers must establish which resources need to be common, which elements need to participate, what each can do, and how disruption or corruption is bounded. Consolidation remains an architectural choice, but its security consequences become part of the trade rather than an assumption hidden behind convenience.

2. What the Principle Means

NIST SP 800-160, Volume 1, Revision 1 states the design principle of *Least Sharing* as follows [1]:

> ***"System resources are shared among system elements only when necessary and among as few elements as possible."***

NIST identifies two criteria: share only if absolutely necessary and minimize sharing if allowed. The first is a trade decision that weighs the costs and benefits of sharing against the increased exposure it creates. The second constrains the extent of sharing. Together, they require both justification for a common resource and a limit on the system elements among which it is shared. A business preference for consolidation does not by itself settle either question.

NIST's discussion includes data, information, system variables, interfaces, functions, and services. Shared mechanisms can create information paths, enable unauthorized access or modification, and adversely affect system capabilities. Corruption of a common state can affect other system elements whose behavior depends on that state. Minimizing these relationships can also simplify design and implementation [1]. The scope therefore extends to common mechanisms as well as the information they hold.

NIST's two criteria—sharing only when necessary and minimizing its extent—provide the conceptual foundation for the following analysis. The first two items address those criteria; the last two items are engineering questions developed in this article to examine the influence and dependence created by shared resources:

- **Necessity.** Identify the mission or protection need served by sharing. Examine alternatives that avoid the common resource or expose a smaller, controlled result. Record the trade and the consequences of the selected arrangement.
- **Extent.** Limit participation to the system elements that require it. Define which portions of the resource are common and which portions remain separate. Include indirect participation through administrators, services, and management interfaces.
- **Influence.** Determine how each system element can affect the resource and, through it, other system elements. Include modification, disclosure, resource exhaustion, timing, ordering, and changes to governing policy.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Dependence.** Determine which required functions cease to operate correctly when the shared resource fails, becomes unavailable, or behaves maliciously. Distinguish a useful common service from a dependency that can defeat several protections at once.

Counting system elements alone does not describe the consequences of their relationship. A common resource shared by two highly consequential system elements can require stronger protection than a resource shared by many low-consequence consumers.

NIST identifies the historically recognized principle of least common mechanism as an instance of *Least Sharing* [1]. Reducing common mechanisms addresses one important source of coupling. The principle of *Least Sharing* also directs attention to shared resources, functions, state, and variables more broadly. An architecture should not restrict its analysis to a list of common software services while ignoring common information or administrative authority.

> ***"Justify the sharing. Minimize its reach. Examine the influence it creates."***

Some sharing is inherent in the intended function. Several system elements may need a common transaction record or a coordinated view of physical operations. The principle calls for the least sharing necessary to satisfy those needs under the relevant constraints. That can mean a narrowly defined interface to selected information rather than direct access to an entire mutable repository. It can also mean accepting a protected common mechanism where alternatives introduce greater overall complexity or risk.

3. What the Principle Does Not Mean

Least Sharing is sometimes equated with avoiding collaboration, duplicating every service, or prohibiting common infrastructure. Those interpretations can undermine the mission and create new weaknesses. The principle requires an engineering trade supported by evidence.

- **Least Sharing does not mean no sharing.** Coordination, communication, accountability, and protection may require common resources. Their system elements and permitted effects must be justified and constrained.
- **Least Sharing does not mean separate copies establish independence.** Copies can remain subject to one writable source, synchronization process, administrator, signing authority, or recovery mechanism. Trace the relationships that can change or disable them.
- **Least Sharing does not mean read-only access eliminates all exposure.** Readers can receive corrupted information, disclose information they obtain, or consume shared capacity. Whether a reader can modify state is only one part of the analysis.
- **Least Sharing does not mean strong access control makes every consolidation acceptable.** Correctly authorized users or processes may still interfere through shared state or consume resources needed by others. A compromised common service can act within the authority entrusted to it.
- **Least Sharing does not mean shared code is shared runtime state.** Separate processes can use the same software design without sharing writable memory. Common code can introduce correlated vulnerabilities, but the runtime sharing claim and the common implementation risk require distinct analysis.
- **Least Sharing does not mean duplication always improves protection.** Additional instances create configuration, patching, consistency, and operational obligations. Unmanaged duplication can increase attack surface and introduce mistakes that outweigh the intended benefit.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Least Sharing does not mean a low system element count proves acceptable risk.** The consequence of the relationship, the authority involved, and the protection of the common mechanism matter as much as the number of system elements.

Logical partitioning can support the principle of *Least Sharing* when it reduces the effective resource exposed to each system element and enforces the intended limits. Separate namespaces in a common database, for example, can constrain ordinary requests. They do not automatically remove dependence on the database engine, administrator, storage, or platform. The claim should identify the level at which sharing has been reduced and the levels at which it remains.

The same caution applies to dedicated infrastructure. A dedicated server can still depend on common power, network equipment, identity services, and remote management. It may substantially reduce one path of influence without providing complete independence. Engineering records should describe the achieved property precisely rather than label the entire system isolated.

Common use of a standard or algorithm also differs from sharing a live resource. Two devices implementing the same published protocol do not necessarily share mutable state or operating authority. Nevertheless, a common defect or compromised supplier can affect both. Applying the design principle of *Least Sharing* helps expose common mechanisms; a complete resilience argument must also consider correlated failures, common implementation vulnerabilities, and supply-chain dependencies.

4. Loss-Driven Interpretation

Least Sharing begins with the losses stakeholders consider unacceptable. NIST's systems security engineering approach connects system purpose, assets, protection needs, and loss concerns to requirements and architecture [1]. The appropriate sharing boundary follows from those concerns. A uniform rule to consolidate or separate every resource cannot account for differences in mission consequence, operating context, and physical safety.

Consider a water treatment facility with two treatment process lines. Each process line must preserve required local control, and each has a supervisory application that supplies selected status information to a common reporting service. Stakeholders identify two unacceptable outcomes: an unauthorized change to treatment control that creates unsafe output, and loss of both process lines' minimum required service through a disturbance that the design is intended to contain to one process line. A shared engineering workstation, writable recipe[1] repository, or unrestricted management service could connect the process lines in ways that defeat those objectives.

The need to see both process lines' status does not establish a need to modify both lines through the same operational resource. Likewise, the need for consistent approved control settings does not establish that both controllers must continuously consume one writable configuration store. Engineers should distinguish required information exchange from unnecessary common authority and state. The alternatives must still preserve coordination, correct configuration, and timely maintenance.

The following engineering activities help determine which resources must be shared, among which system elements, and under what constraints:

[1] A writable *recipe* repository is a storage location containing treatment instructions or operating settings that authorized users or software can change. Recipes could include chemical dosing rates, target pH values, or process sequences. If both process lines obtain settings from that shared repository, an improper change could affect both.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Identify the loss pathways.** Determine how corruption, disclosure, exhaustion, or misuse of a common resource could contribute to unacceptable loss. Include effects on several functions at once and effects that persist after the originating system element is disconnected.
- **Identify the necessary system elements.** Establish which system elements need the resource for their specified functions. Exclude system elements from the sharing arrangement when their needs can be met by a smaller information product, a mediated operation, or another allocation of responsibility.
- **Define permitted influence.** Specify what each system element may contribute, observe, change, consume, or cause. Include rates, capacity allocations, sequencing, and the consequence of incorrect inputs where those affect protection.
- **Establish protective independence.** Identify the functions that must remain effective when the common resource is unavailable or hostile. Trace the management, identity, storage, and recovery dependencies that can defeat that expectation.
- **Compare the alternatives.** Evaluate separation, duplication, constrained interfaces, local validated copies, and protected common mechanisms. Account for cost, complexity, consistency, performance, safety, and residual exposure.

Availability deserves explicit treatment. Shared capacity may allow one workload to deprive another of service without changing its data. A queue filled with legitimate-looking requests can delay protective action. A storage system full of diagnostic records can prevent essential records from being committed. These are sharing consequences even when authentication and authorization operate as configured. Requirements should express the minimum service that each consequential system element must retain.

Confidentiality and integrity also require different arguments. A sanitized status view may limit disclosure while still transmitting incorrect values that affect decisions. A resource protected against unauthorized writes may still be unavailable when needed. The design should identify the property being protected and avoid treating a single control as evidence for every loss pathway.

An effective question is: if the least trusted system element becomes hostile, what can happen to every other system element before detection and response? Where the answer includes unacceptable loss, the design needs a smaller sharing scope, stronger constraints, an independent protective function, or a revised allocation. This connects the principle of *Least Sharing* directly to *Engineering for Compromise* [2].

5. Engineering the Principle into the Architecture

The principle of *Least Sharing* requires a model of resources and the relationships they create. For each consequential common resource, identify its purpose, system elements, permitted operations, capacity limits, administrators, and supporting dependencies. Record why sharing is necessary, the alternatives considered, the allocated protections, and the conditions under which the arrangement remains acceptable. This record provides a basis for design review and later change analysis.

Begin with functions and protection needs. System elements with different trustworthiness requirements, authority, or loss consequences should not inherit a shared resource merely because they use the same vendor product. Determine whether the system can satisfy their needs through separately controlled instances, a smaller common service, or an interface that exposes only the required information or operation. System-level need for a function does not mean every element needs direct access to its implementation.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

Shared mutable state[2] warrants particular scrutiny. A configuration variable, workflow status, model memory, or transaction record can influence many consumers. Define ownership of each state item, who can modify it, and how validity is established. Specify what consumers do when the state item is missing, out-of-date, invalid, or inconsistent. A protected write interface is useful only if its checks constrain the consequences that matter. A syntactically correct value can still be operationally unsafe.

Where system elements need information rather than common modification rights, expose a bounded information product. A supervisory display can receive selected status values without mounting the controllers' configuration storage. A maintenance analyst can receive a diagnostic export without joining the operational management domain. Direction, content, currentness, and permitted use should be specified. A copied or published value remains an input whose trustworthiness must be assessed before consequential use.

The following engineering activities help allocate and constrain necessary sharing in the architecture:

- **Separate unnecessary common state.** Allocate private working memory, configuration, credentials, and task context where sharing provides no required benefit. Ensure that backup, synchronization, and administrative mechanisms do not silently recombine those resources.
- **Narrow necessary interfaces.** Expose the smallest required operations and information. Avoid broad file-system mounts, unrestricted database queries, or general management interfaces when a limited service can meet the need.
- **Bound resource consumption.** Reserve or enforce capacity for critical system elements. Analyze processor scheduling, memory, storage, network bandwidth, queue depth, locks, and service limits where contention could produce loss.
- **Protect the common mechanism.** Minimize its functionality and privilege, control its administration, and make its assumptions explicit. A central service entrusted with several domains can require greater assurance than any individual client.
- **Separate consequential authorities.** Where one common authority could defeat multiple protections, distribute approval or allocate administration to independently protected mechanisms. Confirm that a single lower-level account cannot override the intended distribution.
- **Preserve independent protection and recovery.** Prevent operational system elements from modifying the mechanisms and restoration resources expected to constrain or recover from their compromise. Examine common management paths as well as storage location.

A common identity service illustrates the trade. It can improve account life cycle control and policy consistency, yet its compromise can affect every dependent domain. Separate relying-party policies may limit the operations available to ordinary users while leaving the identity authority highly consequential. Engineers should assess whether critical functions require separately protected authorization, constrained credential acceptance, local continuity arrangements, or additional approval. The answer depends on the loss model and the compromises the system must tolerate.

Management services often create greater coupling than application interfaces. A deployment platform can install code across otherwise separated workloads. A backup administrator can change retention and restoration data. A maintenance service can reconfigure several controllers. These capabilities belong in the

[2] *Mutable state* is information a system retains that can change during operation—for example, control settings, equipment status, or an AI agent's task history. *Shared mutable state* means several system elements use the same changeable information. If one system element changes or corrupts it, the change can affect the behavior of others that rely on it.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

sharing analysis even when they are exercised infrequently. The architecture should show which authority can change each boundary and what prevents one compromise from rewriting the established protections.

Shared infrastructure requires a similarly precise argument. Virtual machines may use separate memory address spaces while depending on a common hypervisor and storage controller. Containers may depend on one host kernel. Separate cloud accounts may remain under a common organizational administrator. Such arrangements can be justified, but the claim must state what the platform enforces and what happens if the platform or its administration is compromised. Logical separation does not remove the need to assess the shared enforcement mechanism.

AI-enabled architectures should allocate task memory, retrieval access, tool credentials, and orchestration state according to purpose. Avoid giving several agents a common writable context merely to simplify coordination. Where coordination requires common information, define what each agent can contribute and which independently enforced checks precede consequential actions. Treat externally sourced material and other agents' outputs as inputs with explicit trust limits. A shared memory label provides no assurance about how its contents influence behavior.

Least Sharing can introduce consistency and maintenance obligations. Separate configuration stores can drift; copied information can become stale; dedicated instances can be patched unevenly. The design should specify approved sources, validation, controlled distribution, and reconciliation without granting every system element authority over the common baseline. Reducing one sharing path while creating an unrestricted synchronization path can recreate the original weakness.

> ***"Separate instances provide a boundary only when their common dependencies support it."***

6. Engineering for Compromise

Engineering for Compromise assumes that selected system elements can behave maliciously and asks which security properties remain dependable [2]. The design principle of *Least Sharing* contributes by removing unnecessary paths through which that behavior can affect other system elements. Necessary common resources become explicit objects of the compromise analysis—rather than trusted connections that disappear from consideration once their normal operation has been described.

For the water treatment facility, assume an adversary controls the supervisory application of one process line. Determine whether it can alter shared configuration, poison the common reporting service's status cache, exhaust that service, reach a shared maintenance mechanism, or obtain credentials used by the other process line. The analysis should use the authority available to the compromised system element, including the delegated functions and information it can place in resources consumed by more trusted elements.

Next, assume that a shared service is compromised. A service that correctly isolates clients during normal operation may be able to violate that isolation when its implementation or administrator becomes hostile. Identify which assurance claims depend on that service remaining trustworthy and which protections are independently enforced. If compromise of the service is outside the tolerated model, it should be noted. If it is inside the model, the architecture needs constraints that can survive its loss of cooperation.

The distinction between system element compromise and mechanism compromise is essential. A database can enforce separate access policies against a hostile client while remaining a common source of failure if the database engine is compromised. A hypervisor can isolate hostile guests while its own compromise

© 2026 RONROSSECURE, LLC | CC BY 4.0.

defeats that isolation. Both are legitimate subjects for engineering, but they support different containment claims and require different evidence.

The following compromise scenarios help engineers evaluate how shared resources can transmit harmful influence and which protections remain effective when system elements, common mechanisms, or supporting authorities become hostile:

- **Compromised consumer.** Attempt to obtain another system element's information, consume its reserved capacity, infer sensitive state through the common resource, or invoke broader operations through confused authority.
- **Compromised contributor.** Attempt to insert malicious or misleading state, change ordering, supply outdated information as current, and influence a downstream action through a trusted consumer.
- **Compromised common mechanism.** Determine whether independently enforced restrictions still protect critical system elements. Identify every claim that fails when the mechanism no longer enforces policy.
- **Compromised administrator.** Exercise the ability to modify membership, permissions, configuration, deployment, and recovery. Test whether one administrative path can erase several nominally separate boundaries.
- **Compromised restoration source.** Determine whether recovery can reintroduce shared corruption or reconnect a hostile dependency. Protect the images, configuration, data, and mechanisms used for restoration, and the decisions that admit recovered system elements into service.

Propagation need not involve code execution on the next system element. A corrupted common variable can cause several legitimate controllers to take harmful action. A poisoned knowledge collection can cause multiple agents to recommend the same unsafe change. A common clock or coordination service can misorder operations. Follow the path from attacker-controlled resource to consequential behavior, including the legitimate functions that transform information into action.

Recovery is particularly vulnerable to hidden sharing. Separate backup copies can remain under the same deletion authority. A recovery environment can depend on the compromised operational identity service. Several supposedly clean instances can load one corrupted baseline. The design principle of *Least Sharing* encourages separation of the operational and recovery resources whose simultaneous compromise would defeat restoration. Additional integrity and authorization mechanisms are needed to establish that the retained recovery basis is trustworthy.

Reducing sharing does not establish that every compromised system element is contained. A private resource can still be misused through its owner's authority, and a necessary interface can transmit harmful inputs. The principle of *Least Sharing* removes and narrows relationships; *Domain Separation*, *Mediated Access*, *Least Privilege*, *Least Functionality*, and other principles establish complementary restrictions. The containment argument must explain their composition under explicit assumptions.

7. Interaction with Other Principles

Least Sharing addresses which resources are common and how widely they are shared. Its strongest relationships are with the design principles that constrain system elements, govern interfaces, protect critical functions, and substantiate the resulting system behavior. Each principle makes a distinct contribution to the architecture, and their composition determines the protection achieved [1].

The following relationships show how complementary design principles reinforce the principle of *Least Sharing* by constraining necessary sharing and preserving the protection of the system as a whole:

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Domain Separation.** Establish domains with protection appropriate to their needs. *Least Sharing* reduces common resources that can reconnect those domains or undermine their independence. A shared management plane deserves the same scrutiny as an overt cross-domain interface.
- **Mediated Access.** Govern necessary use of a common resource and the operations performed through it. The mediator should constrain relevant content, authority, direction, resource consumption, and sequence. Mediation controls an allowed relationship; *Least Sharing* first asks whether that relationship is needed.
- **Least Privilege.** Limit the authority of each system element and of the common mechanism. A system element that needs a status value should not receive write authority over shared control configuration. A common administrator's scope also requires justification.
- **Least Functionality.** Minimize capabilities in the shared mechanism and its clients. Unnecessary administrative, scripting, or data transformation features can enlarge the paths through which one system element affects others.
- **Least Persistence.** Limit how long necessary sharing remains available. A temporary maintenance resource should not become a permanent connection among domains. Ending a session does not remove common state or undo changes already propagated.
- **Distributed Privilege.** Require appropriately allocated authority for consequential operations where one shared authority would be too powerful. Independent approvals must not collapse into credentials or policy controlled through the same unprotected mechanism.
- **Protective Failure and Protective Recovery.** Preserve required protection when a shared resource fails and when capability is restored. A protective response should avoid turning one failed dependency into simultaneous loss of all essential service. Restoration should not automatically recreate unjustified sharing.
- **Compositional Trustworthiness.** Assess the properties of the assembled system, including relationships introduced by common state and services. Separate component claims do not establish that their sharing arrangement preserves the required system and security behavior.
- **Substantiated Trustworthiness.** Support the sharing baseline, protective assumptions, and limits with evidence. The assertion that a service is enterprise approved does not demonstrate that its system elements and consequences are acceptable for a particular mission.

These relationships reveal why the principle of *Least Sharing* cannot be reduced to a single access setting. Access mediation can prevent an unauthorized request yet leave a justified user able to exhaust a shared resource. *Least Privilege* can narrow the user's operations while a shared administrator retains authority over every domain. *Domain Separation* can constrain normal traffic while recovery reconnects the domains through a common writable source. Review must address the complete arrangement.

NIST SP 800-160, Volume 2, Revision 1 offers related cyber resiliency techniques, including *Redundancy*, *Segmentation*, *Diversity*, and *Privilege Restriction* [3]. These resiliency techniques can support selected design decisions, but they do not replace the need to determine whether sharing is necessary and, if so, which resources must be shared and among which system elements. *Redundancy* may improve service continuity while retaining a common failure dependency. *Diversity* may reduce susceptibility to a common implementation defect while both implementations remain subject to the same administrator. The specific mechanism and the property it supports should be named.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

8. System Life Cycle Application

Least Sharing is a life cycle obligation. NIST SP 800-160, Volume 1, Revision 1 provides processes for translating stakeholder concerns into requirements, architecture, implementation, evidence, operation, and disposal [1]. Sharing decisions should remain traceable across these processes. Once consolidation becomes embedded in acquisition and operational arrangements, later attempts to restore independence can be costly or technically difficult.

- **Business or Mission Analysis.** Identify the activities that require coordination and the losses that common dependence could produce. Distinguish the need to exchange selected information from the need to share an entire resource or administrative mechanism.
- **Stakeholder Needs and Requirements Definition.** Establish expectations for collaboration, minimum service, confidentiality, integrity, availability, containment, safety, reconstitution, and recovery. Identify which functions must remain effective despite failure or compromise elsewhere.
- **System Requirements Definition.** Specify necessary system elements, common resources, permitted operations, capacity guarantees, information validity conditions, and protective behavior under failure. State the operating and compromise assumptions under which independence or bounded influence is required.
- **Architecture Definition.** Allocate resources and interfaces to domains and functions. Identify common management, identity, storage, communications, and recovery dependencies. Compare alternatives that remove or narrow sharing and record the selected trade.
- **Design Definition.** Specify ownership of common state, membership administration, mediation, resource allocation, validation, fault handling, and restoration. Ensure that enforcement mechanisms can support the claimed limits without relying on a system element assumed compromised.
- **System Analysis.** Evaluate propagation, contention, information paths, and common failure consequences. Analyze the complexity and consistency costs of separate resources and determine whether the complete design satisfies stakeholder protection needs.
- **Implementation.** Realize the specified isolation, access rules, state ownership, and capacity controls. Remove default shared credentials, broad administrative interfaces, and unnecessary common writable directories. Protect the configuration that establishes the sharing boundary.
- **Integration.** Examine the assembled system for newly introduced common mechanisms. Deployment tooling, monitoring agents, service accounts, synchronization, and supplier support can create sharing absent from component designs. Verify that interactions preserve the allocated properties.
- **Verification.** Demonstrate the required limits on system elements and influence. Exercise unauthorized access, malicious authorized use, resource exhaustion, corrupted shared state, and administrative changes under the stated assumptions. Observe effects at the resources and functions being protected.
- **Transition.** Establish the approved sharing baseline. Remove migration accounts, temporary shared folders used during installation and testing, temporary support paths, and test repositories. Confirm that operational and recovery environments have intended memberships and authorities before use.
- **Validation.** Determine whether the realized arrangement supports mission work while preserving the necessary protection. Include degraded operation, urgent maintenance, incident response, and recovery scenarios in which staff might otherwise bypass inconvenient boundaries.
- **Operation and Maintenance.** Monitor membership, permissions, resource consumption, new integrations, and changes to common services. Review consolidation proposals and emergency exceptions against the original loss concerns. Maintain the evidence supporting the arrangement.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Disposal.** Remove retired system elements, credentials, interfaces, and residual access. Address information in common storage, backups, snapshots, caches, and supplier systems. Ensure that removing one system element does not leave its authority or state available to a successor element without authorization.

The life cycle processes of *Configuration Management* and *Information Management* support these activities by maintaining the resource baseline, sharing justifications, approved exceptions, dependency models, and evidence. The *Risk Management* process informs decisions about unavoidable common exposure. The engineering record should identify the residual consequence accepted, the responsible authority, and the assumptions that make the decision acceptable. A recurring exception should trigger review of the design and operational need.

Acquisition must expose dependencies that the acquirer cannot infer from a product interface. Ask suppliers which resources are shared among customers or mission functions, which authorities can cross boundaries, and which protections survive a client compromise or common-service failure. Identify what the supplier can substantiate and what remains unknown. A claim of dedicated processing may leave identity, storage, support, and recovery common.

Change analysis is essential. A new monitoring service, agent coordinator, centralized configuration manager, or backup platform can reconnect functions that were previously independent. Even a read-only integration can introduce common information, capacity demand, or administrative software. Reassess the affected sharing claims when system elements, authority, mutable state, or underlying dependencies change.

9. Failure Patterns and Anti-Patterns

Least-sharing failures often arise when convenience is mistaken for necessity or visible separation is mistaken for independence. The following patterns deserve attention in architecture reviews and operational assessments:

- **Consolidation without a loss analysis.** Several functions use one service because it is cheaper or familiar. Document the required benefit, compare alternatives, and evaluate simultaneous consequences before accepting the common dependency.
- **Separate names over common mutable state.** Distinct applications, accounts, or namespaces consume one writable configuration or status store. Identify the actual state owners and consumers and constrain changes according to their consequences.
- **One administrator spans every boundary.** Application access is narrow while one management identity can reconfigure all domains and their recovery systems. Limit or distribute that authority and substantiate the independence claimed.
- **Read-only is treated as harmless.** Consumers cannot write directly but can receive poisoned data or overwhelm a common service. Validate consequential inputs and bound consumption as required by the loss model.
- **Recovery shares the operational point of failure.** Production and restoration copies use the same deletion authority or compromised service. Separate and protect the restoration basis against the compromise it is expected to overcome.
- **A diagnostic convenience becomes a bridge.** A support workstation, shared folder, or maintenance credential connects otherwise separated mission functions. Provide a constrained support path and bound its scope and duration.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Duplication retains common control.** Separate instances load the same writable baseline or remain under unrestricted common administration. Analyze those dependencies before claiming independent behavior or recovery.
- **Shared capacity has no protective allocation.** One system element consumes memory, storage, processor time, or queue space required by another. Establish and verify appropriate admission, reservation, scheduling, and exhaustion behavior.
- **Agent coordination becomes unrestricted influence.** Multiple agents write common memory that other agents treat as authoritative instructions. Separate task context and constrain how shared information can lead to consequential tool use.
- **Monitoring creates hidden coupling.** An observability platform collects data and deploys privileged agents across all domains. Include its collection, configuration, update, and administrative capabilities in the architecture and compromise analysis.
- **Synchronization recreates the original exposure.** Private copies are introduced, then an unrestricted synchronization service can overwrite them all. Control the distribution path, validate admitted changes, and identify who can authorize them.

These patterns combine. Separate controllers may have dedicated storage but one maintenance laptop. The laptop may use different user accounts but retain a common privileged deployment credential. Independent backups may then be managed through that same credential. Reviewing each local separation independently can miss the single path that defeats the system-level claim.

> ***"A common dependency can turn several apparently local failures into one mission failure."***

Operational pressure can also broaden sharing gradually. A service first used by two applications becomes an enterprise default. Temporary access becomes permanent group membership. A troubleshooting repository accumulates credentials and configuration from unrelated domains. Treat new system elements and new resource types as architectural changes. The fact that an existing service is approved does not establish that every expansion of its reach is acceptable.

10. Evidence and Assurance

A least-sharing claim requires evidence that sharing is necessary, the number of system elements sharing each resource is minimized, and the permitted relationships preserve specified protection. An access list shows who is configured to use a resource. It does not establish why the resource must be common, whether indirect system elements exist, or whether a compromised service or administrator can affect several protected functions.

A top-level claim might be:

> ***Within the defined system scope, resources are shared only where justified by mission and protection needs and among the fewest system elements necessary to meet those needs. Each permitted sharing relationship constrains influence and dependence to the specified limits under the stated operating conditions, life cycle transitions, and failure and compromise assumptions.***

This claim concerns justified and bounded sharing. It does not by itself establish prevention of every unacceptable loss. Supporting loss-prevention claims must address the complete architecture, including consequential authorized actions, physical behavior, and other paths unrelated to sharing. The argument should make those dependencies explicit rather than allow the design principle's name to imply a broader guarantee.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

The following supporting subclaims organize the assurance argument for justified and bounded sharing and identify the evidence needed to substantiate each part of the top-level claim:

- **Subclaim 1: Complete scope and dependencies.** The inventory identifies consequential common resources, direct and indirect system elements, mutable state, supporting platforms, administrative paths, and recovery dependencies. Evidence includes architecture models, data-flow and authority analyses, configuration inventories, supplier disclosures, and integration inspection.
- **Subclaim 2: Justified necessity and minimal extent.** Each sharing arrangement has a traceable purpose, an analyzed alternative, and a justified system element set. Evidence includes requirements traceability, trade studies, membership rationale, approved exceptions, and review of interfaces that expose more resource or authority than required.
- **Subclaim 3: Enforced limits on influence.** System elements can perform required operations without exceeding allocated information, modification, or control limits. Evidence includes access and state ownership rules, mediation specifications, code review, interface tests, malicious-input exercises, and attempts to reach other system elements through indirect paths.
- **Subclaim 4: Bounded interference and dependence.** Required service and protective functions remain within specified limits despite system element misuse, contention, and the shared-resource failures included in the model. Evidence includes capacity analysis, scheduling and admission specifications, exhaustion tests, failure injection, and observations of protected mission behavior.
- **Subclaim 5: Protected common mechanisms and authorities.** The mechanism and its administration provide the trustworthiness required by the claim. Where their compromise is tolerated, independently enforced restrictions remain effective. Evidence includes privilege analysis, protected configuration, platform assurance, administrative separation, and compromise-oriented testing.
- **Subclaim 6: Preserved protection through change and recovery.** Maintenance, updates, emergency access, restoration, and disposal maintain or deliberately reestablish the approved sharing conditions. Evidence includes transition procedures, change impact analyses, operational drift records, recovery demonstrations, and membership removal tests.

Verification should distinguish unauthorized access to shared resources by system elements from prohibited effects caused by system elements authorized to use those resources. Attempt access through alternate interfaces, inherited group membership, administrative functions, service identities, delegated authority, and shared credentials. In addition, exercise a permitted system element's allowed operations maliciously. Correct denial of an unauthorized request does not establish that an authorized producer cannot corrupt a common input or that an authorized reader cannot exhaust the service.

Shared state requires evidence at both production and consumption. Determine whether modification is controlled, whether consumers validate the properties necessary for their decisions, and how outdated or contradictory values are handled. Where information from one domain is intended only for display, verify that it cannot become an implicit command or configuration input through another interface. Observe the consequential operation, not just the acceptance of the original request.

Capacity and failure tests should follow the minimum-service requirements. Exercise saturation, lock contention, storage exhaustion, slow responses, interrupted synchronization, and loss or degradation of the common resource. Measure the behavior of critical system elements and protective functions under those conditions. A service benchmark obtained with cooperative clients does not substantiate containment of a malicious system element's resource use.

The water treatment facility example provides a concrete assurance context. Assume the design is intended to tolerate compromise of one process line's supervisory application and loss of the common reporting

© 2026 RONROSSECURE, LLC | CC BY 4.0.

service. Local control and independently protected safety functions remain trusted. Each process line supplies selected status through a constrained interface; the reporting service has no control-write path. Control configuration and consequential maintenance authority are separately allocated. This is an illustrative design allocation, not a claim that such arrangements alone establish safe treatment.

For unauthorized control changes, the supporting argument must show that neither the compromised application nor the reporting service can modify control configuration directly, through management authority, or by information that a controller treats as authoritative. Tests attempt those specific paths and examine the actual controller state. The independent checks on consequential control changes require their own evidence. An assumed one-way reporting interface does not provide the necessary assurance if another management path remains available.

For simultaneous service loss, the argument must show that a hostile application or failed reporting service cannot deprive both process lines of required local capacity or necessary control information. Evidence should address shared networks, identity, storage, administration, physical process dependencies, and power to the extent they affect that claim. If an unanalyzed common dependency remains, narrow the claim or complete the analysis. Separate application instances cannot support a broader claim of facility independence.

Recovery testing should confirm that restoration of one process line does not require writable access to the other process line's configuration or reconnect the compromised reporting resource as a source of authority. Demonstrate access to a protected restoration basis under the assumed compromise conditions. Inspect the resulting memberships, credentials, configuration, and interfaces before admitting the restored system element into operation.

"An assurance argument must follow a shared dependency to the consequence and loss it can produce."

The assurance case should connect claims, reasoning, evidence, assumptions, and limitations. Identify common mechanisms that remain trusted, supplier evidence that cannot be independently checked, and operating modes that change the sharing baseline. Tests demonstrate selected behaviors under exercised conditions; analysis supports coverage of relevant paths and states. Neither a clean test nor the absence of an observed incident proves that every sharing relationship is justified or constrained.

Maintain the argument as the system evolves. Adding a system element, widening an interface, changing a recovery platform, or consolidating administration can invalidate earlier evidence. Operational records can help detect drift, but the baseline must continue to explain the current resource relationships and their consequences. Assurance is sustained through change, not completed by an initial access review.

11. Key Engineering Takeaways

The following takeaways summarize the engineering decisions and assurance obligations needed to apply the design principle of *Least Sharing* throughout the system life cycle:

1. *Least Sharing* applies to common functions, state, resources, variables, interfaces, and services, including the mechanisms that administer and restore them.
2. Derive sharing boundaries from unacceptable losses, protection needs, and required collaboration. Organizational convenience does not establish an acceptable mission dependency.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

3. Apply NIST's two criteria: share only when necessary, then limit sharing to as few system elements as possible. Record the trade and the rationale for including each system element in the sharing arrangement.
4. Analyze both influence and dependence. Corruption, disclosure, contention, and common failure can propagate through a resource without a direct interface between system elements.
5. Separate names, accounts, copies, or instances do not establish independence when state, administration, platform enforcement, or recovery remains common.
6. Necessary sharing requires constrained interfaces, appropriate authority, validated consequential inputs, and bounded resource use. Read-only access addresses only part of the problem.
7. *Engineering for Compromise* tests hostile system elements, common mechanisms, administrators, and restoration sources under explicit assumptions. Distinguish the claims supported by each scenario.
8. Compose the principle of *Least Sharing* with the principles of *Domain Separation*, *Mediated Access*, *Least Privilege*, *Least Functionality*, *Least Persistence*, *Protective Failure* and *Protective Recovery*.
9. Preserve sharing limits across integration, maintenance, emergency operation, update, recovery, and disposal. New common dependencies require renewed analysis.
10. Assurance must substantiate necessity, minimal participation, bounded effects, and maintained protection. A least-sharing claim contributes to the wider mission assurance argument without replacing it.

12. Conclusion

Least Sharing turns a familiar efficiency decision into an explicit security engineering obligation. Every common resource creates a relationship among the system elements that use, modify, administer, or depend on it. That relationship can support essential collaboration, but it can also carry corruption, disclose information, enable interference, or concentrate the consequences of one failure. The architecture must explain why the relationship is needed and how far it should reach.

NIST provides a direct foundation: share only when necessary and among as few elements as possible [1]. Applying those criteria requires more than reducing the length of an access list. Engineers must identify the actual resources and system elements, follow common state and authority, specify what influence remains acceptable, and compare architectural alternatives. Logical partitioning and dedicated instances support different claims; both require attention to their underlying dependencies.

Engineering for Compromise sharpens the analysis. Assume that one system element becomes hostile, then examine the common mechanism and its administration. Determine which protections remain dependable and which are lost as a result of that dependency. Reduce unnecessary connections and substantiate the constraints on necessary ones. Preserve a trustworthy recovery basis that the same compromise cannot silently defeat [2].

Producers should make these design choices and their limits visible. Consumers should ask which mission functions depend on common resources, who can change them, how one system element can affect another element, and what evidence demonstrates the intended containment. Those questions apply to conventional enterprise systems, operational technology, Internet-of-Things (IoT) devices, and AI-enabled architectures alike.

A trustworthy secure system enables required collaboration while limiting unjustified common influence. Applying the principle of *Least Sharing* establishes that expectation as an emergent system property to be designed, realized, and sustained throughout the system life cycle.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

"Share what the mission requires. Limit who and what must depend on it. Substantiate the boundary."

Acknowledgments

The author gratefully acknowledges the support provided by Dartmouth College's Institute for Security, Technology, and Society (ISTS). The resources provided by the ISTS contributed to the research and development of this paper. The views and conclusions expressed herein are those of the author and do not necessarily represent the views of Dartmouth College.

References

[1] R. Ross, M. Winstead, and M. McEvilley, Engineering Trustworthy Secure Systems, NIST Special Publication 800-160, Volume 1, Revision 1, National Institute of Standards and Technology, November 2022. https://doi.org/10.6028/NIST.SP.800-160v1r1

[2] R. Ross, "Engineering for Compromise: Designing Systems That Remain Trustworthy Secure Under Adversity," RONROSSECURE, LLC, August 2026. https://ronrossecure.com/library

[3] R. Ross, V. Pillitteri, R. Graubart, D. Bodeau, and R. McQuaid, Developing Cyber-Resilient Systems: A Systems Security Engineering Approach, NIST Special Publication 800-160, Volume 2, Revision 1, National Institute of Standards and Technology, December 2021. https://doi.org/10.6028/NIST.SP.800-160v2r1

© 2026 RONROSSECURE, LLC | CC BY 4.0.