Least Privilege

Allocating Only the Authority Required to Accomplish Specified Functions

Dr. Ron Ross

CEO, RONROSSECURE, LLC
ISTS FELLOW, DARTMOUTH COLLEGE

Abstract

Least privilege is a foundational security design principle for developing trustworthy secure systems. NIST Special Publication 800-160, Volume 1, Revision 1 states that each system element is allocated privileges necessary to accomplish its specified functions but no more. The statement applies to hardware, firmware, software, and personnel, and its engineering implications extend beyond restricting administrator accounts. Privilege determines what a system element can reach, change, invoke, delegate, disable, and cause. Every unnecessary privilege expands the consequences of error, misuse, failure, or compromise. This article examines least privilege as a systems security engineering principle. It connects function allocation to unacceptable loss, privilege decomposition, architecture, engineering for compromise, complementary design principles, life-cycle processes, failure patterns, and assurance evidence. The central proposition is that authority must be justified by a specified function and bounded by the consequences of its exercise. A system element should receive sufficient authority to perform its assigned work, under the conditions in which that work is required, while the architecture prevents that authority from becoming an unnecessary pathway to mission loss. Least privilege is therefore both an allocation discipline and a constraint on the capabilities an adversary inherits when a system element becomes untrustworthy.

1. The Engineering Problem

Modern systems[1] distribute authority across people, processes, services, devices, controllers, external dependencies, and infrastructure. A service reads data; an administrator changes a configuration; a deployment pipeline installs software; a controller directs an actuator; a recovery service restores state; and an automated agent invokes tools. Each function requires some ability to affect the system. The engineering problem is determining how much *authority* that function actually requires and ensuring that the realized system grants no more.

Convenient implementation choices often obscure that determination. One service identity may support many unrelated workloads. A maintenance role may include access to mission data because the platform (sometimes unwittingly) bundles those permissions. A software component may run with administrative authority because separating its privileged operations requires additional design work. Broad privilege simplifies initial deployment while transferring complexity and consequence into operation, compromise containment, and assurance.

"Every unnecessary privilege is an unnecessary opportunity to cause loss."

The distinction between intended use and available authority is decisive. A component may ordinarily read one record even though its credential permits it to read every record. An operator may normally change one parameter even though the interface permits disabling the protective mechanism. Documentation of

[1] *Systems* include information technology (IT) systems, operational technology (OT) systems, Internet-of-Things (IoT) devices, and cyber-physical systems.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

expected behavior does not constrain a malicious component. The architecture must limit what the element can *actually* do.

NIST SP 800-160, Volume 1 treats security in terms of freedom from conditions that can cause asset loss with unacceptable consequences [1]. The design principle of *Least Privilege* addresses the authority that can create those conditions. It asks which system elements need the ability to influence each protected asset and whether the scope of that influence is justified by their *specified functions*. Privilege is therefore a consequential engineering allocation, not merely a setting on a user account.

Insufficient privilege is also an engineering failure. A controller unable to perform a required protective action, an operator unable to manage an emergency, or a recovery service unable to restore essential capability can cause loss through *denied function*. The objective is sufficient, narrowly justified authority. Designers must establish both the lower bound needed for mission performance and the upper bound beyond which authority becomes unnecessary or intolerably consequential.

This problem grows as systems incorporate machine identities, autonomous agents, cloud control planes, and complex service relationships. A modest permission may confer much greater authority through another component. Permission to modify a workload specification may enable execution under a more privileged identity. Permission to replace a recovery artifact may determine which software later executes with system authority. **Effective privilege must include those indirect paths**.

2. What the Principle Means

NIST SP 800-160, Volume 1 states the *Least Privilege* principle as follows: “Each system element is allocated privileges that are necessary to accomplish its specified functions but no more.” System elements include hardware, firmware, software, and personnel. The accompanying NIST discussion identifies two desirable effects: minimizing the impact of failure, corruption, or misuse and simplifying analysis of the system element [1].

Three obligations follow from this formulation. First, functions must be specified well enough to determine what authority is necessary. Second, privilege must be decomposed finely enough that necessary authority can be *allocated* without automatically granting unrelated capability. Third, the system must enforce the allocation so that the element cannot exceed the intended scope of action.

Privilege includes more than access to information. NIST identifies the ability to modify, delete, use, or configure a resource and to authorize, start or enable, or stop or disable a process. In a broader system context, authority may include issuing physical commands, selecting executable software, changing policy, releasing keys, allocating resources, delegating permissions, or altering evidence on which other system elements depend.

A useful privilege specification identifies the system element, the function it performs, the object or resource, the permitted operation, and the conditions of use. Conditions may include parameter limits, system state, mission phase, approval, duration, volume, or the relationship between the requesting system element and the target object. These are engineering refinements of the design principle rather than additional wording in its stated definition.

“Privilege should follow the specified function, not the convenience of the implementation.”

© 2026 RONROSSECURE, LLC | CC BY 4.0.

The phrase “but no more” requires a rationale for each grant. Habit, product defaults, organizational seniority, potential future use, and the possibility that a system element might someday need an operation are insufficient bases. Where broader authority is unavoidable, the engineering record should identify the limitation, its consequences, compensating architectural constraints, and the stakeholder decision that accepts the residual *risk*.

Privilege allocation is also sensitive to decomposition. If one element performs unrelated functions, the union of their required permissions may be excessive for any one activity. Separating functions into protected elements can allow each to receive a smaller authority set. The separation must be real: two processes sharing the same credential or an unrestricted management interface do not necessarily have distinct effective privileges.

Least privilege is maintained as functions, operating conditions, and system relationships *change*. A grant justified during installation may be unjustified during operation. Authority needed for a specific repair may be unnecessary after the repair. The privilege allocation must remain traceable to continuing need rather than accumulate with every deployment, exception, and change.

3. What the Principle Does Not Mean

Least privilege is often reduced to removing local administrator rights or refining access control roles. Those actions can help, but they do not exhaust the principle.

- **Least privilege does not mean human accounts alone.** Machine identities, firmware update mechanisms, hardware controllers, service processes, orchestration systems, suppliers, and autonomous agents also exercise authority. Their privilege can exceed that of any ordinary user.
- **Least privilege does not mean deny everything.** Required functions must remain achievable. Removing permissions without understanding mission dependencies can create availability or safety loss and encourage operators to bypass the intended controls.
- **Least privilege does not mean read-only access is harmless.** Reading sensitive data, model weights, credentials, private keys, or recovery secrets can enable irreversible disclosure or subsequent compromise. Read authority must be justified by the function and consequence.
- **Least privilege does not mean a narrow role name proves a narrow capability.** Inheritance, group membership, resource policies, service delegation, impersonation, and management-plane access may confer authority far beyond the visible role definition.
- **Least privilege does not mean temporary authority is necessarily limited authority.** A short-lived credential can still permit catastrophic action immediately. The design principle of *Least Persistence* constrains duration; *Least Privilege* constrains the authority available during that duration.
- **Least privilege does not mean distributing one broad credential among several system elements.** Sharing that credential expands exposure while preserving the authority it confers. Distinct identities and bounded grants are needed to distinguish and constrain each system element.
- **Least privilege does not mean monitoring substitutes for prevention.** Logging can support detection and accountability, but a recorded destructive action may already have produced unacceptable *loss*. Enforcement must act before the prohibited effect where prevention is required.
- **Least privilege does not mean every authorized operation is safe under malicious use.** A required operation may be abused through repetition, sequence, timing, or combination. Constraints on effects may require mediation, independent approval, or architectural redesign.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

The principle is best understood as a *system property*: each system element's effective *authority* remains necessary for its *specified functions* and bounded by enforceable constraints throughout the relevant states and modes.

4. Loss-Driven Interpretation

A least-privilege *architecture* should begin with the assets stakeholders must protect and the outcomes they must achieve. An inventory of existing permissions is useful, but it cannot determine whether those permissions are adequate. Existing permissions may reflect deployment history rather than stakeholder requirements and protection needs. Loss analysis supplies the basis for deciding which operations are necessary and which create unacceptable exposure.

A *loss-driven* chain for least privilege can be expressed as:

> **Asset → Unacceptable Loss → Loss-Enabling Condition → Security Constraint → Specified Function → Privilege Allocation → Enforcement → Evidence.**

Consider a critical AI inference service whose model weights are strategically valuable intellectual property. One specific unacceptable loss is unauthorized disclosure of those model weights. A loss-enabling condition is an externally exposed request-processing system element having authority to read the weight store or request unrestricted memory dumps. Its specified function is to validate and forward inference requests, not to administer model storage. The security constraint should therefore exclude direct access to plaintext weights, unrestricted debugging, export functions, and the credentials that confer those capabilities.

The inference execution system element presents a harder problem. It may legitimately require access to model weights to perform its function. Least privilege cannot eliminate that necessary access without changing the system's functional allocation. System designers must then ask whether the execution element can also export weights, alter release policy, reach arbitrary destinations, or direct a more privileged management service. The security design principles of *Domain Separation*, *Mediated Access*, *Distributed Privilege*, and restrictions on interfaces may be necessary to prevent required computational access from becoming unrestricted disclosure capability.

Consider the security objective of *integrity* in a physical system. A maintenance tool may need to read diagnostic state and replace an approved configuration. It need not necessarily disable safety interlocks, issue unrestricted actuator commands, or install arbitrary firmware. The privilege allocation should distinguish diagnostic observation, bounded configuration change, software replacement, and operational control. Treating these capabilities as one maintenance role can make a compromised tool sufficient to create a hazardous condition resulting in asset loss or mission failure.

The security objective of *availability* creates equally important constraints. A system service that writes transaction records may not need authority to delete every record, modify retention settings, or exhaust shared storage. Conversely, a recovery service may need enough authority to restore essential data during degraded operation. The engineering analysis must address both malicious exercise of available authority and *loss* caused by inability to perform required work.

A permission can be necessary and still too consequential to allocate to one system element acting alone. If a required function can destroy all recovery copies, release a strategic secret, or disable the final protective barrier, the design should examine independent authorization or a different allocation of functions. Least privilege narrows the authority set; loss analysis determines whether the remaining authority is tolerable.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

"Necessary authority still requires an acceptable consequence of misuse."

The resulting allocation is defensible because every privilege can be traced to a specified function and every high-consequence capability can be traced to a protection decision. Engineers can then explain what was withheld, what remains available, and why the system can tolerate its exercise under the stated design assumptions.

5. Engineering the Principle into the Architecture

Architects realize the design principle of *Least Privilege* through coordinated decisions about function, identity, interfaces, authority, and enforcement. Removing individual permissions late in deployment cannot compensate for an architecture that requires all system elements to act with system-wide power. The following actions guide the engineering process.

- **Specify the functions before granting authority.** Identify each system element's required behavior, inputs, outputs, dependencies, and operating modes. A function description such as "manage the system" is too broad to support a meaningful minimum privilege allocation.
- **Inventory privilege-bearing system elements and resources.** Include personnel, workloads, devices, controllers, update services, deployment tools, recovery components, keys, policy services, and external providers. Identify physical and procedural authority as well as logical permissions.
- **Decompose privileges at the level of consequential operations.** Separate reading from modifying, modifying from deleting, configuration from code execution, and ordinary operations from policy administration. Distinguish access to selected objects from access to an entire class of resources.
- **Separate functions when their combined authority is excessive.** Move high-consequence operations into narrowly scoped functions or protected services. An application that occasionally requires one privileged operation should not automatically execute all of its code with that privilege.
- **Allocate distinct identities where distinct authority is required.** Avoid shared credentials that give every workload the union of all workloads' needs. Bind identity to the relevant system element and invocation context strongly enough to prevent one element from adopting another's authority.
- **Constrain objects and operation parameters.** Restrict resource identifiers, paths, destinations, command vocabularies, ranges, quantities, and permitted state transitions. A generic execution interface can defeat an otherwise narrow access policy by allowing arbitrary effects behind one authorized function call.
- **Analyze effective authority and privilege amplification.** Follow inheritance, delegation, impersonation, writable executable content, credential access, policy changes, and workload creation. Permission to change what a privileged system component executes can be equivalent to permission to exercise that component's authority.
- **Make enforcement unavoidable.** Allocate access controls to protected mechanisms that can constrain the actual effect. Application conventions do not enforce least privilege when direct connections to databases, local consoles, diagnostic ports, or management interfaces provide alternate paths.
- **Bound elevation and delegation.** Define who may request, approve, activate, renew, and revoke additional authority. Prevent a subject from approving its own elevation or delegating a broader capability than it is permitted to exercise. Protect the service that issues the elevated authority.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Design state-specific authority.** Define grants for startup, normal operation, degraded operation, emergency response, maintenance, recovery, and retirement. A role may need different privileges in each state, but state changes themselves require controlled authorization.
- **Protect the mechanisms that define privilege.** Policy stores, identity roots, signing keys, orchestration systems, configuration pipelines, and recovery artifacts determine downstream authority. Their administration and update paths belong within the privilege analysis.
- **Maintain a traceable privilege baseline.** Record the function, object, operation, condition, owner, enforcement point, justification, and verification method for each material privilege grant. Changes should update the rationale and assurance evidence, not merely alter the configuration.
- **Revocation deserves explicit design.** Preventing future authorization does not necessarily stop an operation already underway, invalidate an established session, remove a cached credential, or undo information already disclosed. The architecture should specify which effects revocation can terminate, the maximum delay before it takes effect, and what protective action follows incomplete or uncertain revocation.
- **Resource authority also requires attention.** An element may have no administrative permission yet be able to consume enough memory, processing capacity, queue space, or communication bandwidth to deny critical functions. Quotas, scheduling isolation, bounded requests, and admission constraints can limit that influence. Their values must be derived from mission needs and adverse condition analysis rather than assumed adequate because limits exist.
- **Least privilege should be evaluated at the system's actual enforcement granularity.** If a product permits only an all-or-nothing administrative role, a policy document cannot create finer authority. Systems engineers must change the architecture, add a dependable mediation layer, select a different implementation, or explicitly qualify the claim. The realized mechanism determines the available boundary.

"Analyze what a system element can cause, not only what its permission list says."

---

6. Engineering for Compromise

Engineering for compromise assumes that selected system elements may become untrustworthy and examines whether the architecture can continue to prevent *unacceptable loss* [2]. The design principle of *Least Privilege* directly determines the authority available to an adversary controlling such an element. A legitimately granted permission does not become benign merely because the adversary uses it through an authorized interface.

For each system element, assume complete adversary control within the stated compromise model. Exercise every granted operation, every reachable resource, every permitted parameter range, and every delegation path. Ask what data can be disclosed, what state can be altered, what protective mechanism can be disabled, and what other system element can be induced to act. Include operations that are valid individually but damaging in combination.

This examination distinguishes policy compliance from *loss containment*. An authorization service may correctly permit a workload to delete backups because its role includes deletion. However, if the workload is compromised, correct enforcement can faithfully enable catastrophic *loss*. The system defect lies in the allocation and architectural consequence of authority, even though the enforcement mechanism operates as specified.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

"Assume the adversary will use every privilege the compromised element possesses."

Consider a deployment pipeline with authority to install software in production. Removing its direct access to mission data may appear to limit its privilege. If it can install arbitrary code that later executes under a data-reading identity, the pipeline still possesses an indirect route to the data. The system-wide analysis must include code provenance, approval, signing, release authorization, and the runtime authority of installed components.

Engineering for compromise also examines the privilege-administration plane. A system of carefully restricted workload roles offers little containment if one compromised administrator or policy service can replace all restrictions. High-consequence changes may require the application of the security design principle of *Distributed Privilege*, independently protected approval, constrained security policy languages, protected roots, and recovery mechanisms that cannot be rewritten through the same compromised path.

Least privilege cannot guarantee that a compromise is harmless. Some system elements need substantial authority to provide their assigned functions. A controller element may legitimately issue safety-relevant commands; an inference process may legitimately access sensitive model states. The security argument must identify those unavoidable capabilities and establish which independent constraints remain effective if the system element misuses them. Where no adequate constraint exists, the design basis and residual risk must be explicitly stated.

NIST SP 800-160, Volume 2, Revision 1 provides complementary cyber resiliency concepts including techniques and approaches. The cyber resiliency technique of *Privilege Restriction* includes resiliency approaches such as *Trust-Based Privilege Management*, *Attribute-Based Usage Restriction*, and *Dynamic Privileges* [3]. These approaches can condition or adjust authority using relevant attributes and changing circumstances. The cyber resiliency technique of *Non-Persistence* can further limit the period during which resources or authority are exposed. **They support the analysis but do not substitute for justified function allocation or dependable enforcement.**

Dynamic restriction requires trustworthy inputs and a defined protective response. If changing privilege depends on telemetry supplied by the compromised system element, the adversary may conceal the trigger for restriction. If loss can occur before detection and revocation, a reactive mechanism cannot substantiate preventive containment by itself. Static constraints, independent observations, and bounded operating modes may be needed.

Applying the principle of *Least Privilege* as part of engineering for compromise requires examining both communication paths and the authority one system element can exercise over another [1, 2]. A write permission to a configuration file, access to a signing operation, or authority to restore an image may bridge otherwise separated domains. The aim is to make component compromise, expanded system compromise, and unacceptable mission loss distinct events separated by engineered constraints.

7. Interaction with Other Principles

The security design principle of *Least Privilege* works within an architectural system of principles. Other principles determine where authority is allocated, how it is enforced, how its consequences are bounded, and what evidence supports confidence in the allocation.

- **Domain Separation** places elements with different protection needs behind distinct boundaries. *Least Privilege* determines the authority each element may exercise within and across those boundaries.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Mediated Access** enforces limits on access to and operations on system elements. A minimal privilege specification has no protective effect when the relevant operation can bypass its enforcement point.
- **Distributed Privilege** requires multiple authorized entities for specified privileged operations. It can prevent one compromised system element from independently exercising a necessary but high-consequence capability.
- **Least Sharing** reduces common resources, credentials, and state through which one system element can affect others. A shared identity can defeat otherwise careful function-level allocations.
- **Least Functionality** removes unnecessary capabilities from the system element itself. *Least Privilege* limits authority over the capabilities and resources that remain. Both principles can reduce adversary opportunities, but they address different design decisions.
- **Least Persistence** limits how long selected system elements, resources, or relationships remain available. Used with *Least Privilege*, it constrains both the scope and duration of authority without assuming that short duration makes broad authority safe.
- **Structured Decomposition and Composition** organizes functions into analyzable system elements with defined interfaces. Appropriate decomposition permits fine-grained privilege allocation; inappropriate composition can restore the union of privileges that decomposition sought to separate.
- **Protective Defaults** establishes protective behavior where authorization has not been explicitly justified. Newly created workloads, resources, interfaces, and roles should not silently acquire broad permissions through convenience defaults.
- **Minimal Trusted Elements** reduces the number of system elements whose correct behavior is essential for system protection. Narrowly scoped privileged functions can reduce the amount of code that must be trusted, provided their interfaces and dependencies preserve the constraint.
- **Reduced Complexity** makes privilege rules, inheritance, delegation, and enforcement easier to analyze. A complicated policy may hide authority that no individual rule appears to grant.
- **Continuous Protection** extends privilege constraints through transition, maintenance, failure, and recovery. Temporary operational needs do not eliminate the need for a defined boundary on authority.
- **Protective Failure** and **Protective Recovery** preserve critical constraints under faults and restore justified authority from authenticated states. Recovery must not reinstate obsolete permissions or depend on an unrestricted operational credential.
- **Anomaly Detection** can reveal unusual privilege use, unexpected grants, failed escalation attempts, and policy drift. Detection supplements constraints and helps determine when the current allocation is no longer appropriate.
- **Compositional Trustworthiness** and **Substantiated Trustworthiness** require evidence that system element-level allocations and their interactions preserve the claimed system-level limit on authority. The principle of *Commensurate Rigor* guides the depth of that evidence according to consequence.

The broader lesson is that least privilege does not exist solely in an access control database or operating system. It emerges from the relationship among functional allocation, protected boundaries, constrained interfaces, policy administration, operating states, and the trustworthiness of their composition.

8. System Life Cycle Application

The design principle of *Least Privilege* must be engineered across the complete system life cycle. Privilege accumulates through provisioning, integration, maintenance, emergency exceptions, and retirement unless

© 2026 RONROSSECURE, LLC | CC BY 4.0.

the engineering processes preserve a continuing relationship between specified function and granted authority. The system life cycle processes in NIST SP 800-160, Volume 1 provide that framework [1].

- **Business or Mission Analysis** identifies the capabilities stakeholders require, unacceptable losses, operational responsibilities, and circumstances in which authority is essential. It establishes why a privilege allocation matters to mission outcomes.
- **Stakeholder Needs and Requirements Definition** captures protection needs, legitimate duties, prohibited effects, emergency responsibilities, and accountability expectations. Stakeholders should identify operations whose consequences require independent authorization.
- **System Requirements Definition** converts those needs into verifiable constraints on subjects, objects, operations, parameters, delegation, elevation, revocation, resource use, and state-specific authority. Requirements should include authorized capability as well as prohibited behavior.
- **System Architecture Definition** allocates functions to system elements and establishes protected privilege boundaries, policy authorities, management domains, identities, and enforcement points. Architectural alternatives should be compared by effective authority and compromise consequence.
- **Design Definition** specifies privilege models, interfaces, credentials, policy rules, elevation protocols, approval paths, resource limits, and error handling. It identifies how the implementation will realize the required granularity and preserve the allocation.
- **System Analysis** examines direct and indirect authority, harmful operation sequences, privilege escalation, shared dependencies, administrative reach, and mission effects of restriction. It evaluates whether residual necessary privilege is tolerable under the compromise assumptions.
- **Implementation** realizes the allocation in code, hardware, firmware, configuration, physical controls, and procedures. Development shortcuts, embedded credentials, inherited roles, and installation defaults must not inadvertently expand the authorized scope.
- **Integration** examines the privileges created when system elements are composed. Service discovery, federation, orchestration, shared storage, and automation may introduce authority absent from the isolated system component designs.
- **Verification** demonstrates that specified functions succeed with allocated authority, unnecessary operations fail, escalation is bounded, and revocation behaves as required. Negative tests and compromise-oriented analysis examine the authority an adversary could exercise.
- **Transition** controls initial provisioning, installation privileges, migration, supplier access, and production activation. Temporary deployment authority should be removed before the operational baseline is accepted unless continuing need is explicitly justified.
- **Validation** determines whether the allocation supports stakeholder protection needs and real mission performance. Exercises should include emergencies, degraded operation, concurrent duties, recovery, and maintenance rather than routine workflows alone.
- **Operation** monitors privilege use, out-of-date grants, dormant identities, drift, delegation, and changes in mission responsibilities. Reviews should confirm that each material privilege grant still corresponds to a current specified function.
- **Maintenance** governs diagnostics, vendor access, patching, component replacement, and privilege changes. Exceptions should have a defined purpose, scope, approval, duration, monitoring, and termination condition.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **Disposal** terminates privileges, revokes credentials and trust relationships, removes sensitive material, and prevents retired system components from retaining influence. Shared resources must be examined before deletion so that retirement does not disrupt remaining system elements.

A periodic access review cannot by itself establish life-cycle continuity. Reviews may detect unnecessary privilege grants after they have existed for months or years. Provisioning, deployment, state transitions, and policy updates should enforce the intended allocation when changes occur, while operational review checks whether the underlying function and assumptions remain valid.

"The justification for privilege must survive every change in the system life cycle."

9. Failure Patterns

Systems frequently claim least privilege while retaining broad effective *authority*. Several failure patterns recur across technologies and operating environments.

- **The administrator-by-default pattern.** System components execute with full authority because that avoids permission errors during deployment. The architecture never separates privileged operations from ordinary processing.
- **The accumulated-role pattern.** New responsibilities add privilege grants while old responsibilities leave their permissions behind. The identity's authority describes its deployment history rather than its current function.
- **The shared-service-identity pattern.** Independent workloads use one credential with the union of their permissions. Compromise of the least trustworthy workload inherits the authority needed by the most consequential workload.
- **The nominally-read-only pattern.** A role is treated as low risk because it cannot write, although it can disclose credentials, intellectual property, sensitive operational state, or information that enables a compromise at a later time.
- **The hidden-execution pattern.** A system element cannot directly access a protected asset but can modify software, configuration, scripts, or recovery state used by an element that can. The apparent permission boundary does not bound the resulting authority.
- **The delegated-amplification pattern.** A constrained subject invokes a more privileged service that fails to bind the requested operation to the subject's permitted scope. The service becomes a path for exercising authority the subject was never meant to possess.
- **The temporary-superuser pattern.** Broad emergency or maintenance privilege is justified as short lived. Catastrophic action remains possible during the authorized interval, and termination of the grant may not end established sessions.
- **The self-authorized-elevation pattern.** The system element requesting increased privilege can approve the request, modify its own policy, or control the evidence on which approval depends. The nominal approval step adds no independent constraint.
- **The coarse-platform-role pattern.** A product exposes only broad roles, but documentation describes finer restrictions that the product cannot enforce. The security claim exceeds the implementation's actual granularity.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

- **The recovery-superuser pattern.** Backup and restoration system components can delete recovery copies, overwrite production, or install executable content without independent constraint. Compromise of the recovery infrastructure defeats both operation and restoration.
- **The revocation-assumption pattern.** Removing a role is assumed to remove authority immediately. Cached tokens, copied credentials, running jobs, long-lived connections, and disconnected devices continue to exercise the grant.
- **The required-operation blind spot.** Testing rejects unauthorized calls but never examines malicious repetition or combination of authorized calls. The system element stays within its permission list while producing unacceptable effects.

These patterns share a common defect: the design evaluates a named permission or role while overlooking the authority actually reachable through the system. Corrective analysis should trace operations to their consequences and include the administration of the privilege boundary itself.

"Removing a permission is ineffective when another path in the system confers the same authority."

---

10. Evidence and Assurance

A least-privilege claim requires evidence that allocated authority is both sufficient for specified functions and no broader than necessary within the stated scope. A policy export, role inventory, or privileged-access product demonstrates part of an implementation. It does not by itself demonstrate justified allocation, non-bypassable enforcement, or containment of malicious use.

A top-level claim might be:

> ***Each system element within the defined scope possesses only the effective privileges required for its specified functions under authorized operating conditions, and the allocation and enforcement of those privileges preserve the stated security constraints under the identified failure and compromise assumptions.***

Supporting subclaims should address completeness of the system element and resource inventory; direct and indirect authority; adequacy of functional specifications; justification for each material privilege grant; granularity of privilege decomposition; enforcement; identity binding; elevation and delegation; revocation; policy administration; system life cycle modes; resource influence; and the remaining high-consequence capabilities.

Evidence may include function-to-privilege traceability, subject-object-operation matrices, architecture models, interface specifications, policy analysis, inheritance and delegation graphs, configuration baselines, code review, privileged-function analysis, credential inventories, authorization tests, resource-exhaustion tests, adversarial exercises, revocation measurements, recovery demonstrations, and operational records of privilege grant changes. High-consequence allocations may warrant formal methods or systematic state and sequence analysis.

Verification should show that the system element can perform each specified function using only its allocated privileges. It should exercise initialization, routine operation, error handling, maintenance, and recovery where those functions fall within the specified scope. Removing an unnecessary permission should

© 2026 RONROSSECURE, LLC | CC BY 4.0.

not inadvertently shift the work to an unrecorded shared administrator identity or require operators to defeat the boundary.

Verification should also attempt access to adjacent resources, broader parameter ranges, unrelated administrative operations, forbidden delegation, and policy modification. Tests should also attempt identity substitution, credential extraction, writable-code exploitation, and invocation of a more privileged deputy. The question is whether the effective *authority* matches the privilege allocation, including effects obtained indirectly.

Compromise-oriented verification assumes the system element acts maliciously while retaining its legitimate authority. For an AI agent, this may include adversarial instructions or hostile tool inputs that cause it to attempt every operation available through its tools. Tool permissions must remain enforceable outside the agent's reasoning process; an instruction to avoid a prohibited action is not an independent authority boundary.

For a maintenance service, hostile-subject tests might attempt unauthorized firmware replacement, deletion of evidence, disabling of protections, repeated configuration changes, and alteration of recovery material. The evidence should distinguish operations the service cannot perform from necessary operations that remain dangerous and depend on independent constraints.

Revocation tests should measure behavior at every relevant enforcement point. They should distinguish expiration of a token, withdrawal of a role, closure of a session, cancellation of a system operation, and restoration of state. Information already read or exported cannot be recovered through the revocation of permissions. The *assurance argument* must account for that limit rather than imply that all consequences are reversible.

"An access review documents grants. An assurance argument explains their necessity and consequences."

An *assurance case* connects claims, reasoning, evidence, assumptions, and limitations. It should identify which authorities remain trusted, which compromises the design tolerates, which combinations exceed the design basis, and which operational modes alter the allocation. Claims of "no more than necessary" should acknowledge implementation granularity and accepted exceptions explicitly.

Evidence must be maintained as the system evolves. A new API, role, agent tool, deployment mechanism, supplier, or recovery procedure can create a new privilege path. Material changes should trigger renewed analysis and verification. Operational telemetry can reveal drift and unexpected use, but observations do not prove that unobserved harmful capabilities are absent.

11. Key Engineering Takeaways

The following key engineering takeaways are important when applying the principle of *Least Privilege*.

1. Least privilege applies to every system element that exercises authority, including hardware, firmware, software, personnel, and external supporting elements within the defined scope.
2. Privilege must be traceable to specified functions. Broad descriptions of responsibility cannot justify an unlimited set of capabilities.
3. The allocation must permit required mission behavior while excluding unnecessary authority. Both excessive privilege and insufficient privilege can create loss.

© 2026 RONROSSECURE, LLC | CC BY 4.0.

4. Effective authority includes inheritance, delegation, impersonation, code and configuration changes, credential access, resource consumption, and influence through more privileged system elements.
5. Fine-grained privilege requires appropriate functional decomposition and an implementation capable of enforcing the intended operation and resource boundaries.
6. Engineering for compromise treats every granted privilege as capability available to an adversary controlling the system element. Authorized operations must be analyzed for malicious consequences.
7. Necessary high-consequence authority may require *Distributed Privilege*, *Domain Separation*, *Mediated Access*, or redesign of the function allocation to prevent a compromise from causing unacceptable loss.
8. *Least Persistence* constrains exposure in time; it does not make excessive authority acceptable. Privilege elevation, delegation, and revocation need explicit limits and evidence.
9. Privilege administration, deployment, update, and recovery mechanisms can determine the authority of the whole system and must be included in the compromise analysis.
10. Assurance must show why each material privilege grant is needed, how its scope is enforced, and what happens when the system element misuses every capability still available to it.

The principle of *Least Privilege* provides the allocation discipline that makes authorized capability proportionate to the function it serves.

12. Conclusion

Least Privilege is a concise principle with system-wide consequences. NIST SP 800-160, Volume 1 states that each system element is allocated the privileges necessary to accomplish its specified functions but no more. Realizing that principle requires systems engineers to specify functions, decompose authority, allocate it to appropriate system elements, enforce the resulting limits, and substantiate the allocation with evidence.

The principle connects mission need to permissible influence. Stakeholders define the capabilities they require and the losses they cannot accept. Functional allocation identifies what each system element must do. Privilege allocation determines which operations and resources are necessary for that capability. Architecture constrains indirect authority and protects the mechanisms that grant, change, and revoke it. Assurance provides justified *confidence* that the realized system preserves those decisions.

Engineering for compromise makes the consequences of allocation explicit. Once a system element becomes hostile, its legitimate authority becomes adversary capability. Reducing unnecessary privilege narrows that capability and simplifies the analysis of what compromise can cause. Where necessary privilege remains consequential, the architecture must provide independent constraints or identify the limits of the containment claim.

Least privilege is therefore not complete when an organization removes administrator rights or performs an annual access review. It is sustained when each system element's actual authority remains justified by its current function, when changes cannot inadvertently expand that authority, and when adversarial use has been analyzed against unacceptable *loss*. The boundary must survive the system states in which protection is most difficult: degraded operation, emergency access, maintenance, update, and recovery.

"The engineering purpose of least privilege is to make authority no broader than the function it serves."

© 2026 RONROSSECURE, LLC | CC BY 4.0.

A trustworthy secure system gives its system elements enough authority to accomplish their assigned work while preventing avoidable authority from becoming a pathway to mission loss. The engineering question is not simply whether a system element has permission. It is whether that permission is necessary, whether its effects are bounded, and whether the evidence supports confidence in the answer.

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.