Domain Separation

Building Security Boundaries That Survive Compromise

Dr. Ron Ross

CEO, RONROSSECURE, LLC
ISTS FELLOW, DARTMOUTH COLLEGE

Abstract

Domain separation is one of the foundational architectural principles for trustworthy secure systems. NIST Special Publication 800-160, Volume 1, Revision 1 states the principle succinctly: domains with distinctly different protection needs are physically or logically separated. The engineering implications are considerably deeper. Domain separation determines where influence is permitted, where it is constrained, what interactions must be mediated, which failures or compromises may propagate, and where the architecture must preserve a boundary even when an element on one side becomes untrustworthy. This article examines domain separation as a systems security engineering principle rather than as a networking technique or access-control mechanism. It connects the principle to asset loss, protection needs, security requirements, architecture, engineering for compromise, complementary design principles, life-cycle processes, and assurance evidence. The central proposition is that a meaningful security domain is not defined merely by where components are placed. It is defined by a defensible limit on influence. A boundary is therefore trustworthy only to the extent that there is sufficient evidence that compromise, failure, or misuse within one domain cannot produce prohibited effects in another beyond the limits explicitly accepted by stakeholders.

1. The Engineering Problem

Modern systems are assembled from heterogeneous hardware, software, firmware, people, processes, facilities, networks, services, and external dependencies. These elements do not have uniform protection needs, trustworthiness, privileges, exposure, criticality, or consequences of failure. Yet system architectures routinely connect them because mission and business capabilities depend on interaction.

That creates a fundamental engineering tension: the system must permit enough interaction to accomplish its purpose while preventing those interactions from becoming uncontrolled paths of influence. A system component that processes untrusted external data may need to provide results to a mission-critical function. An administrator may need to maintain a system without acquiring unrestricted access to mission data. A cloud service may need to supply computation without being allowed to control security-critical state. A safety-critical controller may need data from a less trustworthy sensor network without allowing compromise of that network to become a compromise of the controller. The problem is, therefore, not simply connectivity. It is influence.

"The problem is not simply connectivity. It is influence."

Every interface creates some opportunity for one *system element* to affect another. That influence may involve information, commands, timing, resource consumption, configuration, privilege, physical effects, recovery state, or dependencies on shared services. When domains with different protection needs are allowed to influence one another without sufficient constraint, the security characteristics of the more critical domain can collapse toward those of the less trustworthy domain.

NIST SP 800-160, Volume 1, Revision 1 defines security in terms of freedom from conditions that can cause asset loss with unacceptable consequences and emphasizes that systems security engineering must control the events, conditions, and consequences that constitute loss. Domain separation addresses one of the central architectural questions created by that definition, that is, **where must influence stop**?

A well-engineered separation boundary provides an answer before an attack, fault, or misuse occurs. It identifies a set of system elements that share sufficiently similar protection needs, places them within a protected subsystem or *domain*, and constrains interactions across the domain boundary. The resulting architecture limits susceptibility to external influence and contains erroneous or malicious behavior originating within the domain. Domain separation is much more than segmentation. It is a structural means of controlling the reach of adversity.

"Domain separation is much more than segmentation. It is a structural means of controlling the reach of adversity."

Domain separation changes the consequence of initial compromise. In a flat architecture, initial access can become the starting point for discovery, lateral movement, privilege escalation, persistence, and ultimately mission loss. A *domain-separated architecture* can force an adversary to confront a new, independently enforced boundary before influence can propagate to another protection domain. Domain separation, therefore, attacks one of the consequential assumptions underlying modern cyberattacks, that is, **compromise of one reachable element creates a pathway to the rest of the system**. The engineering objective is not to claim that every cross-domain movement is impossible, but to make unauthorized propagation structurally constrained, independently mediated, and sufficiently difficult that initial compromise does not automatically become system compromise.

2. What the Principle Means

NIST SP 800-160 states the *Domain Separation* principle as follows: **"Domains with distinctly different protection needs are physically or logically separated."** The accompanying discussion explains that separation enhances control over system function and data flow by limiting the extent to which an entity or domain can influence, or be influenced by, another entity or domain.

Two complementary purposes are embedded in that formulation.

The first is *susceptibility reduction*: protecting a domain from influence by external entities. A high-assurance control function, for example, should not be freely influenced by a lower-assurance user environment merely because both execute on the same system.

The second is *containment*: protecting external entities from erroneous or malicious behavior occurring inside the domain. A domain that processes hostile content may be expected to fail or become compromised. The architecture should constrain the *consequences* of that event rather than assume the domain will remain trustworthy.

NIST also makes an important structural statement: **domain separation requires a domain to be contained within its own protected subsystem so that elements of the domain are directly accessible only through the procedures or functions of that protected subsystem**. Isolation is the enabling concept. Interfaces are the controlled exceptions.

This distinction is critical. Absolute isolation would eliminate interaction, but useful systems generally cannot operate that way. Domains must exchange information, invoke functions, share selected resources,

or coordinate behavior. The engineering challenge is to preserve separation while permitting *only* those interactions necessary for the required capability.

Accordingly, every cross-domain interface is a design decision that trades some degree of separation for functionality. External requests for resources or functions must be arbitrated at those interfaces. NIST cites mechanisms such as firewalls, data diodes, cross-domain solutions, and encryption, but the principle is intentionally *technology-neutral*. The mechanism is not the principle. The principle is the architectural decision to constrain influence between domains whose protection needs differ.

"The mechanism is not the principle. The principle is the architectural decision to constrain influence."

A domain may be defined by confidentiality needs, integrity needs, mission criticality, safety consequences, privilege, trustworthiness, exposure to adversity, operational role, life-cycle function, or combinations of these factors. Administrative and user domains are a familiar example, but the principle applies much more broadly: safety-critical versus non-safety-critical functions; production versus development; operational versus maintenance; trusted versus untrusted execution; mission data versus public data; control plane versus data plane; recovery infrastructure versus operational infrastructure; and high-assurance security mechanisms versus general-purpose computing.

The principle therefore answers three questions simultaneously: (1) what must be separated, (2) why must it be separated, and (3) through what controlled interactions may the separated domains still cooperate?

3. What the Principle Does Not Mean

Domain separation is often reduced to familiar cybersecurity mechanisms. That interpretation is too narrow.

- **Domain separation does not mean simply creating segmented networks**. Network segmentation can contribute to domain separation, but a segmented network does not by itself establish a trustworthy security boundary. If the separated segments depend on the same privileged management plane, identity service, hypervisor, orchestration system, update infrastructure, or administrator, compromise may cross the supposed boundary without traversing the network path the segmented network was intended to constrain.
- **Domain separation does not mean merely drawing boxes on an architecture diagram.** A boundary has engineering significance only if the realized system enforces the constraints represented by the line. Diagrams can describe intended separation while implementation choices quietly reintroduce shared dependencies, bypass paths, privileged interfaces, or maintenance channels.
- **Domain separation does not mean air-gapping everything.** Physical separation may provide strong isolation, but mission capability often requires controlled interaction. Domain separation is about bounding and mediating necessary interactions, not eliminating every interaction.
- **Domain separation does not mean confidentiality alone.** Separation can protect confidentiality, but it also constrains integrity effects, privilege propagation, availability dependencies, resource interference, control influence, and failure propagation.
- **Domain separation does not mean that everything inside a domain is equally trustworthy.** A domain is a protection construct, not a declaration that every element within it can be trusted. Internal decomposition may still be necessary, and nested domains may have distinct protection needs.

- **Domain separation does not mean that a boundary is trustworthy because a commercial security product is present.** A firewall, gateway, hypervisor, container boundary, encryption mechanism, or cross-domain solution contributes only to the extent that its design, implementation, configuration, operation, and assurance are commensurate with the consequences of boundary failure.
- **Domain separation does not mean that the architecture can ignore life cycle states.** Development, integration, maintenance, troubleshooting, recovery, and disposal frequently introduce interfaces and privileges that do not exist during normal operation. A system that is well separated in its operational state but collapses its domains during maintenance does not possess continuous domain separation.

The principle is therefore best understood as a *system property*: influence across a defined boundary remains within authorized and intended limits throughout the relevant system life cycle states and modes.

4. Loss-Driven Interpretation

The most useful way to decide where domain boundaries belong is to begin with loss rather than technology. NIST SP 800-160 treats assets broadly: system capabilities, people, material resources and infrastructure, intellectual property, data and information, and derivative assets such as reputation and trust. *Protection needs* arise from the relationship among the asset, the type and context of loss, and the consequences of that loss. The architecture should, therefore, not begin by asking, “Where should we put the firewall?” It should begin by asking, “What must *not* be allowed to happen?”

A loss-driven chain for domain separation can be expressed as:

> **Asset → Unacceptable Loss → Hazardous or Loss-Enabling Condition → Security Constraint → Domain Boundary → Mediated Interaction → Evidence.**

Consider an advanced AI model whose weights represent strategically valuable intellectual property. An unacceptable loss might be unauthorized disclosure or extraction of those weights. A loss-enabling condition might be the ability of an Internet-facing service, compromised administrator workstation, or general-purpose orchestration plane to read model memory or storage directly. The resulting security constraint could state that externally exposed or lower-trust domains shall not directly access plaintext model weights or the mechanisms capable of exporting them. That constraint drives architecture: model execution, key management, administrative control, and external request processing may need to occupy distinct domains with narrowly defined interfaces.

The same reasoning applies to *integrity*. Suppose an industrial control system has a safety-critical control function. The unacceptable loss may be physical damage or injury caused by unauthorized actuator commands. A hazardous condition may be that a compromised business network can issue or influence control commands. The security constraint is not merely “install a firewall.” It is that no element in the business domain shall be able to cause an unauthorized control action in the safety-critical domain. The architecture then determines what separation and mediation are required to enforce that constraint.

This framing changes domain separation from a generic best practice into a *traceable engineering response* to loss.

It also exposes where boundaries should be strongest. Domains should not be created merely because components are organizationally different or technologically convenient. They should be created where

differences in protection needs, consequences, trustworthiness, privilege, exposure, or mission role make unconstrained influence unacceptable. The key question is not whether two elements are different. It is whether allowing one to influence the other changes the path to unacceptable loss.

"The key question is not whether two elements are different. It is whether allowing one to influence the other changes the path to unacceptable loss."

---

5. Engineering the Principle into the Architecture

Architects realize domain separation through a sequence of decisions rather than through a single mechanism. The following actions guide and inform the engineering process:

- **Identify candidate domains from protection needs.** Grouping should consider assets, critical functions, trust assumptions, privilege, exposure, assurance requirements, and consequences of failure or compromise. A useful domain boundary usually corresponds to a meaningful change in one or more of those characteristics.
- **Define the allowed influence between domains.** This requires more precision than listing network ports. Architects should identify information flows, control flows, privilege flows, administrative paths, resource dependencies, timing dependencies, update paths, diagnostic paths, recovery paths, and physical interactions. Each cross-domain interaction should have an explicit purpose and authorization basis.
- **Minimize the interfaces.** Every interface weakens isolation to some degree. Interfaces should therefore be treated as security-relevant architectural commitments. Unnecessary paths should be eliminated; necessary paths should be narrow, explicit, and analyzable.
- **Select mediation mechanisms whose trustworthiness is commensurate with the consequences of boundary failure.** Depending on the system, these may include *reference-monitor-like* mechanisms, gateways, one-way transfer devices, cryptographic separation, memory protection, hardware-enforced execution domains, virtualization boundaries, process isolation, capability systems, physically separate equipment, guarded administrative interfaces, or procedural and physical controls.
- **Separate the management of the boundary from the domains it constrains when necessary.** A common architectural mistake is to create strong operational separation while allowing a shared administrative plane to bypass it. If the same administrator credential, orchestration service, firmware update path, hypervisor, or identity provider can unilaterally control both domains, the true boundary may be located above the apparent one.
- **Account for shared resources.** CPU caches, memory controllers, storage, time sources, networks, power, cooling, logging, identity, key management, and monitoring can create coupling. Shared resources are not automatically prohibited, but their influence must be understood and bounded. If compromise or exhaustion of a shared service can defeat the security objective of both domains, the separation claim must acknowledge that dependency.
- **Design for states and modes.** Startup, shutdown, failover, degraded operation, emergency access, maintenance, patching, backup, restoration, and disposal may alter the boundary. Separation requirements must specify which interactions remain permitted in each state and who or what can authorize transitions.

- **Consider whether selected cross-domain relationships should be persistent at all.** NIST SP 800-160, Volume 1, Revision 1 introduces the design principle of *Least Persistence*. This principle can reduce exposure in the temporal dimension. Services, information, privileges, and connectivity can be created when needed and terminated when their mission purpose has been satisfied. Applied to domain separation, this means that architects should constrain not only what may cross a boundary and where the crossing occurs, but also when the crossing exists and for how long. A continuously available cross-domain path presents a continuously available propagation opportunity; an on-demand path can reduce the time during which the lower-trust domain can influence the higher-protection domain.
- **Determine whether the architecture needs the ability to strengthen separation dynamically.** A dynamic approach for isolating or containing selected applications, components, or resources as risk conditions change may be needed by the architecture. Predefined boundaries should establish the normal architecture of permitted influence, but the system may also need a more restrictive protective state in which connectivity is reduced, privileges are revoked, suspect components are quarantined, or resources are reconfigured around damage. This does not replace domain separation; it builds adaptive behavior on top of an already explicit domain model.
- **Make the boundary visible in architecture artifacts and traceable to requirements.** A domain boundary should have an identifiable purpose, protected assets, governing constraints, permitted flows, enforcement mechanisms, assumptions, dependencies, and verification methods. If those cannot be stated clearly, the boundary is probably not yet adequately engineered.

6. Engineering for Compromise

Engineering for compromise (EfC) changes the basic assumption behind domain separation [2]. Traditional defensive reasoning often asks whether the boundary can prevent an adversary from penetrating a domain. EfC asks an additional question: **what happens when an element on either side of the boundary is already hostile?** That assumption is powerful because it converts the domain boundary from a preventive convenience into a *containment* requirement.

Suppose an Internet-facing inference service is compromised. If the service resides in the same trust domain as model weights, key material, privileged management functions, and recovery infrastructure, the compromise may immediately place all of those assets within adversary reach. The architecture has made *penetration* and *unacceptable loss* nearly synonymous.

If, instead, the exposed service occupies a constrained domain and can interact with the model-execution domain only through a narrow mediated interface, the adversary inherits only the authority intentionally granted across that interface. The attacker may control the compromised service completely yet remain unable to read model weights, reconfigure the execution environment, access recovery credentials, or issue arbitrary privileged commands.

This leads to a useful EfC test for every domain boundary:

Assume the less trustworthy domain is fully compromised. What can it now cause the protected domain to do?

The answer should be expressed in terms of reachable assets, authorized operations, possible information flows, control effects, resource effects, and loss scenarios. If compromise of the lower-trust domain makes an unacceptable loss directly reachable, the separation is insufficient for that loss objective.

A second test reverses the direction:

Assume a protected domain fails or behaves maliciously. What can it cause outside itself?

This is the *containment* dimension emphasized by NIST. Some domains exist not only to protect valuable functions from external influence but also to contain potentially dangerous functions, untrusted content, experimental code, high-privilege services, or components whose failure could propagate.

EfC therefore turns domain separation into a *Compromise-Containment Architecture (CCA)*.[1] The boundary defines a maximum intended radius of influence. That radius is not necessarily zero. Mission capability may require selected interactions. But the architecture should make the residual influence deliberate rather than accidental.

A strong separation claim can therefore be phrased in adversarial terms: **compromise of any non-boundary element within Domain A shall not, by itself, enable prohibited behavior or unacceptable loss in Domain B.** That claim can then be decomposed into requirements and subjected to verification and assurance.

Domain separation also changes adversary economics. In a flat architecture, the cost of attack can decline dramatically after the first successful compromise because the adversary can exploit shared trust, credentials, services, and infrastructure to move toward higher-value assets. A domain-separated architecture forces the adversary to repeatedly confront independently enforced constraints. Initial access yields a foothold, not unrestricted reach. Each additional domain boundary creates another engineering obstacle between compromise and unacceptable loss. This is structural cost imposition: the architecture itself repeatedly raises the *work factor* required to progress toward mission objectives rather than relying solely on defenders to detect and stop that progression.

Domain separation provides one of the primary architectural foundations for a *Compromise Containment Boundary (CCB)*. A CCB is not necessarily synonymous with a network segment, enclave, virtual machine, device, or physical boundary. It is the system-level boundary across which the architecture intends to constrain the information, authority, control, functionality, and influence available to a compromised element. Domain separation helps establish that boundary; complementary principles determine how much capability remains available within it and how necessary interactions across it are controlled.

A useful technique for evaluating domain separation under assumed compromise is Compromise Propagation Analysis (CPA). Rather than asking whether an adversary might penetrate a domain, CPA begins with the stronger assumption that an element or domain is already hostile and asks what becomes reachable next. The analysis follows potential propagation through information, identity, privilege, control, shared services, administration, software and firmware distribution, physical processes, maintenance, recovery, and other dependencies. At each proposed domain boundary, engineers ask whether the boundary actually interrupts the causal path toward unacceptable loss. A meaningful domain boundary should change the adversary's reachability. If crossing the boundary requires no new authority, independently enforced mediation, or additional adversary success, the apparent separation may have little containment value.

“A meaningful domain boundary should change the adversary’s reachability.”

Domain separation can also contribute to compromise margin: the amount of adversary success, loss of trust, or component compromise that can occur before an unacceptable-loss state becomes reachable. A system in which compromise of any externally exposed element immediately places critical assets within

[1] The concepts of *Compromise-Containment Architecture*, *Compromise Containment Boundary*, *Compromise Propagation Analysis*, and *compromise margin* are introduced here as engineering constructs for applying the principle of *Domain Separation* under assumed compromise. The concepts are described in greater detail in [2].

reach has little compromise margin. Separating those assets behind independently enforced boundaries can increase that margin by requiring additional, distinct adversary successes before unacceptable loss becomes possible. This is not a universal numerical metric; it is an analytical means of making explicit what degree and combination of compromise the architecture is intended to tolerate.

7. Interaction with Other Principles

The principle of *Domain Separation* cannot stand alone. NIST's design principles are most powerful when composed, and separation depends on several of them to become operationally meaningful.

- *Mediated Access* governs the interfaces created by separation. Once domains are separated, necessary interactions must be arbitrated according to policy. Separation without mediation either prevents useful interaction or creates uncontrolled bypasses.
- *Least Sharing* reduces the number and scope of resources shared across domains. Shared services, memory, storage, identity systems, management planes, and infrastructure create coupling. Reducing unnecessary sharing strengthens the boundary and simplifies the assurance argument.
- *Least Privilege* limits what subjects crossing or operating near a boundary are authorized to do. Domain separation provides coarse-grained confinement between protection domains; least privilege provides fine-grained confinement of authority within those domains. If a cross-domain service has broad privilege in both domains, separation may exist structurally while excessive authority defeats its purpose.
- *Least Functionality* removes unnecessary functions that could become cross-domain paths. Boundary components should do as little as practicable because every additional function expands the attack surface and the behaviors that must be understood and assured.
- *Minimal Trusted Elements* limits the number of elements whose correct behavior is essential to maintaining separation. If dozens of general-purpose components must all remain trustworthy for a boundary to hold, the assurance burden and attack surface become difficult to manage.
- *Reduced Complexity* strengthens and facilitates analyzability. Simple interfaces, explicit flows, and small enforcement mechanisms make it easier to reason about whether influence is actually constrained.
- *Protective Default* ensures that uncertainty or absence of explicit authorization does not silently open the boundary.
- *Protective Failure* addresses what happens when the boundary mechanism or adjacent elements fail.
- *Protective Recovery* addresses restoration of trustworthy separation after compromise or failure.
- *Anomaly Detection* becomes more effective when domain separation makes intended behavior explicit. In a flat architecture, malicious lateral movement may be difficult to distinguish from legitimate internal communication. Domain separation constrains cross-domain interaction to defined interfaces and authorized behaviors. This creates an architectural definition of normality: an attempted interaction that does not conform to the permitted interface, direction, operation, state, or authorization can be treated as a salient anomaly. Domain separation therefore does more than contain compromise; it can improve the signal-to-noise ratio available to detection and response mechanisms.
- *Continuous Protection* extends the requirement across time. A boundary that exists only during normal operation but disappears during maintenance, update, or recovery is not continuously protective.
- *Compositional Trustworthiness* is essential because domain separation is an emergent property. The boundary may depend on hardware isolation, software mediation, cryptography, configuration,

administration, and physical controls. The assurance argument must show that those elements compose to preserve the intended constraint.

- *Substantiated Trustworthiness* requires evidence rather than assumption. The presence of a segmentation mechanism does not prove separation. The system must provide credible evidence that the boundary behaves as claimed under the relevant conditions of adversity.
- *Loss Margins* can be used to reason about how much degradation, bypass, or adversary progress can occur before the separation objective fails and unacceptable loss becomes reachable.

The broader lesson is that domain separation establishes the architectural boundary, but complementary principles determine how strong, narrow, enforceable, recoverable, and trustworthy that boundary becomes.

8. System Life Cycle Application

Domain separation must be preserved throughout the system life cycle, not added during deployment. NIST SP 800-160 identifies fourteen system life cycle processes, and the principle has implications across all of them.

- *Business or Mission Analysis* identifies mission dependencies, high-consequence assets, unacceptable losses, and operational contexts that may require distinct protection domains.
- *Stakeholder Needs and Requirements Definition* captures protection needs and constraints in stakeholder language. Stakeholders may require, for example, that compromise of an externally accessible service not expose mission data or that maintenance personnel not gain access to protected operational information.
- *System Requirements Definition* converts those needs into verifiable requirements. Requirements should specify prohibited influence, authorized cross-domain interactions, performance expectations, states and modes, and assurance needs.
- *System Architecture Definition* is where domain separation becomes structural. Architects allocate functions and assets to domains, define boundaries, identify interfaces and dependencies, and determine where mediation and isolation must occur.
- *Design Definition* refines the architecture into mechanisms, protocols, configurations, physical arrangements, privilege models, key structures, management paths, and failure behavior.
- *System Analysis* evaluates alternatives and tradeoffs. Analysts examine attack paths, failure propagation, shared-resource coupling, performance effects, trust dependencies, and whether the proposed separation actually controls the relevant loss scenarios.
- *Implementation* realizes the mechanisms. Coding, hardware realization, configuration, cryptographic implementation, physical construction, and procedural controls must preserve the design assumptions on which separation depends.
- *Integration* is especially difficult. Individually separated components can become coupled when assembled. New interfaces, shared services, debugging hooks, test harnesses, temporary credentials, or orchestration dependencies can defeat previously valid assumptions.
- *Verification* determines whether the realized system satisfies the separation requirements. This may include inspection, analysis, testing, penetration testing, fault injection, interface testing, configuration review, information-flow analysis, and, for high-assurance cases, formal methods.
- *Transition* must ensure that deployment, initialization, data migration, credential provisioning, and acceptance activities do not introduce bypasses or residual connections.

- *Validation* asks whether the realized separation actually satisfies stakeholder protection needs in the operational context. A technically correct boundary that prevents the mission from functioning or overlooks a critical operational pathway is not a successful solution.
- *Operation* maintains the boundary under real workloads, changing threats, configuration drift, personnel changes, and evolving mission use. Monitoring should detect attempts to cross boundaries and changes that invalidate separation assumptions.
- *Maintenance* is one of the most common points of erosion. Diagnostic ports, vendor access, emergency accounts, temporary network connections, component replacement, and patching can create privileged cross-domain paths. Maintenance procedures must preserve or deliberately control the boundary.
- *Disposal* must maintain separation until protected assets and residual data are securely removed. Decommissioning one domain should not expose assets, keys, logs, configurations, or media belonging to another.

The governing rule is *continuity of intent*: the reason the boundary was created must remain traceable through requirements, architecture, implementation, operation, change, and retirement.

"The governing rule is continuity of intent."

---

9. Failure Patterns

Systems frequently appear to implement domain separation while preserving hidden paths that defeat it. Several *failure patterns* recur.

The diagram-only boundary. Architecture drawings show separate zones, but implementation provides unrestricted administrative, storage, orchestration, or service-to-service access across them.

The shared-root-of-trust problem. Two domains use separate applications and networks but depend on the same highly privileged hypervisor, management controller, identity administrator, or cloud control plane. Compromise of the shared root collapses both domains.

The firewall-equals-separation assumption. A firewall constrains selected network traffic but does not address shared credentials, management interfaces, application-layer authority, data flows through permitted services, or non-network channels.

The bidirectional convenience interface. A boundary intended to permit narrow one-way reporting evolves into a general bidirectional API for operational convenience. Each added command increases the influence one domain can exert over the other.

The maintenance bypass. Normal operations are strongly separated but troubleshooting permits direct administrator access from a general-purpose workstation into both domains. The exceptional path becomes the most attractive attack path.

The common identity plane. Domains have separate computing and networks but share an identity infrastructure whose compromise grants privileges in both. The identity system is therefore part of the separation architecture whether or not the diagram shows it.

The logging paradox. Security logs from a high-protection domain are exported to a lower-trust monitoring domain without considering whether logs contain sensitive information or whether the logging channel can be used to influence the protected domain.

The recovery collapse. Backups, recovery consoles, golden images, key escrow, or disaster-recovery systems span multiple domains with greater privilege than normal operations. Compromise of recovery infrastructure bypasses the operational boundary.

The performance exception. Security checks are bypassed, cached, or weakened because mediation creates latency. The system still appears segmented, but the actual enforcement path no longer matches the security model.

The static-boundary assumption. The system evolves, new services are added, cloud resources move, software is updated, and dependencies change, yet the original domain model is never reassessed.

The trust-by-location fallacy. Components are assumed trustworthy because they are "inside" the protected domain. Internal compromise then becomes unconstrained because no further decomposition, least privilege, or mediation exists.

The internal-attack-surface blind spot. NIST SP 800-160 emphasizes that once penetration is assumed, attack surface is not limited to externally exposed interfaces. It includes internal vulnerabilities reachable through lateral movement and can extend into development, operational, maintenance, and supply-chain environments. A domain-separation analysis that examines only Internet-facing exposure can therefore miss the interfaces and dependencies most likely to matter after the first compromise. The relevant question is which internal resources become reachable from each assumed foothold and whether a domain boundary materially changes that reachability.

These failure patterns share a common feature: the visible boundary is not the true boundary of influence. Engineering analysis must therefore follow authority, dependency, and effect — not merely topology.

"Engineering analysis must follow authority, dependency, and effect—not merely topology."

---

10. Evidence and Assurance

A claim of domain separation is meaningful only if supported by evidence commensurate with the consequences of its failure. NIST SP 800-160 emphasizes that *trustworthiness* is demonstrated through evidence and that assurance is the grounds for justified confidence that claims have been or will be achieved. It also cautions that a meaningful trustworthiness claim cannot rest on an isolated demonstration that a protection mechanism exists. Evidence must support the conclusion that the system was properly specified, designed, implemented, and integrated with sufficient rigor to deliver the claimed system-level behavior.

For domain separation, the top-level claim might be:

***Domain A* and *Domain B* are sufficiently separated such that interactions between them are limited to explicitly authorized and intended flows, and compromise or failure within either domain cannot by itself produce specified unacceptable losses in the other.**

That claim can be decomposed into *subclaims* concerning architecture, interface completeness, mediation, privilege, isolation strength, shared dependencies, configuration, system life cycle states, failure behavior, recovery, and operational monitoring. Evidence may include requirements traceability showing why the boundary exists; architecture models showing all identified cross-domain flows; data-flow and control-flow analyses; interface specifications; privilege and authorization matrices; configuration baselines; hardware isolation documentation; cryptographic design evidence; source-code or implementation review; formal

information-flow models; penetration testing; adversarial testing; fault injection; red-team exercises; inspection of physical separation; tests of maintenance and recovery states; and operational telemetry demonstrating that prohibited flows are rejected or detected.

For higher-consequence systems, the *absence* of anomalous behavior becomes important: evidence is needed not only that authorized flows work but that prohibited influence does not occur within the scope of the claim. Testing alone cannot demonstrate the absence of all errors or vulnerabilities, so assurance may require structured reasoning that combines analysis, testing, inspection, formal methods, and explicit assumptions.

An *assurance case* is particularly useful. It makes the separation claim explicit, records the arguments connecting evidence to that claim, identifies assumptions and dependencies, and exposes where confidence rests on elements outside the immediate boundary. The assurance case must also evolve. A new interface, software update, cloud migration, administrator role, maintenance procedure, shared service, or threat capability may invalidate evidence that was previously sufficient. Domain separation is therefore not a one-time certification result. It is a maintained claim about an evolving system.

Where dynamic segmentation, isolation, or non-persistent cross-domain relationships are part of the design, the assurance case must address the transitions as well as the steady states. Evidence should show that the system can enter a more restrictive isolation state without opening alternate paths, that revoked connections and privileges are actually removed, that quarantined resources cannot continue exerting prohibited influence, and that later restoration does not silently reintroduce a compromised state. System resiliency therefore expands the separation claim from “the boundary holds” to “the boundary remains controlled as the system changes state under adversity.”

Domain separation should also reduce and clarify the system's *trusted footprint*: the set of elements whose malicious failure could invalidate the separation claim or directly enable unacceptable loss. An assurance analysis should therefore ask not simply whether the boundary mechanisms are trustworthy, but how many elements must remain trustworthy for the boundary claim to survive. A separation architecture that depends on numerous general-purpose components, shared administrators, common management services, or opaque infrastructure may possess a much larger trusted footprint than its architecture diagram suggests. High-assurance effort should be concentrated on the *smallest practicable set of elements whose correct behavior is genuinely essential to the separation claim*.

Evidence for domain separation should also address multiple, correlated, and common-mode compromise. Two apparently independent domains may not provide meaningful separation if both depend on the same identity provider, administrator, management plane, hypervisor, build pipeline, recovery authority, or other common mechanism. Independence is therefore not established by visual or organizational separation; it must be demonstrated with respect to the compromise paths relevant to the loss-control claim. The assurance case should identify such common dependencies explicitly and state which combinations of compromise exceed the intended compromise margin.

“Independence is not established by visual or organizational separation; it must be demonstrated against relevant compromise paths.”

---

11. Key Engineering Takeaways

The following key engineering takeaways are important when applying the security design principle of *Domain Separation*.

1. Domain separation is fundamentally about controlling influence, not merely separating networks or components.
2. Domain boundaries should be derived from differences in protection needs, consequences of loss, trustworthiness, privilege, exposure, and mission role.
3. A line on an architecture diagram is not a security boundary unless the realized system enforces the constraints represented by that line.
4. Every cross-domain interface is a deliberate reduction in isolation and should therefore be minimized, explicitly authorized, mediated, and analyzed in both spatial and temporal terms; selected services, privileges, or connections may be safer when established only on demand.
5. Shared management planes, identity systems, hypervisors, recovery systems, and infrastructure can collapse apparent separation and must be included in the trust analysis.
6. Engineering for compromise provides a powerful test: assume one domain is fully hostile and determine what loss remains reachable across the boundary. NIST SP 800-160, Volume 2, Revision 1 extends that logic by treating persistent adversary presence as a design condition and using segmentation and isolation to contain and delay further progression [3].
7. The principle of *Domain Separation* must be composed with the principles of *Mediated Access*, *Least Privilege*, *Least Sharing*, *Least Functionality*, *Minimal Trusted Elements*, *Protective Failure*, *Protective Recovery*, *Compositional Trustworthiness* and *Substantiated Trustworthiness*.
8. Separation must persist across operational, maintenance, update, recovery, transition, and disposal states. A resilient design may also strengthen separation dynamically by revoking privileges, terminating connectivity, quarantining resources, or entering more restrictive protective states [3].
9. Assurance requires evidence that the system is specified, designed, implemented, integrated, operated, and maintained in a manner that preserves the separation claim, including transitions into and out of dynamically isolated or degraded states. The presence of a firewall, gateway, hypervisor, or encryption mechanism is not sufficient evidence by itself.
10. The strongest architectural question is not “Can the adversary get in?” It is “If a system element becomes compromised, where does its influence stop?”

The principle of *Domain Separation* provides one of the most important answers.

12. Conclusion

Domain Separation is a deceptively simple principle with profound architectural consequences. NIST SP 800-160 defines it in one sentence, but realizing it requires systems engineers to reason about assets, unacceptable loss, trust, influence, interfaces, shared dependencies, states, failure, compromise, recovery, and evidence across the entire system life cycle.

The principle is most valuable when it is treated neither as a product feature nor as a networking pattern. It is an engineering decision about where the system must preserve differences in protection need and where adversity must be prevented from propagating freely. In that sense, domain separation is one of the foundational mechanisms by which an architecture creates limits. It establishes places where authority

changes, where information flow changes, where trust assumptions change, and where the consequences of compromise are intended to change.

Engineering for compromise makes the principle indispensable. If compromise is treated as a credible system state rather than an exceptional event, the architecture must contain it. Domain separation defines the structural boundaries within which that containment can occur.

Operational defenses attempt to detect and stop the adversary. Structural defenses determine what the adversary can reach even when operational defenses fail. Domain separation belongs fundamentally to the latter category. It does not have to prevent the first compromise to be successful. Its engineering purpose is to prevent that compromise from becoming unacceptable system-level loss.

A trustworthy boundary is therefore not one that merely looks separate during normal operation. It is one for which there is justified confidence that, under the adversity the system is expected to face, influence remains bounded to the degree necessary to prevent unacceptable loss.

That is the engineering purpose of domain separation: not simply to divide the system, but to determine where compromise stops.

> ***"The engineering purpose of domain separation is not simply to divide the system, but to determine where compromise stops."***

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*, August 2026. https://lnkd.in/eVSMZdCs

[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

