Internet-Draft
Intended status: Informational
Expires: December 2026

M. Norton
Independent

June 2026

SDLP Security Architecture (SDLP RFC 4)
draft-norton-sdlp-sec-arch-01.txt

Abstract

This document defines the security architecture for the Secured Digital Lifecycle Protocol (SDLP). It specifies the security model, threat surfaces, authentication requirements, authorization boundaries, and integrity guarantees that govern all SDLP lifecycle transitions and transformations.

1. Introduction

This document defines the security architecture for the Secured Digital Lifecycle Protocol (SDLP). SDLP requires authenticated, authorized, and integrity-preserving lifecycle transitions for all digital objects. This document establishes the security model and normative requirements for those transitions.

2. Security Model

SDLP security is based on authenticated identity, deterministic lifecycle state transitions, and verifiable lineage. All actors, transformations, and lifecycle events must be cryptographically attributable.

3. Threat Surfaces

SDLP identifies the following primary threat surfaces:

- Identity spoofing
- Unauthorized transformations
- Lineage tampering
- State machine bypass
- Unauthorized distribution
- Unauthorized retention or resurrection

4. Authentication Requirements

All lifecycle transitions MUST be authenticated using verifiable credentials bound to the actor performing the transition.

5. Authorization Boundaries

SDLP defines strict authorization boundaries for:

- Creation
- Activation
- Distribution
- Transformation
- Verification
- Retention
- Retirement

6. Integrity Guarantees

SDLP requires that all transformations preserve lineage, identity, and state integrity. Unauthorized or unverifiable transitions MUST be rejected.

7. IANA Considerations

This document makes no requests of IANA.

8. Security Considerations

This document defines the security architecture for SDLP and therefore consists entirely of security considerations.

Author’s Address

M. Norton
Independent
El Mirage
United States

Email: mark433norton@gmail.com

Table of Contents

39. Decay Mechanics and Use-Based Lifecycle Physics 3
40. Tamper Physics 5
41. State Transition Physics 7
42. Environment Physics 9
43. ObjectKey Physics 11
44. Lineage Physics 13
45. Policy Physics 15
46. Environment Validation Physics 17
47. Policy Enforcement Physics 19
48. Transfer Physics 21
49. Timestamp Physics 23
50. Serialization Physics 25
51. Anti-Resurrection Physics 27
52. Destruction Physics 29

53. Rehydration Prohibition Physics 31
54. Reproduction Physics 33
Author’s Address 35

39. Decay Mechanics and Use-Based Lifecycle Physics

The SDLP decay model defines how SDLP-governed objects lose usable lifecycle energy (P) through normal use, duplication attempts, and environment interactions. Decay is not a punishment or enforcement mechanism. Decay is predictive lifecycle physics: a deterministic response to actions that consume or divide the object's remaining usable energy.

SDLP defines three classes of decay events:

* Play Events: Consumption of the object through normal use.
* Copy Events: Duplication attempts that divide remaining energy.
* Upload Events: Environment transitions treated as copy events.

All decay events reduce the object's remaining lifecycle energy (P). When P reaches zero, the object MUST transition to the Decayed state.

Play Decay:

* Each play event reduces P by 1.
* P = P - 1
* Play events MUST be recorded as lineage entries.
* Play events MUST NOT be reversible or suppressible.
* Play events MUST NOT reset or increase P.

Rewind Threshold Decay:

* A rewind event occurs when the user attempts to move backward within a digital object.
* If the rewind distance is greater than or equal to one-third (1/3) of the object's total length or duration, the object MUST trigger a Restart Event.
* Restart Events MUST deduct one play from P.
* Restart Events MUST reset the object's position to the beginning.
* Restart Events MUST be recorded as lineage entries.
* Restart Events MUST NOT be suppressible or reversible.
* Rewind distances less than one-third of total length MUST NOT trigger a Restart Event.

Copy Decay:

* Copying is a decay event that divides the object's remaining lifecycle energy.
* After a copy event, both the parent and the child retain P/2.
* P = P / 2
* Copy events MUST be recorded as lineage entries.
* Copy events MUST NOT increase total system energy.
* Copy events MUST NOT create parallel or forked lineage chains.

Upload Decay:

* Uploading is treated as a copy event.
* Uploading consumes half of the object’s remaining lifecycle energy.
* P = P / 2
* Upload events MUST be recorded as lineage entries.
* Upload events MUST NOT bypass decay physics.

Float Normalization:

* If any decay event produces a non-integer value of P, the value MUST be rounded to the nearest integer using deterministic, platform-independent rounding rules.
* Rounding MUST be canonical and reproducible across all compliant implementations.
* Fractional representations of P are prohibited.

End-of-Life Thresholds:

* When P reaches zero, the object MUST transition to the Decayed state.
* When P is less than or equal to 1, the object MUST initiate mandatory end-of-life destruction and transition to the Bitdumped state.
* End-of-life destruction MUST occur at a rate of 100% per second and MUST be irreversible.
* Hospice states (P = 1) MUST NOT allow further lifecycle operations.

Decay Thresholds:

* When P reaches a policy-defined threshold, the object may restrict certain lifecycle operations.
* Decay thresholds MUST be deterministic and reproducible.

SDLP objects MUST validate decay physics at the following times:

* Before performing a play event
* Before performing a copy event
* Before performing an upload event
* Before executing any lifecycle operation that depends on P
* Before transitioning to the Decayed or Bitdumped state

SDLP objects MUST reject:

* Attempts to reset P
* Attempts to increase P
* Attempts to bypass decay physics
* Attempts to duplicate without decay
* Attempts to create multiple children from a single decay event
* Attempts to falsify decay lineage entries
* Attempts to represent P as a non-canonical or fractional value

Any attempt to falsify decay physics MUST be treated as a tamper event and MUST trigger mandatory bitdump.

Decay physics MUST be deterministic, canonical, and reproducible across all compliant implementations. Platform-dependent behavior, non-deterministic decay calculations, or alternate decay models are prohibited.

The decay mechanics and use-based lifecycle physics ensure that SDLP-governed objects behave like physical products across a digital medium. By modeling play, rewind, copy, and upload as decay events, SDLP establishes a predictable, tamper-evident, and entropy-consistent lifecycle model.

40. Tamper-Response, Bitdump, and Irreversible Destruction Physics

The SDLP tamper-response model defines the irreversible destruction physics that govern how SDLP-governed objects react to malicious, deceptive, or unauthorized conditions. Tamper-response is not a policy, permission, or enforcement mechanism. Tamper-response is a terminal physics event: a deterministic collapse triggered when the object detects conditions incompatible with secure digital existence.

SDLP defines three classes of tamper conditions:

* Forgery: Invalid signatures, spoofed identity, falsified lineage, or unauthorized modification of canonical fields.

* Manipulation: Debugging, patching, memory injection, binary modification, or attempts to alter execution flow.

* Deception: Environment spoofing, virtualization masking, trust signal falsification, or attempts to bypass validation routines.

When a tamper condition is detected, the object MUST perform one of two responses:

* Refusal: For accidental or non-malicious failures, the object MUST refuse execution and record a lineage entry documenting the failure.

* Bitdump: For malicious, intentional, or deceptive conditions, the object MUST irreversibly destroy itself and transition to the Bitdumped state.

Bitdump is an irreversible terminal state. Bitdump MUST:

* Zeroize all cryptographic material.
* Invalidate the ObjectKey.
* Invalidate all internal state.
* Render the object permanently non-functional.
* Record a final lineage entry if possible.
* Prevent all future execution, transfer, or consumption.

SDLP objects MUST trigger immediate bitdump under the following conditions:

* Debugger attachment
* Binary modification or patching
* Memory injection or code manipulation
* Environment spoofing or falsified trust signals
* Unauthorized virtualization masking
* Attempts to bypass environment validation
* Attempts to disable tamper-response
* Attempts to modify identity fields
* Attempts to falsify lineage entries
* Attempts to reset or increase P
* Attempts to duplicate without decay
* Attempts to create parallel lineage chains

Bitdump physics:

* Bitdump is atomic and cannot be interrupted.
* Bitdump is irreversible and cannot be undone.
* Bitdump MUST NOT leave recoverable state.
* Bitdump MUST NOT allow rollback or resurrection.
* Bitdump MUST NOT allow partial destruction.
* Bitdump MUST NOT allow post-collapse execution.

SDLP objects MUST validate tamper conditions at the following times:

* At startup
* Before performing any lifecycle operation
* Before executing any state transition
* Before accepting a lineage entry
* Before applying a policy
* Continuously during execution

SDLP objects MUST reject:

* Any attempt to override tamper-response
* Any attempt to redefine tamper conditions
* Any attempt to disable bitdump
* Any attempt to recover from bitdump
* Any attempt to reinitialize a bitdumped object

The Bitdumped state is final. SDLP objects in the Bitdumped state:

* Must not execute
* Must not transfer
* Must not consume
* Must not accept lineage entries
* Must not apply policies
* Must not rebind identity
* Must not reinstantiate

The tamper-response and irreversible destruction physics ensure that SDLP-governed objects cannot be manipulated, forged, or deceived. Bitdump establishes the terminal boundary of SDLP lifecycle physics, providing a deterministic and tamper-evident end-of-life condition for all digital objects.

41. Canonical Encoding Rules and Deterministic Serialization

The SDLP canonical encoding model defines the deterministic, unambiguous, and reproducible serialization rules required for all SDLP-governed objects, lineage entries, policies, and identity structures. Canonical encoding ensures that all compliant implementations produce identical byte-level representations for the same logical data. Any deviation from canonical encoding MUST be treated as a tamper condition.

Canonical encoding requirements:

* Encoding MUST be deterministic.
* Encoding MUST be platform-independent.
* Encoding MUST be architecture-independent.
* Encoding MUST be byte-for-byte reproducible.
* Encoding MUST NOT depend on locale, language, or system settings.
* Encoding MUST NOT allow alternate representations.

SDLP defines the following canonical encoding rules:

* Field Order: All fields MUST appear in a strict, predefined order. Reordering, omission, or insertion of fields is prohibited.

* Field Presence: All required fields MUST be present. Optional fields MUST be encoded explicitly as null when absent.

* Field Types: Each field MUST be encoded using its canonical type. Type coercion, implicit conversion, or alternate type encoding is prohibited.

* String Encoding: All strings MUST be encoded as UTF-8 without BOM. Alternate encodings are prohibited.

* Integer Encoding: All integers MUST be encoded as unsigned, big-endian, fixed-width values. Variable-length integers are prohibited.

* Boolean Encoding: Booleans MUST be encoded as a single byte: 0x00 for false, 0x01 for true.

* Hash Encoding: Hashes MUST be encoded as raw bytes, not hex or base64.

* Signature Encoding: Signatures MUST be encoded as raw bytes in canonical order defined by the signature algorithm.

* Timestamp Encoding: Timestamps MUST be encoded as UNIX epoch seconds in big-endian format.

* Null Encoding: Null MUST be encoded as a single canonical byte value (0xFF). Alternate null representations are prohibited.

Canonical serialization rules:

* Serialization MUST produce a single, contiguous byte sequence.
* Serialization MUST NOT include padding, alignment, or metadata.
* Serialization MUST NOT include comments or annotations.
* Serialization MUST NOT include platform-specific markers.
* Serialization MUST NOT include variable-length prefixes unless explicitly defined.

SDLP objects MUST validate canonical encoding at the following times:

* Before computing a hash
* Before verifying a signature
* Before accepting a lineage entry
* Before applying a policy
* Before performing any lifecycle operation
* Before executing any state transition

SDLP objects MUST reject:

* Non-canonical encodings
* Alternate field orders
* Missing or extra fields
* Variable-length integer encodings
* Non-UTF-8 string encodings
* Hex-encoded or base64-encoded hashes
* Non-canonical null representations
* Platform-dependent serialization formats

Any attempt to alter canonical encoding MUST be treated as a tamper event and MUST trigger mandatory bitdump.

Canonical encoding ensures that SDLP-governed objects maintain deterministic identity, reproducible lineage, and verifiable physics across all compliant implementations. By eliminating ambiguity and enforcing strict serialization rules, SDLP establishes a stable and tamper-evident foundation for digital lifecycle governance.

42. Cryptographic Hashing, Signature Algorithms, and Key Requirements

The SDLP cryptographic model defines the hashing algorithms, signature algorithms, and key requirements necessary to ensure deterministic identity, verifiable lineage, tamper-evident state transitions, and irreversible destruction physics. All cryptographic operations MUST be deterministic, canonical, and reproducible across all compliant implementations.

SDLP objects MUST use only approved cryptographic primitives. Use of non-approved algorithms, weakened variants, platform-dependent cryptography, or implementation-specific shortcuts is prohibited.

Hashing Requirements:

* SDLP objects MUST use a cryptographic hash function with at least 256-bit output.
* Hashes MUST be computed over canonical byte-encoded data.
* Hashes MUST be encoded as raw bytes, not hex or base64.
* Hashes MUST be deterministic and reproducible.
* Hashes MUST be included in all signature payloads.
* Hashes MUST be validated before accepting any lineage entry.

Signature Requirements:

* SDLP objects MUST use asymmetric signature algorithms with at least 128-bit security strength.
* Signatures MUST be computed over canonical byte-encoded payloads.
* Signatures MUST be encoded as raw bytes in canonical order.
* Signatures MUST be validated before performing any lifecycle operation.
* Signatures MUST be included in lineage entries, identity fields, and policy structures.
* Signature verification failure MUST be treated as a tamper condition.

Key Requirements:

* ObjectKey: A unique keypair generated at instantiation. Used to sign lineage entries and validate internal state. Must never be replaced, regenerated, or exported.
* DistributorKey: Used to sign instantiation events and product metadata.
* CustomerKey: Used to sign ownership-binding and transfer events.
* PolicyAuthorityKey: Used to sign policy versions.
* EnvironmentKey: Optional hardware-backed key used for trusted execution signals.

Key Handling Rules:

* Keys MUST never be exported in plaintext.
* Keys MUST never be stored in non-secure memory.
* Keys MUST never be regenerated after instantiation.
* Keys MUST never be shared across objects.
* Keys MUST never be derived from platform identifiers.
* Keys MUST be destroyed during bitdump.

Signature Payload Structure:

* All signatures MUST include:
  - Identity fields
  - PreviousHash
  - NewState
  - EventType
  - Timestamp
  - Canonical encoding of all fields
* Omission of any field invalidates the signature.
* Additional fields MUST NOT be added to the payload.

Cryptographic Validation Requirements:

* Before accepting a lineage entry
* Before applying a policy
* Before performing any lifecycle operation
* Before executing any state transition
* Before binding identity
* Continuously during environment validation

SDLP objects MUST reject:

* Weak or deprecated algorithms
* Non-canonical signature formats
* Hashes encoded as hex or base64
* Keys generated outside SDLP-defined processes
* Signatures missing required fields
* Signatures computed over non-canonical data
* Any attempt to bypass cryptographic validation

Any attempt to forge signatures, falsify hashes, or manipulate key material MUST be treated as a tamper condition and MUST trigger mandatory bitdump.

The cryptographic hashing, signature algorithms, and key requirements ensure that SDLP-governed objects maintain deterministic identity, verifiable lineage, and tamper-evident lifecycle physics across all compliant implementations. Cryptography forms the mathematical foundation of SDLP’s digital physics layer.

43. ObjectKey Lifecycle, Protection, and Zeroization Requirements

The ObjectKey is the foundational cryptographic identity of an SDLP-governed object. It is generated at instantiation, used throughout the object’s lifecycle to sign lineage entries and validate internal state, and destroyed irreversibly during bitdump. The ObjectKey MUST be handled according to strict lifecycle, protection, and zeroization requirements to ensure deterministic identity, tamper-evident transitions, and secure destruction physics.

ObjectKey Generation:

* The ObjectKey MUST be generated at instantiation using an approved asymmetric key generation algorithm with at least 128-bit security strength.
* ObjectKey generation MUST occur within a trusted execution environment or equivalent secure enclave.
* ObjectKeys MUST be unique per object and MUST never be reused or derived from other keys.
* ObjectKeys MUST NOT be derived from platform identifiers, user identifiers, or environmental characteristics.

ObjectKey Usage:

* The ObjectKey MUST be used to sign all lineage entries.
* The ObjectKey MUST be used to validate internal state transitions.
* The ObjectKey MUST NOT be used for encryption, key exchange, or any purpose outside SDLP-defined lifecycle operations.
* The ObjectKey MUST NOT sign non-canonical or non-validated data.
* The ObjectKey MUST NOT sign data outside the object’s own lifecycle domain.

ObjectKey Storage:

* The ObjectKey MUST be stored only in secure, non-exportable hardware-backed storage or equivalent protected memory.
* The ObjectKey MUST never be written to disk in plaintext.
* The ObjectKey MUST never be included in backups, snapshots, or memory dumps.
* The ObjectKey MUST NOT be accessible to user-mode processes or external applications.
* The ObjectKey MUST NOT be exportable under any circumstances.

ObjectKey Protection:

* Any attempt to access, export, modify, or replace the ObjectKey MUST be treated as a tamper condition.
* Any attempt to use the ObjectKey outside canonical SDLP operations MUST be treated as a tamper condition.
* Any attempt to bypass secure storage or inject alternate key material MUST trigger mandatory bitdump.
* The ObjectKey MUST be validated continuously during execution to ensure integrity and authenticity.

ObjectKey Rotation and Replacement:

* ObjectKeys MUST never be rotated, replaced, regenerated, or reissued after instantiation.
* Any attempt to rotate or replace the ObjectKey MUST be treated as a tamper condition.
* Objects requiring new key material MUST be instantiated as new objects with new identity and lineage.

ObjectKey Zeroization:

* During bitdump, the ObjectKey MUST be destroyed irreversibly.
* Zeroization MUST remove all key material from memory, secure storage, caches, registers, and hardware-backed enclaves.
* Zeroization MUST be atomic and MUST NOT allow partial destruction.
* Zeroization MUST NOT leave recoverable remnants or allow forensic reconstruction.
* Zeroization MUST occur before the object transitions to the Bitdumped state.

Zeroization Triggers:

* Bitdump events (tamper-induced or decay-induced)
* Hospice collapse (P <= 1)
* Signature forgery detection
* Lineage falsification detection
* Environment spoofing detection
* Unauthorized debugger attachment
* Unauthorized memory inspection or modification

Post-Zeroization Behavior:

* After zeroization, the object MUST be permanently non-functional.
* The object MUST NOT execute, transfer, or consume.
* The object MUST NOT accept lineage entries or apply policies.
* The object MUST NOT rebind identity or reinstantiate.
* The object MUST NOT regenerate or restore the ObjectKey.

The ObjectKey lifecycle, protection, and zeroization requirements ensure that SDLP-governed objects maintain secure, tamper-evident identity throughout their existence and undergo irreversible destruction when required by decay physics or tamper-response physics. The ObjectKey is the cryptographic anchor of SDLP’s digital physics model.

44. Lineage Entry Structure and Canonical Event Sequencing

Lineage is the authoritative, tamper-evident record of all lifecycle events performed by an SDLP-governed object. Each lineage entry represents a single, atomic state transition and MUST be signed by the ObjectKey. Lineage establishes the object’s historical identity, validates decay physics, and ensures deterministic reconstruction of the object’s lifecycle.

Lineage entries MUST be canonical, complete, and strictly ordered. Missing entries, reordered entries, or altered entries invalidate the lineage chain and MUST be treated as tamper conditions.

Lineage Entry Structure:

Each lineage entry MUST contain the following fields in canonical order:

* EntryIndex: A monotonically increasing integer starting at 0.
* PreviousHash: The hash of the previous lineage entry.
* NewState: The resulting state after the event.
* EventType: The type of lifecycle event performed.
* Timestamp: The UNIX epoch time of the event.
* DecayDelta: The change in P resulting from the event.
* P_After: The value of P after applying DecayDelta.
* EnvironmentInfo: Canonical environment validation data.
* PolicyVersion: The active policy version at the time of the event.
* Signature: The ObjectKey signature over all canonical fields.

Required Event Types:

* Instantiation
* Play
* Restart (rewind distance >= one-third threshold)
* Copy
* Upload
* Transfer
* PolicyUpdate
* EnvironmentValidation
* DecayCollapse (P = 0)
* HospiceEntry (P = 1)
* Bitdump (tamper-induced or decay-induced)

Canonical Event Sequencing:

* Lineage entries MUST be strictly sequential.
* EntryIndex MUST increase by exactly 1 for each new entry.
* No gaps, duplicates, or parallel sequences are permitted.
* PreviousHash MUST match the hash of the prior entry.
* NewState MUST reflect the canonical state transition rules.
* Events MUST be recorded immediately and atomically.

Atomicity Requirements:

* A lineage entry MUST be fully written, signed, and validated before any subsequent lifecycle operation may occur.
* Partial or incomplete lineage entries are prohibited.
* If a lineage entry cannot be completed, the object MUST halt and refuse further execution.

Lineage Validation:

SDLP objects MUST validate lineage integrity at the following times:

* At startup
* Before performing any lifecycle operation
* Before applying decay physics
* Before accepting a transfer
* Before applying a policy
* Before executing a state transition
* Continuously during environment validation

SDLP objects MUST reject:

* Missing lineage entries
* Reordered lineage entries
* Duplicate EntryIndex values
* Incorrect PreviousHash values
* Non-canonical field order
* Missing required fields
* Invalid or unverifiable signatures
* Lineage entries inconsistent with decay physics
* Lineage entries inconsistent with canonical state transitions

Any attempt to modify, reorder, falsify, or truncate lineage MUST be treated as a tamper condition and MUST trigger mandatory bitdump.

Lineage as a Forensic Record:

* Lineage MUST allow deterministic reconstruction of the object’s entire lifecycle.
* Lineage MUST provide a complete audit trail of decay, transfer, environment validation, and policy application.
* Lineage MUST be immutable once written.
* Lineage MUST be cryptographically verifiable without external context.

Canonical event sequencing ensures that SDLP-governed objects maintain a complete, tamper-evident, and cryptographically anchored record of their lifecycle. Lineage is the forensic backbone of SDLP’s digital physics model.

45. State Transition Rules and Canonical Lifecycle Graph

SDLP-governed objects MUST follow a deterministic, canonical lifecycle defined by a finite set of states and a strict set of allowable transitions. State transitions MUST be atomic, signed, recorded as lineage entries, and validated before execution. Any deviation from the canonical lifecycle graph MUST be treated as a tamper condition.

Canonical States:

* Instantiated: The object has been created and assigned an ObjectKey.
* Active: The object is usable and may perform lifecycle operations.
* Restricted: The object has reached a policy-defined threshold and may perform only a subset of lifecycle operations.
* Hospice: The object has P = 1 and MUST NOT perform further lifecycle operations except mandatory end-of-life destruction.
* Decayed: The object has P = 0 and MUST transition to Bitdump.
* Bitdumped: The object has undergone irreversible destruction and is permanently non-functional.

Canonical Lifecycle Graph:

The following transitions are permitted:

* Instantiated -> Active
* Active -> Active (play, rewind, copy, upload, transfer, policy update)
* Active -> Restricted (policy-defined threshold)
* Restricted -> Active (policy-defined recovery)
* Active -> Hospice (P = 1)
* Restricted -> Hospice (P = 1)
* Hospice -> Bitdumped (mandatory end-of-life destruction)
* Active -> Decayed (P = 0)
* Restricted -> Decayed (P = 0)
* Decayed -> Bitdumped (mandatory destruction)
* Active -> Bitdumped (tamper-induced)
* Restricted -> Bitdumped (tamper-induced)
* Instantiated -> Bitdumped (tamper-induced)

No other transitions are permitted.

State Transition Rules:

* All transitions MUST be recorded as lineage entries.
* All transitions MUST be signed by the ObjectKey.
* All transitions MUST be validated before execution.
* Transitions MUST be atomic and MUST NOT allow partial state changes.
* Transitions MUST NOT skip intermediate states unless explicitly defined (e.g., tamper-induced Bitdump).
* Transitions MUST NOT be reversible.

State Validation Requirements:

Before performing any state transition, SDLP objects MUST validate:

* Lineage integrity
* Signature validity
* Canonical encoding
* Decay physics consistency
* Policy compliance
* Environment validation
* ObjectKey integrity

Invalid transitions MUST be rejected and treated as tamper conditions.

Hospice State Rules:

* Hospice is entered when P = 1.
* Hospice prohibits all lifecycle operations except mandatory destruction.
* Hospice MUST transition to Bitdumped at 100% per second.
* Hospice MUST NOT allow play, rewind, copy, upload, or transfer.

Decayed State Rules:

* Decayed is entered when P = 0.
* Decayed MUST immediately transition to Bitdumped.
* Decayed MUST NOT allow any lifecycle operations.

Bitdumped State Rules:

* Bitdumped is final and irreversible.
* The object MUST be permanently non-functional.
* The ObjectKey MUST be zeroized.
* The object MUST NOT execute, transfer, or consume.
* The object MUST NOT accept lineage entries or apply policies.
* The object MUST NOT rebind identity or reinstantiate.

Canonical lifecycle enforcement ensures that SDLP-governed objects behave predictably, securely, and consistently across all compliant implementations. The lifecycle graph defines the complete set of allowable transitions and prohibits undefined or ambiguous behavior.

46. Environment Validation, Trust Signals, and Execution Preconditions

SDLP-governed objects MUST validate their execution environment before performing any lifecycle operation. Environment validation ensures that the object is running within a trusted, non-spoofed, non-tampered context. If the environment cannot be validated, the object MUST refuse execution or initiate mandatory bitdump depending on the severity of the violation.

Environment validation MUST be deterministic, canonical, and reproducible across all compliant implementations.

Environment Trust Requirements:

* The environment MUST provide a verifiable trust signal.
* The environment MUST NOT be spoofed, emulated, or replayed.
* The environment MUST NOT allow unauthorized debugger attachment.
* The environment MUST NOT allow unauthorized memory inspection.
* The environment MUST NOT allow modification of executable code.
* The environment MUST NOT allow time manipulation or clock rollback.
* The environment MUST NOT allow virtualization unless explicitly permitted by policy.

Trust Signals:

The environment MUST provide one or more of the following canonical trust signals:

* Hardware-backed attestation (TPM, Secure Enclave, or equivalent)
* EnvironmentKey signature validation
* Verified runtime integrity measurement
* Verified OS integrity measurement
* Verified application container integrity
* Verified anti-debugger state
* Verified anti-tamper state

Trust signals MUST be:

* Cryptographically verifiable
* Non-spoofable
* Non-replayable
* Canonically encoded
* Included in lineage entries

Execution Preconditions:

Before performing any lifecycle operation, the object MUST validate:

* Environment trust signals
* ObjectKey integrity
* Lineage integrity
* Policy compliance
* Decay physics consistency
* Canonical encoding of all inputs
* Timestamp monotonicity

If any precondition fails, the object MUST refuse execution.

Environment Validation Events:

* Environment validation MUST occur at startup.
* Environment validation MUST occur before each lifecycle operation.
* Environment validation MUST occur continuously during execution.
* Environment validation MUST be recorded as lineage entries.

Environment Spoofing Detection:

The following conditions MUST be treated as tamper events:

* Debugger attachment without authorization
* Memory inspection without authorization
* Modification of executable code
* Virtualization when not permitted by policy
* Clock rollback or timestamp manipulation
* Replay of previous trust signals
* Injection of forged trust signals
* OS-level or runtime-level integrity failure

Tamper Response:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST trigger mandatory bitdump.
* Environment spoofing is always a major violation.

Environment Replay Protection:

* Trust signals MUST include nonces or monotonic counters.
* Trust signals MUST NOT be accepted if previously observed.
* Trust signals MUST be bound to the current timestamp.
* Trust signals MUST be bound to the current object identity.

Environment Validation Failure Behavior:

* The object MUST NOT execute lifecycle operations.
* The object MUST NOT modify state.
* The object MUST NOT sign lineage entries.
* The object MUST NOT consume or transfer.
* The object MUST NOT accept policy updates.

Canonical environment validation ensures that SDLP-governed objects operate only within trusted, verifiable, and tamper-resistant contexts. Environment trust is a mandatory prerequisite for all lifecycle operations and forms a critical component of SDLP’s digital physics model.

47. Policy Enforcement, Thresholds, and Runtime Constraints

SDLP policies define the operational constraints, thresholds, and behavioral limits that govern an object’s runtime behavior. Policies MUST be deterministic, canonical, and cryptographically signed by the PolicyAuthorityKey. SDLP-governed objects MUST enforce policy rules before performing any lifecycle operation.

Policies MUST NOT be advisory or optional. Policy enforcement is a mandatory component of SDLP digital physics.

Policy Structure:

Each policy MUST contain the following canonical fields:

* PolicyVersion: A monotonically increasing integer.
* PolicyAuthoritySignature: A signature over all canonical fields.
* Thresholds: Policy-defined operational limits.
* Permissions: Allowed lifecycle operations.
* Restrictions: Prohibited lifecycle operations.
* EnvironmentRules: Required trust conditions.
* DecayRules: Policy-defined decay thresholds.
* TransferRules: Ownership and movement constraints.
* TimestampRules: Clock and monotonicity requirements.

Policy Application:

* Policies MUST be validated before application.
* Policies MUST be recorded as lineage entries.
* Policies MUST NOT be applied retroactively.
* Policies MUST NOT modify historical lineage.
* Policies MUST NOT override decay physics or tamper physics.

Threshold Enforcement:

Policies may define thresholds that restrict or modify runtime behavior. Thresholds MUST be deterministic and MUST NOT conflict with SDLP physics.

Examples of policy-defined thresholds:

* Minimum P required for transfer
* Maximum number of plays per time window
* Maximum rewind distance before restriction
* Environment trust level requirements
* Time-based access windows

Thresholds MUST be enforced at runtime and MUST be validated before each lifecycle operation.

Runtime Constraints:

Policies may impose runtime constraints on:

* Play frequency
* Rewind behavior
* Copy permissions
* Upload permissions
* Transfer eligibility
* Environment validation frequency
* Allowed execution platforms
* Allowed geographic regions (if permitted by higher-level policy)

Runtime constraints MUST be enforced before execution and MUST be recorded as lineage entries when they affect state.

Policy Violations:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST be treated as tamper conditions.
* Attempts to bypass policy enforcement MUST trigger mandatory bitdump.

Policy Update Rules:

* Policies may be updated only by the PolicyAuthorityKey.
* Policy updates MUST be recorded as lineage entries.
* Policy updates MUST NOT reduce security guarantees.
* Policy updates MUST NOT increase P or modify decay history.
* Policy updates MUST NOT alter canonical state transitions.

Policy and Decay Interaction:

* Policies may define additional decay thresholds but may not override core decay physics.
* Policies may restrict operations when P is below a threshold.
* Policies may not prevent hospice entry (P = 1).
* Policies may not prevent mandatory destruction (P <= 1).

Policy and Environment Interaction:

* Policies may require specific trust signals.
* Policies may prohibit execution in untrusted environments.
* Policies may require continuous environment validation.
* Policies may define environment-specific restrictions.

Policy and Transfer Interaction:

* Policies may define transfer eligibility rules.
* Policies may require CustomerKey signatures.
* Policies may restrict transfer frequency or destination.
* Policies may prohibit transfer when P is below a threshold.

Canonical Policy Enforcement:

* Policies MUST be enforced before every lifecycle operation.
* Policies MUST be validated continuously during execution.
* Policies MUST be cryptographically verifiable.
* Policies MUST be immutable once applied.

Policy enforcement ensures that SDLP-governed objects operate within defined constraints, respect lifecycle limits, and maintain secure, predictable behavior across all compliant implementations. Policies form the governance layer of SDLP’s digital physics model.

48. Transfer Mechanics, Ownership Binding, and CustomerKey Requirements

SDLP-governed objects support cryptographically verifiable ownership binding and transfer operations. Ownership is represented by the CustomerKey, which MUST be used to authorize transfers, validate possession, and bind the object to a specific customer identity without revealing personal information.

Ownership binding MUST be deterministic, canonical, and tamper-evident. Transfers MUST be atomic, signed, and recorded as lineage entries.

CustomerKey Requirements:

* The CustomerKey MUST be an asymmetric keypair with at least 128-bit security strength.
* The CustomerKey MUST be generated and controlled by the customer.
* The CustomerKey MUST NOT be derived from personal identifiers.
* The CustomerKey MUST NOT be exportable by the SDLP object.
* The CustomerKey MUST NOT be replaced without a transfer event.
* The CustomerKey MUST NOT be used for encryption or key exchange.

Ownership Binding:

* Ownership is established when the CustomerKey signs a binding event.
* Ownership binding MUST be recorded as a lineage entry.
* Ownership binding MUST include:
  - CustomerKey public component
  - Timestamp
  - PolicyVersion
  - Environment validation data
  - Signature over canonical fields
* Ownership binding MUST be validated before any lifecycle operation.

Transfer Mechanics:

* Transfers MUST be authorized by the current owner using the CustomerKey.
* Transfers MUST be accepted by the recipient using their own CustomerKey.
* Transfers MUST be recorded as lineage entries.
* Transfers MUST be atomic and MUST NOT allow partial completion.
* Transfers MUST NOT modify decay history or increase P.
* Transfers MUST NOT bypass environment validation.

Transfer Preconditions:

Before a transfer may occur, the object MUST validate:

* CustomerKey signature from the current owner
* CustomerKey signature from the recipient
* Environment trust signals
* Policy-defined transfer eligibility
* Decay physics consistency
* Timestamp monotonicity
* Canonical encoding of all transfer fields

Transfer Eligibility Rules:

Policies may define transfer restrictions, including:

* Minimum P required for transfer
* Maximum number of transfers per time window
* Allowed geographic regions
* Allowed execution platforms
* Required environment trust levels
* Prohibition of transfer during Restricted or Hospice states

Transfer Rejection Conditions:

Transfers MUST be rejected if:

* CustomerKey signatures are invalid
* Environment validation fails
* Policy prohibits transfer
* P is below a policy-defined threshold
* The object is in Hospice or Decayed state
* The transfer would violate canonical sequencing
* The transfer would create parallel lineage

Transfer and Lineage:

* Transfers MUST be recorded as lineage entries.
* Transfer entries MUST include:
   - PreviousHash
   - NewState
   - EventType = Transfer
   - CustomerKey (old)
   - CustomerKey (new)
   - Timestamp
   - PolicyVersion
   - EnvironmentInfo
   - Signature
* Transfer entries MUST be validated before execution.

Transfer and Decay Interaction:

* Transfers MUST NOT modify P.
* Transfers MUST NOT reset decay thresholds.
* Transfers MUST NOT bypass hospice or mandatory destruction.
* Transfers MUST NOT occur when P <= 1.

Transfer and Tamper Interaction:

The following MUST be treated as tamper conditions:

* Attempting to transfer without CustomerKey authorization
* Attempting to forge CustomerKey signatures
* Attempting to modify transfer lineage entries
* Attempting to bypass transfer eligibility rules
* Attempting to transfer from an untrusted environment

Tamper-induced transfers MUST trigger mandatory bitdump.

Canonical Ownership Model:

* Ownership MUST be cryptographically verifiable.
* Ownership MUST be transferable only through canonical events.
* Ownership MUST NOT be inferred from platform identity.
* Ownership MUST NOT be overridden by policy.
* Ownership MUST NOT be duplicated or forked.

Transfer mechanics, ownership binding, and CustomerKey requirements ensure that SDLP-governed objects maintain secure, verifiable, and tamper-evident ownership throughout their lifecycle. These rules form the identity and possession layer of SDLP's digital physics model.

49. Timestamp Monotonicity, Clock Integrity, and Anti-Rollback Guarantees

SDLP-governed objects MUST maintain strict timestamp monotonicity and MUST validate clock integrity before performing any lifecycle operation. Time is a critical component of SDLP digital physics, and any attempt to manipulate or spoof temporal data MUST be treated as a tamper condition.

Timestamp Requirements:

* All lineage entries MUST include a canonical UNIX epoch timestamp.
* Timestamps MUST be strictly monotonic.
* Timestamps MUST NOT decrease relative to the previous lineage entry.
* Timestamps MUST be validated against environment trust signals.
* Timestamps MUST be encoded as 64-bit big-endian integers.

Clock Integrity Requirements:

* The environment clock MUST be trusted and verifiable.
* The clock MUST NOT be adjustable by unprivileged processes.
* The clock MUST NOT be subject to rollback.
* The clock MUST NOT be spoofed, virtualized, or replayed.
* The clock MUST be validated continuously during execution.

Anti-Rollback Guarantees:

* If the current timestamp is less than the previous lineage timestamp, the object MUST treat this as a tamper condition.
* Clock rollback MUST trigger mandatory bitdump.
* Replay of previously observed timestamps MUST be rejected.
* Timestamps MUST be bound to environment trust signals to prevent replay attacks.

Timestamp Validation:

Before performing any lifecycle operation, the object MUST validate:

* Monotonicity relative to the previous lineage entry
* Environment-provided time attestation
* Policy-defined time windows
* Canonical encoding of timestamp fields
* Absence of rollback or replay conditions

Time-Based Policy Enforcement:

Policies may define:

* Access windows
* Transfer windows
* Maximum play frequency per time interval
* Maximum rewind frequency per time interval
* Environment validation intervals
* Decay threshold timing rules

Time-based policies MUST be enforced before execution and MUST be recorded as lineage entries when they affect state.

Temporal Tamper Detection:

The following conditions MUST be treated as tamper events:

* Clock rollback
* Clock freeze or stalling
* Clock acceleration beyond policy limits
* Replay of previously observed timestamps
* Forged or manipulated time attestation signals
* Desynchronization between environment time and object time
* Virtualized or emulated time sources

Temporal Tamper Response:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST trigger mandatory bitdump.
* Clock rollback is always a major violation.

Timestamp and Lineage Interaction:

* Each lineage entry MUST have a timestamp greater than or equal to the previous entry.
* Timestamp monotonicity MUST be validated before writing a new lineage entry.
* Non-monotonic timestamps MUST invalidate the lineage chain.
* Timestamps MUST be included in signature payloads.

Timestamp and Decay Interaction:

* Decay physics MUST NOT be bypassed by manipulating time.
* Time-based decay thresholds MUST be enforced strictly.
* Hospice and mandatory destruction MUST NOT be delayed by clock manipulation.
* Time-based restrictions MUST NOT be circumvented by rollback.

Canonical Temporal Model:

* Time MUST be treated as a deterministic, non-reversible dimension.
* Time MUST advance monotonically across the object’s lifecycle.
* Time MUST be cryptographically anchored to environment trust.
* Time MUST be immutable once recorded.

Timestamp monotonicity, clock integrity, and anti-rollback guarantees ensure that SDLP-governed objects maintain temporal consistency, prevent replay attacks, and enforce time-based policy and decay physics. Time is a foundational component of SDLP's digital physics model.

50. Canonical Serialization, Byte Encoding, and Cross-Platform Determinism

SDLP-governed objects MUST use a canonical, platform-independent serialization format for all lifecycle data, lineage entries, policy structures, and cryptographic payloads. Canonical serialization ensures that SDLP objects behave identically across all compliant implementations, regardless of hardware architecture, operating system, or runtime environment.

Canonical serialization is mandatory for all cryptographic operations and MUST be applied before hashing, signing, or validating any data.

Canonical Encoding Rules:

* All integers MUST be encoded as big-endian, fixed-width values.
* All timestamps MUST be encoded as 64-bit big-endian integers.
* All hashes MUST be encoded as raw bytes (not hex, not base64).
* All signatures MUST be encoded as raw bytes in canonical order.
* All strings MUST be encoded as UTF-8 without BOM.
* All boolean values MUST be encoded as a single byte (0x00 or 0x01).
* All arrays MUST be length-prefixed using a 32-bit big-endian integer.
* All structures MUST be encoded in strict field order with no optional reordering.

Prohibited Encoding Variants:

* Hexadecimal or base64 encoding of hashes or signatures
* Little-endian integer encoding
* Variable-width integers
* Platform-native serialization formats
* JSON, XML, CBOR, protobuf, or other schema-based encodings
* Padding, whitespace, or alignment bytes
* Floating-point representations of any field

Canonical Structure Encoding:

Each canonical structure MUST be encoded as:

* FieldCount (32-bit big-endian)
* Field1
* Field2
* ...
* FieldN

Fields MUST be encoded exactly in the order defined by the SDLP specification. Missing fields, reordered fields, or additional fields invalidate the structure.

Canonical Lineage Encoding:

Lineage entries MUST be encoded using the canonical structure rules and MUST include:

* EntryIndex (64-bit big-endian)
* PreviousHash (raw bytes)
* NewState (32-bit big-endian)
* EventType (32-bit big-endian)
* Timestamp (64-bit big-endian)
* DecayDelta (32-bit big-endian)
* P_After (32-bit big-endian)
* EnvironmentInfo (canonical structure)
* PolicyVersion (32-bit big-endian)
* Signature (raw bytes)

Canonical Policy Encoding:

Policies MUST be encoded as:

* PolicyVersion (32-bit big-endian)
* Thresholds (canonical structure)
* Permissions (canonical structure)
* Restrictions (canonical structure)
* EnvironmentRules (canonical structure)
* DecayRules (canonical structure)
* TransferRules (canonical structure)
* TimestampRules (canonical structure)
* PolicyAuthoritySignature (raw bytes)

Cross-Platform Determinism:

* Serialization MUST produce identical byte sequences across all platforms.
* Hashes computed over canonical data MUST be identical across all platforms.
* Signatures computed over canonical data MUST be identical across all platforms.
* Canonical encoding MUST NOT depend on:
  - CPU architecture
  - Endianness
  - Word size
  - Operating system
  - Runtime environment
  - Compiler or interpreter behavior

Canonical Validation:

Before accepting any serialized data, SDLP objects MUST validate:

* Field order
* Field count
* Field encoding
* Integer width and endianness
* Raw byte encoding of hashes and signatures
* Absence of padding or extraneous bytes
* Canonical UTF-8 encoding for strings

Serialization Tamper Detection:

The following conditions MUST be treated as tamper events:

* Non-canonical field order
* Incorrect integer width or endianness
* Hex or base64-encoded hashes or signatures
* Additional or missing fields
* Non-canonical UTF-8 encoding
* Platform-native serialization formats
* Floating-point representations of any field

Tamper Response:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST trigger mandatory bitdump.
* Any attempt to bypass canonical serialization is a major violation.

Canonical serialization, byte encoding, and cross-platform determinism ensure that SDLP-governed objects maintain consistent, verifiable, and tamper-evident behavior across all compliant implementations. Canonical encoding is the foundation of SDLP’s cryptographic and forensic integrity.

51. Rehydration Prohibition, Resurrection Prevention, and Anti-Reinstantiation Guarantees

SDLP-governed objects MUST enforce strict prohibitions against any form of rehydration, resurrection, or reinstantiation after destruction. Once an object enters the Bitdumped state, its lifecycle is permanently terminated. No compliant implementation may allow an object to return to a functional state after destruction.

Rehydration, resurrection, and reinstantiation attempts MUST be treated as tamper events and MUST trigger mandatory bitdump of any object involved in the attempt.

Rehydration Prohibition:

* SDLP objects MUST NOT allow reconstruction from serialized data.
* SDLP objects MUST NOT allow reloading of previous internal state.
* SDLP objects MUST NOT allow reassembly from lineage entries.
* SDLP objects MUST NOT allow rollback to earlier lifecycle states.
* SDLP objects MUST NOT allow restoration of destroyed key material.

Resurrection Prevention:

* Bitdumped objects MUST NOT be reinitialized.
* Bitdumped objects MUST NOT accept new lineage entries.
* Bitdumped objects MUST NOT accept new policies.
* Bitdumped objects MUST NOT validate environment trust signals.
* Bitdumped objects MUST NOT perform any lifecycle operation.

Anti-Reinstantiation Guarantees:

* A destroyed object MUST NOT be instantiated again using the same ObjectKey.
* A destroyed object MUST NOT be instantiated again using the same lineage.
* A destroyed object MUST NOT be instantiated again using the same identity fields.
* Any attempt to recreate an object using prior state MUST be treated as a tamper condition.

Canonical Destruction Finality:

* Bitdump is a terminal lifecycle state.
* Bitdumped objects MUST be permanently non-functional.
* Bitdumped objects MUST NOT be transferable.
* Bitdumped objects MUST NOT be recoverable by any means.
* Bitdumped objects MUST NOT be rehydrated, resurrected, or reinstantiated.

Rehydration Tamper Detection:

The following conditions MUST be treated as tamper events:

* Loading serialized data from a destroyed object.
* Attempting to reconstruct lineage from a destroyed object.
* Attempting to restore ObjectKey material.
* Attempting to bypass destruction proofs.
* Attempting to clone or fork a destroyed object.
* Attempting to rebind identity fields after destruction.

Rehydration Tamper Response:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST trigger mandatory bitdump.
* Any attempt to resurrect a destroyed object is a major violation.

Destruction Proof Requirements:

* A destruction proof MUST be final and immutable.
* A destruction proof MUST NOT allow reconstruction of the object.
* A destruction proof MUST NOT contain recoverable state.
* A destruction proof MUST be verifiable without revealing internal data.

Canonical destruction finality ensures that SDLP-governed objects cannot be revived, reconstructed, or reinstantiated after destruction. These guarantees preserve the integrity of decay physics, tamper physics, and the irreversible nature of the Bitdumped state.

52. Bitdump Mechanics, Zeroization Protocol, and Irreversible Destruction Physics

This section defines the mandatory enforcement behavior associated with Zeroization-class terminal states described in SDLP-arch. While SDLP-arch specifies the architectural meaning of irreversible destruction, this section specifies the normative mechanics, sequencing, and security requirements that govern how destruction is executed. Bitdump is the canonical enforcement mechanism for Predictive Digital Security and MUST be implemented by all SDLP-governed objects.

SDLP-governed objects MUST implement a deterministic, irreversible destruction mechanism known as bitdump. Bitdump is the terminal lifecycle state and represents the permanent, cryptographically provable end of an object’s existence. Once bitdump begins, the object MUST NOT be recoverable, reconstructable, or executable.

Bitdump is a mandatory component of SDLP digital physics and MUST be triggered under specific conditions defined by decay physics, tamper physics, and policy enforcement.

Bitdump Preconditions:

Bitdump MUST be initiated when any of the following occur:

* P <= 1 (mandatory destruction threshold)
* Critical tamper event detected
* Lineage corruption or invalidation
* ObjectKey corruption or unauthorized replacement attempt
* Clock rollback or temporal replay
* Environment spoofing or debugger attachment
* PolicyAuthoritySignature failure
* Explicit policy-defined destruction event

Bitdump MUST NOT be deferrable, reversible, or suppressible.

Zeroization Protocol:

Upon entering bitdump, the object MUST:

* Zeroize all cryptographic key material
* Zeroize all internal state variables
* Zeroize all cached lineage data
* Zeroize all environment trust data
* Zeroize all policy structures
* Zeroize all buffers containing sensitive information

Zeroization MUST:

* Occur in constant time
* Be complete and non-recoverable
* Not rely on garbage collection or runtime heuristics
* Not leave residual data in memory or storage
* Not allow partial or selective zeroization

Destruction Mechanics:

After zeroization, the object MUST:

* Invalidate its ObjectKey permanently
* Invalidate all signatures
* Invalidate all lineage entries
* Invalidate all policy bindings
* Invalidate all environment trust bindings
* Transition to the Destroyed state

The Destroyed state MUST be terminal and MUST NOT allow:

* Execution
* Transfer
* Rewind
* Play
* Policy updates
* Environment validation
* Serialization
* Rehydration
* Reinstantiation

Destruction MUST be absolute.

Destruction Proof:

After bitdump completes, the object MUST emit a destruction proof containing:

* Final lineage hash
* Timestamp of destruction
* Reason code (Decay, Tamper, Policy, or Critical Fault)
* Zeroization confirmation flag

The destruction proof MUST:

* Be verifiable externally
* Not contain sensitive material
* Not reveal internal state
* Not allow reconstruction of the object

Destruction proof MUST NOT be signed by the ObjectKey, as the key is zeroized prior to emission.

Destruction and Lineage:

* Bitdump MUST NOT write a new lineage entry.
* Lineage MUST be considered complete at the last valid entry.
* Destruction proof MUST reference the final lineage hash.
* Lineage MUST NOT be modifiable after destruction.

Destruction and Decay Interaction:

* P <= 1 MUST always trigger bitdump.
* Decay physics MUST NOT be bypassed or delayed.
* Hospice state MUST NOT prevent destruction.
* Policy MUST NOT override mandatory destruction.

Destruction and Tamper Interaction:

* Critical tamper events MUST trigger immediate bitdump.
* Tamper-induced destruction MUST NOT allow any further execution.
* Tamper events MUST NOT allow partial zeroization.

Post-Destruction Behavior:

* The object MUST be permanently non-functional.
* The object MUST NOT respond to any lifecycle operations.
* The object MUST NOT validate, serialize, or execute.
* The object MUST NOT be transferable.
* The object MUST NOT be recoverable by any means.

Bitdump mechanics, zeroization protocol, and irreversible destruction physics ensure that SDLP-governed objects terminate cleanly, securely, and provably. Destruction is a fundamental component of SDLP's digital physics model and guarantees the finality and integrity of the lifecycle.

53. Rehydration Prohibition, Resurrection Prevention, and Anti-Reinstantiation Physics

This section defines the mandatory enforcement behavior associated with destruction finality as described in SDLP-arch. While SDLP-arch establishes the architectural principle that Zeroization-class states are irreversible, this section specifies the normative security requirements that prevent any form of rehydration, resurrection, or reinstantiation after destruction. These guarantees preserve the integrity of decay physics, tamper physics, and the irreversible nature of the Bitdumped state.

SDLP-governed objects MUST enforce strict prohibitions against any form of rehydration, resurrection, or reinstantiation after destruction. Once an object enters the Bitdumped state, its lifecycle is permanently terminated. No compliant implementation may allow an object to return to a functional state after destruction.

Rehydration, resurrection, and reinstantiation attempts MUST be treated as tamper events and MUST trigger mandatory bitdump of any object involved in the attempt.

Rehydration Prohibition:

* SDLP objects MUST NOT allow reconstruction from serialized data.
* SDLP objects MUST NOT allow reloading of previous internal state.
* SDLP objects MUST NOT allow reassembly from lineage entries.
* SDLP objects MUST NOT allow rollback to earlier lifecycle states.
* SDLP objects MUST NOT allow restoration of destroyed key material.

Resurrection Prevention:

* Bitdumped objects MUST NOT be reinitialized.
* Bitdumped objects MUST NOT accept new lineage entries.
* Bitdumped objects MUST NOT accept new policies.
* Bitdumped objects MUST NOT validate environment trust signals.
* Bitdumped objects MUST NOT perform any lifecycle operation.

Anti-Reinstantiation Guarantees:

* A destroyed object MUST NOT be instantiated again using the same ObjectKey.
* A destroyed object MUST NOT be instantiated again using the same lineage.
* A destroyed object MUST NOT be instantiated again using the same identity fields.
* Any attempt to recreate an object using prior state MUST be treated as a tamper condition.

Canonical Destruction Finality:

* Bitdump is a terminal lifecycle state.
* Bitdumped objects MUST be permanently non-functional.
* Bitdumped objects MUST NOT be transferable.
* Bitdumped objects MUST NOT be recoverable by any means.
* Bitdumped objects MUST NOT be rehydrated, resurrected, or reinstantiated.

Rehydration Tamper Detection:

The following conditions MUST be treated as tamper events:

* Loading serialized data from a destroyed object.
* Attempting to reconstruct lineage from a destroyed object.
* Attempting to restore ObjectKey material.
* Attempting to bypass destruction proofs.
* Attempting to clone or fork a destroyed object.
* Attempting to rebind identity fields after destruction.

Rehydration Tamper Response:

* Minor violations MUST cause immediate refusal of execution.
* Major violations MUST trigger mandatory bitdump.
* Any attempt to resurrect a destroyed object is a major violation.

Destruction Proof Interaction:

* Destruction proofs MUST NOT contain recoverable state.
* Destruction proofs MUST NOT allow reconstruction of the object.
* Destruction proofs MUST be verifiable without revealing internal data.
* Destruction proofs MUST be immutable once emitted.

Canonical destruction finality ensures that SDLP-governed objects cannot be revived, reconstructed, or reinstantiated after destruction. These guarantees preserve the integrity of decay physics, tamper physics, and the irreversible nature of the Bitdumped state.

54. Initialization Enforcement and Pre-Init Termination Mechanics

This section defines the enforcement requirements governing the Initialization phase of an SDLP instance. Initialization is the mandatory trust boundary at which an encoded instance evaluates its host environment to determine whether lifecycle activation may proceed. If any Initialization precondition fails, the instance MUST perform Pre-Init Termination. Because the instance has not yet entered the lifecycle, Pre-Init Termination is a security measure rather than a lifecycle event.

54.1 Initialization Preconditions

An SDLP instance MUST evaluate the following conditions prior to Initialization. Failure of any condition MUST result in Pre-Init Termination.

* Environment Integrity: The host environment MUST NOT exhibit corruption, instability, or incomplete initialization.

* Trust Anchors: Required trust anchors MUST be present, valid, and cryptographically verifiable.

* Time Integrity: The environment MUST provide a time source that satisfies SDLP time-integrity requirements.

* Lineage Preconditions: The environment MUST provide the lineage context necessary to validate the instances lineage seed.

* Policy Preconditions: The environment MUST provide the policy bindings required for activation.

* Anti-Tamper Preconditions: The environment MUST NOT exhibit tamper, replay, debugging, or instrumentation characteristics.

54.2 Initialization Validation Procedure

Prior to activation, an SDLP instance MUST perform the following validation steps in the order defined:

1. Verify environment integrity.
2. Validate trust anchors.
3. Validate time integrity.
4. Validate lineage prerequisites.
5. Validate policy prerequisites.
6. Evaluate anti-tamper conditions.
7. Confirm that no prohibited conditions are present.

If all steps succeed, Initialization MAY proceed. If any step fails, the instance MUST perform Pre-Init Termination.

54.3 Pre-Init Termination Requirements

When Initialization preconditions are not met, the instance MUST:

* Immediately cease all activation attempts.
* Zeroize all volatile cryptographic material.
* Emit a Bitdump containing a Pre-Init Termination reason code.
* Emit an LDR indicating that the instance never entered the lifecycle.
* Enter a terminal state from which no activation is possible.

Pre-Init Termination MUST NOT be suppressible, deferrable, or overrideable by any user, platform, or external process.

54.4 Prohibited Conditions

The following conditions MUST result in immediate Pre-Init Termination:

* Debugger presence or instrumentation.
* Replay of Initialization context.
* Rooted, jailbroken, or privilege-escalated environments.
* Missing or invalid trust anchors.
* Invalid or unverifiable time source.
* Corrupted or incomplete environment state.
* Missing lineage or policy prerequisites.
* Any condition that would compromise lifecycle integrity.

54.5 Post-Termination Behavior

After Pre-Init Termination:

* The instance MUST NOT attempt re-Initialization.
* The instance MUST NOT enter any lifecycle state.
* The instance MUST NOT accept further input.
* The instance MUST NOT be capable of rehydration, resurrection, or reinstantiation.
* The instance MUST be treated as a terminated object for all purposes of SDLP enforcement.

54.6 Security Rationale

Initialization is the final opportunity for an SDLP instance to prevent activation in an unsafe environment. Because user behavior and platform integrity cannot be relied upon as security controls, Pre-Init Termination ensures that:

* protected content cannot be extracted,
* identity and lineage cannot be compromised,
* policy cannot be bypassed,
* lifecycle integrity cannot be subverted, and
* unsafe environments cannot coerce activation.

54.7 Summary

Initialization Enforcement ensures that SDLP instances activate only in environments capable of upholding SDLP security guarantees. Pre-Init Termination provides mandatory protection against unsafe or adversarial conditions. Together, these mechanics preserve the integrity of identity, lineage, policy, and lifecycle continuity across the SDLP ecosystem.

55. Security Considerations

SDLP is a security architecture. All sections of this document are security-relevant. The protocol defines strict lifecycle controls, irreversible destruction rules, tamper-detection mechanisms, trust evaluation requirements, and deterministic serialization guarantees intended to prevent duplication, forgery, rollback, replay, and unauthorized reinstantiation of SDLP-governed objects.

Implementations MUST ensure that all cryptographic operations, environmental attestation steps, trust evaluations, and lifecycle transitions are performed exactly as specified. Deviations from the SDLP physics model may result in security vulnerabilities, including unauthorized object recovery, lineage corruption, or bypass of Bit-Drop destruction guarantees.

Implementations MUST also ensure that environment trust signals, timestamp sources, policy authorities, and distributor credentials are authenticated and cannot be spoofed, replayed, or substituted. Failure to validate these signals may allow unauthorized activation, lifecycle manipulation, or evasion of destruction requirements.

SDLP-compliant systems MUST treat all untrusted environments as hostile. Any detection of tampering, corruption, or malicious interference MUST trigger SafeMode, Restricted state, or immediate Bit-Drop as defined by the SDLP lifecycle physics and trust model.

Implementers MUST ensure that logs, policies, and event payloads are immutable, append-only, and cryptographically verifiable. Any break in log-chain integrity, policy signature validity, or event authenticity MUST be treated as a security-critical condition.

SDLP relies on deterministic behavior for all lifecycle, compliance, and destruction operations. Non-deterministic behavior, ambiguous state transitions, or inconsistent serialization may introduce

exploitable inconsistencies and MUST be avoided.

Implementations MUST assume that adversaries may attempt to exploit timing, ordering, or environmental inconsistencies to bypass lifecycle controls. All SDLP operations MUST therefore be atomic, reproducible, and resistant to manipulation.

Finally, implementers MUST ensure that Bit-Drop is implemented as an instantaneous, irreversible destruction operation. Partial destruction, delayed destruction, or recoverable destruction semantics are prohibited and constitute a violation of SDLPs security guarantees.

56. IANA Considerations

This document has no IANA actions.

57. Acknowledgments

The author acknowledges the contributions of reviewers, implementers, and members of the SDLP community whose feedback helped refine the architecture and clarify the digital physics model.

58. Changelog

draft-norton-sdlp-sec-arch-00

* Initial version of the SDLP Security Architecture.
* Includes structural definitions, object model, identifiers, lineage primitives, policy structures, environment model, and lifecycle framework.
* Physics Layer (Sections 39-54) will be introduced in draft-norton-sdlp-sec-arch-01.

59. Author’s Address

M. Norton
Email: mark433norton@gmail.com