RFC-027: Vault Migration

Status: Draft **Date:** January 2026 **Author:** Derrell Piper ddp@eludom.net
Implementation: Proposed

Abstract

This RFC specifies vault migration for the Library of Cyberspace: how to move vaults between hosts, storage backends, and administrative domains while preserving content integrity, capability chains, and audit continuity. Migration is essential for long-term preservation.

Motivation

Vaults must outlive their infrastructure:

- **Hardware failure** - Disks die, servers decommission
- **Provider changes** - Cloud vendors come and go
- **Jurisdiction** - Legal requirements may force relocation
- **Performance** - Move closer to users
- **Cost** - Cheaper storage becomes available

Migration must be:

- **Complete** - All objects, metadata, capabilities
- **Verifiable** - Cryptographic proof of integrity
- **Resumable** - Handle interruptions gracefully
- **Auditable** - Full trail of what moved where

Migration Types

Full Migration

Move entire vault to new location:

```
(define (full-migration source-vault target-host)
  "Migrate entire vault to new host"
  (let ((migration-id (generate-migration-id)))
    (audit-append action: `(migration-start ,migration-id)
                  type: 'full
                  source: source-vault
                  target: target-host)

    ;; Phase 1: Snapshot
```

```
    (let ((snapshot (vault-snapshot source-vault)))
      ;; Phase 2: Transfer
      (transfer-snapshot snapshot target-host)
      ;; Phase 3: Verify
      (verify-transfer snapshot target-host)
      ;; Phase 4: Cutover
      (cutover source-vault target-host migration-id))))
```

Partial Migration

Move subset of objects:

```
(define (partial-migration source-vault target-host predicate)
  "Migrate objects matching predicate"
  (let ((objects (soup-query source-vault predicate)))
    (for-each (lambda (obj)
                (migrate-object obj target-host))
              objects)))
```

Live Migration

Migrate while vault remains active:

```
(define (live-migration source-vault target-host)
  "Migrate without downtime"
  ;; Phase 1: Initial sync
  (let ((checkpoint (initial-sync source-vault target-host)))
    ;; Phase 2: Catch up changes
    (let loop ((last-checkpoint checkpoint))
      (let ((changes (changes-since source-vault last-checkpoint)))
        (if (< (length changes) *catchup-threshold*)
            ;; Phase 3: Final cutover (brief pause)
            (atomic-cutover source-vault target-host changes)
            (loop (sync-changes changes target-host)))))))
```

Object Transfer

Export Format

```
;; Sealed archive format (RFC-018)
(define (export-object hash)
  `(vault-object
    (hash ,hash)
    (content ,(cas-get hash))
    (soup-entry ,(soup-get hash))
    (references ,(object-references hash))
```

```
    (signatures ,(object-signatures hash))))

(define (export-batch hashes)
  (let ((objects (map export-object hashes)))
    (seal-archive objects
      compression: 'zstd
      encryption: target-vault-key)))
```

Transfer Protocol

```
(define (transfer-object obj source target)
  "Transfer single object with verification"
  (let* ((hash (object-hash obj))
         (data (export-object hash)))
    ;; Send to target
    (vault-send target data)
    ;; Get acknowledgment with hash
    (let ((ack (vault-receive target)))
      (unless (equal? (ack-hash ack) hash)
        (error "Transfer verification failed" hash))
      (audit-append action: `(object-transferred ,hash)
                    source: source
                    target: target))))
```

Streaming Transfer

```
(define (stream-transfer source target)
  "Stream objects for efficient bulk transfer"
  (let ((stream (open-transfer-stream source target)))
    (for-each (lambda (hash)
                (let ((obj (export-object hash)))
                  (stream-write stream obj)
                  (when (stream-buffer-full? stream)
                    (stream-flush stream))))
              (vault-all-hashes source))
    (stream-close stream)))
```

Metadata Migration

Soup Entries

```
(define (migrate-soup-entry hash source target)
  "Migrate soup metadata for object"
  (let ((entry (soup-get source hash)))
    (when entry
```

```
      (soup-put target hash entry)
      ;; Verify
      (unless (equal? (soup-get target hash) entry)
        (error "Soup entry migration failed" hash)))))
```

Index Migration

```
(define (migrate-indexes source target)
  "Rebuild indexes on target"
  ;; Option 1: Export and import index data
  (let ((index-data (export-indexes source)))
    (import-indexes target index-data))

  ;; Option 2: Rebuild from soup (slower but guaranteed correct)
  (rebuild-indexes target))
```

Audit Trail

```
(define (migrate-audit source target)
  "Migrate audit trail (critical for provenance)"
  (let ((audit-entries (audit-export source)))
    ;; Audit entries are append-only, transfer in order
    (for-each (lambda (entry)
                (audit-import target entry))
              audit-entries)
    ;; Add migration event to both
    (let ((migration-entry
           `(migration-audit
             (source ,source)
             (target ,target)
             (entries ,(length audit-entries))
             (timestamp ,(current-time)))))
      (audit-append source migration-entry)
      (audit-append target migration-entry))))
```

Capability Migration

Certificate Transfer

```
(define (migrate-capabilities source target)
  "Migrate SPKI certificates"
  (let ((certs (soup-query source type: 'certificate)))
    (for-each (lambda (cert)
                ;; Certificates are content-addressed, transfer directly
                (let ((hash (cert-hash cert)))
```

```
                (migrate-object hash source target)))
            certs)))
```

Key Migration

```
;; Private keys require special handling
(define (migrate-vault-keys source target ceremony-witnesses)
  "Migrate vault keys via key ceremony"
  ;; Never transfer private keys directly!
  ;; Use threshold re-sharing or key ceremony
  (let* ((old-shares (gather-key-shares source))
         (new-key (generate-new-key target))
         (transition-cert (create-key-transition-cert
                           old-key: (vault-public-key source)
                           new-key: new-key
                           witnesses: ceremony-witnesses)))
    ;; Sign transition with old key
    (sign-transition old-shares transition-cert)
    ;; Record in both vaults
    (audit-append source action: `(key-transition ,transition-cert))
    (audit-append target action: `(key-transition ,transition-cert))))
```

Delegation Chain Continuity

```
(define (verify-delegation-chains target)
  "Verify all delegation chains remain valid after migration"
  (let ((certs (soup-query target type: 'certificate)))
    (for-each (lambda (cert)
                (let ((chain (build-chain cert)))
                  (unless (valid-chain? chain)
                    (warn "Broken delegation chain" cert))))
              certs)))
```

Verification

Content Verification

```
(define (verify-migration source target)
  "Verify all content transferred correctly"
  (let ((source-hashes (vault-all-hashes source))
        (target-hashes (vault-all-hashes target)))

    ;; Check completeness
    (let ((missing (set-difference source-hashes target-hashes)))
      (unless (null? missing)
```

```
        (error "Missing objects after migration" missing)))

    ;; Check integrity (sample verification for large vaults)
    (let ((sample (random-sample source-hashes 1000)))
      (for-each (lambda (hash)
                  (unless (equal? (cas-get source hash)
                                  (cas-get target hash))
                    (error "Content mismatch" hash)))
                sample))

    ;; Check soup metadata
    (for-each (lambda (hash)
                (unless (equal? (soup-get source hash)
                                (soup-get target hash))
                  (error "Soup mismatch" hash)))
              (random-sample source-hashes 1000))))
```

Merkle Verification

```
(define (merkle-verify-migration source target)
  "Verify migration via Merkle root comparison"
  (let ((source-root (vault-merkle-root source))
        (target-root (vault-merkle-root target)))
    (unless (equal? source-root target-root)
      ;; Find divergence
      (let ((divergence (find-merkle-divergence source target)))
        (error "Merkle verification failed" divergence)))))
```

Audit Verification

```
(define (verify-audit-continuity source target)
  "Verify audit trail continuity"
  (let ((source-audit (audit-export source))
        (target-audit (audit-export target)))
    ;; Target should have all source entries plus migration events
    (unless (subsequence? source-audit target-audit)
      (error "Audit trail discontinuity"))))
```

Cutover

Atomic Cutover

```
(define (atomic-cutover source target migration-id)
  "Atomically switch from source to target"
  ;; Phase 1: Quiesce source
```

```
  (vault-readonly! source)

  ;; Phase 2: Final sync
  (let ((final-changes (changes-since source (last-sync-point))))
    (sync-changes final-changes target))

  ;; Phase 3: Verify
  (verify-migration source target)

  ;; Phase 4: Update DNS/routing
  (update-vault-routing source target)

  ;; Phase 5: Activate target
  (vault-activate! target)

  ;; Phase 6: Record completion
  (audit-append target action: `(migration-complete ,migration-id)))
```

Rollback

```
(define (migration-rollback migration-id)
  "Rollback failed migration"
  (let ((migration (get-migration migration-id)))
    (case (migration-phase migration)
      ((transfer)
       ;; Just abandon target
       (vault-delete! (migration-target migration)))
      ((cutover)
       ;; Restore routing to source
       (update-vault-routing (migration-target migration)
                             (migration-source migration))
       (vault-readonly! (migration-target migration))
       (vault-activate! (migration-source migration)))
      ((complete)
       (error "Cannot rollback completed migration")))))
```

Incremental Migration

Checkpoint System

```
(define (migration-checkpoint migration-id progress)
  "Save migration progress for resumption"
  (let ((checkpoint
         `(checkpoint
           (migration-id ,migration-id)
```

```
            (timestamp ,(current-time))
            (transferred ,(progress-transferred progress))
            (remaining ,(progress-remaining progress))
            (last-hash ,(progress-last-hash progress)))))
    (cas-put (serialize checkpoint))))

(define (resume-migration migration-id)
  "Resume interrupted migration"
  (let ((checkpoint (latest-checkpoint migration-id)))
    (if checkpoint
        (continue-from-checkpoint checkpoint)
        (error "No checkpoint found" migration-id))))
```

Change Tracking

```
;; Track changes during migration
(define (changes-since vault timestamp)
  "Get all changes since timestamp"
  (soup-query vault
              modified: (> timestamp)
              order-by: '((modified asc))))

(define (sync-changes changes target)
  "Apply changes to target"
  (for-each (lambda (change)
              (case (change-type change)
                ((create modify)
                 (migrate-object (change-hash change) target))
                ((delete)
                 (cas-delete target (change-hash change)))))
            changes))
```

Storage Backend Migration

Backend Abstraction

```
;; Migrate between storage backends
(define (backend-migration vault old-backend new-backend)
  "Migrate vault to different storage backend"
  (for-each (lambda (hash)
              (let ((data (backend-get old-backend hash)))
                (backend-put new-backend hash data)
                (verify-backend-transfer hash old-backend new-backend)))
            (backend-all-hashes old-backend)))
```

```
;; Example: S3 to local filesystem
(define (s3-to-local vault bucket path)
  (backend-migration vault
                     (make-s3-backend bucket)
                     (make-fs-backend path)))
```

Format Conversion

```
;; Migrate between storage formats
(define (format-migration vault old-format new-format)
  "Convert storage format during migration"
  (for-each (lambda (hash)
              (let* ((old-data (read-format old-format hash))
                     (new-data (convert-format old-data new-format)))
                (write-format new-format hash new-data)))
            (vault-all-hashes vault)))
```

Federation Migration

Vault Federation Changes

```
(define (federation-migrate vault old-federation new-federation)
  "Migrate vault between federations"
  ;; Leave old federation
  (federation-leave old-federation vault)

  ;; Update vault metadata
  (vault-set-federation! vault new-federation)

  ;; Join new federation
  (federation-join new-federation vault)

  ;; Announce to peers
  (federation-announce new-federation vault))
```

Cross-Federation Transfer

```
(define (cross-federation-transfer hash source-fed target-fed)
  "Transfer object between federations"
  (let* ((source-vault (federation-locate source-fed hash))
         (target-vault (federation-select target-fed))
         (obj (vault-get source-vault hash)))
    (vault-put target-vault obj)
    (federation-announce target-fed hash target-vault)))
```

Security Considerations

Migration Authorization

```
;; Migration requires appropriate capability
(spki-cert
  (issuer vault-admin)
  (subject migration-operator)
  (capability
    (action migrate)
    (object vault-id))
  (validity (not-after "2026-02-01")))

(define (authorized-migration? operator source target)
  (and (has-capability? operator 'migrate source)
       (has-capability? operator 'write target)))
```

Encryption in Transit

```
(define (secure-transfer source target)
  "Transfer with encryption"
  (let ((session-key (establish-session-key source target)))
    (for-each (lambda (hash)
                (let* ((data (cas-get source hash))
                       (encrypted (encrypt session-key data)))
                  (send-encrypted target encrypted)))
              (vault-all-hashes source))))
```

Migration Audit

```
;; All migrations are fully audited
(define (audited-migration source target)
  (let ((migration-id (generate-migration-id)))
    (audit-append action: 'migration-authorized
                  migration-id: migration-id
                  operator: (current-principal)
                  source: source
                  target: target)
    ;; ... perform migration ...
    (audit-append action: 'migration-complete
                  migration-id: migration-id
                  objects: (count-migrated)
                  verification: (verification-result))))
```

References

1. Live Migration of Virtual Machines - Clark et al.
2. RFC-018: Sealed Archive Format
3. RFC-022: Key Ceremony Protocol
4. RFC-024: Network Protocol

Changelog

- **2026-01-07** - Initial draft

Implementation Status: Draft **Dependencies:** vault, cas, soup, federation
Integration: Vault operations, federation, disaster recovery