RFC-030: Encryption at Rest

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

Abstract

This RFC specifies encryption at rest for the Library of Cyberspace: how vaults protect stored data using modern cryptography while maintaining content-addressability, key management, and operational flexibility. All sensitive data is encrypted before touching persistent storage.

Motivation

Data at rest faces threats:

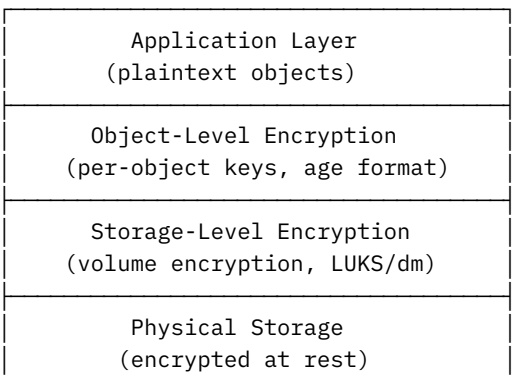
- **Physical theft** - Disks, servers, backups stolen
- **Insider access** - Unauthorized admin access
- **Legal compulsion** - Forced disclosure orders
- **Decommissioning** - Data remnants on old hardware

Encryption must be:

- **Transparent** - Applications unaware of encryption
- **Performant** - Minimal overhead
- **Recoverable** - Key loss doesn't mean data loss
- **Auditable** - All key operations logged

Encryption Model

Layered Encryption



└──────────────────────────────────────┘

Encryption Scope

```scheme
;; What gets encrypted
(define encryption-policy
  `((content . always)           ; Object content - always encrypted
    (soup-metadata . optional)   ; Metadata - configurable
    (indexes . optional)         ; Search indexes - configurable
    (audit-log . always)         ; Audit trail - always encrypted
    (keys . never-plaintext)))   ; Keys never stored plaintext
```

Key Hierarchy

Key Types

```scheme
(define key-hierarchy
  '(master-key                    ; Root of trust
    └── vault-key                 ; Per-vault encryption
        ├── content-key           ; Object encryption
        ├── metadata-key          ; Soup encryption
        ├── index-key             ; Index encryption
        └── audit-key))           ; Audit log encryption

(define (derive-key parent purpose)
  "Derive child key from parent"
  (hkdf-sha256 parent
               salt: (purpose->salt purpose)
               info: (string->utf8 (symbol->string purpose))
               length: 32))
```

Master Key

```scheme
;; Master key from key ceremony (RFC-022)
(define (initialize-master-key shares threshold)
  "Reconstruct master key from shares"
  (let ((master (shamir-combine shares threshold)))
    ;; Derive vault key
    (let ((vault-key (derive-key master 'vault)))
      ;; Store encrypted vault key
      (store-encrypted-vault-key vault-key master)
      ;; Clear master from memory
      (secure-clear! master)
      vault-key)))
```

Key Derivation

```
(define (derive-content-key vault-key object-hash)
  "Derive unique key for each object"
  (hkdf-sha256 vault-key
               salt: (string->utf8 object-hash)
               info: #u8(99 111 110 116 101 110 116)  ; "content"
               length: 32))

(define (derive-chunk-key content-key chunk-index)
  "Derive key for each chunk within object"
  (hkdf-sha256 content-key
               salt: (integer->bytevector chunk-index)
               info: #u8(99 104 117 110 107)  ; "chunk"
               length: 32))
```

Object Encryption

Age Format

```
;; Use age for object encryption (RFC-018)
(define (encrypt-object data recipient-keys)
  "Encrypt object using age format"
  (age-encrypt data recipient-keys))

(define (decrypt-object encrypted identity)
  "Decrypt object using age format"
  (age-decrypt encrypted identity))
```

Symmetric Encryption

```
;; For internal encryption with derived keys
(define (encrypt-symmetric data key)
  "Encrypt with ChaCha20-Poly1305"
  (let ((nonce (generate-nonce 12)))
    (let ((ciphertext (chacha20-poly1305-encrypt data key nonce)))
      (bytevector-append nonce ciphertext))))

(define (decrypt-symmetric encrypted key)
  "Decrypt ChaCha20-Poly1305"
  (let ((nonce (subbytevector encrypted 0 12))
        (ciphertext (subbytevector encrypted 12)))
    (chacha20-poly1305-decrypt ciphertext key nonce)))
```

Streaming Encryption

```
(define (encrypt-stream input-port output-port key)
  "Stream encryption for large objects"
  (let ((chunk-size 65536)
        (chunk-index 0))
    (let loop ()
      (let ((chunk (read-bytevector chunk-size input-port)))
        (unless (eof-object? chunk)
          (let ((chunk-key (derive-chunk-key key chunk-index)))
            (write-bytevector (encrypt-symmetric chunk chunk-key) output-port)
            (secure-clear! chunk-key))
          (set! chunk-index (+ chunk-index 1))
          (loop))))))
```

Storage Integration

Encrypted CAS

```
(define (cas-put-encrypted data)
  "Store encrypted object in CAS"
  (let* ((hash (content-hash data))            ; Hash plaintext
         (key (derive-content-key (vault-key) hash))
         (encrypted (encrypt-symmetric data key)))
    (secure-clear! key)
    (storage-put hash encrypted)               ; Store ciphertext
    (soup-put hash encrypted: #t)
    hash))

(define (cas-get-decrypted hash)
  "Retrieve and decrypt object from CAS"
  (let* ((encrypted (storage-get hash))
         (key (derive-content-key (vault-key) hash))
         (data (decrypt-symmetric encrypted key)))
    (secure-clear! key)
    ;; Verify hash
    (unless (equal? hash (content-hash data))
      (error "Hash verification failed after decryption"))
    data))
```

Encrypted Soup

```
(define (soup-put-encrypted hash metadata)
  "Store encrypted soup entry"
  (let* ((serialized (serialize metadata))
```

```
         (key (derive-key (vault-key) 'metadata))
         (encrypted (encrypt-symmetric serialized key)))
    (secure-clear! key)
    (soup-storage-put hash encrypted)))

(define (soup-get-decrypted hash)
  "Retrieve and decrypt soup entry"
  (let* ((encrypted (soup-storage-get hash))
         (key (derive-key (vault-key) 'metadata))
         (serialized (decrypt-symmetric encrypted key)))
    (secure-clear! key)
    (deserialize serialized)))
```

Encrypted Indexes

```
;; Searchable encryption for indexes
(define (index-put-encrypted term hash)
  "Add encrypted index entry"
  (let* ((key (derive-key (vault-key) 'index))
         (encrypted-term (deterministic-encrypt term key))
         (encrypted-hash (encrypt-symmetric hash key)))
    (secure-clear! key)
    (index-storage-put encrypted-term encrypted-hash)))

;; Deterministic encryption enables equality search
(define (deterministic-encrypt data key)
  "Deterministic encryption for searchable indexes"
  (let ((derived-key (hkdf-sha256 key salt: data length: 32)))
    (siv-encrypt data derived-key)))
```

Key Management

Key Storage

```
;; Keys encrypted with master key
(define (store-vault-key vault-key master-key)
  "Store vault key encrypted with master key"
  (let ((encrypted (encrypt-symmetric vault-key master-key)))
    (write-file (vault-key-path) encrypted)))

;; Keys can also be stored as SPKI-encrypted blobs
(define (store-key-for-principal key principal)
  "Store key encrypted for specific principal"
  (let* ((encrypted (age-encrypt key (list (principal-public-key principal))))
         (hash (cas-put encrypted)))
```

```
    (soup-put hash
      type: 'encrypted-key
      recipient: principal)
    hash))
```

Key Rotation

```
(define (rotate-vault-key)
  "Rotate vault encryption key"
  (let ((old-key (vault-key))
        (new-key (generate-key 32)))

    (audit-append action: 'key-rotation-start)

    ;; Re-encrypt all objects
    (for-each (lambda (hash)
                (let* ((encrypted (storage-get hash))
                       (data (decrypt-symmetric encrypted old-key))
                       (new-encrypted (encrypt-symmetric data new-key)))
                  (storage-put hash new-encrypted)))
              (all-object-hashes))

    ;; Update vault key
    (set-vault-key! new-key)
    (secure-clear! old-key)

    (audit-append action: 'key-rotation-complete)))
```

Key Escrow

```
;; Threshold key escrow for recovery
(define (escrow-vault-key vault-key custodians threshold)
  "Split vault key among custodians"
  (let ((shares (shamir-split vault-key (length custodians) threshold)))
    (for-each (lambda (custodian share)
                (let ((encrypted-share
                       (age-encrypt share (list (custodian-public-key custodian)))))
                  (send-to-custodian custodian encrypted-share)))
              custodians shares)))

(define (recover-vault-key custodian-shares)
  "Recover vault key from custodian shares"
  (let ((decrypted-shares
         (map (lambda (share custodian)
                (age-decrypt share (custodian-identity custodian)))
              custodian-shares custodians)))
```

```
    (shamir-combine decrypted-shares)))
```

Access Control

Key Access

```
;; Keys protected by capabilities
(spki-cert
  (issuer vault-admin)
  (subject data-processor)
  (capability
    (action decrypt)
    (object (type 'content)))
  (validity (not-after "2027-01-01")))

(define (authorized-decrypt? principal hash)
  "Check if principal can decrypt object"
  (has-capability? principal 'decrypt hash))
```

Envelope Encryption

```
;; Object encrypted with DEK, DEK encrypted with KEK
(define (envelope-encrypt data recipients)
  "Envelope encryption for multiple recipients"
  (let* ((dek (generate-key 32))                    ; Data Encryption Key
         (encrypted-data (encrypt-symmetric data dek))
         (encrypted-deks                            ; KEK per recipient
          (map (lambda (recipient)
                 (age-encrypt dek (list recipient)))
               recipients)))
    `(envelope
      (encrypted-data . ,encrypted-data)
      (encrypted-keys . ,encrypted-deks))))

(define (envelope-decrypt envelope identity)
  "Decrypt envelope"
  (let* ((encrypted-deks (assoc-ref envelope 'encrypted-keys))
         (dek (find-and-decrypt-dek encrypted-deks identity))
         (encrypted-data (assoc-ref envelope 'encrypted-data)))
    (decrypt-symmetric encrypted-data dek)))
```

Secure Memory

Key Handling

```
;; Keys must be handled carefully in memory
(define (with-key key-source proc)
  "Use key with automatic cleanup"
  (let ((key (key-source)))
    (dynamic-wind
      (lambda () #t)
      (lambda () (proc key))
      (lambda () (secure-clear! key)))))

(define (secure-clear! bytevector)
  "Securely zero memory"
  (bytevector-fill! bytevector 0)
  ;; Prevent compiler optimization
  (memory-barrier))
```

Memory Protection

```
;; Lock key pages in memory (prevent swap)
(define (allocate-secure-memory size)
  (let ((mem (allocate-memory size)))
    (mlock mem size)
    mem))

(define (free-secure-memory mem size)
  (secure-clear! mem)
  (munlock mem size)
  (free-memory mem))
```

Volume Encryption

LUKS Integration

```
;; Storage volume encryption
(define (setup-encrypted-volume device passphrase)
  "Initialize LUKS encrypted volume"
  (run-command "cryptsetup" "luksFormat" device)
  (run-command "cryptsetup" "luksOpen" device "vault-storage"))

(define (mount-encrypted-volume passphrase)
  "Mount encrypted storage volume"
  (run-command "cryptsetup" "luksOpen"
               (storage-device)
```

```
              "vault-storage"
              stdin: passphrase)
  (run-command "mount" "/dev/mapper/vault-storage" (storage-path)))
```

Hardware Security Modules

```
;; HSM for key protection
(define (hsm-generate-key hsm-slot)
  "Generate key in HSM"
  (pkcs11-generate-key
    session: (hsm-session)
    slot: hsm-slot
    mechanism: 'aes-256
    extractable: #f))

(define (hsm-encrypt data key-handle)
  "Encrypt using HSM-held key"
  (pkcs11-encrypt
    session: (hsm-session)
    key: key-handle
    mechanism: 'aes-gcm
    data: data))
```

Audit

Encryption Audit

```
(define (audit-encryption-operation op hash principal)
  "Log encryption operation"
  (audit-append
    action: op
    hash: hash
    principal: principal
    timestamp: (current-time)
    ;; Never log keys or plaintext!
    ))

;; Audit all key operations
(define (audit-key-operation op key-id principal)
  (audit-append
    action: op
    key-id: key-id
    principal: principal
    timestamp: (current-time)))
```

Compliance Reporting

```
(define (encryption-compliance-report)
  "Generate encryption compliance report"
  `((total-objects . ,(count-objects))
    (encrypted-objects . ,(count-encrypted-objects))
    (encryption-coverage . ,(/ (count-encrypted-objects) (count-objects)))
    (key-rotation-date . ,(last-key-rotation))
    (algorithm . "ChaCha20-Poly1305")
    (key-length . 256)))
```

Performance

Encryption Overhead

```
;; Benchmark encryption overhead
(define (benchmark-encryption size iterations)
  (let ((data (generate-random-bytes size))
        (key (generate-key 32)))
    (time-iterations iterations
      (encrypt-symmetric data key))))
```

Hardware Acceleration

```
;; Use AES-NI when available
(define (select-cipher)
  (if (cpu-supports-aes-ni?)
      'aes-256-gcm
      'chacha20-poly1305))
```

Security Considerations

Key Compromise

```
;; If key is compromised
(define (handle-key-compromise compromised-key-id)
  "Emergency response to key compromise"
  ;; Immediate key rotation
  (rotate-vault-key)

  ;; Audit trail
  (audit-append action: 'key-compromise
                key-id: compromised-key-id
                priority: 'critical)
```

```
  ;; Notify administrators
  (alert-key-compromise compromised-key-id)

  ;; Re-encrypt with new key
  (re-encrypt-all-objects))
```

Side Channels

```
;; Constant-time operations
(define (constant-time-compare a b)
  "Compare without timing leaks"
  (let ((result 0))
    (do ((i 0 (+ i 1)))
        ((= i (bytevector-length a)) (= result 0))
      (set! result (bitwise-ior result
                    (bitwise-xor (bytevector-u8-ref a i)
                                 (bytevector-u8-ref b i)))))))
```

References

1. age - Modern file encryption
2. LUKS - Linux Unified Key Setup
3. RFC-022: Key Ceremony Protocol
4. RFC-018: Sealed Archive Format

Changelog

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

Implementation Status: Draft **Dependencies:** age, chacha20-poly1305, hkdf **Integration:** Storage layer, key management, audit