RFC-021: Capability Delegation Patterns

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

Abstract

This RFC specifies capability delegation patterns for the Library of Cyberspace: how principals grant, attenuate, and revoke capabilities using SPKI certificates, content-addressed objects, and the soup metadata layer. Capabilities flow through delegation chains with monotonically decreasing authority.

Motivation

Authorization in distributed systems is hard:

- **ACLs don't scale** - Central lists become bottlenecks
- **Identity is fragile** - Names change, keys rotate
- **Ambient authority is dangerous** - "Run as root" is not a security model
- **Revocation is an afterthought** - Usually bolted on badly

SPKI (Simple Public Key Infrastructure) solved this decades ago:

- **Capabilities, not identities** - What you can do, not who you are
- **Local names** - No global namespace required
- **Delegation chains** - Authority flows from source to delegate
- **Threshold signatures** - M-of-N for critical operations

The Library integrates SPKI with content-addressed storage, creating capabilities that are themselves content-addressed and introspectable.

Capability Model

Principals

A principal is anything that can hold authority:

```
;; Key principal - most common
(make-principal
  (type key)
  (algorithm ed25519)
  (public-key #${...32 bytes...}))

;; Hash principal - content-addressed object
```

```
(make-principal
  (type hash)
  (algorithm sha256)
  (digest #${...32 bytes...}))

;; Threshold principal - M-of-N group
(make-principal
  (type threshold)
  (threshold 3)
  (members (key1 key2 key3 key4 key5)))

;; Name principal - local binding
(make-principal
  (type name)
  (issuer parent-key)
  (name "alice"))
```

Capabilities

A capability is a transferable right to perform an action:

```
(capability
  (action read)
  (object (hash sha256 "content-hash...")))

(capability
  (action write)
  (object (tree sha256 "subtree-root...")))

(capability
  (action sign)
  (object (tag "releases/*")))

(capability
  (action delegate)
  (object (capability ...)))  ; Meta-capability
```

Certificates

SPKI certificates bind capabilities to principals:

```
(spki-cert
  (issuer vault-master-key)
  (subject alice-key)
  (capability
    (action read)
    (object (tree sha256 "docs-root...")))
  (validity
    (not-before "2026-01-01")
    (not-after "2027-01-01"))
  (delegation #t))  ; Alice can re-delegate
```

Delegation Chains

Chain Structure

Authority flows through certificate chains:

```
Master Key
    │
    ▯ (cert: full access to vault)
Admin Key
    │
    ▯ (cert: read/write to docs/, can delegate)
Developer Key
    │
    ▯ (cert: read-only to docs/public/, cannot delegate)
Contractor Key
```

Chain Validation

```
(define (validate-chain chain target-capability)
  "Validate delegation chain grants capability"
  (let loop ((certs chain)
             (current-authority 'unlimited))
    (if (null? certs)
        (capability-subset? target-capability current-authority)
        (let ((cert (car certs)))
          (and (verify-signature cert)
               (valid-time? cert)
               (not-revoked? cert)
               (capability-subset? (cert-capability cert) current-authority)
               (if (cert-delegation? cert)
                   (loop (cdr certs) (cert-capability cert))
                   (and (null? (cdr certs))
                        (capability-subset? target-capability
                                           (cert-capability cert))))))))))
```

Monotonic Attenuation

Delegated capabilities can only decrease:

```
;; Alice has read/write to entire vault
(spki-cert
  (issuer master)
  (subject alice)
  (capability (action (read write)) (object vault-root))
  (delegation #t))

;; Alice delegates read-only to Bob (valid - attenuation)
(spki-cert
  (issuer alice)
  (subject bob)
  (capability (action read) (object vault-root))
  (delegation #t))

;; Bob tries to delegate write to Carol (INVALID - amplification)
(spki-cert
  (issuer bob)
  (subject carol)
  (capability (action write) (object vault-root))  ; REJECTED
  (delegation #f))

;; Bob delegates read to subtree only (valid - further attenuation)
(spki-cert
  (issuer bob)
  (subject carol)
  (capability (action read) (object docs-subtree))
  (delegation #f))
```

Delegation Patterns

Pattern 1: Direct Delegation

Simplest form - one hop:

```
(define (delegate-direct issuer-key subject-key capability
                         #!key validity can-delegate)
  (sign-cert issuer-key
    `(spki-cert
      (issuer ,(key->principal issuer-key))
      (subject ,(key->principal subject-key))
      (capability ,capability)
      (validity ,validity)
      (delegation ,can-delegate))))
```

Pattern 2: Role-Based Delegation

Delegate to roles, bind keys to roles:

```
;; Define role
(spki-cert
  (issuer vault-admin)
  (subject (name vault-admin "reviewer"))
  (capability (action read) (object (tag "pending/*")))
  (delegation #f))

;; Bind key to role
(spki-cert
  (issuer vault-admin)
  (subject alice-key)
  (capability (name vault-admin "reviewer"))
  (validity (not-after "2026-06-01")))

;; Alice now has reviewer capability via role binding
```

Pattern 3: Threshold Delegation

Require multiple parties:

```
;; Critical operation requires 3-of-5
(spki-cert
  (issuer root-key)
  (subject (threshold 3 (key1 key2 key3 key4 key5)))
  (capability (action delete) (object vault-root))
  (delegation #f))

;; Exercise requires gathering signatures
(define (threshold-exercise capability signers)
  (let ((signatures (map (lambda (key) (sign key capability)) signers)))
    (if (>= (length signatures) 3)
        (execute-capability capability signatures)
        (error "Insufficient signatures"))))
```

Pattern 4: Time-Bounded Delegation

Temporary access:

```
;; Conference access - 3 days only
(spki-cert
  (issuer organizer-key)
  (subject attendee-key)
  (capability (action read) (object conference-materials))
  (validity
    (not-before "2026-03-15T09:00:00Z")
    (not-after "2026-03-17T18:00:00Z"))
  (delegation #f))
```

Pattern 5: Conditional Delegation

Capability with restrictions:

```
;; Can read, but only from specific IP range
(spki-cert
  (issuer admin-key)
  (subject service-key)
  (capability (action read) (object api-data))
  (condition
    (source-ip "10.0.0.0/8"))
  (delegation #f))

;; Can write, but only objects under 1MB
(spki-cert
  (issuer admin-key)
  (subject uploader-key)
  (capability (action write) (object uploads))
  (condition
    (max-size 1048576))
  (delegation #f))
```

Pattern 6: Proxy Delegation

Delegate through intermediary:

```
;; Alice delegates to proxy service
(spki-cert
  (issuer alice-key)
  (subject proxy-key)
  (capability (action read) (object alice-files))
  (delegation #t)
  (condition (proxy-for alice-key)))

;; Proxy can act on Alice's behalf
;; but chain shows Alice as original authority
```

Content-Addressed Capabilities

Hash as Capability

Knowledge of a content hash is itself a capability (RFC-020):

```
;; Possessing this hash grants read access
(define secret-doc-hash "sha256:7f83b1657ff1fc...")

;; The hash is unguessable (256 bits of entropy)
;; Sharing the hash = sharing the capability
(define (share-capability recipient hash)
  (encrypted-send recipient hash))
```

Capability Certificates for Hashes

Formalize hash-based access:

```
(spki-cert
  (issuer vault-key)
  (subject reader-key)
  (capability
    (action read)
    (object (hash sha256 "specific-doc...")))
  (validity (not-after "2027-01-01")))
```

Tree Capabilities

Grant access to Merkle subtree:

```
(spki-cert
  (issuer vault-key)
  (subject team-key)
  (capability
    (action read)
    (object (tree sha256 "project-root..."))
    (propagate #t))  ; Includes all referenced objects
  (delegation #t))
```

Sealed Capabilities

Encrypt capability for specific recipient:

```
(define (seal-capability cap recipient-pubkey)
  "Encrypt capability so only recipient can use it"
  (let* ((serialized (serialize cap))
         (encrypted (age-encrypt serialized recipient-pubkey)))
    (cas-put encrypted)))

(define (unseal-capability sealed-hash identity)
  "Decrypt and exercise capability"
  (let* ((encrypted (cas-get sealed-hash))
         (decrypted (age-decrypt encrypted identity))
         (cap (deserialize decrypted)))
    cap))
```

Revocation

Revocation Lists

```
(spki-crl
  (issuer vault-admin)
  (revoked
    ((cert-hash "sha256:revoked1...")
     (reason key-compromise)
     (revoked-at 1767700000))
    ((cert-hash "sha256:revoked2...")
     (reason superseded)
     (revoked-at 1767700100))))
```

Online Revocation Check

```
(define (check-revocation cert)
  "Check if certificate is revoked"
  (let* ((cert-hash (sha256 (serialize cert)))
         (issuer (cert-issuer cert))
         (crl (fetch-crl issuer)))
    (not (member cert-hash (crl-revoked-hashes crl)))))
```

Tombstone Revocation

Using CAS tombstones (RFC-020):

```
(define (revoke-capability-tombstone cert-hash reason)
  "Revoke by tombstoning the certificate"
  (cas-tombstone cert-hash
    reason: reason
    actor: (current-principal)))
```

Short-Lived Certificates

Avoid revocation by using short validity:

```
;; 1-hour certificate, no revocation needed
(spki-cert
  (issuer service-key)
  (subject session-key)
  (capability (action api-access))
  (validity
    (not-before ,(current-time))
    (not-after ,(+ (current-time) 3600)))
  (delegation #f))
```

Soup Integration

Certificates in the Soup

All certificates are soup objects:

```
(soup-object
  (name "cert/alice-read-docs")
  (type certificate)
  (size "412B")
  (crypto (ed25519 sha256 "cert-hash..."))
  (issuer "vault-admin")
  (subject "alice")
  (capability "read docs/*")
  (expires "2027-01-01"))
```

Querying Capabilities

```
;; Find all certificates for a subject
(soup-query type: 'certificate subject: alice-key)

;; Find all certificates granting write access
(soup-query type: 'certificate capability: 'write)

;; Find expiring certificates
(soup-query type: 'certificate
            expires-before: (+ (current-time) (* 7 24 3600)))

;; Find certificates from specific issuer
(soup-query type: 'certificate issuer: vault-admin)
```

Capability Introspection

```
;; What can Alice do?
(define (principal-capabilities principal)
  (let ((certs (soup-query type: 'certificate subject: principal)))
    (map cert-capability certs)))

;; Who can read this object?
(define (object-readers hash)
  (let ((certs (soup-query type: 'certificate
                           capability: `(read ,hash))))
    (map cert-subject certs)))
```

```scheme
;; Visualize delegation graph
(define (delegation-graph root-principal)
  (let ((certs (soup-query type: 'certificate issuer: root-principal)))
    (map (lambda (cert)
           (cons (cert-subject cert)
                 (delegation-graph (cert-subject cert))))
         certs)))
```

Authorization Decisions

Simple Check

```scheme
(define (authorized? principal action object)
  "Check if principal can perform action on object"
  (let ((chains (find-authorization-chains principal action object)))
    (any valid-chain? chains)))
```

Chain Discovery

```scheme
(define (find-authorization-chains principal action object)
  "Find all certificate chains granting capability"
  (let ((target-cap (capability action object)))
    (let search ((current principal) (chain '()))
      (let ((certs (soup-query type: 'certificate subject: current)))
        (append-map
          (lambda (cert)
            (if (capability-grants? (cert-capability cert) target-cap)
                (list (reverse (cons cert chain)))
                (if (cert-delegation? cert)
                    (search (cert-issuer cert) (cons cert chain))
                    '())))
          certs)))))
```

Cached Authorization

```scheme
(define auth-cache (make-lru-cache 10000))

(define (authorized?/cached principal action object)
  (let ((key (list principal action object)))
    (or (lru-get auth-cache key)
        (let ((result (authorized? principal action object)))
          (lru-put! auth-cache key result)
          result))))
```

Delegation Ceremonies

Key Ceremony

For critical delegations:

```
(define (key-ceremony capability threshold witnesses)
  "Conduct witnessed delegation ceremony"
  (let* ((ceremony-id (generate-ceremony-id))
         (ceremony-record
          `(ceremony
            (id ,ceremony-id)
            (capability ,capability)
            (threshold ,threshold)
            (witnesses ,witnesses)
            (started-at ,(current-time)))))

    ;; Record ceremony start
    (audit-append action: `(ceremony-start ,ceremony-id))

    ;; Gather witness signatures
    (let ((signatures (gather-witness-signatures ceremony-record witnesses)))
      (if (>= (length signatures) threshold)
          (let ((cert (finalize-ceremony ceremony-record signatures)))
            (audit-append action: `(ceremony-complete ,ceremony-id))
            cert)
          (begin
            (audit-append action: `(ceremony-failed ,ceremony-id))
            (error "Insufficient witnesses"))))))
```

Emergency Revocation

```
(define (emergency-revoke cert-hash reason)
  "Emergency revocation with audit trail"
  (let ((revocation
         `(emergency-revocation
           (cert ,cert-hash)
           (reason ,reason)
           (revoked-by ,(current-principal))
           (revoked-at ,(current-time)))))

    ;; Immediate tombstone
    (cas-tombstone cert-hash reason: reason)

    ;; Add to CRL
    (crl-append cert-hash reason)
    ;; Audit with high priority
    (audit-append
      action: `(emergency-revoke ,cert-hash)
      motivation: reason
      priority: 'critical)

    ;; Notify affected parties
    (notify-revocation cert-hash)))
```

Security Considerations

Capability Leakage

```
;; Capabilities can leak through:
;; 1. Logging - don't log capability tokens
;; 2. URLs - don't put capabilities in query strings
;; 3. Errors - don't include capabilities in error messages

(define (safe-log message capability)
  (log (string-append message " [capability:REDACTED]")))
```

Confused Deputy

```
;; Always verify capability matches intended action
(define (execute-action action object capability)
  (unless (capability-grants? capability action object)
    (error "Capability does not grant this action"))
  (perform-action action object))
```

Time-of-Check vs Time-of-Use

```
;; Validate immediately before use, not before
(define (safe-execute capability action)
  (let ((validated (validate-capability capability)))
    ;; No window between check and use
    (atomically
      (unless validated
        (error "Invalid capability"))
      (perform-action action))))
```

Delegation Depth Limits

```
;; Prevent infinite delegation chains
(define max-delegation-depth 10)

(define (validate-chain-depth chain)
  (when (> (length chain) max-delegation-depth)
    (error "Delegation chain too deep")))
```

Implementation Notes

Certificate Storage

```
;; Certificates stored in CAS
(define (store-cert cert)
  (let ((signed (sign-cert cert)))
    (cas-put (serialize signed))))

;; Indexed by issuer and subject for fast lookup
(define cert-by-issuer (make-hash-table))
(define cert-by-subject (make-hash-table))
```

Performance

- Certificate validation is expensive - cache results
- Chain discovery can be slow - index by issuer/subject
- Revocation checks add latency - use short-lived certs when possible

---

References

1. SPKI/SDSI 2.0 - RFC 2693
2. A Logic of Authentication - Burrows, Abadi, Needham
3. Capability Myths Demolished - Miller, Yee, Shapiro
4. RFC-004: SPKI Authorization
5. RFC-020: Content-Addressed Storage
6. RFC-007: Threshold Signature Governance

Changelog

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

Implementation Status: Draft **Dependencies:** spki, vault, cas **Integration:** Vault authorization, soup queries, audit trail