# RFC-022: Key Ceremony Protocol

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

---

## Abstract

This RFC specifies key ceremony protocols for the Library of Cyberspace: the ritualized generation, distribution, and activation of cryptographic keys with witnessed verification. Key ceremonies establish roots of trust through transparent, auditable, multi-party processes.

---

## Motivation

Keys are the foundation of all cryptographic trust:

- **Generate wrong** - Everything built on them fails
- **Store wrong** - Compromise propagates everywhere
- **Distribute wrong** - Man-in-the-middle forever
- **No witnesses** - No one believes you did it right

Key ceremonies solve this through:

- **Witnessed generation** - Multiple parties verify randomness
- **Split custody** - No single point of compromise
- **Documented process** - Reproducible and auditable
- **Air-gapped execution** - Network isolation during critical operations

The Library requires key ceremonies for:

- Vault master keys
- Threshold signing keys
- Root certificates
- Recovery keys

---

## Ceremony Types

### Type 1: Single Key Generation

For individual vault keys:

```
┌─────────────────────────────────────────┐
│          SINGLE KEY CEREMONY            │
├─────────────────────────────────────────┤
│ Participants: 1 generator, 2+ witnesses │
```

```
| Duration: ~30 minutes                    |
| Output: 1 keypair                        |
└──────────────────────────────────────────┘
```

### Type 2: Threshold Key Generation

For M-of-N shared keys (RFC-007, RFC-008):

```
┌──────────────────────────────────────────┐
│        THRESHOLD KEY CEREMONY            │
├──────────────────────────────────────────┤
| Participants: N shareholders, 2+ witnesses |
| Duration: ~2 hours                       |
| Output: N shares, 1 public key           |
└──────────────────────────────────────────┘
```

### Type 3: Root Certificate Ceremony

For establishing certificate authority:

```
┌──────────────────────────────────────────┐
│       ROOT CERTIFICATE CEREMONY          │
├──────────────────────────────────────────┤
| Participants: 3-5 officers, auditor      |
| Duration: ~4 hours                       |
| Output: Root cert, subordinate certs     |
└──────────────────────────────────────────┘
```

### Type 4: Recovery Key Ceremony

For generating disaster recovery keys:

```
┌──────────────────────────────────────────┐
│        RECOVERY KEY CEREMONY             │
├──────────────────────────────────────────┤
| Participants: Board members, escrow      |
| Duration: ~3 hours                       |
| Output: Recovery shares in escrow        |
└──────────────────────────────────────────┘
```

---

## Ceremony Environment

### Physical Requirements

```
SECURE ROOM CHECKLIST:
[ ] No network connectivity
[ ] No wireless devices
```

```
[ ] Faraday shielding (ideal)
[ ] Physical access control
[ ] Video recording (optional)
[ ] Tamper-evident bags for media
[ ] Multiple witnesses present
```

### Equipment

```
(ceremony-equipment
  (primary
    (air-gapped-computer "dedicated ceremony machine")
    (hardware-rng "dice, hardware token, or both")
    (secure-storage "HSM or encrypted USB drives")
    (printer "for paper backup"))
  (verification
    (independent-computer "verify signatures")
    (hash-calculator "verify digests"))
  (documentation
    (camera "photograph key steps")
    (ceremony-log "paper record")
    (witness-signatures "attestation forms")))
```

### Software

```
(ceremony-software
  (os "minimal Linux, verified hash")
  (key-generator "library keygen or gpg")
  (shamir-split "ssss or library implementation")
  (hash-tool "sha256sum")
  (encryption "age for share protection"))
```

---

## Single Key Ceremony

### Pre-Ceremony

```
(define (pre-ceremony-checklist ceremony-id)
  '((verify-room "Secure room confirmed")
    (verify-equipment "All equipment present and functional")
    (verify-participants "All participants identified")
    (verify-software "Software hashes match expected")
    (document-start "Record ceremony start time")
    (collect-devices "All phones/devices collected")))
```

### Entropy Generation

```
(define (generate-entropy witnesses)
  "Generate verifiable randomness"
  (let* (;; Hardware RNG contribution
         (hw-entropy (read-hardware-rng 32))

         ;; Dice rolls - each witness rolls
         (dice-entropy
          (apply bytes-append
                 (map (lambda (w)
                        (let ((rolls (witness-dice-rolls w 20)))
                          (sha256 (rolls->bytes rolls))))
                      witnesses)))

         ;; Timestamp contribution
         (time-entropy (sha256 (timestamp->bytes (current-time))))

         ;; Combine all sources
         (combined (bytes-append hw-entropy dice-entropy time-entropy)))

    ;; Final mixing
    (sha256 combined)))

(define (witness-dice-rolls witness count)
  "Witness rolls dice, records each result"
  (let loop ((i 0) (rolls '()))
    (if (= i count)
        (reverse rolls)
        (let ((roll (read-dice-roll witness)))
          (announce (format "Witness ~a roll ~a: ~a" witness i roll))
          (loop (+ i 1) (cons roll rolls))))))
```

### Key Generation

```
(define (generate-ceremony-key entropy purpose)
  "Generate key from ceremony entropy"
  (let* (;; Derive key material
         (key-material (hkdf entropy
                             salt: (string->bytes purpose)
                             info: "ceremony-key"
                             length: 32))

         ;; Generate Ed25519 keypair
         (keypair (ed25519-keypair-from-seed key-material))
         (public-key (keypair-public keypair))
```

```
        (secret-key (keypair-secret keypair)))

    ;; Announce public key
    (announce "Public key generated:")
    (announce (bytes->hex public-key))

    ;; Return both (secret handled carefully)
    keypair))
```

### Verification

```
(define (verify-key-generation keypair witnesses)
  "Witnesses verify key was generated correctly"

  ;; Sign test message
  (let* ((test-message "ceremony verification test")
         (signature (ed25519-sign (keypair-secret keypair) test-message)))

    ;; Each witness verifies on independent machine
    (for-each
      (lambda (witness)
        (let ((verified (witness-verify witness
                                        (keypair-public keypair)
                                        test-message
                                        signature)))
          (announce (format "Witness ~a verification: ~a"
                            witness
                            (if verified "PASS" "FAIL")))
          (unless verified
            (error "Verification failed - abort ceremony"))))
      witnesses)))
```

### Key Storage

```
(define (store-ceremony-key keypair ceremony-id storage-method)
  "Securely store generated key"
  (case storage-method
    ((hsm)
     ;; Store in hardware security module
     (hsm-store ceremony-id (keypair-secret keypair)))

    ((encrypted-usb)
     ;; Encrypt to USB drive
     (let ((passphrase (generate-passphrase 6)))
       (announce "Passphrase for USB backup:")
       (announce passphrase)
```

```
      (write-encrypted-usb ceremony-id
                           (keypair-secret keypair)
                           passphrase)))

    ((paper)
     ;; Print paper backup
     (print-paper-backup ceremony-id
                         (keypair-secret keypair)
                         (keypair-public keypair)))

    ((shamir)
     ;; Split into shares (see threshold ceremony)
     (shamir-split-key keypair ceremony-id))))
```

### Ceremony Record

```
(define (create-ceremony-record ceremony-id keypair witnesses)
  "Create signed ceremony record"
  (let ((record
         `(key-ceremony
           (id ,ceremony-id)
           (type single-key)
           (timestamp ,(current-time))
           (public-key ,(bytes->hex (keypair-public keypair)))
           (algorithm ed25519)
           (witnesses
            ,(map (lambda (w)
                    `(witness
                      (name ,(witness-name w))
                      (public-key ,(witness-pubkey w))
                      (attestation "I witnessed this key ceremony")))
                  witnesses))
           (entropy-sources (hardware-rng dice timestamps))
           (storage-method ,(storage-method)))))

    ;; Sign by ceremony officer
    (let ((signed (sign-ceremony-record record)))
      ;; Counter-sign by witnesses
      (for-each
        (lambda (w)
          (witness-countersign signed w))
        witnesses)

      ;; Store in audit trail
      (audit-append
        action: `(key-ceremony ,ceremony-id)
```

```
        motivation: "Key generation ceremony completed")

      signed)))
```

---

## Threshold Key Ceremony

### Share Distribution

```
(define (threshold-ceremony n k ceremony-id)
  "Generate threshold key with N shares, K required"

  ;; Generate master entropy (same as single key)
  (let* ((entropy (generate-entropy witnesses))
         (keypair (generate-ceremony-key entropy ceremony-id))

         ;; Split secret into shares
         (shares (shamir-split (keypair-secret keypair) n k)))

    ;; Distribute shares to shareholders
    (for-each
      (lambda (i share shareholder)
        (announce (format "Distributing share ~a to ~a" i shareholder))
        (distribute-share share shareholder ceremony-id))
      (iota n)
      shares
      shareholders)

    ;; Verify reconstruction (with K volunteers)
    (verify-threshold-reconstruction shares k keypair)

    ;; Create ceremony record
    (create-threshold-ceremony-record ceremony-id
                                      keypair
                                      n k
                                      shareholders
                                      witnesses)))
```

### Share Protection

```
(define (distribute-share share shareholder ceremony-id)
  "Encrypt share for specific shareholder"
  (let* (;; Encrypt share to shareholder's key
         (encrypted (age-encrypt share (shareholder-pubkey shareholder)))

         ;; Create share package
```

```
        (package
         `(share-package
           (ceremony-id ,ceremony-id)
           (shareholder ,(shareholder-id shareholder))
           (share-number ,(share-index share))
           (encrypted-share ,(bytes->base64 encrypted))
           (verification-hash ,(sha256 share)))))

  ;; Print or store
  (deliver-share-package package shareholder)))
```

### Reconstruction Test

```
(define (verify-threshold-reconstruction shares k expected-keypair)
  "Verify shares reconstruct correctly"

  ;; Select K volunteers
  (let* ((volunteers (take (shuffle shareholders) k))
         (volunteer-shares (map get-share-from-volunteer volunteers))

         ;; Reconstruct secret
         (reconstructed (shamir-reconstruct volunteer-shares))

         ;; Verify matches
         (matches (equal? reconstructed (keypair-secret expected-keypair))))

    (announce (format "Reconstruction test with ~a shareholders: ~a"
                      k
                      (if matches "PASS" "FAIL")))

    (unless matches
      (error "Reconstruction failed - shares may be corrupted"))

    ;; Securely clear reconstructed secret from memory
    (secure-clear reconstructed)))
```

---

## Root Certificate Ceremony

### Certificate Generation

```
(define (root-certificate-ceremony ceremony-id validity-years)
  "Generate root certificate authority"

  (let* (;; Generate root keypair
         (root-keypair (ceremony-generate-key "root-ca"))
```

```
        ;; Create self-signed root certificate
        (root-cert
         `(spki-cert
           (issuer ,(keypair-public root-keypair))
           (subject ,(keypair-public root-keypair))
           (capability (action sign-certificates))
           (validity
             (not-before ,(current-time))
             (not-after ,(+ (current-time)
                            (* validity-years 365 24 3600))))
           (extensions
             (basic-constraints (ca #t) (path-length 2))
             (key-usage (cert-sign crl-sign)))))

        ;; Sign root certificate
        (signed-root (sign-cert root-keypair root-cert)))

    ;; Witnesses verify root certificate
    (verify-root-certificate signed-root witnesses)

    ;; Generate subordinate CA certificates
    (let ((subordinates (generate-subordinate-certs root-keypair)))

      ;; Create ceremony record
      (create-root-ceremony-record ceremony-id
                                   signed-root
                                   subordinates
                                   witnesses))))
```

### Trust Anchor Publication

```
(define (publish-trust-anchor root-cert ceremony-record)
  "Publish root certificate as trust anchor"

  ;; Multiple publication channels
  (publish-to-vault root-cert ceremony-record)
  (publish-to-website root-cert ceremony-record)
  (publish-to-transparency-log root-cert ceremony-record)

  ;; Generate human-readable fingerprint
  (let ((fingerprint (cert-fingerprint root-cert)))
    (announce "Root certificate fingerprint:")
    (announce (fingerprint->words fingerprint))
    (announce (fingerprint->hex fingerprint))))
```

## Recovery Key Ceremony

### Escrow Protocol

```
(define (recovery-key-ceremony n k escrow-agents)
  "Generate recovery keys with escrow"

  (let* (;; Generate recovery keypair
         (recovery-keypair (ceremony-generate-key "recovery"))

         ;; Split into shares
         (shares (shamir-split (keypair-secret recovery-keypair) n k))

         ;; Encrypt each share to escrow agent
         (escrowed-shares
          (map (lambda (share agent)
                 (escrow-share share agent))
               shares
               escrow-agents)))

    ;; Store escrow metadata (not shares)
    (create-escrow-record recovery-keypair escrowed-shares)

    ;; Test recovery (with k agents)
    (test-recovery-process escrowed-shares k recovery-keypair)))

(define (escrow-share share agent)
  "Encrypt share for escrow agent"
  `(escrowed-share
    (agent ,(agent-id agent))
    (encrypted ,(age-encrypt share (agent-pubkey agent)))
    (verification ,(sha256 share))
    (escrow-date ,(current-time))))
```

### Recovery Execution

```
(define (execute-recovery escrowed-shares agents)
  "Execute recovery using escrowed shares"

  ;; Verify quorum present
  (unless (>= (length agents) recovery-threshold)
    (error "Insufficient agents for recovery"))

  ;; Each agent decrypts their share
  (let* ((decrypted-shares
```

```
        (map (lambda (agent escrowed)
               (agent-decrypt-share agent escrowed))
             agents
             (filter-by-agent escrowed-shares agents)))

       ;; Verify share hashes
       (_ (verify-share-hashes decrypted-shares escrowed-shares))

       ;; Reconstruct
       (recovered-secret (shamir-reconstruct decrypted-shares)))

  ;; Audit recovery event
  (audit-append
    action: '(recovery-executed)
    motivation: "Disaster recovery initiated"
    priority: 'critical)

  recovered-secret))
```

---

## Ceremony Audit Trail

### Ceremony Log Format

```
(ceremony-log
  (ceremony-id "KC-2026-001")
  (type root-certificate)
  (date "2026-01-07")
  (location "Secure facility, Building A, Room 101")

  (participants
    (officer "Alice Smith" (role ceremony-officer))
    (officer "Bob Jones" (role key-custodian))
    (witness "Carol White" (role witness))
    (witness "David Brown" (role witness))
    (auditor "Eve Green" (role auditor)))

  (timeline
    (event "09:00" "Room secured, equipment verified")
    (event "09:15" "Participants identified, devices collected")
    (event "09:30" "Software verification complete")
    (event "09:45" "Entropy generation started")
    (event "10:00" "Dice rolls complete, 100 rolls recorded")
    (event "10:15" "Key generation complete")
    (event "10:30" "Share distribution complete")
    (event "10:45" "Reconstruction test PASSED")
```

```
    (event "11:00" "Ceremony concluded"))

  (artifacts
    (public-key "sha256:abc123...")
    (ceremony-record "sha256:def456...")
    (video-hash "sha256:789xyz..."))

  (attestations
    (signed-by "Alice Smith" "sha256:sig1...")
    (signed-by "Bob Jones" "sha256:sig2...")
    (signed-by "Carol White" "sha256:sig3...")
    (signed-by "David Brown" "sha256:sig4...")
    (signed-by "Eve Green" "sha256:sig5...")))
```

### Soup Integration

```
(soup-object
  (name "ceremony/KC-2026-001")
  (type key-ceremony)
  (size "4.2KB")
  (crypto (ed25519 sha256 "ceremony-hash..."))
  (ceremony-type root-certificate)
  (participants 5)
  (date "2026-01-07")
  (status completed)
  (public-key "sha256:abc123..."))
```

---

## Security Considerations

### Entropy Quality

```
;; Multiple entropy sources required
(define (verify-entropy-quality entropy-sources)
  (unless (>= (length entropy-sources) 3)
    (error "Insufficient entropy sources"))
  (unless (member 'hardware-rng entropy-sources)
    (warn "No hardware RNG - ceremony quality reduced"))
  (unless (member 'dice entropy-sources)
    (warn "No dice rolls - verifiability reduced")))
```

### Side Channel Protection

```
;; Constant-time operations during key generation
(define (secure-key-operations)
  '((constant-time-comparison "for all secret comparisons")
```

```
    (memory-clearing "zero secrets after use")
    (no-branching-on-secrets "avoid timing leaks")
    (power-analysis-resistance "for HSM operations")))
```

### Witness Collusion

```
;; Mitigations for witness collusion
(define witness-requirements
  '((minimum-witnesses 2)
    (independent-verification "each witness verifies independently")
    (no-communication "witnesses cannot confer during ceremony")
    (video-recording "optional but recommended")
    (external-auditor "for high-value ceremonies")))
```

### Ceremony Compromise Recovery

```
(define (ceremony-compromise-response ceremony-id)
  "Response if ceremony is compromised"
  (let ((cert (get-ceremony-certificate ceremony-id)))
    ;; Immediate revocation
    (emergency-revoke cert "Ceremony compromise suspected")

    ;; Notify all relying parties
    (broadcast-revocation cert)

    ;; Schedule new ceremony
    (schedule-replacement-ceremony ceremony-id)

    ;; Forensic preservation
    (preserve-ceremony-artifacts ceremony-id)))
```

---

## Implementation

### Ceremony Script

```
(define (run-ceremony type)
  "Main ceremony execution script"
  (let ((ceremony-id (generate-ceremony-id)))
    (display-banner ceremony-id type)

    ;; Pre-ceremony
    (run-checklist (pre-ceremony-checklist ceremony-id))

    ;; Main ceremony
    (case type
```

```
      ((single-key)
       (single-key-ceremony ceremony-id))
      ((threshold)
       (threshold-ceremony (prompt "N?") (prompt "K?") ceremony-id))
      ((root-cert)
       (root-certificate-ceremony ceremony-id (prompt "Validity years?")))
      ((recovery)
       (recovery-key-ceremony (prompt "N?") (prompt "K?")
                              (collect-escrow-agents))))

    ;; Post-ceremony
    (run-checklist (post-ceremony-checklist ceremony-id))

    (announce "Ceremony complete.")
    ceremony-id))
```

### Offline Tool

```
# Ceremony tool - runs air-gapped
$ seal-ceremony --type threshold --shares 5 --threshold 3

Library of Cyberspace - Key Ceremony Tool
=========================================

Ceremony ID: KC-2026-001
Type: Threshold (5-of-3)

Pre-ceremony checklist:
[x] Network disabled
[x] Wireless disabled
[x] Software verified
[x] Witnesses present (3)

Generating entropy...
  Hardware RNG: 32 bytes collected
  Dice rolls: Witness 1, roll 1: _
```

---

## References


1. DNSSEC Root Key Ceremony
2. RFC 2693 - SPKI Certificate Theory
3. RFC-007: Threshold Signature Governance
4. RFC-008: Shamir Secret Sharing
5. RFC-021: Capability Delegation
6. Key Ceremony Best Practices - NIST SP 800-57

---

## Changelog

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

---

**Implementation Status:** Draft **Dependencies:** shamir, spki, age, audit **Integration:** Vault initialization, threshold governance, disaster recovery