RFC-034: Audit Log Protection

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

Abstract

This RFC specifies protection mechanisms for the Library of Cyberspace audit log: how the system prevents resource exhaustion, log flooding, and denial of service attacks against the audit trail while maintaining its integrity as the authoritative record of all vault operations.

Motivation

The audit log is sacred:

- **Evidence** - Legal and forensic record
- **Accountability** - Who did what when
- **Recovery** - Reconstruct state after failure
- **Trust** - Foundation of the security model

But the audit log is also a target:

- **Flooding** - Generate noise to hide malicious activity
- **Exhaustion** - Fill storage to halt operations
- **Evasion** - Overwhelm to prevent logging of real attacks
- **Amplification** - Small action triggers large log entries

The audit log must protect itself while never failing to record.

Threat Model

Attack Vectors

```
(define audit-threats
  '((flooding
     (description "Generate massive log volume")
     (goal "Hide malicious activity in noise")
     (method "Rapid legitimate-looking operations"))

    (exhaustion
     (description "Fill audit storage")
     (goal "Halt vault operations")
     (method "Sustained high-volume logging"))
```

```
    (amplification
     (description "Small input, large log output")
     (goal "Asymmetric resource consumption")
     (method "Operations that log disproportionately"))

    (corruption
     (description "Tamper with log entries")
     (goal "Alter historical record")
     (method "Exploit write access to log storage"))

    (truncation
     (description "Force log rotation/deletion")
     (goal "Remove evidence")
     (method "Fill logs to trigger cleanup"))))
```

Attacker Capabilities

```
;; Assume attacker can:
;; - Authenticate as a valid principal
;; - Perform many legitimate operations
;; - Control timing of operations
;; - Observe system behavior

;; Assume attacker cannot:
;; - Directly write to audit storage
;; - Forge signatures on log entries
;; - Break cryptographic primitives
```

Rate Limiting

Per-Principal Audit Limits

```
(define audit-rate-limits
  `((entries-per-second . 100)
    (bytes-per-second . 102400)      ; 100KB/s
    (entries-per-minute . 1000)
    (bytes-per-minute . 10485760)))  ; 10MB/min

(define principal-audit-buckets (make-hash-table))

(define (get-audit-bucket principal)
  (or (hash-table-ref principal-audit-buckets principal #f)
      (let ((bucket (make-token-bucket
                     (assoc-ref audit-rate-limits 'entries-per-second)
```

```
                    (assoc-ref audit-rate-limits 'entries-per-second))))
        (hash-table-set! principal-audit-buckets principal bucket)
        bucket)))

(define (audit-rate-check principal entry-size)
  "Check if principal can generate audit entry"
  (let ((bucket (get-audit-bucket principal)))
    (bucket 1)))  ; Consume one token
```

Audit Entry Costing

```
;; Different operations have different audit costs
(define audit-costs
  `((read . 1)
    (write . 2)
    (delete . 5)
    (query . 1)
    (admin . 10)
    (key-operation . 50)
    (emergency . 0)))  ; Always allowed

(define (audit-cost action)
  (or (assoc-ref audit-costs action) 1))

(define (consume-audit-budget principal action)
  "Deduct from principal's audit budget"
  (let ((cost (audit-cost action))
        (bucket (get-audit-bucket principal)))
    (if (bucket cost)
        #t
        (begin
          ;; Log the rate limit event (always succeeds)
          (audit-append-privileged
            action: 'audit-rate-limited
            principal: principal
            attempted-action: action)
          #f))))
```

Burst Handling

```
;; Allow short bursts but limit sustained rate
(define (make-audit-limiter)
  (let ((short-term (make-token-bucket 200 100))   ; 100/s, burst 200
        (long-term (make-sliding-window 10000 60))) ; 10k/min
    (lambda (cost)
      (and (short-term cost)
```

```
            (long-term)))))
```

Storage Protection

Reserved Audit Space

```
;; Always reserve space for audit
(define *audit-reserved-space* (* 1024 1024 1024))  ; 1GB minimum

(define (audit-storage-available)
  (let ((total (storage-available))
        (audit-used (audit-storage-used)))
    (max 0 (- total *audit-reserved-space*))))

(define (check-audit-storage entry-size)
  "Ensure space for audit entry"
  (let ((available (audit-storage-available)))
    (when (< available entry-size)
      ;; Emergency: halt non-audit operations
      (enter-audit-protection-mode)
      ;; But always log
      #t)))
```

Audit Protection Mode

```
(define audit-protection-mode? (make-parameter #f))

(define (enter-audit-protection-mode)
  "Emergency mode: only allow audit writes"
  (audit-protection-mode? #t)
  (audit-append-privileged
    action: 'audit-protection-enabled
    reason: 'storage-exhaustion)
  ;; Reject all non-essential operations
  (set-vault-mode! 'audit-only))

(define (exit-audit-protection-mode)
  "Resume normal operations"
  (when (> (audit-storage-available) (* 2 *audit-reserved-space*))
    (audit-protection-mode? #f)
    (set-vault-mode! 'normal)
    (audit-append-privileged
      action: 'audit-protection-disabled)))
```

Tiered Storage

```
;; Audit log tiers
(define audit-tiers
  '((hot . ((retention . 86400)      ; 1 day
            (storage . ssd)
            (compression . none)))
    (warm . ((retention . 604800)    ; 7 days
             (storage . hdd)
             (compression . zstd-fast)))
    (cold . ((retention . 31536000)  ; 1 year
             (storage . archive)
             (compression . zstd-max)))
    (glacier . ((retention . #f)     ; Forever
                (storage . offsite)
                (compression . zstd-max)))))

(define (tier-audit-entry entry age)
  "Move entry to appropriate storage tier"
  (let ((tier (find (lambda (t)
                      (or (not (assoc-ref (cdr t) 'retention))
                          (< age (assoc-ref (cdr t) 'retention))))
                    audit-tiers)))
    (migrate-to-tier entry (car tier))))
```

Entry Size Limits

Maximum Entry Size

```
(define *max-audit-entry-size* 65536)  ; 64KB

(define (validate-audit-entry entry)
  "Validate entry before logging"
  (let ((size (serialized-size entry)))
    (when (> size *max-audit-entry-size*)
      (error 'audit-entry-too-large
             `((size . ,size)
               (limit . ,*max-audit-entry-size*))))))
```

Entry Truncation

```
(define (truncate-audit-entry entry max-size)
  "Truncate oversized entry while preserving critical fields"
  (let ((critical '(action principal timestamp hash signature)))
    (let loop ((entry entry)
```

```
          (fields (entry-fields entry)))
      (if (<= (serialized-size entry) max-size)
          entry
          (let ((removable (find (lambda (f)
                                   (not (member f critical)))
                                 fields)))
            (if removable
                (loop (assoc-remove entry removable)
                      (delete removable fields))
                ;; Can't truncate further, log warning
                (begin
                  (audit-append-privileged
                    action: 'audit-entry-truncation-failed
                    original-size: (serialized-size entry))
                  entry)))))))
```

Payload Summarization

```
(define (summarize-large-payload payload)
  "Summarize large payloads for audit"
  (if (> (bytevector-length payload) 1024)
      `((type . summarized)
        (original-size . ,(bytevector-length payload))
        (hash . ,(content-hash payload))
        (preview . ,(subbytevector payload 0 256)))
      payload))
```

Aggregation

Event Aggregation

```
;; Aggregate similar events to reduce volume
(define aggregation-window 60)  ; seconds
(define pending-aggregations (make-hash-table))

(define (aggregate-audit-event action principal details)
  "Aggregate similar events within window"
  (let* ((key (list action principal))
         (existing (hash-table-ref pending-aggregations key #f)))
    (if existing
        ;; Add to existing aggregation
        (begin
          (set-cdr! (assoc 'count existing)
                    (+ 1 (assoc-ref existing 'count)))
          (set-cdr! (assoc 'last-seen existing)
```

```
                    (current-time)))
          ;; Start new aggregation
          (hash-table-set! pending-aggregations key
            `((action . ,action)
              (principal . ,principal)
              (count . 1)
              (first-seen . ,(current-time))
              (last-seen . ,(current-time))
              (sample . ,details))))))

(define (flush-aggregations)
  "Flush aggregated events to audit log"
  (hash-table-walk pending-aggregations
    (lambda (key agg)
      (when (> (- (current-time) (assoc-ref agg 'first-seen))
               aggregation-window)
        (audit-append-direct
          action: 'aggregated-events
          original-action: (assoc-ref agg 'action)
          principal: (assoc-ref agg 'principal)
          count: (assoc-ref agg 'count)
          window-start: (assoc-ref agg 'first-seen)
          window-end: (assoc-ref agg 'last-seen)
          sample: (assoc-ref agg 'sample))
        (hash-table-delete! pending-aggregations key)))))
```

Sampling

```
;; Sample high-volume events
(define (should-sample? action principal)
  (let ((rate (current-rate action principal)))
    (cond
      ((< rate 10) #t)              ; Always log low-rate
      ((< rate 100) (< (random 1.0) 0.5))  ; 50% sample
      ((< rate 1000) (< (random 1.0) 0.1)) ; 10% sample
      (else (< (random 1.0) 0.01)))))      ; 1% sample

(define (audit-with-sampling action principal details)
  "Log with sampling for high-volume events"
  (if (should-sample? action principal)
      (audit-append action: action
                    principal: principal
                    details: details
                    sampled: #t)
      ;; Still count for rate tracking
      (increment-audit-counter action principal)))
```

Anomaly Detection

Baseline Establishment

```
(define (establish-audit-baseline principal duration)
  "Establish normal audit patterns for principal"
  (let ((events (soup-query type: 'audit-entry
                            principal: principal
                            timestamp: (> (- (current-time) duration)))))
    `((events-per-hour . ,(/ (length events) (/ duration 3600)))
      (common-actions . ,(top-n (map audit-action events) 10))
      (typical-hours . ,(typical-active-hours events))
      (established-at . ,(current-time)))))
```

Anomaly Detection

```
(define (detect-audit-anomaly principal current-rate baseline)
  "Detect anomalous audit patterns"
  (let ((expected (assoc-ref baseline 'events-per-hour))
        (threshold 3.0))  ; 3x normal is anomalous
    (when (> current-rate (* threshold expected))
      (audit-append-privileged
        action: 'audit-anomaly-detected
        principal: principal
        current-rate: current-rate
        expected-rate: expected
        severity: (if (> current-rate (* 10 expected))
                      'critical
                      'warning))
      ;; Potentially throttle
      (when (> current-rate (* 10 expected))
        (throttle-principal principal)))))
```

Flood Detection

```
(define (detect-audit-flood)
  "Detect system-wide audit flooding"
  (let ((current-rate (global-audit-rate))
        (threshold (* 10 (baseline-global-rate))))
    (when (> current-rate threshold)
      ;; Identify top contributors
      (let ((top-principals (top-audit-principals 10)))
        (audit-append-privileged
          action: 'audit-flood-detected
          global-rate: current-rate
```

```
          threshold: threshold
          top-principals: top-principals)
        ;; Throttle top contributors
        (for-each throttle-principal
                  (map car top-principals))))))
```

Privileged Logging

System Events

```
;; Some events bypass rate limits
(define privileged-actions
  '(audit-rate-limited
    audit-protection-enabled
    audit-protection-disabled
    audit-anomaly-detected
    audit-flood-detected
    key-ceremony
    emergency-revoke
    security-alert))

(define (audit-append-privileged . args)
  "Log privileged event (bypasses rate limits)"
  (let ((entry (apply make-audit-entry args)))
    ;; Always log, never rate limit
    (audit-write-direct entry)
    ;; Alert on privileged events
    (when (member (assoc-ref entry 'action) '(security-alert emergency-revoke))
      (send-security-alert entry))))
```

Guaranteed Delivery

```
(define (audit-write-guaranteed entry)
  "Write audit entry with guaranteed delivery"
  ;; Write to multiple locations
  (let ((locations (list (primary-audit-log)
                         (secondary-audit-log)
                         (remote-audit-log))))
    (let ((successes (filter (lambda (loc)
                               (guard (ex (else #f))
                                 (audit-write loc entry)
                                 #t))
                             locations)))
      (when (< (length successes) 2)
        ;; Failed to write to enough locations
```

```
        (emergency-audit-alert entry successes)))))
```

Integrity Protection

Append-Only Enforcement

```
;; Audit log is strictly append-only
(define (audit-write entry)
  "Append entry to audit log"
  (let ((log (current-audit-log)))
    ;; Verify we're at the end
    (unless (at-log-end? log)
      (error 'audit-log-not-at-end))
    ;; Append with sequence number
    (let ((seq (next-sequence-number log)))
      (write-entry log (cons `(sequence . ,seq) entry))
      ;; Sign the entry
      (sign-entry log entry))))
```

Chain Integrity

```
;; Each entry references previous (blockchain-style)
(define (chain-entry entry previous-hash)
  `((previous-hash . ,previous-hash)
    (sequence . ,(+ 1 (previous-sequence)))
    (timestamp . ,(current-time))
    ,@entry
    (hash . ,(compute-entry-hash entry previous-hash))))

(define (verify-audit-chain log)
  "Verify audit log chain integrity"
  (let loop ((entries (audit-entries log))
             (expected-prev #f))
    (if (null? entries)
        #t
        (let ((entry (car entries)))
          (and (or (not expected-prev)
                   (equal? (assoc-ref entry 'previous-hash) expected-prev))
               (loop (cdr entries)
                     (assoc-ref entry 'hash)))))))
```

Recovery

Log Reconstruction

```
(define (reconstruct-audit-log backups)
  "Reconstruct audit log from multiple backups"
  (let ((entries (merge-backup-entries backups)))
    ;; Deduplicate by sequence number
    (let ((deduped (deduplicate-by-sequence entries)))
      ;; Verify chain
      (unless (verify-audit-chain deduped)
        (error 'audit-chain-broken))
      deduped)))
```

Gap Detection

```
(define (detect-audit-gaps log)
  "Find gaps in audit sequence"
  (let loop ((entries (audit-entries log))
             (expected-seq 1)
             (gaps '()))
    (if (null? entries)
        gaps
        (let ((seq (assoc-ref (car entries) 'sequence)))
          (if (= seq expected-seq)
              (loop (cdr entries) (+ seq 1) gaps)
              (loop (cdr entries) (+ seq 1)
                    (cons (list expected-seq (- seq 1)) gaps)))))))
```

References

1. NIST SP 800-92: Guide to Computer Security Log Management
2. RFC-003: Cryptographic Audit Trail
3. RFC-032: Rate Limiting and Quotas
4. RFC-028: Error Handling and Recovery

Changelog

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

Implementation Status: Draft **Dependencies:** audit, storage, monitoring
Integration: All vault operations, security monitoring, forensics