Tiered Security Email Protocol (TSEP)

A Multi-Level Security Framework for Email Communications

Version: 1.0
Date: February 21, 2026
Status: Draft Specification

Abstract

This whitepaper proposes the Tiered Security Email Protocol (TSEP), a backward-compatible extension to SMTP that enables end-to-end encrypted email with graduated security levels. TSEP decouples identity and key management from email infrastructure by delegating authentication to third-party identity providers (banks, enterprise identity systems, consumer authentication apps). The protocol defines three security levels optimized for different use cases: cached-key mode for routine communications, online verification for sensitive messages, and time-bound verification for high-stakes transactions.

Key Innovations:

- Multi-tier security model balancing convenience and protection
- Identity provider-mediated key management
- Offline-capable encrypted email (Level 1)
- Real-time authentication for sensitive content (Levels 2-3)
- Cryptographic non-repudiation and audit trails
- Pragmatic migration path from existing email infrastructure

---

1. Introduction

1.1 Problem Statement

Modern email faces a fundamental tension: it must remain an open, interoperable protocol while protecting increasingly sensitive communications against sophisticated threats including AI-powered phishing, man-in-the-middle attacks, and sender impersonation.

Current solutions have critical limitations:

- **S/MIME**: Complex certificate management, poor mobile support, limited adoption
- **PGP/GPG**: Steep learning curve, web-of-trust complexity, key management burden
- **Proprietary Apps**: Fragmented ecosystems, poor interoperability, vendor lock-in
- **Basic Encryption**: Lacks graduated security levels, authentication, policy enforcement

1.2 Design Goals

1. **Graduated Security**: Different threat models require different security/convenience tradeoffs
2. **Backward Compatibility**: Works within existing SMTP infrastructure
3. **Delegate Complexity**: Identity providers handle authentication and key management
4. **User-Friendly**: Minimal friction for common use cases
5. **Policy-Aware**: Organizational policies enforceable by identity providers
6. **Auditable**: Cryptographic proof of delivery, reading, and authentication
7. **Offline-Capable**: Routine communications don't require online verification

1.3 Scope

This specification defines:

- Three-tier security level architecture
- Message envelope format and encryption standards
- Key exchange and discovery protocols
- Identity provider integration interfaces
- Client and server implementation requirements

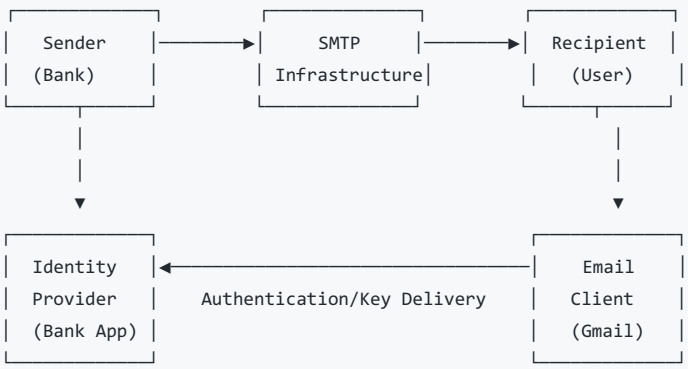
Out of scope:

- Specific identity provider implementations
- Email client UI/UX requirements
- Regulatory compliance mappings (jurisdiction-specific)

---

2. Architecture Overview

2.1 Core Components



2.2 Security Levels

Level	Name	Key Delivery	Authentication	Use Case
1	Cached Key	One-time sync	Initial setup only	Routine communications
2	Online Verification	Per-message	Real-time 2FA	Sensitive actions
3	Time-Bound	Per-message + constraints	Real-time 2FA + policy	High-stakes transactions

2.3 Trust Model

- **Email Infrastructure**: Untrusted (adversarial model)
- **Identity Providers**: Trusted for key custody and authentication
- **End Devices**: Trusted when properly attested
- **Sender Identity**: Cryptographically verified via signatures
- **Message Integrity**: Protected by authenticated encryption

---

3. Security Level Specifications

3.1 Level 1: Cached Key Mode

Purpose: Protect routine communications with minimal friction

Initialization Flow:

```
1. User installs identity provider app (e.g., bank app)
2. App generates user keypair (X25519 for encryption)
3. App establishes secure channel with email client via OAuth 2.0
4. Public keys exchanged bidirectionally
5. User's encrypted private key cached in email client
6. Association stored in identity provider's registry
```

Message Flow:

```
Sender:
1. Retrieve recipient public key from registry
2. Encrypt message body: AES-256-GCM with X25519 key exchange
3. Sign envelope with sender's Ed25519 private key
4. Send via standard SMTP

Recipient:
1. Receive encrypted message
2. Retrieve cached private key from local keystore
3. Verify sender signature
4. Decrypt message
5. Display (works offline)
```

Security Properties:

- ✅ End-to-end encryption (email provider cannot read)
- ✅ Sender authentication (signature verification)
- ✅ Offline reading capability
- ✅ Fast, transparent UX
- ⚠ Vulnerable to device compromise
- ⚠ No per-message authentication

Key Storage: Private key encrypted at rest using device keychain (iOS Keychain, Android Keystore, Windows DPAPI)

3.2 Level 2: Online Verification Mode

Purpose: Require real-time authentication for sensitive messages

Message Flow:

```
Sender:
1. Encrypt message (as Level 1)
2. Mark security_level: 2 in envelope
3. Include identity provider auth endpoint
4. Send via SMTP

Recipient:
1. Receive encrypted message
2. Email client displays preview (encrypted state)
3. User taps "Unlock" → triggers auth request
4. Identity provider app prompts authentication (biometric/PIN)
5. Upon approval, provider issues time-limited decryption token
6. Email client fetches ephemeral private key using token
7. Decrypt and display
8. Ephemeral key securely erased after decryption
```

Authentication Protocol:

```
 Email Client → Identity Provider:
POST /api/v1/email/decrypt-auth
{
  "message_id": "msg_abc123",
  "recipient_key_id": "sha256:...",
  "context": {
    "sender": "noreply@bank.com",
    "subject_hash": "sha256:...",
    "timestamp": "2025-11-13T10:30:00Z"
  }
}

Identity Provider Response:
{
  "challenge_id": "chg_xyz789",
  "push_sent": true,
  "expires_in": 120
}

(User approves on phone)

Email Client → Identity Provider:
POST /api/v1/email/decrypt-key
{
  "challenge_id": "chg_xyz789",
  "decryption_token": "tok_....."
}

Identity Provider Response:
{
  "ephemeral_private_key": "...",
  "expires_at": "2025-11-13T10:35:00Z"
}
```

Security Properties:

- ✅ All Level 1 properties
- ✅ Per-message user authentication
- ✅ Device and location policy enforcement
- ✅ Audit trail of all decryption events
- ✅ Instant revocation capability
- ⚠ Requires online connectivity
- ⚠ Slight UX friction (1-2 seconds)

3.3 Level 3: Time-Bound Verification Mode

Purpose: Maximum security for high-stakes communications

Extended Properties:

```
{
  "security_policy": {
    "expires_at": "2025-11-13T10:45:00Z",
    "max_decrypt_count": 1,
    "read_receipt_required": true,
    "geofence": {
      "allowed_countries": ["US", "CA"],
      "blocked_regions": []
    },
    "device_requirements": {
      "min_trust_score": 0.9,
      "attestation_required": true
    }
  }
}
```

Message Flow:

```
1. All Level 2 authentication steps
2. Additional validation:
   - Time window check (message expiry)
   - Geolocation verification
   - Device trust attestation
   - Risk scoring
3. Ephemeral key with stricter constraints:
   - Max lifetime: 60 seconds
   - Single-use only
   - Requires attestation signature
4. Decryption in secure view:
   - Screenshot prevention (platform-dependent)
   - Copy/paste disabled
   - Auto-destruct after viewing
5. Cryptographic read receipt generated and sent
```

Read Receipt Format:

```
{
  "message_id": "msg_abc123",
  "recipient": "user@example.com",
  "read_at": "2025-11-13T10:32:15Z",
  "content_hash": "sha256:...",
  "signature": "Ed25519:...",
  "authentication_proof": {
    "method": "biometric",
    "device_id": "dev_...",
    "location": "US-CA",
    "trust_score": 0.95
  }
}
```

Security Properties:

- ✅ All Level 2 properties
- ✅ Time-limited message exposure
- ✅ Single-view enforcement (configurable)
- ✅ Geographic restrictions
- ✅ Cryptographic proof of reading
- ✅ Sender can revoke before opening
- ✅ Non-repudiation for compliance

---

4. Message Envelope Specification

4.1 Envelope Structure (JSON)

```
{
  "tsep_version": "1.0",
  "message_id": "msg_550e8400-e29b-41d4-a716-446655440000",
  "security_level": 2,

  "headers": {
    "from": "noreply@bank.com",
    "to": "user@example.com",
    "subject": "Important Account Update",
    "date": "2025-11-13T10:30:00Z",
    "message_type": "account_notification"
  },

  "encryption": {
    "algorithm": "X25519-ChaCha20-Poly1305",
    "recipient_key_id": "sha256:abc123...",
    "ephemeral_public_key": "base64:...",
    "nonce": "base64:...",
    "encrypted_body": "base64:...",
    "encrypted_attachments": [
      {
        "filename": "statement.pdf",
        "content_type": "application/pdf",
        "encrypted_data": "base64:..."
      }
    ]
  },

  "signature": {
    "algorithm": "Ed25519",
    "sender_key_id": "sha256:def456...",
    "signed_data": "base64:...",
    "timestamp": "2025-11-13T10:30:00Z"
  },

  "security_policy": {
    "requires_online_auth": true,
    "auth_provider": {
      "name": "Example Bank",
      "app_id": "com.examplebank.app",
      "auth_endpoint": "https://auth.examplebank.com/api/v1/email",
      "public_key_id": "sha256:..."
    },

    "constraints": {
      "expires_at": "2025-11-13T11:30:00Z",
      "max_decrypt_count": 1,
      "geofence": {
        "type": "country_whitelist",
        "countries": ["US", "CA"]
      },
      "device_attestation_required": true
    },

    "read_receipt": {
      "required": true,
      "endpoint": "https://receipts.examplebank.com/api/v1/confirm"
    }
  }
}
```

4.2 MIME Type

```
Content-Type: application/vnd.tsep.encrypted+json; version=1.0
```

4.3 Email Transport

The TSEP envelope is encoded as base64 and embedded in a standard email body:

```
From: noreply@bank.com
To: user@example.com
Subject: Secure Message (TSEP)
Content-Type: multipart/mixed; boundary="tsep_boundary"

--tsep_boundary
Content-Type: text/plain; charset=utf-8

This message requires a TSEP-compatible email client to view.
Download at: https://tsep.org/clients

--tsep_boundary
Content-Type: application/vnd.tsep.encrypted+json; version=1.0
Content-Transfer-Encoding: base64

[BASE64_ENCODED_TSEP_ENVELOPE]

--tsep_boundary--
```

---

5. Key Management

5.1 Key Types

Key Type	Algorithm	Purpose	Lifetime
User Encryption	X25519	Asymmetric encryption	1 year (rotated)
User Signing	Ed25519	Message authentication	1 year (rotated)
Sender Signing	Ed25519	Prove sender identity	2 years
Ephemeral Session	X25519	Per-message key exchange	Single use

5.2 Key Storage

Level 1 (Cached Keys):

- Private keys stored in platform keychain/keystore
- Encrypted at rest with device hardware key
- Access controlled by OS-level permissions

Levels 2-3 (Identity Provider Custody):

- Private keys stored in Hardware Security Modules (HSMs)
- Key shards distributed across multiple secure enclaves
- Requires user authentication to access
- Never transmitted in plaintext

5.3 Key Discovery Protocol

DNS-Based Discovery (Recommended):

```
_tsep._tcp.example.com. 86400 IN TXT (
  "v=TSEP1"
  "k=https://keys.example.com/.well-known/tsep-keys"
  "p=sha256:abc123..."
)
```

HTTPS Key Server:

```
GET https://keys.example.com/.well-known/tsep-keys/user@example.com

Response:
{
  "email": "user@example.com",
  "public_keys": [
    {
      "key_id": "sha256:abc123...",
      "algorithm": "X25519",
      "public_key": "base64:...",
      "created_at": "2025-01-15T00:00:00Z",
      "expires_at": "2026-01-15T00:00:00Z",
      "status": "active"
    }
  ],
  "identity_provider": {
    "name": "Example Bank",
    "auth_endpoint": "https://auth.examplebank.com",
    "public_key": "base64:..."
  },
  "security_levels_supported": [1, 2, 3]
}
```

5.4 Key Rotation

Rotation Schedule:

- User keys: Annual rotation
- Sender keys: Biennial rotation
- Ephemeral keys: Single use, immediate destruction

Rotation Protocol:

```
1. Generate new keypair (user or provider)
2. Publish new public key to discovery service
3. Transition period: Accept both old and new keys (30 days)
4. Update all identity provider associations
5. Revoke old key after transition period
6. Archive old key for decrypting historical messages
```

---

6. Identity Provider Integration

6.1 Provider Responsibilities

1. **Key Custody**: Secure generation and storage of user private keys
2. **Authentication**: Verify user identity before key release
3. **Policy Enforcement**: Geographic, device, and risk-based controls
4. **Audit Logging**: Immutable record of all key access events
5. **Key Rotation**: Automated key lifecycle management
6. **Recovery**: Account recovery mechanisms
7. **Interoperability**: Standard API compliance

6.2 Authentication API Specification

Endpoint: `POST /api/v1/email/auth`

Request:

```
{
  "message_id": "msg_...",
  "recipient_email": "user@example.com",
  "recipient_key_id": "sha256:...",
  "security_level": 2,
  "context": {
    "sender": "noreply@bank.com",
    "subject_hash": "sha256:...",
    "timestamp": "2025-11-13T10:30:00Z"
  },
  "client_attestation": {
    "platform": "iOS",
    "attestation_data": "base64:...",
    "device_id": "dev_..."
  }
}
```

Response (Challenge):

```
{
  "challenge_id": "chg_...",
  "challenge_type": "push_notification",
  "expires_in": 120,
  "status": "pending"
}
```

Polling Endpoint: `GET /api/v1/email/auth/{challenge_id}`

Response (Approved):

```
{
  "status": "approved",
  "decryption_token": "tok_...",
  "expires_at": "2025-11-13T10:35:00Z",
  "authentication_proof": {
    "method": "biometric",
    "timestamp": "2025-11-13T10:30:15Z",
    "device_trusted": true,
    "risk_score": 0.1
  }
}
```

6.3 Key Delivery API

Endpoint: `POST /api/v1/email/decrypt-key`

Request:

```
{
  "decryption_token": "tok_...",
  "message_id": "msg_...",
  "recipient_key_id": "sha256:..."
}
```

Response:

```
{
  "ephemeral_private_key": "base64:...",
  "expires_at": "2025-11-13T10:31:00Z",
  "single_use": true,
  "key_algorithm": "X25519"
}
```

6.4 Provider Certification

Identity providers must undergo certification to participate in TSEP ecosystem:

Security Requirements:

- SOC 2 Type II compliance
- Annual third-party penetration testing
- HSM or secure enclave for key storage
- Incident response plan and disclosure policy
- Insurance coverage for breaches

Technical Requirements:

- API uptime: 99.9% SLA
- Authentication latency: < 2 seconds (p95)
- Key delivery latency: < 500ms (p95)
- Support for WebAuthn/FIDO2

Audit Requirements:

- Immutable audit logs (5 year retention)
- Real-time monitoring and alerting
- Quarterly security audits
- Transparent incident reporting

---

7. Client Implementation Guidelines

7.1 Email Client Requirements

MUST Support:

- TSEP envelope parsing and validation
- X25519 key exchange and ChaCha20-Poly1305 encryption
- Ed25519 signature verification
- Secure key storage in platform keychain
- Identity provider OAuth integration
- Deep linking to identity provider apps

SHOULD Support:

- Offline decryption (Level 1 caching)
- Read receipt generation
- Key rotation notifications
- Multiple identity provider accounts

MAY Support:

- Geolocation-based policy enforcement
- Screenshot prevention (platform-dependent)
- Secure enclaves for sensitive operations

7.2 Initial Setup Flow

```
1. User installs TSEP-compatible email client
2. User installs identity provider app (e.g., bank app)
3. Identity provider app offers "Enable Secure Email"
4. User consents → OAuth flow initiated
5. Email client receives OAuth token
6. Secure channel established (TLS 1.3 + certificate pinning)
7. Identity provider generates user keypair
8. Public keys exchanged bidirectionally
9. Private key (encrypted) provisioned to email client (Level 1)
10. Association stored in both systems
11. Setup complete → User notified
```

7.3 Message Handling Logic

```
def handle_incoming_email(email):
    # Check if TSEP envelope present
    tsep_envelope = extract_tsep_envelope(email)

    if not tsep_envelope:
        # Regular email, handle normally
        return display_email(email)

    # Parse and validate envelope
```

```
    # Parse and validate envelope
    envelope = parse_envelope(tsep_envelope)
    validate_envelope_schema(envelope)

    # Verify sender signature
    sender_public_key = fetch_sender_key(envelope.signature.sender_key_id)
    if not verify_signature(envelope, sender_public_key):
        raise SecurityError("Invalid sender signature")

    # Route to appropriate security level handler
    level = envelope.security_level

    if level == 1:
        return handle_level1_message(envelope)
    elif level == 2:
        return handle_level2_message(envelope)
    elif level == 3:
        return handle_level3_message(envelope)
    else:
        raise ValueError(f"Unknown security level: {level}")

def handle_level1_message(envelope):
    # Retrieve cached private key
    private_key = keystore.get(envelope.encryption.recipient_key_id)

    if not private_key:
        raise KeyError("Private key not found. Please sync with identity provider.")

    # Decrypt
    plaintext = decrypt(
        envelope.encryption.encrypted_body,
        private_key,
        envelope.encryption.ephemeral_public_key,
        envelope.encryption.nonce
    )

    return display_decrypted_message(plaintext, envelope.headers)

def handle_level2_message(envelope):
    # Show preview in encrypted state
    show_secure_preview(envelope.headers, security_level=2)

    # Wait for user to tap "Unlock"
    if not user_requested_unlock():
        return

    # Request authentication from identity provider
    auth_result = request_authentication(
        envelope.security_policy.auth_provider,
        envelope.message_id,
        envelope.headers
    )

    if not auth_result.approved:
        show_error("Authentication denied")
        return

    # Fetch ephemeral private key
    ephemeral_key = fetch_ephemeral_key(
        auth_result.decryption_token,
        envelope.message_id
    )

    # Decrypt
    plaintext = decrypt(
        envelope.encryption.encrypted_body,
```

```
        ephemeral_key,
        envelope.encryption.ephemeral_public_key,
        envelope.encryption.nonce
    )

    # Securely erase ephemeral key
    secure_erase(ephemeral_key)

    # Send read receipt if required
    if envelope.security_policy.read_receipt.required:
        send_read_receipt(envelope, auth_result)

    return display_decrypted_message(plaintext, envelope.headers)

def handle_level3_message(envelope):
    policy = envelope.security_policy

    # Validate time constraints
    if datetime.now() > policy.constraints.expires_at:
        show_error("Message has expired")
        return

    # Validate geofence
    if policy.constraints.geofence:
        user_location = get_user_location()
        if not is_location_allowed(user_location, policy.constraints.geofence):
            show_error("Message not accessible from current location")
            return

    # All Level 2 authentication steps
    auth_result = request_authentication(
        envelope.security_policy.auth_provider,
        envelope.message_id,
        envelope.headers,
        security_level=3
    )

    if not auth_result.approved:
        show_error("Authentication denied")
        return

    # Device attestation (if required)
    if policy.constraints.device_attestation_required:
        attestation = generate_device_attestation()
        if not verify_attestation(attestation, policy):
            show_error("Device trust insufficient")
            return

    # Fetch ephemeral key
    ephemeral_key = fetch_ephemeral_key(
        auth_result.decryption_token,
        envelope.message_id,
        single_use=True
    )

    # Decrypt
    plaintext = decrypt(
        envelope.encryption.encrypted_body,
        ephemeral_key,
        envelope.encryption.ephemeral_public_key,
        envelope.encryption.nonce
    )

    secure_erase(ephemeral_key)
```

```
    # Display in secure viewer
    if policy.constraints.max_decrypt_count == 1:
        display_secure_view(
            plaintext,
            screenshot_disabled=True,
            copy_disabled=True,
            auto_destruct_after=300  # 5 minutes
        )
        mark_message_consumed(envelope.message_id)
    else:
        display_decrypted_message(plaintext, envelope.headers)

    # Generate and send read receipt
    receipt = generate_read_receipt(
        envelope.message_id,
        plaintext,
        auth_result
    )
    send_read_receipt(receipt, policy.read_receipt.endpoint)
```

---

8. Security Considerations

8.1 Threat Model

In Scope:

- Passive network eavesdropping
- Email provider compromise
- Sender impersonation (phishing)
- Message tampering
- Device theft (mitigated via authentication)
- Compromised email client

Out of Scope:

- Identity provider compromise (assumes trusted)
- Zero-day exploits in cryptographic primitives
- Physical coercion of users
- Quantum computing attacks (post-quantum migration planned)

8.2 Cryptographic Algorithms

Current Recommended:

- **Encryption**: X25519 (ECDH) + ChaCha20-Poly1305 (AEAD)
- **Signing**: Ed25519
- **Hashing**: SHA-256 (SHA3-256 as alternative)
- **Key Derivation**: HKDF-SHA256

Post-Quantum Migration Path:

- Hybrid key exchange: X25519 + Kyber768
- Hybrid signatures: Ed25519 + Dilithium3
- Timeline: Transition by 2030

8.3 Attack Vectors & Mitigations

Compromised Device (Level 1):

- Attack: Attacker extracts cached private key
- Mitigation: Hardware keychain protection, biometric access
- Additional: Switch to Level 2 for sensitive messages

Man-in-the-Middle (Key Exchange):

- Attack: Attacker substitutes public keys
- Mitigation: Certificate transparency for identity providers
- Additional: Out-of-band key fingerprint verification

Replay Attacks:

- Attack: Attacker re-sends authentication token
- Mitigation: Nonces, timestamp validation, single-use tokens
- Additional: Sequence numbers for ordered messages

Phishing (Sender Impersonation):

- Attack: Attacker claims to be legitimate sender
- Mitigation: Cryptographic signatures, verified sender registry
- Additional: Client-side visual trust indicators

Coerced Authentication (Level 2/3):

- Attack: Attacker forces user to authenticate
- Mitigation: Duress PIN support, biometric liveness detection
- Additional: Anomaly detection by identity provider

8.4 Privacy Considerations

Metadata Leakage:

- Email headers (To, From, Subject) visible to email infrastructure
- Mitigation: Subject line hashing, minimal headers

Traffic Analysis:

- Email size and timing patterns may leak information
- Mitigation: Padding, decoy traffic (future consideration)

Identity Provider Tracking:

- Provider sees authentication events and message metadata
- Mitigation: Minimal logging, user data controls, audit transparency

---

9. Operational Considerations

9.1 Performance Targets

Operation	Target Latency	Notes
Level 1 Decryption	< 100ms	Client-side only
Level 2 Authentication	< 2s (p95)	Includes user interaction
Level 3 Authentication	< 3s (p95)	Additional policy checks
Key Discovery	< 500ms	DNS or HTTP(S) lookup
Message Encryption	< 200ms	Sender-side operation

9.2 Scalability

Email Client:

- Key cache: O(1) lookup by key ID
- Signature verification: ~1ms per message (Ed25519)
- Offline capable for Level 1

Identity Provider:

- HSM throughput: > 1000 operations/second
- Authentication concurrency: > 10,000 simultaneous challenges
- Global distribution: Multi-region redundancy

9.3 Monitoring & Alerting

Key Metrics:

- Authentication success rate (target: > 99%)
- Decryption latency (p50, p95, p99)
- Key rotation completion rate
- Failed signature verification rate
- Policy violation frequency

Alerting Triggers:

- Spike in authentication failures (possible attack)
- Increased decryption latency (performance degradation)
- Unusual access patterns (compromised credentials)
- Certificate expiry warnings

---

10. Migration & Adoption

10.1 Deployment Phases

Phase 1: Pilot (Months 1-6)

- Select 2-3 identity providers (banks, enterprise)
- Develop reference implementations
- Deploy to limited user base (10,000 users)
- Gather feedback, iterate on specification

Phase 2: Early Adoption (Months 7-18)

- Onboard additional identity providers
- Email client integrations (Gmail, Outlook, Apple Mail)
- Expand to 1M+ users
- Establish certification program

Phase 3: Mainstream (Months 19-36)

- Default encryption for supported institutions
- Consumer authentication app support (Google, Apple)
- Cross-institution interoperability testing
- 10M+ users

Phase 4: Universal (Years 4+)

- IETF standardization
- Regulatory recognition (eIDAS, NIST)
- Deprecation of unencrypted sensitive email
- 100M+ users

10.2 Backward Compatibility

Graceful Degradation:

- Non-TSEP clients receive plaintext notice + download link
- Senders can specify fallback behavior
- Hybrid mode: TSEP + traditional email in parallel (transition period)

Interoperability:

- TSEP coexists with S/MIME, PGP
- Opportunistic encryption: use TSEP if supported, fallback otherwise
- Bridge services for legacy systems

10.3 Cost Analysis

Identity Provider Costs:

- HSM infrastructure: $50K-$500K initial investment
- API hosting: $5K-$50K/month (depending on scale)
- Certification: $25K-$100K one-time
- Ongoing operations: $10K-$100K/month

Email Client Costs:

- Development: 2-6 engineer-months per client
- Maintenance: 0.5-1 engineer (ongoing)
- Negligible hosting costs (client-side crypto)

User Costs:

- Zero (free feature for users)
- Slight increase in data usage (encrypted envelopes ~20% larger)

---

11. Governance & Standardization

11.1 Standards Bodies

Primary: IETF (Internet Engineering Task Force)

- Working Group: "Secure Email Transport" (SET-WG)
- Charter: Develop TSEP as RFC standard
- Timeline: Draft RFC by Q2 2026, Final RFC by Q4 2026

Supporting Organizations:

- W3C: Web standards for browser-based clients
- IEEE: Hardware security requirements
- NIST: Cryptographic algorithm guidance
- ISO/IEC: International harmonization (ISO 27001 integration)

11.2 Governance Model

TSEP Consortium:

```
Structure:
├── Steering Committee
│   ├── Technical Advisory Board (cryptographers, security experts)
│   ├── Identity Provider Representatives (5 seats)
│   ├── Email Client Representatives (3 seats)
│   └── End User Advocacy (2 seats)
├── Certification Authority
│   ├── Provider Certification Program
│   ├── Security Audit Oversight
│   └── Compliance Monitoring
└── Standards Maintenance
    ├── Protocol Evolution
    ├── Backward Compatibility
    └── Interoperability Testing
```

Decision Making:

- Technical changes: Consensus-based (IETF model)
- Security advisories: Fast-track process (48-hour window)
- Certification criteria: Annual review with public comment period

11.3 Intellectual Property

Patent Policy:

- All participants commit to royalty-free licensing
- W3C RF (Royalty Free) patent policy adopted
- Defensive patent pool for TSEP implementations

Open Source:

- Reference implementations under Apache 2.0 license
- Test vectors and compliance suites publicly available
- Community contributions encouraged

---

12. Compliance & Regulatory Considerations

12.1 Data Protection Regulations

GDPR (EU):

- ✅ Data minimization: Only essential metadata collected
- ✅ Right to erasure: Key revocation + message deletion
- ✅ Data portability: Export keys in standard format
- ✅ Consent: Explicit opt-in for Level 1 caching
- ✅ Processor agreements: Identity providers as processors

CCPA (California):

- ✅ Right to know: Users can query all key access logs
- ✅ Right to delete: Account deletion removes all keys
- ✅ Opt-out: Users can disable cached keys anytime

HIPAA (Healthcare):

- ✅ Encryption in transit and at rest
- ✅ Access controls: Authentication required (Levels 2-3)
- ✅ Audit trails: Comprehensive logging
- ✅ Business Associate Agreements: Identity provider BAAs

12.2 Financial Regulations

PCI-DSS:

- Cardholder data transmitted via Level 2+ only
- Key storage in HSMs (meets PCI requirement 3.5)
- Quarterly vulnerability scanning

SOX (Sarbanes-Oxley):

- Non-repudiation via Level 3 read receipts
- Immutable audit trails

- Document retention policies enforced

AML/KYC:

- Identity provider verification satisfies KYC requirements
- Transaction records linked to authenticated identities

12.3 eSignature Laws

ESIGN Act (US) / eIDAS (EU):

- Level 3 read receipts qualify as "electronic signatures"
- Cryptographic evidence satisfies non-repudiation requirements
- Audit trails meet record-keeping obligations

Qualified Electronic Signatures:

- Identity providers can offer eIDAS-qualified signatures
- Integration with national identity systems (eID)

---

13. Use Case Examples

13.1 Banking & Financial Services

Routine Communications (Level 1):

- Monthly account statements
- Balance notifications
- Product marketing
- General updates

Sensitive Operations (Level 2):

- Password reset confirmations
- Account setting changes
- Small transaction confirmations (<$1,000)
- Two-factor authentication backup codes

High-Stakes Transactions (Level 3):

- Wire transfer approvals (>$10,000)
- Account closure confirmations
- Beneficiary changes
- Legal document delivery (loan agreements)

Implementation Example:

```
{
  "use_case": "wire_transfer_approval",
  "security_level": 3,
  "security_policy": {
    "expires_at": "2025-11-13T12:00:00Z",
    "max_decrypt_count": 1,
    "geofence": {
      "allowed_countries": ["US"]
    },
    "device_requirements": {
      "min_trust_score": 0.95,
      "attestation_required": true
    },
    "read_receipt_required": true,
    "additional_verification": {
      "type": "transaction_code",
      "method": "sms_otp"
    }
  },
  "message_content": {
    "subject": "Approve Wire Transfer: $50,000 to Jane Smith",
    "body_template": "wire_transfer_approval",
    "transaction_details": {
      "amount": 50000,
      "currency": "USD",
      "recipient": "Jane Smith",
      "account_last_four": "1234",
      "initiated_by": "John Doe",
      "timestamp": "2025-11-13T10:30:00Z"
    }
  }
}
```

13.2 Healthcare

Level 1 (Non-PHI):

- Appointment reminders (without clinical details)
- General health tips
- Insurance information
- Facility updates

Level 2 (Basic PHI):

- Lab results (non-critical)
- Prescription ready notifications
- Referral information
- Billing statements

Level 3 (Sensitive PHI):

- Cancer diagnosis results
- HIV/STI test results
- Mental health records
- Genetic testing results

HIPAA Compliance Mapping:

```
TSEP Level → HIPAA Requirement
Level 1    → Encryption in transit (§164.312(e)(1))
Level 2    → Access controls (§164.312(a)(1))
Level 3    → Audit controls (§164.312(b)) + Integrity (§164.312(c)(1))
```

13.3 Legal Services

Level 1:

- Newsletter and firm updates
- CLE seminar invitations
- General legal information

Level 2:

- Client engagement letters
- Non-privileged case updates
- Invoice delivery

Level 3:

- Attorney-client privileged communications
- Settlement offers
- Confidential discovery materials
- Contracts requiring signature

Attorney-Client Privilege Protection:

- Level 3 ensures end-to-end encryption
- Read receipts prove delivery without compromising privilege
- Audit trails satisfy ethical disclosure requirements

13.4 Enterprise Communications

Level 1 (Internal):

- Company announcements
- Team newsletters
- Event invitations
- HR policy updates

Level 2 (Confidential):

- Performance reviews
- Salary information
- Proprietary research data
- Customer information

Level 3 (Trade Secrets):

- M&A communications
- Unreleased product designs
- Financial projections
- Board resolutions

Integration with Enterprise Systems:

```
 Identity Provider: Azure AD / Okta
Policies:
- Level 1: All authenticated employees
- Level 2: Department-specific access
- Level 3: Executive approval required

Device Requirements:
- Corporate-managed devices only
- MDM enrollment verified
- Conditional access policies enforced
```

13.5 Government Communications

Unclassified (Level 1):

- Public notices
- Constituent newsletters
- Event announcements

For Official Use Only (Level 2):

- Inter-agency communications
- Draft policy documents
- Budget information

Classified (Level 3):

- National security information
- Law enforcement sensitive data
- Diplomatic communications

Compliance Requirements:

- FIPS 140-2 Level 3+ HSMs
- FedRAMP High certification
- NIST SP 800-53 controls
- Cross-Domain Solution (CDS) integration

---

14. Implementation Reference

14.1 Reference Libraries

TypeScript/JavaScript:

```
// TSEP Client Library
import { TSEPClient, SecurityLevel } from '@tsep/client';

const client = new TSEPClient({
  identityProvider: {
    name: 'Example Bank',
    authEndpoint: 'https://auth.bank.com/api/v1/email',
    publicKey: 'base64:...'
  },
  keyStore: new PlatformKeyStore() // iOS Keychain, Android Keystore
});

// Initialize with identity provider
await client.initialize();

// Receive encrypted email
const encryptedEmail = await emailClient.receiveEmail();
const tsepMessage = await client.parseMessage(encryptedEmail);

// Decrypt based on security level
if (tsepMessage.securityLevel === SecurityLevel.CACHED) {
  const plaintext = await client.decryptCached(tsepMessage);
  displayMessage(plaintext);
} else if (tsepMessage.securityLevel === SecurityLevel.ONLINE_VERIFICATION) {
  const plaintext = await client.decryptWithAuth(tsepMessage);
  displayMessage(plaintext);
} else if (tsepMessage.securityLevel === SecurityLevel.TIME_BOUND) {
  const plaintext = await client.decryptTimeBound(tsepMessage);
  displaySecureView(plaintext, { autoDestruct: true });
}
```

Python:

```
 # TSEP Server Library (for sending)
from tsep import TSEPSender, SecurityLevel, SecurityPolicy

sender = TSEPSender(
    sender_email='noreply@bank.com',
    signing_key=load_ed25519_key('sender_private.pem')
)

# Send Level 1 message
message = sender.create_message(
    to='user@example.com',
    subject='Monthly Statement',
    body='Your statement is attached.',
    attachments=['statement.pdf'],
    security_level=SecurityLevel.CACHED
)

await sender.send(message)

# Send Level 3 message
policy = SecurityPolicy(
    expires_at=datetime.now() + timedelta(minutes=15),
    max_decrypt_count=1,
    geofence={'allowed_countries': ['US']},
    read_receipt_required=True
)

message = sender.create_message(
    to='user@example.com',
    subject='Wire Transfer Approval Required',
    body=render_template('wire_transfer.html', amount=50000),
    security_level=SecurityLevel.TIME_BOUND,
    security_policy=policy
)

await sender.send(message)
```

14.2 Test Vectors

Test Case 1: Level 1 Encryption/Decryption

```
{
  "description": "Basic Level 1 message encryption",
  "sender_private_key": "ed25519:302e020100300506032b657004220420...",
  "recipient_public_key": "x25519:302a300506032b656e032100...",
  "plaintext": "This is a test message for Level 1 encryption.",
  "expected_envelope": {
    "tsep_version": "1.0",
    "security_level": 1,
    "encryption": {
      "algorithm": "X25519-ChaCha20-Poly1305",
      "ephemeral_public_key": "base64:...",
      "nonce": "base64:...",
      "encrypted_body": "base64:..."
    },
    "signature": {
      "algorithm": "Ed25519",
      "signed_data": "base64:..."
    }
  }
}
```

Test Case 2: Signature Verification

```
{
  "description": "Verify Ed25519 signature on envelope",
  "sender_public_key": "ed25519:302a300506032b6570032100...",
  "envelope_data": "base64:...",
  "signature": "base64:...",
  "expected_valid": true
}
```

Test Case 3: Level 3 Time Expiry

```
{
  "description": "Message should fail to decrypt after expiry",
  "message_id": "msg_test_expiry",
  "expires_at": "2025-11-13T10:00:00Z",
  "attempt_decrypt_at": "2025-11-13T10:01:00Z",
  "expected_error": "MessageExpiredError"
}
```

14.3 Compliance Testing Suite

Security Tests:

- ☐ Key generation uses cryptographically secure RNG
- ☐ Private keys never transmitted unencrypted
- ☐ Ephemeral keys securely erased after use
- ☐ Signature verification fails for tampered messages
- ☐ Replay attacks detected via nonce validation
- ☐ Expired messages cannot be decrypted
- ☐ Geofence restrictions enforced correctly

Interoperability Tests:

- ☐ Messages encrypted by Implementation A decrypt correctly in Implementation B
- ☐ Key discovery works across DNS and HTTPS methods
- ☐ Multiple identity providers can authenticate same user
- ☐ Read receipts format compatible across implementations

Performance Tests:

- ☐ Level 1 decryption completes in < 100ms
- ☐ Level 2 authentication completes in < 2s (p95)
- ☐ Key discovery completes in < 500ms
- ☐ Client handles 1000 cached keys efficiently

---

15. Future Enhancements

15.1 Post-Quantum Cryptography

Timeline: Transition begins 2026, complete by 2030

Migration Strategy:

```
Phase 1 (2026): Hybrid mode
- X25519 + Kyber768 (key exchange)
- Ed25519 + Dilithium3 (signatures)
- Backward compatible with non-PQC clients

Phase 2 (2028): PQC preferred
- New keys generated as PQC by default
- Classical crypto maintained for legacy support

Phase 3 (2030): PQC only
- Classical crypto deprecated
- Security advisories for non-upgraded clients
```

Algorithm Selection:

- **KEM**: Kyber (NIST selected)
- **Signatures**: Dilithium (NIST selected)
- **Fallback**: SPHINCS+ for ultra-conservative use cases

15.2 Decentralized Identity

Self-Sovereign Identity (SSI) Integration:

```
User Controls:
- Generate and manage own keys (no provider custody)
- Verifiable credentials from multiple issuers
- Selective disclosure of attributes

Implementation:
- W3C DID (Decentralized Identifier) standard
- Verifiable Credentials for identity attestation
- Blockchain or distributed ledger for DID resolution

Example DID:
did:tsep:123456789abcdefghi

DID Document:
{
  "@context": "https://www.w3.org/ns/did/v1",
  "id": "did:tsep:123456789abcdefghi",
  "authentication": [{
    "id": "did:tsep:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "publicKeyMultibase": "zH3C2AVv..."
  }],
  "service": [{
    "id": "did:tsep:123456789abcdefghi#email",
    "type": "SecureEmail",
    "serviceEndpoint": "https://example.com/.well-known/tsep"
  }]
}
```

15.3 Advanced Policy Language

Policy-as-Code:

```
{
  "policy_version": "2.0",
  "rules": [
    {
      "condition": "message.amount > 10000 AND message.type == 'wire_transfer'",
      "required_security_level": 3,
      "additional_auth": ["sms_otp", "yubikey"],
      "approval_workflow": {
        "approvers": ["manager", "compliance_officer"],
        "quorum": 2
      }
    },
    {
      "condition": "user.location.country NOT IN ['US', 'CA', 'UK']",
      "action": "deny",
      "reason": "Geographic restriction"
    },
    {
      "condition": "time.hour < 9 OR time.hour > 17",
      "required_security_level": 3,
      "reason": "After-hours transaction"
    }
  ]
}
```

15.4 AI-Powered Security

Anomaly Detection:

- Behavioral biometrics during authentication
- Typing patterns, device usage patterns
- Outlier detection for suspicious access

Risk Scoring:

```
risk_score = calculate_risk({
    'device_trust': 0.95,
    'location_familiar': True,
    'time_of_day_normal': True,
    'velocity_check': 'pass',  # No rapid-fire attempts
    'sender_reputation': 0.98,
    'message_classification': 'routine'
})

if risk_score > 0.7:
    require_step_up_auth()
elif risk_score > 0.9:
    block_and_alert()
```

Phishing Detection:

- Analyze message content for social engineering patterns
- Compare sender claims against verified identity database
- Flag suspicious URLs or attachments

15.5 Cross-Protocol Bridges

Integration with Messaging Apps:

- Signal: TSEP envelope transported over Signal protocol
- WhatsApp: End-to-end encrypted message wrapper
- Telegram: Secret chats with TSEP authentication

Legacy System Support:

- SMTP-to-TSEP gateway for organizations in transition
- Fallback mode with clear security indicators
- Upgrade prompts for recipients

---

16. Security Considerations (Extended)

16.1 Advanced Attack Scenarios

Attack: Coerced Authentication

```
 Scenario: Attacker forces victim to authenticate at gunpoint

Mitigations:
1. Duress PIN: User enters alternate PIN that appears to work
   but triggers silent alarm and sends dummy data

2. Biometric Liveness Detection: Ensure user is willing participant
   - Heart rate variability analysis
   - Facial micro-expressions

3. Time-delay Authentication: High-stakes messages require
   authentication + 1-hour cooling-off period

4. Location Anomaly Detection: Unexpected location triggers
   additional verification or blocks access
```

Attack: Quantum Computer (Future)

```
Scenario: Adversary with quantum computer breaks X25519/Ed25519

Mitigations:
1. Hybrid PQC migration (see Section 15.1)
2. Forward secrecy: Past messages remain secure
3. Crypto-agility: Rapid algorithm rotation capability
4. Sentinel systems: Monitor for quantum computing advances
```

Attack: Supply Chain Compromise

```
Scenario: Malicious code injected into email client or identity provider

Mitigations:
1. Code signing: All updates cryptographically signed
2. Reproducible builds: Independent verification of binaries
3. Client attestation: Runtime integrity verification
4. Vendor transparency: Public security audits, bug bounty programs
```

Attack: Social Engineering of Identity Provider

```
Scenario: Attacker convinces provider employee to release keys

Mitigations:
1. Multi-party authorization: Multiple employees required for sensitive operations
2. Hardware security: Keys in HSMs inaccessible to employees
3. Audit logging: All access attempts logged immutably
4. Anomaly detection: Unusual patterns trigger investigation
5. No-knowledge architecture: Provider cannot extract usable keys
```

16.2 Cryptographic Agility

Algorithm Negotiation:

```
{
  "supported_algorithms": {
    "encryption": [
      "X25519-ChaCha20-Poly1305",  // Primary
      "X25519-AES256-GCM",         // Alternative
      "Kyber768-ChaCha20-Poly1305" // Post-quantum
    ],
    "signing": [
      "Ed25519",      // Primary
      "Dilithium3"    // Post-quantum
    ],
    "hashing": [
      "SHA256",       // Primary
      "SHA3-256"      // Alternative
    ]
  },
  "deprecation_schedule": {
    "RSA-2048": "2026-01-01",  // No longer accepted
    "SHA1": "deprecated"        // Already phased out
  }
}
```

Emergency Algorithm Rotation:

```
Trigger: Critical vulnerability discovered in ChaCha20

Process:
1. Security advisory issued (within 24 hours)
2. All providers update to alternative algorithm (48 hours)
3. Clients auto-update encryption preferences
4. Legacy algorithm blocked after transition period (30 days)
5. Post-mortem and protocol update
```

16.3 Operational Security

Key Ceremony:

```
For high-assurance deployments (government, large financial institutions):

1. Physical Security:
   - Faraday cage environment
   - Video recording of entire ceremony
   - Multiple witnesses from different organizations

2. Key Generation:
   - HSMs in FIPS 140-2 Level 4 mode
   - Entropy from multiple hardware RNGs
   - Deterministic key derivation for backup/audit

3. Key Splitting:
   - Shamir secret sharing (M-of-N threshold)
   - Shards distributed to geographically separate locations
   - Reconstruction requires physical presence + biometrics

4. Documentation:
   - Cryptographic hashes of all generated keys published
   - Certificate transparency log entries
   - Independent auditor sign-off
```

Incident Response Plan:

```
Severity Levels:

P0 (Critical): Key compromise, crypto break, nation-state attack
  - Response Time: 1 hour
  - Actions: Emergency key rotation, service lockdown, public disclosure

P1 (High): Vulnerability allowing unauthorized message access
  - Response Time: 24 hours
  - Actions: Patch deployment, affected user notification, audit

P2 (Medium): Policy bypass, non-critical vulnerability
  - Response Time: 7 days
  - Actions: Scheduled patch, internal review

P3 (Low): Performance issue, minor bug
  - Response Time: 30 days
  - Actions: Include in next release
```

---

17. Economic Model

17.1 Stakeholder Value Proposition

For Users (Email Recipients):

- **Free**: No direct cost for enhanced security

- **Value**: Protection against phishing, privacy, peace of mind
- **Convenience**: Levels 1 and 2 require minimal extra effort

For Senders (Banks, Healthcare, Enterprise):

- **Cost**: $0.01 - $0.05 per Level 2/3 message (authentication overhead)
- **Savings**: Reduced fraud losses ($1B+ annually for financial sector)
- **Compliance**: Avoid regulatory fines (GDPR: up to 4% revenue)
- **Brand**: Enhanced customer trust and satisfaction

For Identity Providers:

- **Revenue Streams**:
  - Per-authentication fees: $0.01 - $0.03 per Level 2/3 message
  - Enterprise licenses: $10K - $500K annually
  - Premium features: Advanced analytics, custom policies
- **Cost**: Infrastructure ($50K-$500K initial, $10K-$100K monthly)
- **Margin**: 60-70% gross margin at scale

For Email Client Vendors:

- **Cost**: Development (2-6 engineer-months), negligible ongoing
- **Value**: Competitive differentiation, retention of enterprise customers
- **Revenue**: Potential premium tier for TSEP features

17.2 Total Cost of Ownership (TCO)

Enterprise Deployment (10,000 employees, 1M messages/month):

Year 1:

- Identity provider integration: $100,000
- Email client upgrades: $50,000
- User training: $25,000
- Per-message costs (50% Level 2+): $5,000/month = $60,000/year
- **Total Year 1**: $235,000

Years 2-5:

- Per-message costs: $60,000/year
- Maintenance: $20,000/year
- **Annual TCO**: $80,000/year

Savings:

- Reduced phishing losses: $500,000/year (avg. enterprise)
- Compliance cost avoidance: $200,000/year
- Productivity gains: $100,000/year (less time verifying legitimacy)
- **Net Savings**: $640,000/year after Year 1

ROI: 272% first year, 800% annually thereafter

17.3 Market Opportunity

Total Addressable Market (TAM):

- Global email users: 4.5 billion
- High-value email segments:
  - Financial services: 500M users
  - Healthcare: 800M users
  - Enterprise: 1.2B users
  - Government: 200M users
- **TAM**: 2.7B users requiring secure email

Serviceable Obtainable Market (SOM):

- 10% adoption by 2030: 270M users
- Average revenue per user (ARPU): $2/year (identity provider fees)
- **Market Size**: $540M annually by 2030

---

18. Conclusion

The Tiered Security Email Protocol (TSEP) addresses a critical gap in email security by providing graduated protection appropriate to message sensitivity while maintaining the openness and ubiquity that makes email valuable. By delegating identity and key management to specialized providers, TSEP avoids the complexity that has hindered previous secure email adoption.

18.1 Key Innovations

1. **Three-tier security model**: Balances convenience (Level 1) with protection (Levels 2-3)
2. **Identity provider delegation**: Leverages existing authentication infrastructure
3. **Backward compatibility**: Works within SMTP, coexists with existing email
4. **Policy enforcement**: Organizations control access without sacrificing usability
5. **Cryptographic assurance**: Non-repudiation, audit trails, verified identities

18.2 Path Forward

Immediate Actions (2025-2026):

- Form IETF working group and draft RFC
- Develop reference implementations (open source)
- Pilot with 2-3 financial institutions
- Establish TSEP Consortium for governance

Near-term Goals (2026-2027):

- 10M+ users on TSEP
- 5+ identity providers certified
- Major email client support (Gmail, Outlook, Apple Mail)
- Regulatory recognition (NIST, eIDAS)

Long-term Vision (2028-2030):

- 100M+ users globally
- Default encryption for sensitive communications
- Post-quantum cryptography migration complete
- TSEP as de facto standard for authenticated email

18.3 Call to Action

For Identity Providers: Join the pilot program to shape the standard and gain early-mover advantage in secure email authentication.

For Email Client Vendors: Integrate TSEP to differentiate your product and meet growing enterprise security demands.

For Standards Bodies: Support TSEP as an IETF RFC and harmonize with national/regional identity frameworks.

For Enterprises: Advocate for TSEP support from your email providers and identity systems to protect your communications.

For Users: Request TSEP from your banks, healthcare providers, and employers to ensure your sensitive messages are truly private.

---

19. References

19.1 Standards & Specifications

[RFC5322] Resnick, P., "Internet Message Format", RFC 5322, October 2008.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.

[RFC8551] Schaad, J., Ramsdell, B., and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification", RFC 8551, April 2019.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, January 2016.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, January 2017.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, May 2015.

19.2 Cryptographic Standards

[NIST-PQC] National Institute of Standards and Technology, "Post-Quantum Cryptography Standardization", 2024.

[FIPS-140-3] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules", FIPS PUB 140-3, March 2019.

[SP800-56A] National Institute of Standards and Technology, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography", NIST SP 800-56A Rev. 3, April 2018.

19.3 Security Research

[Bernstein2006] Bernstein, D.J., "Curve25519: new Diffie-Hellman speed records", PKC 2006.

[Perrin2016] Perrin, T., "The XEdDSA and VXEdDSA Signature Schemes", Signal Foundation, 2016.

[Green2014] Green, M. and Rubin, A., "Secure Email: Breaking the Barriers", Johns Hopkins University, 2014.

19.4 Identity & Authentication

[WebAuthn] W3C, "Web Authentication: An API for accessing Public Key Credentials", W3C Recommendation, March 2019.

[OAuth2.0] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[eIDAS] European Parliament, "Regulation on electronic identification and trust services for electronic transactions in the internal market", 2014.

Appendix A: Cryptographic Implementation Details

A.1 Key Derivation

X25519 Key Exchange:

```
def derive_shared_secret(private_key, peer_public_key):
    """
    Derive shared secret using X25519 ECDH
    """
    shared_secret = x25519(private_key, peer_public_key)
    return shared_secret

def derive_encryption_key(shared_secret, salt, info):
    """
    Derive encryption key using HKDF-SHA256
    """
    return HKDF(
        algorithm=SHA256(),
        length=32,  # 256-bit key
        salt=salt,
        info=info
    ).derive(shared_secret)
```

Message Encryption:

```
def encrypt_message(plaintext, recipient_public_key, sender_private_key):
    """
    Encrypt message using X25519 + ChaCha20-Poly1305
    """
    # Generate ephemeral keypair
    ephemeral_private = generate_x25519_private_key()
    ephemeral_public = x25519_public_key(ephemeral_private)

    # Derive shared secret
    shared_secret = x25519(ephemeral_private, recipient_public_key)

    # Derive encryption key
    salt = os.urandom(32)
    encryption_key = derive_encryption_key(
        shared_secret,
        salt,
        info=b"TSEP-v1-encryption"
    )

    # Encrypt
    nonce = os.urandom(12)  # 96-bit nonce for ChaCha20
    cipher = ChaCha20Poly1305(encryption_key)
    ciphertext = cipher.encrypt(nonce, plaintext, associated_data=None)

    return {
        'ephemeral_public_key': ephemeral_public,
        'salt': salt,
        'nonce': nonce,
        'ciphertext': ciphertext
    }
```

A.2 Signature Generation

Ed25519 Signing:

```
def sign_envelope(envelope_data, sender_private_key):
    """
    Sign TSEP envelope with Ed25519
    """
    #
```

```
# TSEP Protocol - Essential Code Examples

## 1. Initial Setup & Key Exchange

### Client-Side: Email Client Initialization
```javascript
// Email client establishes connection with identity provider
import { TSEPClient } from '@tsep/client';

async function setupSecureEmail() {
  // Initialize TSEP client
  const client = new TSEPClient({
    emailAddress: 'user@example.com',
    keyStore: new PlatformKeyStore(), // iOS Keychain, Android Keystore, etc.
  });

  // OAuth flow to authenticate with identity provider
  const authCode = await client.initiateOAuth({
    provider: 'bank.com',
    scope: ['email:encrypt', 'email:decrypt'],
    redirectUri: 'emailclient://oauth-callback'
  });

  // Complete OAuth and establish secure channel
  const session = await client.completeOAuth(authCode);

  // Generate or retrieve user keypair
  const userKeypair = await client.getOrGenerateKeypair({
    algorithm: 'X25519',
    keyId: `user_${Date.now()}`
  });

  // Exchange keys with identity provider
  await session.exchangeKeys({
    userPublicKey: userKeypair.publicKey,
    requestProviderPublicKey: true
  });

  // For Level 1: Receive and cache encrypted private key
  const encryptedPrivateKey = await session.receivePrivateKey();
  await client.keyStore.save({
    keyId: userKeypair.keyId,
    encryptedPrivateKey: encryptedPrivateKey,
    providerPublicKey: session.providerPublicKey
  });

  console.log('✓ Secure email setup complete');
  return { success: true, keyId: userKeypair.keyId };
}
```

Server-Side: Identity Provider Key Exchange Handler

```
from cryptography.hazmat.primitives.asymmetric import x25519, ed25519
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import os
```
```

```
class IdentityProviderKeyManager:
    def __init__(self, hsm_client):
        self.hsm = hsm_client
        self.key_registry = KeyRegistry()

    async def handle_key_exchange(self, user_id, user_public_key, email_client_info):
        """
        Handle key exchange request from email client
        """
        # Generate user's keypair in HSM
        user_private_key = self.hsm.generate_x25519_key()
        user_public_key_computed = user_private_key.public_key()

        # Store in secure registry
        await self.key_registry.store({
            'user_id': user_id,
            'key_id': f'user_{user_id}_{int(time.time())}',
            'private_key_ref': user_private_key.hsm_reference,  # HSM reference, not actual key
            'public_key': user_public_key_computed.public_bytes_raw(),
            'created_at': datetime.utcnow(),
            'status': 'active'
        })

        # For Level 1 caching: Encrypt private key for email client
        # Derive device-specific encryption key
        device_key = await self.derive_device_key(email_client_info)
        encrypted_private_key = await self.encrypt_key_for_device(
            user_private_key,
            device_key
        )

        # Return provider's public signing key and encrypted user private key
        provider_signing_key = await self.hsm.get_provider_signing_key()

        return {
            'provider_public_key': provider_signing_key.public_key().public_bytes_raw(),
            'user_key_id': key_id,
            'encrypted_private_key': encrypted_private_key,
            'supported_security_levels': [1, 2, 3],
            'auth_endpoints': {
                'level2': 'https://auth.bank.com/api/v1/email/auth',
                'level3': 'https://auth.bank.com/api/v1/email/auth-premium'
            }
        }

    async def derive_device_key(self, email_client_info):
        """
        Derive device-specific key for encrypting cached private key
        """
        # Use device attestation to derive unique key
        device_id = email_client_info['device_id']
        attestation = email_client_info['attestation_data']

        # Verify attestation (platform-specific)
        if not self.verify_attestation(attestation):
            raise SecurityError("Invalid device attestation")

        # Derive key using HKDF
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=os.urandom(32),
            info=f"TSEP-device-key-{device_id}".encode()
        )
        return hkdf.derive(attestation.public_key)
```

2. Message Encryption (Sender)

Encrypting and Sending TSEP Message

```
// Bank's email sending service
import { TSEPSender } from '@tsep/sender';
import { ed25519, x25519 } from '@noble/curves';

class BankEmailService {
  constructor() {
    this.sender = new TSEPSender({
      senderEmail: 'noreply@bank.com',
      signingKey: loadSigningKey(), // Bank's Ed25519 private key
    });
  }

  async sendSecureEmail(params) {
    const {
      recipientEmail,
      subject,
      body,
      attachments = [],
      securityLevel = 1
    } = params;

    // 1. Look up recipient's public key
    const recipientKey = await this.lookupPublicKey(recipientEmail);

    // 2. Generate ephemeral keypair for this message
    const ephemeralPrivate = x25519.utils.randomPrivateKey();
    const ephemeralPublic = x25519.getPublicKey(ephemeralPrivate);

    // 3. Derive shared secret using ECDH
    const sharedSecret = x25519.getSharedSecret(
      ephemeralPrivate,
      recipientKey.publicKey
    );

    // 4. Derive encryption key using HKDF
    const salt = crypto.getRandomValues(new Uint8Array(32));
    const encryptionKey = await this.deriveKey(sharedSecret, salt);

    // 5. Encrypt message body and attachments
    const nonce = crypto.getRandomValues(new Uint8Array(12));
    const ciphertext = await this.encryptChaCha20Poly1305(
      body,
      encryptionKey,
      nonce
    );

    // Encrypt attachments
    const encryptedAttachments = await Promise.all(
      attachments.map(att => this.encryptAttachment(att, encryptionKey))
    );

    // 6. Create TSEP envelope
    const envelope = {
      tsep_version: '1.0',
      message_id: `msg_${crypto.randomUUID()}`,
      security_level: securityLevel,

      headers: {
```

```
    headers: {
      from: 'noreply@bank.com',
      to: recipientEmail,
      subject: subject,
      date: new Date().toISOString(),
      message_type: 'transaction_notification'
    },

    encryption: {
      algorithm: 'X25519-ChaCha20-Poly1305',
      recipient_key_id: recipientKey.keyId,
      ephemeral_public_key: this.toBase64(ephemeralPublic),
      salt: this.toBase64(salt),
      nonce: this.toBase64(nonce),
      encrypted_body: this.toBase64(ciphertext),
      encrypted_attachments: encryptedAttachments
    },

    // Add security policy for Level 2+
    ...(securityLevel >= 2 && {
      security_policy: this.buildSecurityPolicy(securityLevel, params)
    })
  };

  // 7. Sign the envelope
  envelope.signature = await this.signEnvelope(envelope);

  // 8. Wrap in SMTP email
  const email = this.wrapInEmail(envelope, recipientEmail);

  // 9. Send via SMTP
  await this.smtpClient.send(email);

  return {
    message_id: envelope.message_id,
    sent_at: envelope.headers.date
  };
}

async signEnvelope(envelope) {
  // Create canonical representation for signing
  const canonicalData = this.canonicalize({
    message_id: envelope.message_id,
    headers: envelope.headers,
    encryption: envelope.encryption
  });

  // Sign with Ed25519
  const signature = ed25519.sign(
    canonicalData,
    this.signingKey.privateKey
  );

  return {
    algorithm: 'Ed25519',
    sender_key_id: this.signingKey.keyId,
    signed_data: this.toBase64(signature),
    timestamp: new Date().toISOString()
  };
}

buildSecurityPolicy(level, params) {
  const policy = {
    requires_online_auth: true,
    auth_provider: {
      name: 'Example Bank',
```

```
        app_id: 'com.examplebank.app',
        auth_endpoint: 'https://auth.examplebank.com/api/v1/email',
        public_key_id: this.signingKey.keyId
      }
    };

    if (level === 3) {
      // Time-bound policies
      policy.constraints = {
        expires_at: new Date(Date.now() + 15 * 60 * 1000).toISOString(), // 15 min
        max_decrypt_count: 1,
        geofence: {
          type: 'country_whitelist',
          countries: params.allowedCountries || ['US']
        },
        device_attestation_required: true
      };

      policy.read_receipt = {
        required: true,
        endpoint: 'https://receipts.examplebank.com/api/v1/confirm'
      };
    }

    return policy;
  }
}
```

---

3. Message Decryption (Recipient)

Level 1: Cached Key Decryption

```
async function decryptLevel1Message(envelope) {
  // 1. Retrieve cached private key
  const privateKey = await keyStore.get(
    envelope.encryption.recipient_key_id
  );

  if (!privateKey) {
    throw new Error('Private key not found. Please sync with identity provider.');
  }

  // 2. Verify sender signature BEFORE decryption
  const senderPublicKey = await fetchPublicKey(
    envelope.signature.sender_key_id
  );

  const isValid = await verifySignature(
    envelope,
    senderPublicKey,
    envelope.signature.signed_data
  );

  if (!isValid) {
    throw new SecurityError('Invalid sender signature - possible tampering');
  }

  // 3. Derive shared secret
  const sharedSecret = x25519.getSharedSecret(
    privateKey,
    base64ToBytes(envelope.encryption.ephemeral_public_key)
  );

  // 4. Derive decryption key
  const decryptionKey = await deriveKey(
    sharedSecret,
    base64ToBytes(envelope.encryption.salt)
  );

  // 5. Decrypt message
  const plaintext = await decryptChaCha20Poly1305(
    base64ToBytes(envelope.encryption.encrypted_body),
    decryptionKey,
    base64ToBytes(envelope.encryption.nonce)
  );

  return {
    plaintext: new TextDecoder().decode(plaintext),
    metadata: {
      sender: envelope.headers.from,
      subject: envelope.headers.subject,
      verified: true,
      security_level: 1
    }
  };
}
```

Level 2: Online Verification Flow

```
async function decryptLevel2Message(envelope) {
  const policy = envelope.security_policy;

  // 1. Show encrypted preview
  showSecurePreview({
    from: envelope.headers.from,
    subject: envelope.headers.subject,
```

```
  requiresAuth: true
});

// 2. Request authentication from identity provider
const authRequest = await fetch(policy.auth_provider.auth_endpoint + '/auth', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    message_id: envelope.message_id,
    recipient_email: envelope.headers.to,
    recipient_key_id: envelope.encryption.recipient_key_id,
    security_level: envelope.security_level,
    context: {
      sender: envelope.headers.from,
      subject_hash: await sha256(envelope.headers.subject),
      timestamp: envelope.headers.date
    },
    client_attestation: await generateClientAttestation()
  })
});

const authResponse = await authRequest.json();
const challengeId = authResponse.challenge_id;

// 3. Wait for user authentication (push notification)
// Poll for approval or use WebSocket for real-time update
const approval = await pollForApproval(
  policy.auth_provider.auth_endpoint,
  challengeId,
  timeout = 120000 // 2 minutes
);

if (!approval.approved) {
  throw new Error('Authentication denied by user');
}

// 4. Fetch ephemeral private key with decryption token
const keyRequest = await fetch(policy.auth_provider.auth_endpoint + '/decrypt-key', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    decryption_token: approval.decryption_token,
    message_id: envelope.message_id
  })
});

const keyResponse = await keyRequest.json();
const ephemeralPrivateKey = base64ToBytes(keyResponse.ephemeral_private_key);

// 5. Decrypt (same as Level 1)
const sharedSecret = x25519.getSharedSecret(
  ephemeralPrivateKey,
  base64ToBytes(envelope.encryption.ephemeral_public_key)
);

const decryptionKey = await deriveKey(
  sharedSecret,
  base64ToBytes(envelope.encryption.salt)
);

const plaintext = await decryptChaCha20Poly1305(
  base64ToBytes(envelope.encryption.encrypted_body),
  decryptionKey,
  base64ToBytes(envelope.encryption.nonce)
);
```

```
  // 6. CRITICAL: Securely erase ephemeral key
  secureErase(ephemeralPrivateKey);
  secureErase(decryptionKey);

  // 7. Send read receipt if required
  if (policy.read_receipt?.required) {
    await sendReadReceipt(envelope, approval, plaintext);
  }

  return {
    plaintext: new TextDecoder().decode(plaintext),
    metadata: {
      authenticated_at: approval.timestamp,
      auth_method: approval.method,
      security_level: 2
    }
  };
}

// Helper: Poll for authentication approval
async function pollForApproval(endpoint, challengeId, timeout) {
  const startTime = Date.now();

  while (Date.now() - startTime < timeout) {
    const response = await fetch(`${endpoint}/auth/${challengeId}`);
    const status = await response.json();

    if (status.status === 'approved') {
      return status;
    } else if (status.status === 'denied') {
      throw new Error('User denied authentication');
    }

    // Wait 1 second before next poll
    await sleep(1000);
  }

  throw new Error('Authentication timeout');
}
```

---

4. Identity Provider Authentication API

Authentication Challenge Handler

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import asyncio

app = FastAPI()

class AuthRequest(BaseModel):
    message_id: str
    recipient_email: str
    recipient_key_id: str
    security_level: int
    context: dict
    client_attestation: dict

class DecryptKeyRequest(BaseModel):
    decryption_token: str
    message_id: str
```

```
@app.post("/api/v1/email/auth")
async def create_auth_challenge(request: AuthRequest):
    """
    Create authentication challenge and send push notification
    """
    # 1. Validate client attestation
    if not verify_client_attestation(request.client_attestation):
        raise HTTPException(status_code=403, detail="Invalid client attestation")

    # 2. Look up user by email/key
    user = await db.get_user_by_email(request.recipient_email)
    if not user:
        raise HTTPException(status_code=404, detail="User not found")

    # 3. Retrieve message metadata for display
    message_info = await db.get_message_metadata(request.message_id)

    # 4. Create authentication challenge
    challenge_id = f"chg_{generate_secure_id()}"
    challenge = {
        'challenge_id': challenge_id,
        'user_id': user.id,
        'message_id': request.message_id,
        'security_level': request.security_level,
        'context': request.context,
        'created_at': datetime.utcnow(),
        'expires_at': datetime.utcnow() + timedelta(minutes=2),
        'status': 'pending'
    }

    await redis.setex(
        f"auth_challenge:{challenge_id}",
        120,  # 2 minute TTL
        json.dumps(challenge)
    )

    # 5. Send push notification to user's device(s)
    await push_service.send({
        'user_id': user.id,
        'type': 'email_decrypt_request',
        'title': 'Secure Email Authentication',
        'body': f'Decrypt email from {request.context["sender"]}?',
        'data': {
            'challenge_id': challenge_id,
            'sender': request.context['sender'],
            'subject_hash': request.context['subject_hash'],
            'security_level': request.security_level,
            'message_classification': message_info.get('classification', 'general')
        },
        'actions': ['Approve', 'Deny']
    })

    # 6. Log authentication request
    await audit_log.record({
        'event': 'AUTH_CHALLENGE_CREATED',
        'user_id': user.id,
        'message_id': request.message_id,
        'security_level': request.security_level,
        'client': request.client_attestation.get('client_id'),
        'timestamp': datetime.utcnow()
    })

    return {
        'challenge_id': challenge_id,
        'challenge_type': 'push_notification',
```

```
        'expires_in': 120,
        'status': 'pending'
    }

@app.get("/api/v1/email/auth/{challenge_id}")
async def check_auth_status(challenge_id: str):
    """
    Poll endpoint for checking authentication status
    """
    challenge = await redis.get(f"auth_challenge:{challenge_id}")

    if not challenge:
        raise HTTPException(status_code=404, detail="Challenge not found or expired")

    challenge_data = json.loads(challenge)

    return {
        'status': challenge_data['status'],
        'decryption_token': challenge_data.get('decryption_token'),
        'timestamp': challenge_data.get('approved_at'),
        'method': challenge_data.get('auth_method')
    }

@app.post("/api/v1/email/user-approve")
async def user_approves_auth(challenge_id: str, auth_method: str):
    """
    Called by mobile app when user approves authentication
    """
    # 1. Retrieve challenge
    challenge = await redis.get(f"auth_challenge:{challenge_id}")
    if not challenge:
        raise HTTPException(status_code=404, detail="Challenge expired")

    challenge_data = json.loads(challenge)

    # 2. Verify user authentication (biometric/PIN already done in app)
    user = await db.get_user(challenge_data['user_id'])

    # 3. Check security policies for Level 3
    if challenge_data['security_level'] == 3:
        policy_check = await evaluate_policy(user, challenge_data)
        if not policy_check['allowed']:
            raise HTTPException(status_code=403, detail=policy_check['reason'])

    # 4. Generate time-limited decryption token
    token_expiry = 5 if challenge_data['security_level'] >= 2 else 60  # minutes
    decryption_token = await generate_jwt({
        'challenge_id': challenge_id,
        'user_id': user.id,
        'message_id': challenge_data['message_id'],
        'recipient_key_id': challenge_data.get('recipient_key_id'),
        'security_level': challenge_data['security_level'],
        'exp': datetime.utcnow() + timedelta(minutes=token_expiry)
    })

    # 5. Update challenge status
    challenge_data['status'] = 'approved'
    challenge_data['approved_at'] = datetime.utcnow().isoformat()
    challenge_data['auth_method'] = auth_method
    challenge_data['decryption_token'] = decryption_token

    await redis.setex(
        f"auth_challenge:{challenge_id}",
        120,
        json.dumps(challenge_data)
    )
```

```
    )

    # 6. Audit log
    await audit_log.record({
        'event': 'AUTH_APPROVED',
        'user_id': user.id,
        'challenge_id': challenge_id,
        'message_id': challenge_data['message_id'],
        'auth_method': auth_method,
        'timestamp': datetime.utcnow()
    })

    return {'success': True}

@app.post("/api/v1/email/decrypt-key")
async def provide_decryption_key(request: DecryptKeyRequest):
    """
    Provide ephemeral private key for message decryption
    """
    # 1. Validate and decode decryption token
    try:
        token_data = verify_jwt(request.decryption_token)
    except JWTError:
        raise HTTPException(status_code=403, detail="Invalid or expired token")

    # 2. Check if token already used (for Level 3 single-use)
    token_key = f"used_token:{request.decryption_token[:32]}"
    if await redis.exists(token_key):
        raise HTTPException(status_code=403, detail="Token already used")

    # 3. Retrieve user's private key from HSM
    private_key_shard = await hsm.retrieve_key_shard(
        token_data['recipient_key_id']
    )

    # 4. Mark token as used (if single-use required)
    if token_data['security_level'] == 3:
        await redis.setex(token_key, 300, '1')  # 5 minute TTL

    # 5. Audit log
    await audit_log.record({
        'event': 'DECRYPTION_KEY_PROVIDED',
        'user_id': token_data['user_id'],
        'message_id': token_data['message_id'],
        'key_id': token_data['recipient_key_id'],
        'timestamp': datetime.utcnow()
    })

    return {
        'ephemeral_private_key': base64.b64encode(private_key_shard).decode(),
        'expires_at': token_data['exp'],
        'single_use': token_data['security_level'] == 3,
        'key_algorithm': 'X25519'
    }

async def evaluate_policy(user, challenge_data):
    """
    Evaluate security policies for Level 3 messages
    """
    # Device trust check
    device = await db.get_user_device(user.id)
    if device.trust_score < 0.95:
        return {'allowed': False, 'reason': 'Device trust insufficient'}

    # Geolocation check (if specified in message policy)
    user_location = await get_user_location(user.id)
```

```
    message_policy = challenge_data.get('policy', {})

    if 'geofence' in message_policy:
        allowed_countries = message_policy['geofence'].get('countries', [])
        if user_location['country'] not in allowed_countries:
            return {'allowed': False, 'reason': 'Geographic restriction'}

    # Risk scoring
    risk_score = await calculate_risk_score(user, challenge_data)
    if risk_score > 0.7:
        return {'allowed': False, 'reason': 'Risk score too high'}

    return {'allowed': True}
```

---

5. Key Discovery Protocol

DNS-Based Public Key Discovery

```
// Query DNS for recipient's public key
async function lookupPublicKeyDNS(email) {
  const domain = email.split('@')[1];
  const dnsQuery = `_tsep._tcp.${domain}`;

  try {
    // Query DNS TXT record
    const records = await dns.resolveTxt(dnsQuery);

    // Parse TSEP record
    const tsepRecord = records.find(r => r[0].startsWith('v=TSEP1'));
    if (!tsepRecord) {
      throw new Error('TSEP not supported by domain');
    }

    // Extract key server URL
    const keyServerMatch = tsepRecord[0].match(/k=([^;]+)/);
    const keyServerUrl = keyServerMatch[1];

    // Fetch key from HTTPS endpoint
    const response = await fetch(`${keyServerUrl}/${email}`);
    const keyData = await response.json();

    return {
      publicKey: base64ToBytes(keyData.public_keys[0].public_key),
      keyId: keyData.public_keys[0].key_id,
      algorithm: keyData.public_keys[0].algorithm,
      identityProvider: keyData.identity_provider
    };

  } catch (error) {
    // Fallback to HTTPS-only discovery
    return await lookupPublicKeyHTTPS(email);
  }
}

// HTTPS Key Server Implementation
async function lookupPublicKeyHTTPS(email) {
  const domain = email.split('@')[1];
  const keyServerUrl = `https://${domain}/.well-known/tsep-keys/${email}`;

  const response = await fetch(keyServerUrl);
  if (!response.ok) {
    throw new Error(`Key lookup failed: ${response.status}`);
  }

  const keyData = await response.json();

  // Verify key data signature (certificate transparency)
  await verifyKeyDataSignature(keyData);

  return {
    publicKey: base64ToBytes(keyData.public_keys[0].public_key),
    keyId: keyData.public_keys[0].key_id,
    algorithm: keyData.public_keys[0].algorithm,
    expiresAt: keyData.public_keys[0].expires_at,
    identityProvider: keyData.identity_provider
  };
}
```

Key Server Implementation

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/.well-known/tsep-keys/{email}")
async def get_public_key(email: str):
    """
    Serve public key for email address
    """
    # Look up user's active public key
    user = await db.get_user_by_email(email)
    if not user:
        return JSONResponse(status_code=404, content={"error": "User not found"})

    active_keys = await db.get_active_keys(user.id)

    return {
        "email": email,
        "public_keys": [
            {
                "key_id": key.key_id,
                "algorithm": "X25519",
                "public_key": base64.b64encode(key.public_key).decode(),
                "created_at": key.created_at.isoformat(),
                "expires_at": key.expires_at.isoformat(),
                "status": "active"
            }
            for key in active_keys
        ],
        "identity_provider": {
            "name": "Example Bank",
            "auth_endpoint": "https://auth.examplebank.com/api/v1/email",
            "public_key": base64.b64encode(PROVIDER_PUBLIC_KEY).decode()
        },
        "security_levels_supported": [1, 2, 3],
        "tsep_version": "1.0"
    }
```

---

6. Read Receipt Generation (Level 3)

```
async function generateReadReceipt(envelope, authResult, plaintext) {
  // Create receipt data
  const receipt = {
    message_id: envelope.message_id,
    recipient: envelope.headers.to,
    sender: envelope.headers.from,
    read_at: new Date().toISOString(),

    // Hash of plaintext for non-repudiation
    content_hash: await sha256(plaintext),

    // Authentication proof
    authentication_proof: {
      method: authResult.method,
      timestamp: authResult.timestamp,
      device_id: await getDeviceId(),
      location: await getUserLocation(),
      trust_score: authResult.trust_score
    }
  };

  // Sign receipt with recipient's private key
  const signature = await signWithPrivateKey(receipt, recipientPrivateKey);
  receipt.signature = signature;

  return receipt;
}

async function sendReadReceipt(receipt, endpoint) {
  await fetch(endpoint, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(receipt)
  });
}
```

---

7. Security Utilities

Secure Memory Erasure

```
function secureErase(sensitiveData) {
  if (sensitiveData instanceof Uint8Array) {
    // Overwrite with random data
    crypto.getRandomValues(sensitiveData);
    // Overwrite with zeros
    sensitiveData.fill(0);
  } else if (typeof sensitiveData === 'string') {
    // For strings, overwrite the memory if possible (browser limitations)
    sensitiveData = null;
  }

  // Hint to garbage collector (not guaranteed)
  if (global.gc) global.gc();
}
async function generateClientAttestation() {
  // Platform-specific attestation
  if (platform.isIOS()) {
    // Use DeviceCheck or Apple App Attest
    return await generateIOSAttestation();
  } else if (platform.isAndroid()) {
    // Use SafetyNet or Play Integrity API
    return await generateAndroidAttestation();
```

```
  } else {
    // Fallback: browser or unknown platform
    return await generateWebAttestation();
  }
}

// iOS Device Attestation Example
async function generateIOSAttestation() {
  // This is a placeholder. In production, use native bridge to DeviceCheck/AppAttest.
  // Example: call native module and return attestation object.
  return {
    platform: 'iOS',
    device_id: await getDeviceId(),
    attestation_data: await getIOSDeviceCheckToken(),
    public_key: await getDevicePublicKey()
  };
}

// Android Device Attestation Example
async function generateAndroidAttestation() {
  // This is a placeholder. In production, use native bridge to SafetyNet/Play Integrity.
  return {
    platform: 'Android',
    device_id: await getDeviceId(),
    attestation_data: await getAndroidSafetyNetToken(),
    public_key: await getDevicePublicKey()
  };
}

// Web Attestation Example (fallback)
async function generateWebAttestation() {
  // Use WebAuthn or similar for browser-based attestation
  return {
    platform: 'Web',
    device_id: await getDeviceId(),
    attestation_data: await getWebAuthnAssertion(),
    public_key: await getDevicePublicKey()
  };
}

// Helpers (placeholders, must be implemented per platform/environment)
async function getDeviceId() {
  // Return a unique device identifier (platform-specific)
  // For demo, use a random UUID
  if (window && window.crypto && window.crypto.randomUUID) {
    return window.crypto.randomUUID();
  }
  return 'device-' + Math.random().toString(36).slice(2);
}

async function getIOSDeviceCheckToken() {
  // Call native bridge to get DeviceCheck/AppAttest token
  // Placeholder: return dummy token
  return 'dummy-ios-devicecheck-token';
}

async function getAndroidSafetyNetToken() {
  // Call native bridge to get SafetyNet/Play Integrity token
  // Placeholder: return dummy token
  return 'dummy-android-safetynet-token';
}

async function getWebAuthnAssertion() {
  // Use WebAuthn API for browser attestation
  // Placeholder: return dummy assertion
```

```
  return 'dummy-webauthn-assertion';
}

async function getDevicePublicKey() {
  // Return device public key (used for attestation)
  // Placeholder: return dummy key
  return new Uint8Array(32).fill(1);
}
```