

Server:

Certificate Creation Process in SimpleEnroll

The SimpleEnroll function implements a secure certificate enrollment system that creates X.509 certificates signed by a Certificate Authority (CA). Here's a step-by-step breakdown of the process:

1. Initial Request Processing

- The function accepts a POST request with JSON containing:
 - **key**: An RSA 2048-bit public key in PEM format
 - **data**: Optional base64-encoded JSON data containing identity information and quantum-resistant keys

2. Public Key Validation

The system first validates the provided public key:

```
AsymmetricKeyParameter? publicKeyRequestor =  
(AsymmetricKeyParameter)BouncyCastleHelper.fromPEM(key);
```

If the key is invalid or not in proper PEM format, the request is rejected.

3. Identity Verification Process

The system supports two modes:

- **Anonymous mode**: Users can get certificates without providing identity
- **Identity-verified mode**: Users must provide an email address and verification token

For identity verification:

1. The user provides an email address (identity) and a verification token
2. The system checks if anonymous enrollment is allowed (`GLOBALS.Anonymous`)
3. If identity is required, it validates the email domain against allowed domains
4. The system looks for a token file in S3: `{origin}/tokens/{identity}.token`
5. Token validation includes:
 - Checking if the token matches the stored token
 - Verifying the token hasn't expired (1-hour validity)
 - Ensuring the identity in the token file matches the provided identity
6. Upon successful validation, the token file is deleted

4. Unique Alias Generation

The system generates a unique three-word alias using a sophisticated algorithm:

- Selects three random words from a dictionary of ~10,000 common English words

- Format: `word1-word2-word3.{origin}` (e.g., `happy-cloud-tree.example.com`)
- Includes a rarity system:
 - **Legendary**: All three words are the same (1/10,000 chance)
 - **Epic**: Two words match (10/10,000 chance)
 - **Rare/Uncommon/Common**: Based on probability distribution
- Checks S3 to ensure the alias doesn't already exist
- Retries up to 10 times if collisions occur

5. Certificate Authority Key Retrieval

The system retrieves the CA's private key for signing:

```
byte[] cakeysBytes = System.IO.File.ReadAllBytes($"subcakeys.{origin}.pem");
byte[] cakeysDecrypted = BouncyCastleHelper.DecryptWithKey(cakeysBytes,
    GLOBALS.password.ToBytes(), GLOBALS.origin.ToBytes());
```

- The CA keys are stored encrypted using AES-GCM
- Decryption requires the server password and origin as additional authenticated data

6. X.509 Certificate Creation

The certificate is created with these specifications:

- **Subject DN**: `CN={alias}` (the three-word alias)
- **Issuer DN**: `CN={origin}` (the CA's domain)
- **Serial Number**: Random 64-bit integer
- **Validity Period**: 397 days (just over 13 months)
- **Subject Alternative Names**:
 - DNS name: The alias
 - RFC822 name: The email address (if provided)
- **Key Usage**: KeyEncipherment only
- **Basic Constraints**: CA:FALSE (not a CA certificate)
- **Custom Extension** (OID 1.3.6.1.4.1.57055): Contains the base64-encoded data including:
 - Quantum-resistant keys (Kyber for encryption, Dilithium for signatures)
 - Identity information
- **Signature Algorithm**: SHA512withRSA

7. Certificate Storage

The signed certificate is stored in S3:

- Primary location: `{origin}/cert/{alias}.pem`
- If identity provided: `{origin}/identity/{identity}/{alias}.pem`

8. Response

The system returns:

- **alias**: The generated three-word alias
- **origin**: The CA's domain
- **publickey**: The user's public key (echoed back)
- **certificate**: The signed X.509 certificate in PEM format

Importance of Valid SSL Certificate on Hosted Domain

The valid SSL certificate on the hosted domain is **critically important** for several reasons:

1. **Trust Chain Establishment**: The domain's SSL certificate establishes the initial trust relationship. When clients connect to retrieve certificates, they can verify they're talking to the legitimate certificate authority.
2. **Man-in-the-Middle Protection**: Without HTTPS secured by a valid SSL certificate, attackers could intercept certificate enrollment requests and issue fraudulent certificates.
3. **Identity Verification**: The domain in the SSL certificate becomes part of the certificate hierarchy. All issued certificates have the domain as their issuer, creating a verifiable chain of trust.
4. **API Security**: All API endpoints (enrollment, retrieval, verification) are protected by HTTPS, ensuring:
 - Confidentiality of public keys during transmission
 - Integrity of certificates being downloaded
 - Authentication of the certificate authority server
5. **Token Protection**: Identity verification tokens are transmitted over HTTPS, preventing interception and replay attacks.
6. **Certificate Validation**: The system includes a **VerifyAliasAsync** method that retrieves and validates certificates over HTTPS. This process relies on the SSL certificate to ensure the validation request reaches the authentic CA.

The system essentially creates a two-tier PKI where:

- The domain's SSL certificate (from a trusted CA) establishes web trust
- The custom CA certificate (self-signed but verified through the SSL-protected API) establishes application-level trust

This design allows the system to bootstrap trust from the web PKI into a custom PKI for secure communications, making the valid SSL certificate a fundamental security requirement.

Client

Client-Side Certificate Creation Process (Create.cs)

The client-side certificate creation process is a sophisticated implementation that combines traditional RSA cryptography with quantum-resistant algorithms. Here's a step-by-step breakdown:

1. Command-Line Interface

The client uses a command-line verb system:

```
[Verb("create", HelpText = "Create an alias.")]
```

Options include:

- `-e` or `--email`: Optional email address to associate with the alias
- `-t` or `--token`: Email validation token (required if email is provided)

2. Key Generation Process

The client generates three different key pairs for comprehensive security:

a) RSA Key Pair (Classical Cryptography)

```
AsymmetricCipherKeyPair keyPair = BouncyCastleHelper.GenerateKeyPair(2048);
```

- Generates a 2048-bit RSA key pair
- Used for traditional encryption and digital signatures
- Compatible with existing PKI infrastructure

b) Kyber Key Pair (Quantum-Resistant Encryption)

```
AsymmetricCipherKeyPair KyberKeyPair =  
BouncyCastleQuantumHelper.GenerateKyberKeyPair();
```

- Uses Kyber1024 parameters (highest security level)
- Provides quantum-resistant encryption capabilities
- Based on Module Learning with Errors (MLWE) problem
- NIST-approved post-quantum cryptography standard

c) Dilithium Key Pair (Quantum-Resistant Signatures)

```
AsymmetricCipherKeyPair DilithiumKeyPair =  
BouncyCastleQuantumHelper.GenerateDilithiumKeyPair();
```

- Uses Dilithium5 parameters (highest security level)
- Provides quantum-resistant digital signatures
- Based on Module Learning with Errors and Short Integer Solution problems
- NIST-approved post-quantum cryptography standard

3. Data Payload Preparation

The client creates a structured payload containing:

```
var data = new CustomExtensionData{
    KyberKey = Convert.ToBase64String(KyberPublicKey),
    DilithiumKey = Convert.ToBase64String(DilithiumPublicKey),
    Email = opts.Email ?? string.Empty,
    Token = opts.Token ?? string.Empty
};
```

This data is:

1. Serialized to JSON
2. Encoded to UTF-8 bytes
3. Base64-encoded for transmission

4. Certificate Request Submission

The client sends a POST request to the server's `/simpleenroll` endpoint:

```
{
  "key": "RSA public key in PEM format",
  "data": "Base64-encoded JSON containing quantum keys and identity info"
}
```

5. Server Response Processing

The server returns:

- `alias`: The unique three-word identifier
- `certificate`: The signed X.509 certificate
- `origin`: The certificate authority's domain
- `publickey`: Echo of the submitted public key

6. Local Storage of Cryptographic Materials

The client stores all cryptographic materials securely:

a) Certificate Storage

```
Storage.StoreCert(alias, j.Certificate);
```

- Stored as PEM file: `{alias}.pem`
- Location: `~/local/share/surepack/aliases/` (Linux) or equivalent

b) Private Key Storage

Each private key is encrypted before storage:

```
Storage.StorePrivateKey($"{alias}.rsa", privateKeyPem, Globals.Password);
Storage.StorePrivateKey($"{alias}.kyber", kyberPrivateKeyPem, Globals.Password);
Storage.StorePrivateKey($"{alias}.dilithium", dilithiumPrivateKeyPem,
Globals.Password);
```

Encryption mechanism:

- Uses AES-GCM encryption
- Key: User's password (converted to bytes)
- Nonce: Domain name from alias (provides uniqueness)
- Each key type stored separately: `.rsa`, `.kyber`, `.dilithium`

7. Certificate Verification

After receiving the certificate, the client immediately verifies it:

```
(bool valid, byte[] rootFingerprint) = await
BouncyCastleHelper.VerifyAliasAsync(domain, alias, "");
```

This verification process:

1. Retrieves the CA certificates from the server
2. Validates the certificate chain
3. Checks certificate validity dates
4. Verifies the CA's signing authority
5. Returns the root CA's fingerprint

8. Root Fingerprint Storage

```
string rootFingerprintHex = Convert.ToBase64String(rootFingerprint);
Storage.StorePrivateKey($"{alias}.root", rootFingerprintHex, Globals.Password);
```

- The root CA's fingerprint is stored encrypted
- Used for future certificate validations
- Provides trust anchor for the custom PKI

Security Features and Considerations

1. Hybrid Cryptography Approach

The system implements a "belt and suspenders" approach:

- **RSA**: Current standard, widely supported

- **Kyber**: Quantum-resistant encryption
- **Dilithium**: Quantum-resistant signatures

This ensures security against both current and future (quantum) threats.

2. Local Key Generation

All private keys are generated locally on the client:

- Private keys never leave the client device
- Only public keys are sent to the server
- Ensures true end-to-end security

3. Password-Based Encryption

Private keys are encrypted using the user's password:

- Protects keys at rest
- Password never transmitted to server
- Each key encrypted with unique nonce (domain-based)

4. Certificate Chain Validation

The client performs thorough validation:

- Verifies the entire certificate chain
- Checks certificate validity periods
- Validates CA signing authority
- Ensures domain name consistency

5. Storage Organization

```
~/.local/share/surepack/  
├── aliases/  
│   ├── {alias}.pem           (certificate)  
│   ├── {alias}.rsa           (encrypted RSA private key)  
│   ├── {alias}.kyber         (encrypted Kyber private key)  
│   ├── {alias}.dilithium     (encrypted Dilithium private key)  
│   └── {alias}.root          (encrypted root fingerprint)  
└── settings.json
```

6. Error Handling and User Experience

- Progress reporting for UI integration
- Comprehensive error messages
- Password retry mechanism
- Graceful failure handling

Importance of SSL Certificate (Client Perspective)

From the client's perspective, the server's valid SSL certificate is crucial for:

1. **Initial Trust Bootstrap:** The HTTPS connection validates the server's identity before any certificate enrollment
2. **Secure Key Transmission:** The RSA public key and quantum public keys are transmitted over HTTPS, preventing interception
3. **Token Security:** Email verification tokens are protected during transmission
4. **Certificate Retrieval:** When verifying certificates, the client relies on HTTPS to ensure authentic CA certificates
5. **Man-in-the-Middle Prevention:** Without HTTPS, an attacker could intercept the enrollment request and issue fraudulent certificates

The client's `VerifyAliasAsync` method specifically makes HTTPS calls to retrieve and validate certificates, making the SSL certificate a fundamental security requirement for the entire PKI system to function securely.