Distributed Credential Protection

Summary

Design a distributed credential protecion scheme so that the credential generated and proected in one server node could be used in an entire cluster.

Motivation

For TLS protocols, an initial connection will negotiate the security parameters and then establishe the security channel between client and server. The initial connection is expensive because a lot of cryptographic operations are involved. The negotiated parameters could be cached in a protected credential. The credential could be reused for subsequent connections. The credential reuse could significantly improve the performance the the subsequent connections.

We want to extend the benefit from connections between the same client and server to connections between the same client and an entire cluster. Although the credential is created and protected in one server node, it must be usable on any server node in the cluster. As require that each node should share the secrets/keys that are used to protec the credential.

The distributed credential protection scheme should take care of the key genetation, key rotation and synchronization across the clusters of computers.

Related Cryptographic Algorithms and Terms

Secure hash function

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. A secure hash algorithm is a hash function that is suitable for use in cryptography.

```
    HASH(m) -> MD

    Options:
        Hash    a hash function; HashLen donotes the length of the
                hash function output in octets.
    Input:
        m       the message

    Output:
        MD      a fixed-size message digest (of HashLen octets)

    The output MD is calculated by mapping the input message to
    fixed-size values with specific hash algorithm.  The output
    could be notated as follows:

        MD = HASH(m)
```

Keyed-Hash Message Authentication Code (HMAC)

In cryptography, an HMAC(Keyed-hash message authentication code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key.

```
HMAC-Hash(m, K) -> MAC

Options:
    Hash    a hash function; HashLen donotes the length of the
            hash function output in octets.
Input:
    m       the message
    K       the secret key

Output:
    MAC     a fixed-size message authentication code (of HashLen octets)

The output MAC could be notated as follows:

    MAC = HMAC-Hash(m, K)
```

HMAC-based key derivation function (HKDF)

HKDF is a HMAC-based key derivation function (KDF). It is used to take some source of initial keying material and derive from it one or more cryptographically strong secret keys.

HKDF follows the "extract-then-expand" paradigm, where the KDF logically consists of two modules. The first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key K. The second stage "expands" the key K into several additional pseudorandom keys (the output of the KDF).

1. HKDF-Extract(salt, IKM) -> PRK

```
Options:
   Hash    a hash function; HashLen denotes the length of the
           hash function output in octets

Inputs:
   salt    optional salt value (a non-secret random value);
           if not provided, it is set to a string of HashLen zeros.
   IKM     input keying material

Output:
   PRK     a pseudorandom key (of HashLen octets)

The output PRK is calculated as follows:

   PRK = HMAC-Hash(salt, IKM)

or notated as follows:

   PRK = HKDF-Extract(salt, IKM)
```

2. HKDF-Expand(PRK, info, L) -> OKM

```
Options:
   Hash    a hash function; HashLen denotes the length of the
           hash function output in octets

Inputs:
   PRK     a pseudorandom key of at least HashLen octets
           (usually, the output from the extract step)
```

```
   info    optional context and application specific information
           (can be a zero-length string)
   L       length of output keying material in octets
           (<= 255*HashLen)

Output:
   OKM      output keying material (of L octets)

The output OKM could be notated as follows:

   OKM = HKDF-Expand(PRK, info, L)
```

Authenticated encryption with associated data (AEAD)

Authenticated encryption with associated data (AEAD) is a form of encryption which simultaneously assure the confidentiality and authenticity of data.

```
AEAD-Encrypt(key, nonce, additional_data, plaintext) -> AEADEncrypted

Input:
    key         the secret key for the encryption
    nonce       a unique value for each encryption operation
    additional_data     the associated data
    plaintext   the message to be encrypted

Output:
    AEADEncrypted   the ciphertext

AEAD-Decrypt(key, nonce,
             additional_data, AEADEncrypted)  ->  plaintext

Input:
    key         the secret key for the encryption
    nonce       a unique value for each encryption operation
    additional_data     the associated data
    AEADEncrypted       the ciphertext

Output:
    plaintext   the decrypted message
```

Distributed Credential Protecion Scheme

The basic idea of this proposal is deriving credential protecion key from the server authentication possessions. In general, the server possesses some private information for the server authentication on TLS connections, for example, the RSA/EC private key. Every server in the distributed system should be able to access the possession.

With the proposed scheme, the credential protecion keys will be automatically generated and rotated in each server, and automatically synchronized among the distributed system.

This proposal can be considered as a triple stage process.

1. Select the server possessions for key generation;
2. Design the key rotation scheme;
3. Design the credential protecion scheme.

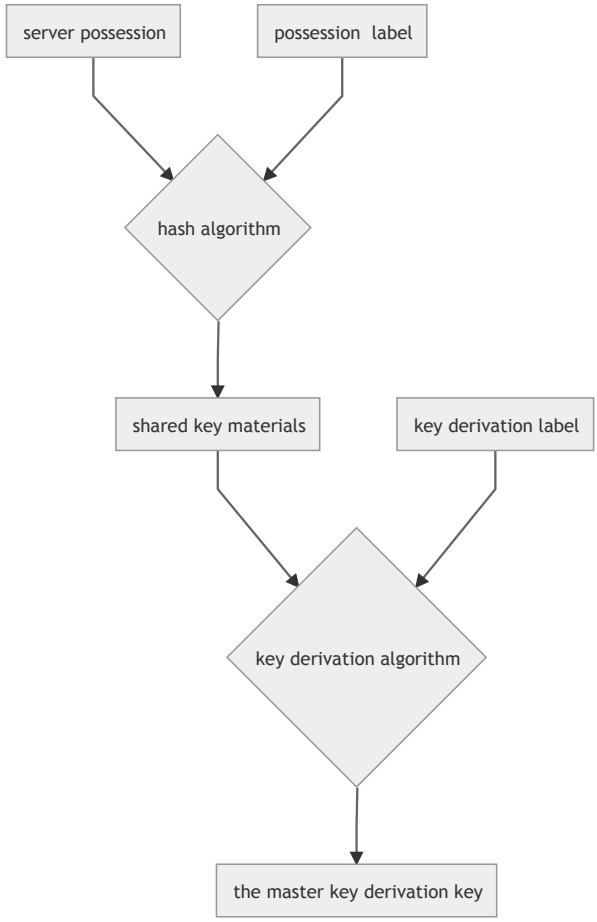
Select the possessions and derive the master key derivation key.

Each node in the cluster should possesses some private information for server authentication. In TLS context, we select to used the private key and public key as the server possessions, which is the same in each server node in the cluster.

```
    Possession: encoded public key | encoded private key
```

The server possessions cannot be used directly for credential protection. Instead, the keying material should be generated from the server possessions.

```
    Option:
        Hash:       hash algorithm
        Label:      a choosen octests label for key derivation
    Input:
        Possession:  the server possession

    // Derive the shared key materials from the server possession
    SKM = HASH(Possession || SKMLabel)
        Hash:       the hash algorithm
        SKMLabel:   label for this derivation

    // Derive the master key derivation key.
    KDK = HKDF_Extract(KDKLabel, SKM)
        KDKLabel:   label for this derivation, as salf for HKDF_Extract

    // Clean the shared key materials
    SKM = zeros
```



Time based key rotation scheme

If the maximum key use is limited, or could be exceeded, the key should be rotated before reach the limit. At a time, each node in the cluster should use the credential protection key. The secret key should be updated and synchronized among the cluster. A key rotation scheme defines the key update and synchronization policy so that the credential protection keys do not exceed the maximum use-limit or the timeout limit.

The time based key rotation schme based on the follow two assumptions:

1. The system clock in each node should be synchronized. It is doable by using the Network Time Protocol (NTP).
2. If the system clock is not precise synchronized, the secure parameters could be re-negotiated, without breaking the connection.

Here is the time-based key rotation schme:

1. Define the key rotation timeout, for example one week, or two weeks.

```
private static final long KEY_DERIVATION_PERIOD = TimeUnit.DAYS.toMillis(7);
```

2. When a node inserted into the system, calculate how many periods (or timeouts) have passed since a past-time (for example, 01/01/1970).

```
long periodsSince1970 = System.currentTimeMillis() / KEY_DERIVATION_PERIOD;
```

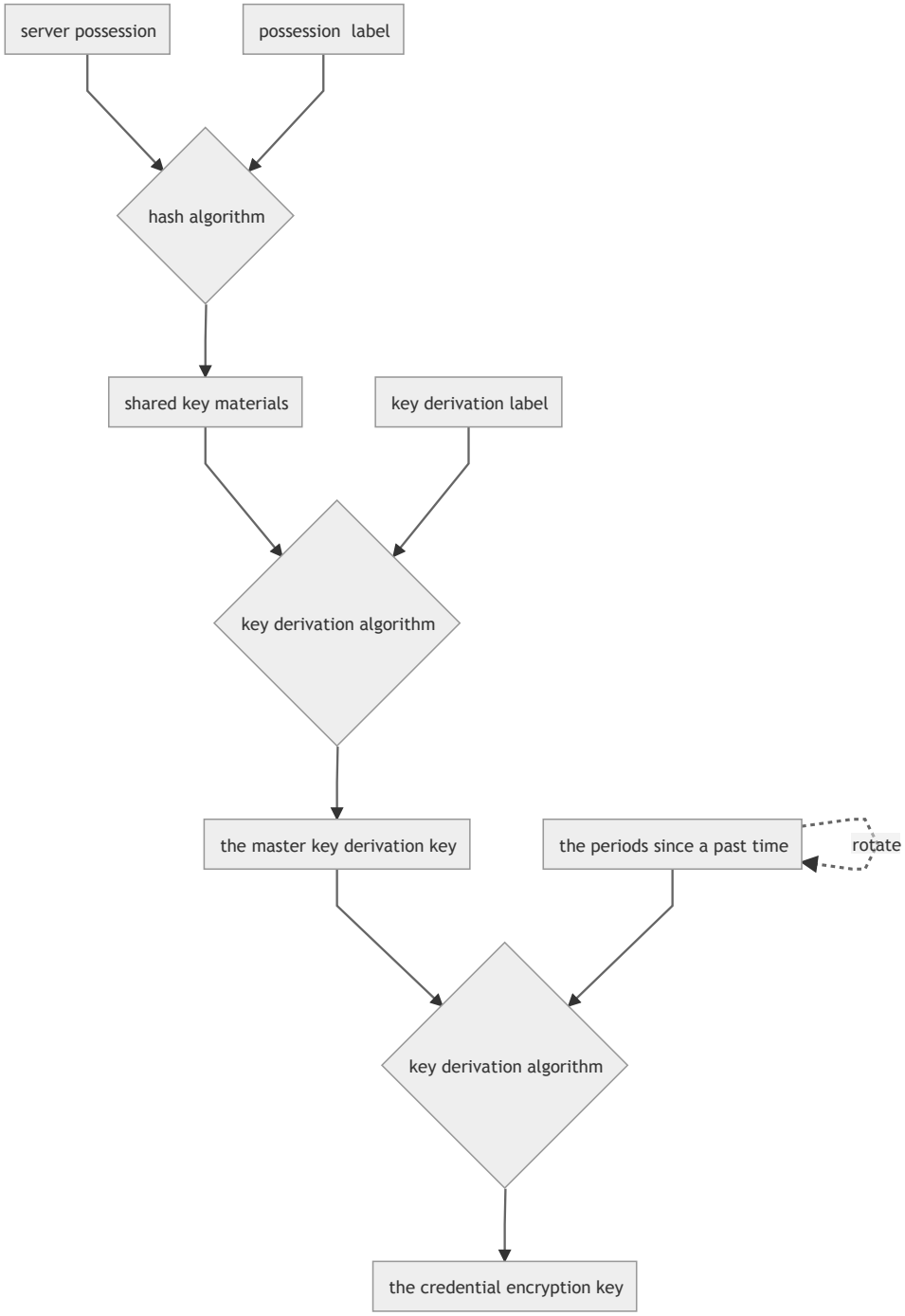
3. Derive the secret key with a deterministic key derivation algorithm (for example, HKDF).

```
    Option:
        Hash:       hash algorithm used for the key derivation

    Input:
        PRK         a pseudorandom key, which is derived from the server possessions
        info        the periods since a past time.
        L           length of output keying material in octets.

    // Derive the credential encryption key.
    TEK = HKDF-Expand(PRK, info, L)
```

```
SecretKey credentialEncryptionKey = HKDF.of(scheme.hashAlg).expand(
                        ppk,
                        Utilities.toByteArray(periodsSince1970),
                        keyScheme.keySize, keyScheme.keyAlg);
```

4. When the system clock moving onto the next period, update the credential encryption key, as described in #3.

If combining the server possession part together, the scenarios is as showned in the following diagram:



Credential protecion scheme

The protected credential is defined as:

```
    Credential: periods | nonce | EncryptedParameters
        periods             : how many periods (or timeouts) have passed since a pas
        nonce               : a random number for each encryption (for AEAD cipher)
        EncryptedParameters : encrypted negotiated-parameters for the initial connec
```

Here is the scheme for the credential creation:

```
    Input:
        key                 :   the credential encryption key
        nonce               :   a random number for each encryption
```

```
        plaintextParameters :   plaintext parameters before encryption.
        periods             :   the peirods have passed since a past-time.

    Output:
       the protected credential

    // Protect the negotiated parameters
    //
    // additional_data: the millis at the end of the peirods since a past time.
    EncryptedParameters =
        AEAD-Encrypt(key, nonce, additional_data, plaintextParameters)

    // Construct the protected credential
    ProtectedCredential = periods | nonce | EncryptedParameters
```

Here is the scheme for credential reuse:

```
    Input:
        ProtectedCredential :   the protected credential

    Output:
       plaintext parameters before encryption, and the peirods have passed since a pa

    // Decapsulate the protected credential, and get the peirods, nonce and Encrypte
    periods | nonce | EncryptedParameters

    // Get the key specified for this periods.
    ondutyKey = ...   // Rotate the key if needed.

    // Decrypt the EncryptedParameters
    //
    // additional_data: the millis at the end of the peirods since a past time.
    plaintextParameters =
        AEAD-Decrypt(ondutyKey, nonce, additional_data, EncryptedParameters)
```