Access Token Design

Version 1.0

Requirements

Currently, DNs do not enforce any access control on accesses to its data blocks. This makes it possible for an unauthorized client to read a data block as long as she can supply its block ID. It's also possible for anyone to write arbitrary data blocks to DNs.

When users request file accesses on NN, file permission checking takes place. Authorization decisions are made with regard to whether the requested accesses to those files (and implicitly, to their corresponding data blocks) are permitted. However, when it comes to subsequent data block accesses on DNs, those authorization decisions are not made available to DNs and consequently, such accesses are not verified. DNs are not capable of making those decisions independently since they don't have concepts of files, let alone file permissions.

In order to implement data access policies consistently across HDFS services, there is a need for a mechanism by which authorization decisions made on NN can be enforced on DNs and any unauthorized access is declined.

Design

We use Access Tokens to pass data access authorization information from NN to DN. One can think of Access Tokens as capabilities; an Access Token enables its owner to access certain data block. It is issued by NN and used on DN. Access Tokens should be generated in such a way that their authenticity can be verified by DN.

In general, tokens can be generated in 2 ways.

A. Using a public-key scheme, where NN chooses a pair of private/public keys and uses the private key to sign a token. The signature becomes an integral part of the token. DN is given NN's public key, which can be used to verify the signature of a token. Since only the NN knows the private key, only the NN can generate a valid token.
B. Using a symmetric key scheme, where NN and all DNs share a secret key. For each token, NN computes a keyed hash (also known as message authentication code or MAC) using the key. Hereinafter, the hash value is referred to as the token authenticator. Token authenticator becomes an integral part of the token. When DN receives a token, it uses its own copy of the secret key to re-compute the token authenticator and compares it with the one included in the token. If they match, the

token is verified as authentic. Since only NN and DNs know the key, no third party can forge tokens.

Method A has the advantage that DN doesn't have to store any secret key and it provides stronger security in the sense that even if a DN is compromised, the attacker still can't forge valid tokens. However, generating and verifying public-key signatures are expensive compared to symmetric-key operations.

We use method B to generate Access Tokens. Method B has better performance, but it has the disadvantage that if a DN is compromised, the attacker can use the compromised secret key to forge valid tokens. However, in HDFS setups where all DNs are protected uniformly in the same way (e.g., inside the same data center and protected by the same firewall policy), it may not make a fundamental difference. In other words, if an attacker is able to compromise one DN, he/she can compromise all DNs in the same way, without having to figure out the secret key.

Access Tokens are ideally non-transferable, i.e., only the owner can use it. This means we don't have to worry if a token gets stolen, for example during transit. One way to make it non-transferable is to include the owner's ID in the token and require whoever uses the token to authenticate as the owner specified in the token. In the current implementation, we include the owner's ID in the token, but DN doesn't verify it. Authentication and verification of owner ID can be added later if needed.

Access Tokens are meant to be lightweight and short-lived. No need to renew or revoke an Access Token. When a cached Access Token expires, simply get a new one. Access Tokens should be cached only in memory and never written to disk. A typical use case is as follows. An HDFS client asks NN for block ids/locations for a file. NN verifies that the client is authorized to access the file and sends back block ids/locations along with an Access Token for each block. Whenever the HDFS client needs to access a block, it sends the block id along with its associated Access Token to a DN. DN verifies the Access Token before allowing access to the block. The HDFS client may cache Access Tokens received from NN in memory and only get new tokens from NN when the cached ones expire or accessing non-cached blocks.

An Access Token has the following format, where *keyID* identifies the secret key used to generate the token, and *accessModes* can be any combination of *READ, WRITE, COPY, REPLACE*.

- *TokenID = {expirationDate, keyID, ownerID, blockID, accessModes}*
- *TokenAuthenticator = HMAC(key, TokenID)*
- *Access Token = {TokenID, TokenAuthenticator}*

An Access Token is valid on all DNs regardless where the data block is actually stored. The secret key used to compute token authenticator is randomly chosen by NN and sent to DNs when they first register with NN. There is a key rolling mechanism that updates

this key on NN and pushes the new key to DNs at regular intervals. The key rolling mechanism works as follows.

1. NN randomly chooses one key to use at start-up. Let's call it the current key. At regular intervals, NN randomly chooses a new key to be used as the current key and retires the old one. The retired keys have to be kept around for as long as the tokens generated by them are valid. Each key is associated with an expiry date accordingly. NN keeps the set of all currently unexpired keys in memory. Among them, only the current key is used for token generation (and token validation). The others are used for token validation only.
2. When DN starts up, it gets the set of all currently unexpired keys from NN during registration. In case a DN re-starts, it's ready to validate all unexpired tokens and there is no need to persist any keys on disk.
3. When NN updates its current key, it first removes any expired ones from the set and then adds the new current key to the set. The new current key will be used for token generation from now on. Each DN will get the new set of keys on their next heartbeats with NN.
4. When DN receives a new set of keys, it first removes expired keys from its cache and then adds the received ones to its cache. In case of duplicate copies, the new copy will replace the old one.
5. When NN restarts, it will lose all its old keys (since they only existed in memory). It will generate new ones to use. However, since DN still keeps old keys in its cache till they expire, old tokens can continue to be used. Only when both NN and DN restart, does the client have to re-fetch tokens from NN.