Hadoop Security Design

Owen O'Malley, Kan Zhang, Sanjay Radia,
Ram Marti, and Christopher Harrell
Yahoo!
{owen,kan,sradia,rmarti,cnh}@yahoo-inc.com

October 2009

Contents

1 Overview 2
1.1 Security risks . . . 2
1.2 Requirements . . . 2
1.3 Design considerations . . . 3

2 Use Cases 3
2.1 Assumptions . . . 3
2.2 High Level Use Cases . . . 4
2.3 Unsupported Use Cases . . . 6
2.4 Detailed Use Cases . . . 6

3 RPC 8

4 HDFS 8
4.1 Delegation Token . . . 10
4.1.1 Overview . . . 10
4.1.2 Design . . . 10
4.2 Block Access Token . . . 12
4.2.1 Requirements . . . 12
4.2.2 Design . . . 12

5 MapReduce 14
5.1 Job Submission . . . 14
5.2 Task . . . 15
5.2.1 Job Token . . . 15
5.3 Shuffle . . . 15
5.4 Web UI . . . 16

6 Higher Level Services 16
6.1 Oozie . . . 16

7 Token Secrets Summary **17**
7.1 Delegation Token 17
7.2 Job Token 17
7.3 Block Access Token 17

8 API and Environment Changes **18**

1 Overview

1.1 Security risks

We have identified the following security risks, among others, to be addressed first.

1. Hadoop services do not authenticate users or other services. As a result, Hadoop is subject to the following security risks.
   (a) A user can access an HDFS or MapReduce cluster as any other user. This makes it impossible to enforce access control in an uncooperative environment. For example, file permission checking on HDFS can be easily circumvented.
   (b) An attacker can masquerade as Hadoop services. For example, user code running on a MapReduce cluster can register itself as a new TaskTracker.
2. DataNodes 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 DataNodes.

1.2 Requirements

1. Users are only allowed to access HDFS files that they have permission to access.
2. Users are only allowed to access or modify their own MapReduce jobs.
3. User to service mutual authentication to prevent unauthorized NameNodes, DataNodes, JobTrackers, or TaskTrackers.
4. Service to service mutual authentication to prevent unauthorized services from joining a cluster's HDFS or MapReduce service.
5. The acquisition and use of Kerberos credentials will be transparent to the user and applications, provided that the operating system acquired a Kerberos Ticket Granting Tickets (TGT) for the user at login.
6. The degradation of GridMix performance should be no more than 3%.

1.3 Design considerations

We choose to use Kerberos for authentication (we also complement it with a second mechanism as explained later). Another widely used mechanism is SSL. We choose Kerberos over SSL for the following reasons.

1. **Better performance** Kerberos uses symmetric key operations, which are orders of magnitude faster than public key operations used by SSL.

2. **Simpler user management** For example, revoking a user can be done by simply deleting the user from the centrally managed Kerberos KDC (key distribution center). Whereas in SSL, a new certificate revocation list has to be generated and propagated to all servers.

2 Use Cases

2.1 Assumptions

1. For backwards compatibility and single-user clusters, it will be possible to configure the cluster with the current style of security.

2. Hadoop itself does not issue user credentials or create accounts for users. Hadoop depends on external user credentials (e.g. OS login, Kerberos credentials, etc). Users are expected to acquire those credentials from Kerberos at operating system login. Hadoop services should also be configured with suitable credentials depending on the cluster setup to authenticate with each other.

3. Each cluster is set up and configured independently. To access multiple clusters, a client needs to authenticate to each cluster separately. However, a single sign on that acquires a Kerberos ticket will work on all appropriate clusters.

4. Users will not have access to root accounts on the cluster or on the machines that are used to launch jobs.

5. HDFS and MapReduce communication will not travel on untrusted networks.

6. A Hadoop job will run no longer than 7 days (configurable) on a MapReduce cluster or accessing HDFS from the job will fail.

7. Kerberos tickets will not be stored in MapReduce jobs and will not be available to the job's tasks. Access to HDFS will be authorized via delegation tokens as explained in section 4.1.

2.2 High Level Use Cases

1. **Applications accessing files on HDFS clusters** Non-MapReduce applications, including *hadoop fs*, access files stored on one or more HDFS clusters. The application should only be able to access files and services they are authorized to access. See figure 1. Variations:
   (a) Access HDFS directly using HDFS protocol.
   (b) Access HDFS indirectly though HDFS proxy servers via the HFTP FileSystem or HTTP get.

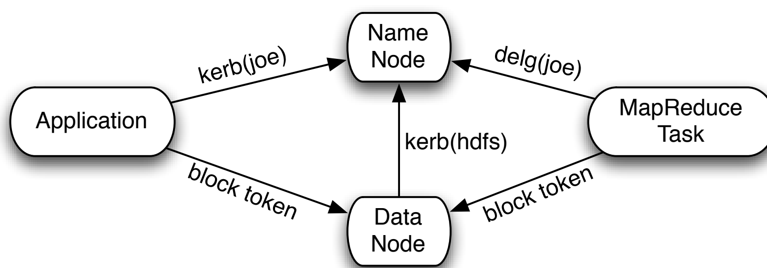

Figure 1: HDFS High-level Dataflow

2. **Applications accessing third-party (non-Hadoop) services** Non-MapReduce applications and MapReduce tasks accessing files or operations supported by third party services. An application should only be able to access services they are authorized to access. Examples of third-party services:
   (a) Access NFS files
   (b) Access ZooKeeper
3. **User submitting jobs to MapReduce clusters** A user submits jobs to one or more MapReduce clusters. Jobs can only be submitted to queues the user is authorized to use. The user can disconnect after job submission and may re-connect to get job status. Jobs may need to access files stored on HDFS clusters as the user as described in case 1). The user needs to specify the list of HDFS clusters for a job at job submission. Jobs should only be able to access only those HDFS files or third-party services authorized for the submitting user. See figure 2. Variations:
   (a) Job is submitted via JobClient protocol
   (b) Job is submitted via Web Services protocol (Phase 2)

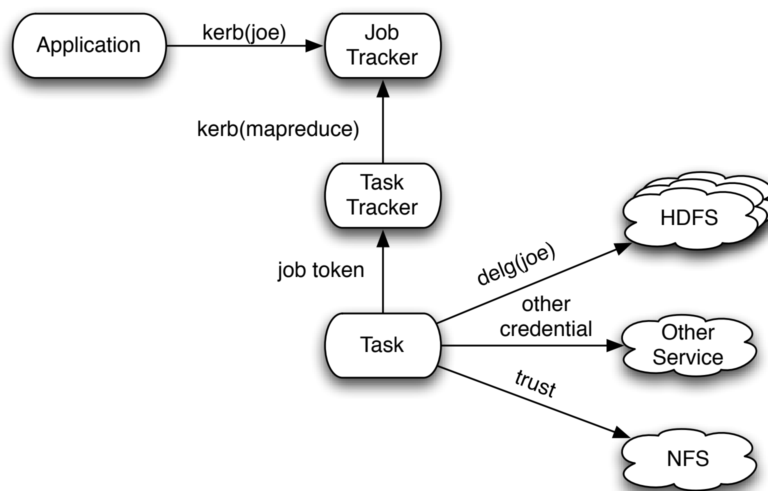

Figure 2: MapReduce High-level Dataflow

4. **User submitting workflows to Oozie** A user submits a workflow to Oozie. The user is authenticated via a pluggable mechanism. Oozie uses Kerberos-based RPC to access the JobTracker and NameNode by authenticating as the Oozie service. The JobTracker and NameNode are configured to allow the Oozie principal to act as a super-user and work on behalf of other users as in figure 3.

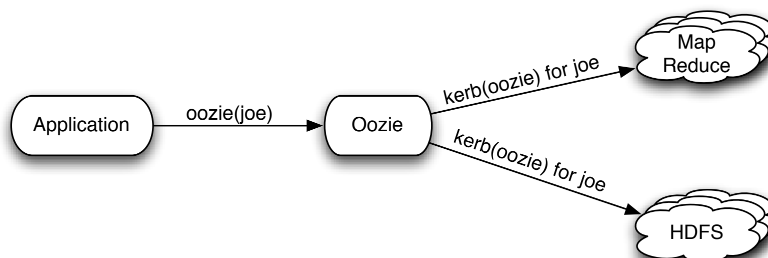

Figure 3: Oozie High-level Dataflow

5. **Headless account doing use cases 1, 2, 3, and 4.** The only difference between a headless account and other accounts is that the headless accounts will have their keys accessible via a keytab instead of using the user's password.

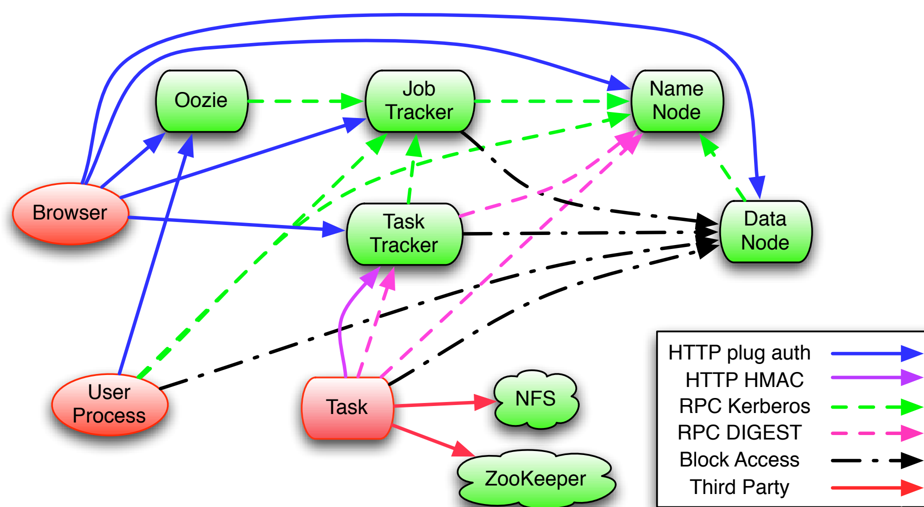

Figure 4: Authentication Dataflow

2.3 Unsupported Use Cases

The following use cases will not be supported by the first security release of Hadoop.

1. Using HFTP/HSFTP protocol to access HDFS clusters without a HDFS delegation token.
2. Accessing the Hadoop services securely across the untrusted internet.

2.4 Detailed Use Cases

1. **User principal setup.**
   (a) User principals are tied to the operating system login account.
   (b) Headless accounts are created for production jobs.
2. **Cluster setup** Admin creates and configures a Hadoop cluster in the Grid
   (a) Configures cluster for Kerberos authentication.
   (b) Admin adds/changes specific users and groups to a cluster's service authorization list
      i. Only these users/groups will have access to the cluster regardless of whether file or job queue permission allows access.
      ii. Admin adds himself and a group to the superuser and supergroup.

   (c) Admin creates job queues and their ACLs.

3. **User runs application which accesses HDFS files and third-party services.** user runs non-MapReduce applications; these applications can access
   (a) HDFS Files in the local cluster.
   (b) HDFS files in remote HDFS clusters.
   (c) Third party services using sensitive third-party credentials. Users should be able to securely store those third-party credentials on the submitting machines.
      i. ZooKeeper
4. **User submits a workflow job to a workflow engine and the workflow job needs to access the Grid** User submits a workflow job to a workflow engine. The workflow job can be long-lived and when it's run, the workflow engine may a) access files on the Grid, and b) submit Hadoop jobs. Authorization on the Grid is checked against the user's account (not that of the workflow engine).
5. **Admin submits a workflow job to a workflow engine using a headless account and the workflow job needs to access the Grid** Same as above, except that authorization on the Grid is checked against the headless account.
6. **User configures a cron job that accesses the Grid** User configures a cron job on a machine that accesses HDFS files or submits MapReduce jobs periodically. In general, the user should use Oozie instead of cron, but if they must use cron, they will have a headless Kerberos principal such as "joe/cron" and its associated keytab file. This headless principal will have the full privileges of the user's account, but without requiring an interactive login.
7. **Using HDFS proxy** Admin configures a bank of HDFS proxy servers for a group of Grid clusters located in the same data center. HDFS proxy supports HSFTP protocol and is used for server-to-server bulk data transfer using dedicated headless accounts. The authentication/authorization mechanism for HDFS proxy is IP address and a database of roles stored in LDAP. A role is configured for an approved IP address, with each IP address having at most one role. Information about a role includes its allowed IP addresses, allowed clusters and allowed file directories. Upon access, proxy sever looks up a role in LDAP based on the client IP and verifies the request is permitted by the role.
8. Queue Admin (phase 2)
   (a) **Job queue ACL setup** An administrator changes a job queue ACL to allow a specific user or group (called the queue owner) to be able to manage the ACL on the queue.

(b) **Managing job queue ACL** Queue owner changes a job queue ACL. per user/group for the cluster

3 RPC

Hadoop clients access services via Hadoop's RPC library. Currently the user's login name is sent across as part of the connection setup and is not authenticated. For authenticated clusters, all RPC's will connect using Simple Authentication and Security Layer (SASL). SASL negotiates a sub-protocol to use and Hadoop will support either using Kerberos (via GSSAPI) or DIGEST-MD5. Most Hadoop services, other than the NameNode, will only support Kerberos authentication, which will be the standard authentication mechanism.

The mechanisms are:

1. **Kerberos** The user gets a service ticket for the service and authenticates using SASL/GSSAPI. This is the standard Kerberos usage and mutually authenticates the client and the server.

2. **DIGEST-MD5** When the client and server share a secret, they can use SASL/DIGEST-MD5 to authenticate to each other. This is much cheaper than using Kerberos and doesn't require a third party such as the Kerberos KDC. The two uses of DIGEST-MD5 will be the HDFS delegation tokens in section 4.1 and the MapReduce job tokens in section 5.2.1.

The client will load any Kerberos tickets that are in the user's ticket cache. MapReduce will also create a token cache that will be loaded by the task. When the application creates an RPC connection, it will use a token, if an appropriate one is available. Otherwise, it will use the Kerberos credentials.

4 HDFS

Communication between the client and the HDFS service is composed of two halves:

- RPC connection from the client to the NameNode
- Block transfer from the client to DataNodes

The RPC connection can be authenticated via Kerberos or via a delegation token, which is described in detail in section 4.1. Delegation tokens are shared secrets between the NameNode and the client that allow subsequent authenticated access without utilizing the Kerberos Key Servers. In order to obtain a delegation token, the client must use a Kerberos authenticated connection. The block transfer is authenticated using a block access token, which is described in detail in section 4.2. Each block access token is specific to a block and is generated by the NameNode.

Although we could solely use Kerberos authentication for the NameNode RPC, the delegation tokens have some critical advantages:

1. **Performance** On a MapReduce cluster, there can be thousands of Tasks running at the same time. If they use Kerberos to authenticate to a NameNode, they need either a delegated TGT (ticket granting ticket) or a delegated service ticket. If using delegated TGT, the Kerberos KDC could become a bottleneck, since each task needs to get a Kerberos service ticket from the KDC using the delegated TGT. Using delegation tokens will save those network traffic to the KDC. Another option is to use a delegated service ticket. Delegated service tickets can be used in a similar fashion as delegation tokens, i.e., without the need to contact an online third party like the KDC. However, Java GSS-API doesn't support service ticket delegation. We may need to use a 3rd party (native) Kerberos library, which requires significantly more development efforts and makes code less portable.

2. **Credential renewal** For Tasks to use Kerberos, the Task owner's Kerberos TGT or service ticket needs to be delegated and made available to the Tasks. Both TGT and service ticket can be renewed for long-running jobs (up to max lifetime set at initial issuing). However, during Kerberos renewal, a new TGT or service ticket will be issued, which needs to be distributed to all running Tasks. If using delegation tokens, the renewal mechanism can be designed in such a way that only the validity period of a token is extended on the NameNode, but the token itself stays the same. Hence, no new tokens need to be issued and pushed to running Tasks. Moreover, renewing Kerberos tickets has to be done before current validity period expires, which puts a timing constraint on the renewal operation. Our delegation tokens can be renewed (or revived) after current validity period expires (but within the max lifetime) by the designated renewer. Being able to renew an expired delegation token is not considered a security weakness since (unlike Kerberos) only the designated renewer can renew a token. A stolen token can't be renewed by the attacker.

3. **Less damage when credential is compromised** A user's Kerberos TGT may be used to access services other than HDFS. If a delegated TGT is used and compromised, the damage is greater than using an HDFS-only credential (delegation token). On the other hand, using a delegated service ticket is equivalent to using a delegation token.

4. **Compatible with non-Kerberos authentication schemes** An additional benefit of using Hadoop proprietary delegation tokens for delegation, as opposed to using Kerberos TGT/Service tickets, is that Kerberos is only used at the "edge" of Hadoop. Delegation tokens don't depend on Kerberos and can be coupled with non-Kerberos authentication mechanisms (such as SSL) used at the edge.

Another consideration is that on a HDFS cluster, file permissions are stored on NameNode, but not on DataNodes. This means only the NameNode is able to authorize accesses to files. During a file access, a client first contacts

the NameNode to find out which DataNodes have the data blocks of the file and then connects to those DataNodes directly to access the data blocks. To enforce file permissions set on the NameNode, DataNodes need to know from the NameNode if the client is authorized to access those blocks. We plan to use an **block access token** mechanism for this purpose. When the client accesses the NameNode for files, authorization is checked on the NameNode and per block access tokens are generated based on file permissions. These block access tokens are returned to the client along with locations of their respective blocks. When the client accesses DataNodes for those data blocks, block access tokens are passed to DataNodes for verification. Only the NameNode issues block access tokens, which are verifiable by DataNodes.

4.1 Delegation Token

4.1.1 Overview

After initial authentication to NameNode using Kerberos credentials, a user may obtain a delegation token, which can be given to user jobs for subsequent authentication to NameNode as the user. The token is in fact a secret key shared between the user and NameNode and should be protected when passed over insecure channels. Anyone who gets it can impersonate the user on NameNode. Note that a user can **only** obtain new delegation tokens by authenticating using Kerberos.

When a user obtains a delegation token from NameNode, the user should tell NameNode who is the designated token renewer. The designated renewer should authenticate to NameNode as itself when renewing the token for the user. Renewing a token means extending the validity period of that token on NameNode. No new token is issued. The old token continues to work. To let a MapReduce job use a delegation token, the user needs to designate JobTracker as the token renewer. All the Tasks of the same job use the same token. JobTracker is responsible for keeping the token valid till the job is finished. After that, JobTracker may optionally cancel the token.

4.1.2 Design

Here is the format of delegation token.

```
TokenID = {ownerID, renewerID, issueDate, maxDate, sequenceNumber}
TokenAuthenticator = HMAC-SHA1(masterKey, TokenID)
Delegation Token = {TokenID, TokenAuthenticator}
```

NameNode chooses $masterKey$ randomly and uses it to generate and verify delegation tokens. NameNode keeps all active tokens in memory and associates each token with an $expiryDate$. If $currentTime > expiryDate$, the token is considered expired and any client authentication request using the token will be rejected. Expired tokens will be deleted from memory. A token is also deleted from memory when the owner or the renewer cancels the token. The

sequenceNumber is a global counter in the NameNode that is incremented as each delegation token is created to ensure that each delegation token is unique.

Using Delegation Token When a client (e.g., a Task) uses a delegation token to authenticate, it first sends $TokenID$ to NameNode (but never sends the associated $TokenAuthenticator$ to NameNode). $TokenID$ identifies the token the client intends to use. Using $TokenID$ and $masterKey$, NameNode can re-compute $TokenAuthenticator$ and the token. NameNode checks if the token is valid. A token is valid if and only if the token exists in memory and $currentTime < expiryDate$ associated with the token. If the token is valid, the client and NameNode will try to authenticate each other using their own $TokenAuthenticator$ as the secret key and DIGEST-MD5 as the protocol. Note that during authentication, one party never reveals its own $TokenAuthenticator$ to the other party. If authentication fails (which means the client and NameNode do not share the same $TokenAuthenticator$), they don't get to know each other's $TokenAuthenticator$.

Token Renewal Delegation tokens need to be renewed periodically to keep them valid. Suppose JobTracker is the designated renewer for a token. During renewal, JobTracker authenticates to NameNode as JobTracker. After successful authentication, JobTracker sends the token to be renewed to NameNode. NameNode verifies that:

1. JobTracker is the renewer specified in $TokenID$,
2. $TokenAuthenticator$ is correct, and
3. $currentTime < maxDate$

specified in $TokenID$. Upon successful verification, if the token exists in memory, which means the token is currently valid, NameNode sets its new $expiryDate$ to $min(currentTime + renewPeriod, maxDate)$. If the token doesn't exist in memory, which indicates NameNode has restarted and therefore lost memory of all previously stored tokens, NameNode adds the token to memory and sets its $expiryDate$ similarly. The latter case allows jobs to survive NameNode restarts. All JobTracker has to do is to renew all tokens with NameNode after NameNode restarts and before relaunching failed Tasks.

Note that the designated renewer can revive an expired (or canceled) token by simply renewing it, if $currentTime < maxDate$ specified in the token. This is because NameNode can't tell the difference between a token that has expired (or has been canceled) and a token that is not in the memory because NameNode restarted. Since only the designated renewer can revive an expired (or canceled) token, this doesn't seem to be a security problem. An attacker who steals the token can't renew or revive it.

The $masterKey$ needs to be updated periodically. NameNode only needs to persist the $masterKey$ on disk, not the tokens.

4.2 Block Access Token

4.2.1 Requirements

Originally in Hadoop, DataNodes did not enforce any access control on accesses to its data blocks. This made 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 DataNodes.

When users request file accesses on NameNode, 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 DataNodes, those authorization decisions are not made available to DataNodes and consequently, such accesses are not verified. DataNodes 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 NameNode can be enforced on DataNodes and any unauthorized access is declined.

4.2.2 Design

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

Block access tokens are generated using a symmetric-key scheme where the NameNode and all of the DataNodes share a secret key. For each token, NameNode 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 DataNode 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 NameNode and DataNodes know the key, no third party can forge tokens.

We considered the possibility of using a public-key scheme to generate the tokens, but it is more computationally expensive. It would have the advantage that a compromised DataNode would not have a secret that an attacker could use to forge valid tokens. However, in HDFS deployments where all DataNodes are protected uniformly (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 DataNode, they can compromise all DataNodes in the same way, without determining the secret key.

Block 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 DataNode doesn't verify it. Authentication and verification of owner ID can be added later if needed.

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

A block 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-SHA1(key, TokenID)
Block Access Token = {TokenID, TokenAuthenticator}
```

A block access token is valid on all DataNodes regardless where the data block is actually stored. The secret key used to compute token authenticator is randomly chosen by NameNode and sent to DataNodes when they first register with NameNode. There is a key rolling mechanism that updates this key on NameNode and pushes the new key to DataNodes at regular intervals. The key rolling mechanism works as follows.

1. NameNode randomly chooses one key to use at start-up. Let's call it the current key. At regular intervals, NameNode 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. NameNode 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 DataNode starts up, it gets the set of all currently unexpired keys from NameNode during registration. In case a DataNode re-starts, it's ready to validate all unexpired tokens and there is no need to persist any keys on disk.

3. When NameNode 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 DataNode will get the new set of keys on their next heartbeats with NameNode.

4. When DataNode 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 NameNode restarts, it will lose all its old keys (since they only existed in memory). It will generate new ones to use. However, since DataNode still keeps old keys in its cache till they expire, old tokens can continue to be used. Only when both NameNode and DataNode restart, does a client have to re-fetch tokens from NameNode.

There are two additional cases where the DataNode and Balancer generate block access tokens without the NameNode. The first case is the NameNode asking a DataNode to replicate some blocks to another DataNode. In this case, DataNode generates a token before sending requests to the other DataNode. In the first case, Balancer may ask a DataNode to replicate blocks to other DataNodes, and the Balancer generates the appropriate tokens.

5 MapReduce

MapReduce security is both much simpler and more complicated than HDFS security. All of the authentication from the client to the JobTracker when submitting or tracking jobs is done using Kerberos via RPC. However, the tasks of the submitted job must run with the user's identity and permissions. MapReduce stores the information about the pending and executing jobs in HDFS, and therefore depends on HDFS remaining secure.

Unlike HDFS, currently MapReduce has no authorization model other than Service Level Authorization (SLA) and the ability to restrict users to submit to certain queues. As part of increasing security, only the user will be able to kill their own jobs or tasks.

5.1 Job Submission

For job submission, the client will write the job configuration, the input splits, and the meta information about the input splits into a directory in their home directory. This directory will be protected as read, write, and execute solely by the user. The client will then use RPC to pass the location of the directory and the security credentials to the JobTracker. Because the job directory is under the user's home directory, the usage counts against their quota instead of the generic pool.

Jobs may access several different HDFS and other services. Therefore, the security credentials for the job will be stored in a Map with string keys and binary values. The job's delegation tokens will be keyed by the NameNode's URL. The security credentials will be stored in the JobTracker's system directory in HDFS, which is only readable by the "mapreduce" service principal.

To ensure that the delegation tokens do not expire, the JobTracker will renew them periodically. When the job is finished, all of the delegation tokens will be invalidated.

In order to read the job configuration, the JobTracker will use the user's delegation token for HDFS. It will read the parts of the job configuration that it needs and store it in RAM. The JobTracker will also generate a random sequence of bytes to use as the job token, which is described in section 5.2.1.

5.2 Task

The task runs as the user who submitted the job. Since the ability to change user ids is limited to root there is a relatively small setuid program written in C that launches the task's JVM as the correct user. It also moves the local files and handles killing the JVM if the task is killed. Running with the user's user id ensures that one user's job can not send operating system signals to either the TaskTracker or other user's tasks. It also ensures that local file permissions are sufficient to keep information private.

5.2.1 Job Token

When the job is submitted, the JobTracker will create a secret key that is only used by the tasks of the job when identifying themselves to the framework. This token will be stored as part of the job in the JobTracker's system directory on HDFS and distributed to the TaskTrackers via RPC. The TaskTrackers will write the job token onto the local disk in the job directory, which is only visible to the job's user. This token will be used for the RPC via DIGEST-MD5 when the Task communicates with the TaskTracker to requests tasks or report status.

Additionally, this token can be used by Pipes tasks, which run as sub-processes of the MapReduce tasks. Using this shared secret, the child and parent can ensure that they both have the secret.

5.3 Shuffle

When a map task finishes, its output is given to the TaskTracker that managed the map task. Each reduce in that job will contact the TaskTracker and fetch its section of the output via HTTP. The framework needs to ensure that other users may not obtain the map outputs. The reduce task will compute the HMAC-SHA1 of the requested URL and the current timestamp and using the job token as the secret. This HMAC-SHA1 will be sent along with the request and the TaskTracker will only serve the request if the HMAC-SHA1 is the correct one for that URL and the timestamp is within the last N minutes.

To ensure that the TaskTracker hasn't been replaced with a trojan, the response header will include a HMAC-SHA1 generated from the requesting HMAC-SHA1 and secured using the job token. The shuffle in the reduce can verify that the response came from job itself.

The advantage of using HMAC-SHA1 over DIGEST-MD5 for the authentication of the shuffle is that it avoids a roundtrip between the server and client. This is an important consideration since there are many shuffle connections, each of which is transferring a small amount of data.

5.4 Web UI

A critical part of MapReduce's user interface is via the JobTracker's web UI. Since the majority of users use this interface, it must also be secured. We will implement a pluggable HTTP user authentication mechanism. This will allow each deploying organization to configure their own browser based authentication in the Hadoop configuration files.

We considered using SPNEGO, which is the standard approach to using Kerberos authenticated web browser access. However, there is no support for SPNEGO in Jetty 6 and the standard browsers have support turned off by default. Therefore, it seems better to let each organization do its own browser based authentication.

Once the user is authenticated, the servlets will need to check the user name against the owner of the job to determine and enforce the allowable operations. Most of the servlets will remain open to everyone, but the ability view the stdout and stderr of the tasks and to kill jobs and tasks will be limited to the job owner.

6 Higher Level Services

There are several higher level services that act as proxies for user requests of the Hadoop services. Because all access to HDFS and MapReduce is via Kerberos, the proxy services will need to have Kerberos service principals. The HDFS and MapReduce services will be configured with a list of Kerberos principals that are *super-users*, which are trusted to act as other users. The proxy will authenticate as itself, but then access functionality as if it was the other user. Each super-user principal will be configured with a group that it may act on behalf of and a set of IP addresses that will be trusted for that principal. When the super-user makes an RPC connection, the first method will be *doAs* to set the user for the connection. The server must reject calls to doAs that:

1. request a user that is not a valid user for that service
2. request a user that is not in the user group that is blessed for that super-user.
3. originate from an IP address other than the blessed set for that super-user

6.1 Oozie

Oozie is a workflow manager that accepts workflows from users and submits the steps in those workflows to HDFS, MapReduce, and SSH. Oozie accepts work-

flows over a HTTP interface that uses pluggable authentication. Oozie will run with the Kerberos service principal “oozie” and use that principal to authenticate to the HDFS and MapReduce services. Both the HDFS and MapReduce services will provide new methods that allow a super-user to act on behalf of others.

We considered the possibility of storing keytab files with headless principals for each user that intends to use Oozie. The headless principals would be named “X/oozie” for each principal “X” and have the same privileges as the normal principal. This was rejected because it introduce a large operational burden to maintain the set of headless principals.

7 Token Secrets Summary

Although the tokens have been discussed independently, they share common pieces. All tokens consist of two parts, the identifier, which contains the information specific to that kind of token, and a password. The password is generated using HMAC-SHA1 on the token identifier and a secret key. The secret keys are 20 bytes and are generated using Java’s SecureRandom class.

7.1 Delegation Token

The secret is kept in the NameNode and is stored persistently in the NameNode’s state files (fs image and edits log). The persistent copy of the secret is used in case the NameNode needs to restart. A new secret is rolled every 24 hours and the last 7 days worth of secrets are kept so that previously generated delegation tokens will be accepted. The generated token is created by the NameNode and passed to the client.

7.2 Job Token

The secret is kept in the JobTracker and is **not** stored persistently. In the case of JobTracker restart, the JobTracker will generate a new secret. The job token is generated by the JobTracker when the job is submitted and is stored with the job and will be usable even after a JobTracker restart.

7.3 Block Access Token

The secrets are generated by the NameNode and distributed to all of the DataNodes, SecondaryNameNode, and Balancer as part of their heart beats. The secrets are not persistent and a new one is generated every 10 hours. The generated token is sent from the NameNode to the client when they open or create a file. The client sends the entire token to the DataNode as part of the request.

8 API and Environment Changes

For the most part, security will not involve making incompatible changes to the API's of Hadoop. However, there are some changes that can not be avoided. The identified changes are:

1. All applications that access HDFS or MapReduce will need to have a Kerberos ticket in the ticket cache. The user is responsible for calling kinit before starting the application.
2. A configuration variable will be added to the MapReduce job configurations "mapreduce.job.hdfs-servers" that contains a comma separated list of the NameNodes that will be accessed by the job. As part of job submission the framework will request a new delegation token for each NameNode. This variable will be updated automatically by FileInputFormats, FileOutputFormats, DistCp, and the distributed cache.
3. getOwner in FileStatus will be fully qualified. Thus, instead of "omalley", the owner will be "omalley@APACHE.ORG".
4. **hadoop fs -ls** will show fully qualified user names, except for users in the default realm. The default realm is configured on the client side.
5. All of the web UI pages, except for the MapReduce front page and the cluster status pages (HDFS and MapReduce) will require authentication.
6. Applications will no longer be able to work around the pseudo security.
7. MapReduce jobs will have owner, group, other permissions to limit who can kill, reprioritize, or view a job.
8. The directory structure for tasks will change so that the only the user and the TaskTracker have permission to see the work directory.
9. The downloaded copy of the distributed cache files are protected from other users, unless the source file is publically visible.
10. HFTP and HSFTP will only be supported if the application has a delegation token from the NameNode. There will not be an HTTP-based mechanism to acquire a delegation token.
11. The UserGroupInformation API will change to reflect the new usage. The JAAS Subject object will have different Principal classes.
12. For users to use Oozie, they will need to be added to the "oozie" group. The same is true of other higher level services.

For servers that access Hadoop, there will be additional changes. Those changes include:

1. The NameNode and JobTracker will need to be configured to treat the server's principal name as a superuser, which is trusted to act as other users.

2. The server will need to acquire Kerberos tickets by calling a new login method in UserGroupInformation that takes a filename that references a Kerberos keytab and the Kerberos principal name to choose out of that keytab file. Keytab files store passwords for headless applications.

3. To work as another user, the server will need to UserGroupInformation.doAs to execute a given method and the invoked RPC calls as another user.