Evaluation of Security in Hadoop

MAHSA TABATABAEI

Master’s Degree Project
Stockholm, Sweden December 22, 2014

XR-EE-LCN 2014:013
ABSTRACT

There are different ways to store and process large amount of data. Hadoop is widely used, one of the most popular platforms to store huge amount of data and process them in parallel. While storing sensitive data, security plays an important role to keep it safe. Security was not that much considered when Hadoop was initially designed. The initial use of Hadoop was managing large amount of public web data so confidentiality of the stored data was not an issue. Initially users and services in Hadoop were not authenticated; Hadoop is designed to run code on a distributed cluster of machines so without proper authentication anyone could submit code and it would be executed. Different projects have started to improve the security of Hadoop. Two of these projects are called project Rhino and Project Sentry [1].

Project Rhino implements splittable crypto codec to provide encryption for the data that is stored in Hadoop distributed file system. It also develops the centralized authentication by implementing Hadoop single sign on which prevents repeated authentication of the users accessing the same services many times. From the authorization point of view Project Rhino provides cell-based authorization for Hbase [2].

Project Sentry provides fine-grained access control by supporting role-based authorization which different services can be bound to it to provide authorization for their users [3].

It is possible to combine security enhancements which have been done in the Project Rhino and Project Sentry to further improve the performance and provide better mechanisms to secure Hadoop.

In this thesis, the security of the system in Hadoop version 1 and Hadoop version 2 is evaluated and different security enhancements are proposed, considering security improvements made by the two aforementioned projects, Project Rhino and Project Sentry, in terms of encryption, authentication, and authorization. This thesis suggests some high-level security improvements on the Centralized authentication system (Hadoop Single Sign on) implementation made by Project Rhino.

Keywords: Hadoop, security, Project Rhino, Project Sentry, splittable crypto codec
ACKNOWLEDGEMENT

I would like to sincerely thank my supervisors at Ericsson, Yi Cheng and Christian Schaefer for their great support in guiding me through the right track by setting weekly meetings while doing thesis. Yi, helped me a lot with her patience through security analytical materials and also in structuring the thesis report. Christian was really helpful in providing me with a holistic view of security concepts and also helped me to have more technical view of the security tools.

I would also like to thank my academic supervisor, Panos Papadimitratos, for his useful review of the report which was great in further improve it.

I am thanking to the security team at Ericsson to provide a good working environment and also sharing their security knowledge with me.

Finally, I am grateful to my friends and family who supported me through the thesis work.
CONTENTS

Acronyms

1 Introduction
  1.1 Goal of thesis ................................................. 3
  1.2 Related work ................................................. 3
  1.3 Structure of the report ........................................ 3

2 Overview of Hadoop
  2.0.1 Hadoop Architecture ........................................ 5
  2.0.2 Hadoop V.2 ................................................. 6

3 Existing Hadoop security mechanisms
  3.1 Hadoop Version 1 ............................................... 9
    3.1.1 Writing file into HDFS .................................... 9
    3.1.2 Reading file from HDFS .................................. 10
    3.1.3 Submitting job to MapReduce .............................. 12
    3.1.4 Map/Reduce Task Execution ............................... 14
  3.2 Detailed Review of Authentication in Hadoop .................. 17
    3.2.1 Delegation token ....................................... 17
    3.2.2 Block Access token ...................................... 19
    3.2.3 JobToken ................................................ 21
  3.3 Security Evaluation of Delegation Token and Block Access Token 22
    3.3.1 Delegation token ....................................... 22
    3.3.2 Block Access Token ...................................... 22
  3.4 Evaluation and review of Authorization in Hadoop ............ 24
    3.4.1 NameNode implements HDFS file Permission .............. 24
    3.4.2 POSIX Access Control List ................................ 24
    3.4.3 Job Queue ................................................ 25
    3.4.4 service access control list .............................. 26
  3.5 Review of Network Encryption in Hadoop ....................... 27
    3.5.1 SASL QoP for RPC connections ............................ 28
    3.5.2 Encryption during HDFS file transfer .................... 28
    3.5.3 SSL for Web consoles and MapReduce shuffle operations 29
    3.5.4 Summary of Tokens ...................................... 30
  3.6 Hadoop Version 2 ............................................. 31
3.6.1 MapReduce V2 job submission ........................................... 31
3.6.2 Communication between ResourceManager and ApplicationMaster: .................................................. 32
3.6.3 MapTask Execution .......................................................... 34
3.6.4 Job Completion .............................................................. 36
3.6.5 Summary of tokens .......................................................... 37

4 Project Rhino ................................................................. 39
4.1 Security Enhancements made by Rhino ................................. 39
4.2 Encryption Enhancements made in Project .............................. 40
   4.2.1 Key storage and management ........................................ 42
   4.2.2 MapReduce Key distribution .......................................... 42
   4.2.3 Protecting Encryption Key ........................................... 42
4.3 Authentication Enhancement :Centralized Hadoop Single Sign on 45
   4.3.1 Client interaction with HDFS in HSSO ............................. 45
   4.3.2 Service Authentication to client ..................................... 55
   4.3.3 Client interaction with Mapreduce in HSSO ...................... 56
   4.3.4 Use of Delegation token in HSSO ................................... 58
   4.3.5 Credential Management Framework: ............................... 58
4.4 Authorization Enhancements ................................................ 59
   4.4.1 The way project Rhino adds cell level security to Hbase .. 59
4.5 Evaluation -Security Concerns with Hadoop Single Sign on ...... 60
   4.5.1 Client Authentication to services ................................. 60
   4.5.2 Summary of possible attacks in HSSO ............................ 61

5 Project Sentry ................................................................. 63
5.1 Security Enhancements ...................................................... 63
   5.1.1 Authentication ........................................................ 63
   5.1.2 Authorization .......................................................... 63
5.2 Sentry Provides authorization for Hive ................................. 64
   5.2.1 Hive Architecture ..................................................... 64
5.3 Existing Hive authorization .................................................. 66
   5.3.1 1.Traditional Hive Authorization ................................... 66
   5.3.2 2.Storage Based Authorization ...................................... 67
5.4 Meta Store Security- Accessing the user passwords: ................. 68
   5.4.1 Proposed Solution: ................................................... 68
   5.4.2 Authorization process in Metastore ............................... 68
5.5 Sentry Architecture .......................................................... 69
5.6 Detailed Authorization process in Sentry .............................. 72
   5.6.1 Authentication of the user who runs Sentry .................... 72
   5.6.2 How Sentry provides fine grained authorization for users ... 72
5.6.3 Proposed improvement for authentication of the Hive users accessing Sentry 74

6 Conclusion 77
6.0.4 Future work 79
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Hadoop Version1 architecture</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Hadoop Version2 architecture</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Writing file into HDFS</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Reading File from HDFS</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Requesting for a new job</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Map-Reduce Task Execution</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Map-Reduce Task Execution (Cont’D)</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Encryption during HDFS file transfer</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Submitting Job</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>Communication between Application Master and Resource Manager</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>Launching Container</td>
<td>33</td>
</tr>
<tr>
<td>3.10</td>
<td>Map-Task Execution</td>
<td>35</td>
</tr>
<tr>
<td>3.11</td>
<td>Job Completion</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Encryption enhancement in Hadoop</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Protecting Encryption key</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Protecting Encryption key</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Client authentication and authorization process in HSSO</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Malicious user gets block access token</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Obtaining certificate from Root Certificate Authority</td>
<td>50</td>
</tr>
<tr>
<td>4.7</td>
<td>Client and server authentication using TLS</td>
<td>51</td>
</tr>
<tr>
<td>4.8</td>
<td>Cluster Access Token reuse</td>
<td>54</td>
</tr>
<tr>
<td>4.9</td>
<td>Client interaction with MapReduce in HSSO</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Hive architecture</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Hive architecture</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Sentry Architecture</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Authentication and Authorization of user who access Sentry</td>
<td>73</td>
</tr>
<tr>
<td>5.5</td>
<td>Improve Authentication in Sentry</td>
<td>74</td>
</tr>
<tr>
<td>5.6</td>
<td>Improve Authentication in Sentry</td>
<td>75</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 This table shows ACL entry types in POSIX . . . . . . . . . . . . . 25
3.2 This table shows Masking Access Permission . . . . . . . . . . . . 25
3.3 This table shows summary of token in Hadoop V1 . . . . . . . . . . 30
3.4 This table shows summary of token in Hadoop V2 . . . . . . . . . . 37

4.1 Summary of possible attacks in HSSO . . . . . . . . . . . . . . . . 61

6.1 Comparison between security system in Hadoop, Project Rhino, and Sentry . . . . . . . . . . . . . . . . . . . . . 78
ACRONYMS

SASL  Simple Authentication and Security Layer
RPC   Remote Procedure Call
ACL   Access Control List
QOP   Quality Of Protection
HDFS  Hadoop Distributed File System
YARN  Yet Another Resource Negotiator
TGT   Ticket Granting Ticket
JWT   JSON Web Token
HSSO  Hadoop Single Sign On
POSIX Portable Operating System Interface
SSL   Secure Socket Layer
AES   Advanced Encryption Standard
IDP   Identity Discovery Provider
RSA   Rivest Shamir Adelman
SHA   Secure Hash Algorithm
TLS   Transport Layer Security
HMAC  Hashed Message Authentication Code
JWS   Jason Web Signature
RDBMS Relational DataBase Management System
LDAP  Light Weight Directory Access Protocol
We live in the era of big data. One important concern is the leakage of the personal/private information. Nowadays, organizations collect information, analyse them, and make decisions based on large amounts of data. It is obvious that security should be considered while storing and processing large amount of sensitive data.

Providing security means not only protecting what data is leaving our network, but also access to data within the network should be controlled. Different security attacks can happen in a network internally and externally. Not paying attention to the security problems may lead to terrible consequences such as regulatory fines, bad reputation, and financial problems [1][4].

One of the most common platforms used to store and process large amount of data is Hadoop. When Hadoop was originally designed, security was not considered. Initially users and services in Hadoop were not authenticated; Hadoop is designed to run code on a distributed cluster of machines so without proper authentication anyone could submit code and it would be executed. Well-intended users could make mistake by deleting large amount of data. Mapreduce had no authentication or authorization so the malicious user could lower the priority of other Hadoop jobs to run his own job faster or also could kill other Mapreduce jobs. All users or programs had the same level of access to the data in the cluster, any job could access any kind of data in the cluster and any user could read any data set. Because of these security concerns and as Hadoop became a more popular platform to store and process large amount of data, the security professionals started to think of more robust security system for Hadoop. Security system of Hadoop has been improving since it was designed. Different projects have started to evolve the security of Hadoop. Some of these projects are Rhino and Sentry [2][5].

Up to now we have not seen any deep analysis and evaluation of the proposed security enhancements made by project Rhino and Project Sentry. This thesis aims to make the detailed analysis of the security enhancements made by project Rhino and Sentry, compare them with the existing security system of Hadoop, and offer a solution to further improve the security. In this report
some possible improvements are offered to enhance the authentication system in Hadoop single sign on in Project Rhino (Discussed in chapter 4). Also, in project Sentry a high level solution is offered to store the user’s credentials safe (Discussed in chapter 5).
1.1 GOAL OF THESIS

This thesis project mainly focuses on the Hadoop Security with the following goals:

1. Evaluate the existing security system in Hadoop version1 and Version2.

2. Analyse, evaluate, and compare security enhancements made by two projects, Project Rhino and Sentry, to the existing security system in Hadoop.

3. Discuss possible improvements for the security enhancements offered by Project Rhino, Sentry, and Hadoop with respect to confidentiality, authentication, and authorization.

1.2 RELATED WORK

Hadoop Security model has continued to improve since it was designed. Hadoop is becoming a popular platform to store and process large amount of data. As security professionals have pointed out to the possible security vulnerabilities and security risk within Hadoop, this results in security improvements by different projects. Lots of vendors are developing security enhanced distribution of Hadoop which is an improvement to the existing security system of Hadoop.

Project Accumulo, provides fine-grained authorization and provides data access at cell-Level to ensure that only authorized users can view and manipulate data points. Project Knox provides perimeter security and makes Hadoop Security setup easier. It also supports authentication and token verification scenarios for Hadoop. Currently Hadoop is offered to the customers as a collection of different independent services with different security mechanisms. This will make it difficult for the customers to interact with Hadoop. The goal of this project is to make a unified coverage for all the existing Hadoop projects [6].

1.3 STRUCTURE OF THE REPORT

The rest of the report is organised as follows: Chapter 2 includes an overview of Hadoop Version 1 and Hadoop Version 2 mainly describing the architecture
of Hadoop. Chapter 3 describes the existing security system in Hadoop version 1 and Hadoop Version 2. Besides, evaluation and further suggestion to improve the existing security system is provided. Chapter 4 discusses project Rhino and evaluates the enhancements made by this project to improve the Hadoop security. Besides, some solutions are offered to improve the security proposals made by Project Rhino. Chapter 5 includes the study and evaluation of the security enhancements for Hadoop made by project Sentry. It also includes some solutions to improve the security proposals made by project Sentry. Chapter 6 concludes the report and suggests about the future work.
Hadoop is an open-source framework to store and process large amounts of data. Hadoop has been developed under an Apache License. It is implemented to scale from a single cluster to thousands of servers. Hadoop consists of two main modules: the MapReduce and the Hadoop Distributed File System (HDFS) [4].

MapReduce is a programming model which assigns work to each of the nodes in the cluster and process data. HDFS is a fault tolerant distributed file system that connects the file systems on many data nodes to make one big file system. It is fault tolerant.

Below, some benefits which Hadoop brings are described [7]:

1. Scalable: Based on the requirements in different organizations, new nodes can be added without the need to change formats of the data, the way jobs are written or the way data is loaded.
2. Fault tolerant: When one of the data nodes fails, work will be redirected to another location where the data has been stored and the process will continue uninterrupted.

2.0.1 Hadoop Architecture

This section describes main parts of Hadoop (as shown in Figure 2.1). The main components in Hadoop Version1 are [8][9]:

JobTracker: It directs the MapReduce task to the nodes in the cluster that contain data.
TaskTracker: It accepts Map-Task and Reduce-Task from the JobTracker.
NameNode: NameNode makes a directory of all files in HDFS so that it is aware where each file is located in the HDFS.
DataNode: DataNode stores data in Hadoop File system.
In Hadoop Version2, HDFS remains pretty much the same as it was in Hadoop Version1. Hadoop Version2 consists of two main parts: The cluster resource management, which is called Yet Another Resource Negotiator (YARN), and MapReduce.

YARN consists of two different modules: Resource manager and Node Manager, see Figure 2.2. Resource manager is only responsible to allocate resources to various jobs. It does not need to take care of job monitoring or updating status. Resources such as memory, bandwidth, and the CPU time are stored in one unit which is called Resource Container. In YARN, the TaskTracker was replaced with the NodeManager [10].

MapReduce capabilities of the JobTracker were replaced by the MapReduce Application Master. This manages each MapReduce job and terminates when the job completes.

**Hadoop Version2 enhancements**

Hadoop Version 2 is developed to make some enhancements to Hadoop Version 1 [10].
Some of new features of Hadoop V2 will be described briefly:

- **NameNode federation**: In Hadoop V1 HDFS consists of a single NameNode and many DataNodes. Name Node is responsible for most of the file system operations such as creating a file, writing or reading a file. On the cluster with lots of nodes, only one single NameNode to keep the directory of files and managing the DataNodes could cause bottleneck problems. In Hadoop V2 multiple NameNodes could help to scale the IO operation.

- **Resource Allocation Model**: In Hadoop V1 nodes in the cluster were capable of running predefined number of Map slots and Reduce slots. This resource allocation is not feasible since different tasks have different slot requirements. Hadoop V2 solves this problem by offering more flexible resource allocation. It provides resource container which each of the containers can supply resources dynamically based on the task requirement. Besides, YARN is capable of running not only MapReduce program but also any other distributed computing program. This will provide more flexibility in choosing the proper distributed program based on the task requirements.
In this section, different steps of how the client reads/writes file into HDFS and how the user submits the job will be explained. Meanwhile, the existing security mechanisms in Hadoop during these processes will be evaluated.

### 3.1 HADOOP VERSION 1

#### Writing file into HDFS

First, the user logs into the enterprise network will be authenticated via Kerberos and obtains Kerberos Ticket Granting Ticket (TGT). Then, the user creates the file on the local file system. The different steps in writing the file are:

1. User sends Remote Procedure Call (RPC) request to NameNode to create the file.
2. User is authenticated to NameNode through Kerberos service ticket and will receive delegation token.
3. After client is authenticated successfully by NameNode, NameNode will make a record of the new file. NameNode sends block access token as well as the block ID and block locations to the authenticated client.
4. The client sends datablocks to DataNode through Data transfer protocol which is a Streaming Socket Protocol. DataNode verifies the block access token and then writes the data into blocks. During this process, client connects to NameNode repeatedly to get the location of allocated blocks and block access tokens through Kerberos service ticket (See Figure 3.1) [11].
First, user logs into the enterprise network and will get the Kerberos TGT. Different steps in reading the file are described below, See Figure 3.2.

1. Client contacts NameNode through RPC to request block locations. Client will be authenticated to NameNode using Kerberos Service Ticket.

2. After client authenticated by NameNode, NameNode sends client the delegation token.

3. Name Node will check the file permission, if that is ok it will return the block IDs, block locations, and block access tokens for the first few blocks.

4. The client will send the block access token to the DataNode using data transfer protocol.

5. After that DataNode verifies the block access token and it sends the block to the client. Client talks to NameNode continuously using Kerberos Service Ticket, receives the location of other blocks and block access tokens [11].
Figure 3.2: Reading File from HDFS
3.4.3 Submitting job to MapReduce

First, the user logs in to the network, is authenticated by Kerberos, and receives the Kerberos TGT. User/Client submits a job to analyse the data in the file(s). Different steps in submitting jobs represented in Figure 3.3 are depicted as below:


2. Then, JobTracker returns the new ID to the client.

3. Client copies the resources needed to run the job to JobTracker’s system directory in HDFS (in a directory named after the job id). In its first RPC call to NameNode, Client gets a delegation token in response and copies it to the same directory.

4. Client then submits the job to the JobTracker and informs the JobTracker that job is ready to execute.

5. JobTracker creates a job token.

6. JobToken is used to identify tasks to the framework and is stored together with job resources in its system directory in HDFS. JobTracker initializes the job and creates map/reduce tasks for it. After that user logs out [11].
Figure 3.3: Requesting for a new job
3.4.4 Map/Reduce Task Execution

In this section, different steps in executing Map/Reduce task are described (Figure 3.4).

1. TaskTracker sends heartbeat to the JobTracker to receive task.

2. TaskTracker receives the task and makes a local directory for the task, copies job resources which includes JobToken and delegation token to its local directory. Maptask runs through the job owners account and reads data from HDFS. Following steps illustrates the reading procedure after TaskTracker receives the task.

3. Maptask contacts NameNode through RPC and authenticates itself to NameNode with the delegation token.

4. NameNode checks the file permission, if it is ok will return block IDs, block locations and also block access tokens to the Map Task.

5. MapTask sends block access token to DataNode.

6. DataNode verifies the token, if ok transfers the block to MapTask.

7. MapTask analyzes the data and stores the map output in the local task directory.


9. Map/Reduce task reports status to its TaskTracker, using job token.

10. TaskTracker sends heartbeats to JobTracker, including the status of all tasks run by the TaskTracker. JobTracker combines status updates to produce a global view of the status of all jobs and their tasks.

11. Client polls JobTracker for the latest status of the job via RPC, using Kerberos service ticket. JobTracker changes the status to “successful” after receiving notification that the last task of the job is complete.

12. JobTracker requests NameNode to cancel the job’s delegation token. JobTracker cleans up its working state for the job and instructs TaskTrackers to do the same (Figure 3.5).
Figure 3.4: Map-Reduce Task Execution
Figure 3.5: Map-Reduce Task Execution (Cont’D)
3.2  Detailed Review of Authentication in Hadoop

- The client and the Hadoop services (NameNode, DataNode, JobTracker, and TaskTracker) authenticate each other using Kerberos.

- Hadoop Services authenticate each other with the use of Kerberos.

- MapReduce task and NameNode authenticate each other with the use of delegation Token.

- MapReduce task and TaskTracker authenticate each other using JobToken [12].

The following section describes the details of Delegation token, Block Access token, and Job Token [12].

3.2.1  Delegation token

The user who submits the job will be authenticated to NameNode by Kerberos. User might log out and job can be performed later. Jobs need to be authenticated to NameNode to access HDFS. So, the kerberos credentials should be passed to JobTracker to be authenticated to NameNode. There are some problems with transferring Kerberos Credentials to JobTracker.

First of all, on the Hadoop cluster many tasks are running concurrently. If tasks use Kerberos to authenticate to NameNode, they need Kerberos TGT or Kerberos Service Ticket. When using Kerberos TGT, KDC could become a bottleneck since every task needs to obtain Kerberos Service ticket using Kerberos TGT. This will result in performance problem. Using delegation token could save the network traffic to KDC since the users are authenticated once, receive delegation token and use it for later authentication to the services [12].

Second problem is related to the credential renewal. Tasks needs to have the user’s (who is running the task) Kerberos TGT or Kerberos service ticket to be authenticated to Kerberos. Both the TGT and Kerberos Service ticket can be renewed to be used. But, during Kerberos renewal a new TGT or service ticket will be created which needs to be propagated to all the tasks which are running. With the use of delegation token the renewal of the token is done in a way that the validity period on the token can be extended on the NameNode while the token stays the same itself. So, there is no need to create new tokens and send them to the running tasks [13].
Format of delegation token

First the user is authenticated to NameNode using Kerberos. After authentication is done, client will get the delegation token from NameNode. Delegation token will be given to Job to be used for the later authentications [11].

Format of the delegation token is described below [11]:
TokenID = ownerID, renewerID, issueDate, maxDate, sequenceNumber
TokenAuthenticator = HMAC-SHA1(masterKey, TokenID)
Delegation Token = TokenID, TokenAuthenticator

master Key:
masterkey is chosen by NameNode randomly. NameNode use this masterkey to create delegation token, known only by NameNode. Masterkey is stored in NameNode.

maxDate:
NameNode stores all the active delegation tokens in its memory. each of the tokens is associated with expiry date. When current time exceeds the expiry date, token is considered expired. Expired tokens can not be used in authentication and will be removed from the memory.

Sequence Number:
Sequence number is a global number which is used in NameNode and will be incremented when each delegation token is created and will insure the uniqueness of the delegation token.

Renewer ID:
Delegation tokens need to be renewed to be valid. Renewer ID is used to identify the designated renewer. If the designated renewer is JobTracker, JobTracker first authenticates itself to NameNode. After authentication is done, JobTracker sends the token to be renewed by NameNode.

After client is authenticated using Kerberos credentials, receives the delegation token from NameNode. Delegation Token is used for the client for later authentication to NameNode.

Delegation token used for authentication

This section describes how delegation is used by Task to be authenticated to the NameNode:
1. First client sends TokenID to NameNode

2. NameNode then recomputes the TokenAuthenticator using the TokenID and the master key. Then NameNode will check if the token is among existing valid tokens in memory. If the token exists in the memory and also the current time is less than the Expiry Date, then the token is considered valid.

3. After token is validated, NameNode and client start authenticating each other. They use Token Authenticator as the shared secret between them to authenticate each other using DIGEST-MD5. If authentication fails, it means that the client and NameNode do not have the same authenticator.

3.2.2 Block Access token

Block Access tokens are used with DataNode to provide access control to its DataBlocks. Initially, Hadoop did not provide access control to DataBlocks. So, any malicious user could read or write into datablocks just by using the BlockID. Block access token will solve the mentioned problem.

**Format of Block Access token**

Block access token structure is described below [11]:

TokenID = expirationDate, keyID, ownerID, blockID, accessModes

TokenAuthenticator = HMAC-SHA1 (key, TokenID)

Block Access Token = TokenID, TokenAuthenticator

KeyID in TokenID is used to identify the key which is used to generate the block access token. Also, access modes could be READ, WRITE, COPY, REPLACE.

Block access tokens are generated by the NameNode. NameNode shares a symmetric-key (Key used in generating TokenAuthenticator) with all the DataNodes which is used in generating the Block access token. NameNode computes the keyed hash of the Token ID (by using the shared secret) called TokenAuthenticator.

The Block access token consisting of both TokenID and TokenAuthenticator will be sent to DataNode. DataNode will recompute the TokenAuthenticator using Token ID and the key which it shares with NameNode.
If the calculated TokenAuthenticator is the same as TokenAuthenticator included in the Block Access Token, then token is considered as valid since just the NameNode and DataNodes have the shared secret to compute the TokenAuthenticator.

Proposal: Public key cryptography in generating Block Access Token

Another way to generate the block access tokens is to use Public-key Cryptography. NameNode will use a pair of public-private key pairs and will use the private key to sign the tokens. This signature will be included in the token. Then, DataNodes will only need the public key to verify the signature. Just NameNode is aware of the private keys to create the signature. So, DataNodes do not need to keep any secrets.

Comparing with the symmetric key encryption, public-key cryptography provides simpler key management and can maintain safety of keys. In symmetric key encryption all the DataNodes share a secret key with NameNode to provide authentication with the Block Access Token. If one DataNode is compromised, then the malicious user can have access to the DataBlocks in all other DataNodes which is a security threat. However, Public-Key cryptography is more expensive than the symmetric cryptography in creating the signature and providing security.

Block access tokens are short-lived tokens. When a block access token expires, a new token could replace the old one. Block access tokens will be cached in the client’s memory and are not stored in the disk.
3.2.3 JobToken

JobToken is created by JobTracker to provide authentication between Map/Reduce tasks and TaskTrackers. This token is used when task reports its status to TaskTracker (Figure 3.5). Another usage of JobToken is when reduceTask wants to fetch the map output from the TaskTracker. After mapTask finishes, map output is given to the TaskTracker. Then each reducTask in that job contacts TaskTracker to fetch the map output. JobToken is used by reduceTask to authenticate itself to the TaskTracker. This way it is insured that no other unauthorized user can fetch the map output.

When job is submitted, JobTracker will generate a secret key. This secret key is stored in JobTracker. It is not stored persistently. If the JobTracker restarts, a new secret will be generated [14].

Format of JobToken

JobToken format is the following [14]:

JobToken ID = (JobID)
TokenAuthenticator= HMAC-SHA1 (Key, TokenID)
JobToken = TokenID, TokenAuthenticator

When job is submitted, JobTracker will generate the JobToken. JobToken (TokenID, TokenAuthenticator) will be stored as part of the job credential in JobTracker directory in HDFS. TaskTrackers will read the JobToken from the HDFS directory and will write the JobToken in the local disk of job directory. This directory is only visible to the one who submitted the job.

Also, Map/Reduce Task will read the JobToken from the local directory. Map/Reduce task sends HMAC-SHA1 (URL + current time + TokenAuthenticator) to the TaskTracker to be authenticated to the TaskTracker. TaskTracker computes the HMAC-SHA1 (URL + current time + TokenAuthenticator) and will compare it with the HMAC-SHA1 (URL + current time + TokenAuthenticator) which was sent in the request. If the two hash value are the same, then it proves that Mapreduce has got the same TokenAuthenticator which is shared with TaskTracker; After that, Task Tracker and JobTracker start authenticating each other using HMAC-SHA1 cryptographic hash function (RFC 2104) with the TokenAuthenticator as the shared secret.
3.3 SECURITY EVALUATION OF DELEGATION TOKEN AND BLOCK ACCESS TOKEN

3.3.1 Delegation token

Delegation token is a secret which is shared between NameNode and client. So it should be protected while being transferred. One way to secure delegation token when transferring it from NameNode to the client, is to secure the transmission using Transport Layer Security (TLS) / Secure Socket Layer (SSL). This way it can protect stealing of the delegation token. This is discussed in detail in section 4.3 (Figure 4.6 and Figure 4.7). It should be mentioned that in the existing security system in Hadoop, Delegation token is sent from NameNode to the client through RPC protocol with Simple Authentication and Security Layer (SASL) client authentication [11]. Also, Quality Of Protection (QOP) is supported by encrypting delegation token which provides confidentiality of the token.

When task uses delegation token to be authenticated to the NameNode, delegation token itself is never sent in clear text. Only token ID is sent by Task to NameNode. So, at this step there is no risk for delegation token to be eavesdropped.

Another security risk is related to the misuse of delegation token which the authenticated client could share delegation token with other users who have not been authenticated. This problem could be solved by binding the client ID to the delegation token so when the client presents delegation token to the NameNode it will save the client ID corresponding to that delegation token. If another user except the authenticated client wants to use the delegation token, NameNode compares the user ID with the authenticated client ID. If they are not the same, then the user will not be authenticated. (This is discussed in detail in Figure 4.8).

3.3.2 Block Access Token

Block access token which includes TokenID and TokenAuthenticator, is sent in clear text from NameNode to Task and then to the DataNode to check if the tasks can have access to the blocks in DataNode. So, there is the risk that the malicious user eavesdrops the Block Access token and uses BlockAccess token to access the Datablocks in DataNodes.

One security solution could be to secure the transmission using TLS/SSL so
that the BlockAccess token cannot be eavesdropped by the malicious user. This solution is discussed in detail in chapter 4.3
3.4 EVALUATION AND REVIEW OF AUTHORIZATION IN HADOOP

3.4.1 NameNode implements HDFS file Permission

HDFS file permission is based on the traditional UNIX permission bits. Each file has three permission sets which include the read, write, and execute. Also, three different user classes will be defined including: Owner, Group, and Others.

Based on the class the user belongs to, HDFS sets access permission. For instance, if the user is the owner HDFS sets owner class permission. If the user is not the owner but is the member of the group class, HDFS sets the group class permission. If the user is not the owner or is not the member of the group class, then other class permission will be enforced [15].

Although this model is sufficient for many organizations, it has some limitations too. One of the limitations is that in the traditional UNIX model only one group is defined which the permissions could be defined for that group. Thus, there will be a problem if there is a need to define more groups and set permission for each of them.

To solve this problem and provide more fine grained permission model, Portable Operating System Interface (POSIX) access control list is offered [15]. Access control list will provide the solution to define different permission models for different hierarchy of the users and groups. This will allow that for each file, different groups and users could have different permissions.

3.4.2 POSIX Access Control List

In POSIX Access Control List, six types of Access Control List (ACL) entries will be defined. Each of these entries, defines permission for one user or a group of users. The "owner" will define permissions for the user who owns a file or directory. The "owning group" will define permission for the file’s owning group. Every "named user" entry defines permission for the user who is specified in the entry’s qualifier list. Every "named group" entry defines permission for the users who are specified in the entry’s qualifier field. "Other" entry defines permission for all of the other users (Figure 3.1) [15].

Mask entry provides more fine grained access for users and puts limitation on the permission which are granted by the named user and named group.
Table 3.1: This table shows ACL entry types in POSIX

<table>
<thead>
<tr>
<th>Type</th>
<th>Text Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>user::rwx</td>
</tr>
<tr>
<td>named user</td>
<td>user:name:rwx</td>
</tr>
<tr>
<td>owning group</td>
<td>group::rwx</td>
</tr>
<tr>
<td>named group</td>
<td>group:name:rwx</td>
</tr>
<tr>
<td>mask</td>
<td>mask::rwx</td>
</tr>
<tr>
<td>other</td>
<td>other::rwx</td>
</tr>
</tbody>
</table>

This is done by defining which permissions are effective and which are masked. Permissions which are defined in one of the entries which were mentioned above and also in the mask, are considered effective. Permissions defined only in the actual entry or just in mask, are not considered effective. This means that these permissions should not be granted. Permissions which are defined in the owner and owning group are always considered effective [15].

Table 3.2: This table shows Masking Access Permission

<table>
<thead>
<tr>
<th>Entry Type</th>
<th>Text Form</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>named user</td>
<td>user:geeko:r-x</td>
<td>r-x</td>
</tr>
<tr>
<td>mask</td>
<td>mask::rw-</td>
<td>rw-</td>
</tr>
<tr>
<td>effective perm-</td>
<td>effective permissions</td>
<td>r--</td>
</tr>
</tbody>
</table>

3.4.3 Job Queue

A job queue Access Control List controls who can submit the job, kills the job or modify the queue properties. In the default mode of Hadoop any user can submit the job. Also, only Administrator can kill the job or see the job. Thus, Job Queue ACL provides fine grained authorization in submitting job [16].

ACL for jobs are defined with the cluster administrator; Administrator instructs Jobtracker to reload its own access control list. Access control list defines which users or groups can submit job. Access Control list is defined in hadoop-policy.xml which consists of the name of the users, groups, and their permission to submit job. JobTracker will check the access control list before...
allowing the user to submit job. Also, JobTracker and TaskTracker will check the access control before allowing the user to see the job configuration files or modify the job credentials [16].

### 3.4.4 service access control list

Controls which users and groups can use the service such as NameNode and JobTracker. Service access control list works pretty much the same as Job access control list [16].
Some organizations are concerned about the security of sensitive data. Encrypting the sensitive data can provide security and confidentiality for data. Initially in Hadoop, everything was transferred in clear text which caused some security problem. Clients and DataNodes in Hadoop transfer data with the use of Data Transfer Protocol. This transfer is not encrypted either. When transferring sensitive data such as bank Account information, data should be encrypted. Otherwise, active eavesdroppers could tamper with the confidentiality and data integrity. But, Hadoop provides the capability to encrypt the network communication. Network encryptions in Hadoop are listed below [17]:

- Simple Authentication And Security Layer (SASL) Quality Of Protection (QOP) for RPC connections
- Encryption during HDFS file transfer
- SSL for Web consoles and MapReduce shuffle operations

In the following sections these three methods will be discussed.
3.5.1 | SASL QoP for RPC connections

SASL is used as the security protocol for Hadoop RPC operations. Simple Authentication and Security Layer provides security such as authentication, encryption or message integrity.

SASL framework can provide different quality of protection services. For instance, SASL mechanism such as MD5-DIGEST have the option to provide either authentication, message integrity, confidentiality or any mixture of these three properties [17].

3.5.2 | Encryption during HDFS file transfer

As it was mentioned above, NameNode issues Block Access Tokens for the clients who uses this token to be authenticated to the DataNode and request for DataBlocks. By Default Data Transfer Protocol does not use RPC and SASL quality of Protection [17].

To solve this problem existing Data Transfer Protocol is wrapped with SASL handshake. When this configuration is enabled, NameNode generates Data Encryption Keys and sends it along with Block Locations, Block IDs, and Block Access token to the clients. This will be used for MD5-DIGEST as credentials. When using encryption for HDFS Data Transfer, RPC encryption should also be enabled to protect the encryption keys while being exchanged because encryption keys are transferred over Hadoop RPC, See Figure 3.6.

![Figure 3.6: Encryption during HDFS file transfer](image-url)
SSL for Web consoles and MapReduce shuffle operations

After map task finishes, map-output will be given to TaskTracker. Reduce task will contact TaskTracker and get its section of output through HTTP. Here there could be the risk that other users access map outputs. In order to prevent malicious user access the map-output, the reduce task will compute Hashed Message Authentication Code (HMAC) -SHA1 of the URL request and the current time stamp using JobToken as the secret. Task tracker will compute HMAC-SHA1 of the URL using job token and compare it with the HMAC-SHA1 hash value sent in the request. If the computed HMAC-SHA1 is the one in the URL, TaskTracker will serve the request. TaskTracker will put the computed HMAC-SHA1 of the URL in the header response. This will allow the reduce task to authenticate the TaskTracker [17].

HTTPS is used to protect the shuffle protection by encrypting HTTP. In order to encrypt HTTP protocol, SSL needs to be configured. Since the nodes in Hadoop cluster will be added and removed, if a node is removed from the cluster its public certificate key should be removed from all other nodes. If one node is added to the cluster, its public certificate key should be added to all other nodes. Java Key store should be implemented in a way to be able to reload certificate public keys if certificate public keys are added or deleted [17].
### Summary of Tokens

In this section a summary of the tokens used for authentication/authorization of the services/clients in Hadoop V1 is provided in Table 3.3.

<table>
<thead>
<tr>
<th>Token type</th>
<th>Created By</th>
<th>Created for</th>
<th>Created by Using</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delegation token</td>
<td>NameNode</td>
<td>Client and Map/Reduce Task</td>
<td>A master key known only to NameNode</td>
<td>Authentication with NameNode</td>
</tr>
<tr>
<td>Block access token</td>
<td>NameNode</td>
<td>Client and Map/Reduce task</td>
<td>A secret key shared by NameNode and all DataNodes</td>
<td>Access control with DataNode</td>
</tr>
<tr>
<td>Job token</td>
<td>JobTracker</td>
<td>Map/Reduce tasks</td>
<td>A secret key generated by JobTracker</td>
<td>Authenticating with TaskTracker</td>
</tr>
</tbody>
</table>

Table 3.3: This table shows summary of token in Hadoop V1
3.6 | HADOOP VERSION 2

In Hadoop2, HDFS Read and Write file are the same as HDFS V1. So, in this section Job submission in MapReduce 2 from the security point of view will be described.

3.6.1 | MapReduce V2 job submission

This section describes how the client submits job to MapReduce V2, See Figure 3.7.

1. First, Client contacts ResourceManager to initiate the process of running the application. In this RPC step mutual authentication between the client and ResourceManager is done using SASL Kerberos.

2. Resource manager responses with a unique Application ID with the information about the cluster resource capabilities which client will need in requesting resources to run the application. Client will also get the Resource Manager Delegation Token.

3. Then the client submits the application to Resource Manager. For this RPC step, Mutual authentication between client and Resource Manager is done using SASL Kerberos [18].

![Figure 3.7: Submitting Job](image-url)
3.6.2 | Communication between ResourceManager and ApplicationMaster:

This section describes the different steps done when Resource manager communicates with ApplicationMaster, See Figure 3.8.

1. After Resource manager receives the Application submission content from the client, it will find the container which meets the resource requirement to run the Application Master. Then it will contact Node manager for the container to start the Application Master Process. This will be done by SASL Mutual authentication between Resource Manager and Node manager through RPC using Kerberos Tickets.

2. First Application Master needs to register itself with the (Scheduler component of) Resource Manager. This is done through RPC protocol with SASL client authentication using Application Master token. This step includes a handshake with information such as the RPC port that Application Master will be listening, the tracking URL to monitor application status.

3. Application Master sends HeartBeat to the Resource Manager and requests for resource allocation.

4. After Resource Manager receives the request from Application Master, it computes the list of containers based on the scheduling policy and will send the allocation Response to Application Master. After that Application Master is ready to complete the task which was assigned to it. This will be done through RPC-SASL mutual authentication between Application Master and the Resource Manager using Application Master token.

5. Application Master will contact the associated node to start its task with the help of Resource List. This will be done through mutual authentication between Application Master and Node Manager using NodeManager token, See Figure 3.9 [18].
Figure 3.8: Communication between Application Master and Resource Manager

Figure 3.9: Launching Container
3.6.3 | MapTask Execution

This section discusses different steps while MapTask runs and reads data from the HDFS (as shown in Figure 3.10).

1. MapTask contacts NameNode and requests for Block Access Token. This will be done through RPC SASL mutual authentication using delegation token.

2. NameNode checks file permission, if ok will return Block Access Token and Block ID.

3. MapTask will send Block Access Token to DataNode using Block data transfer protocol with the use of block token.

4. After Data Node verifies it, it will send block to MapTask. This will be done through Block data transfer protocol with the use of block token.

5. MapTask will save the Map Output in the local directory.

6. Reduce Task will read the Map Output from via HTTP with the use of job token [18].
Figure 3.10: Map-Task Execution
3.6.4 | Job Completion

In this section different steps in completing the job are described (Figure 3.11).
- Map Reduce task reports its status to Application Master using Job Token.
- When the mapreduce job is complete, Application Master will send its status to Resource Manager through RPC using ApplicationMaster token [18].
### Summary of tokens

In this section summary of the tokens used for authentication/authorization of the services/clients in Hadoop V1 are described in Table 3.4.

<table>
<thead>
<tr>
<th>Token type</th>
<th>Created By</th>
<th>Created for</th>
<th>Created by Using</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Master token</td>
<td>Resource Manager</td>
<td>Application Master</td>
<td>A Secret key generated by Resource Manager</td>
<td>Authentication with Resource Manager</td>
</tr>
<tr>
<td>NodeManager token</td>
<td>Resource Manager</td>
<td>Application Master</td>
<td>A Secret Key shared by Resource Manager and all Node Managers</td>
<td>Authentication with Node Manager</td>
</tr>
<tr>
<td>Job token</td>
<td>Application Master</td>
<td>MapReduce Task</td>
<td>A secret key generated by Application Master</td>
<td>Authentication with Application Master</td>
</tr>
<tr>
<td>Block access token</td>
<td>NameNode</td>
<td>Client and MapReduce Task</td>
<td>A secret key shared by NameNode and all DataNodes</td>
<td>Access control by DataNodes</td>
</tr>
<tr>
<td>NameNode Delegation token</td>
<td>NameNode</td>
<td>Client and MapReduce task</td>
<td>A master key generated by NameNode</td>
<td>Authentication to NameNOde</td>
</tr>
</tbody>
</table>

*Table 3.4: This table shows summary of token in Hadoop V2*
Project Rhino is an opensource project which aims to improve the data protection mechanism in Hadoop from different security aspects.

4.1 SECURITY ENHANCEMENTS MADE BY RHINO

Encryption Enhancements

Project Rhino provides the ability to encrypt or decrypt the data stored in HDFS. Hadoop cluster consists of a large number of nodes which are shared among different groups in the organization. Sensitive data such as bank account information or any personal data needs to be stored encrypted. Thus, it is important to store the data securely in HDFS.

Project Rhino designs a framework which can support different cryptographic algorithms to implement encryption. One cryptographic algorithm which is used in this framework is Advanced Encryption Standard (AES). Cryptographic codec implementations will be implemented based on AES [19].

Authentication enhancements

Project Rhino supports different authentication mechanisms such as public-key cryptography. In current security system in Hadoop users and services are authenticated using Kerberos. Project Rhino provides authentication of the users into a centralized service by developing Hadoop Single Sign on. Authentication enhancements made by project Rhino will be described more in Hadoop Single Sign On Section [19].

Authorization Enhancements

Project Rhino adds cell level access control in Hbase. Existing security system in Hadoop supports table level or column level security. In the following sec-
tion, more detailed steps are described about encryption, authorization, and authentication [19].

4.2 | ENCRYPTION ENHANCEMENTS MADE IN PROJECT

Compression can be used in storing files in HDFS. It will reduce size of the files, saves the space for storing files, and make the data transfer faster through the system. Thus, there will be reduced system load which results in improved performance by speeding up the time needed to finish mapreduce jobs. Various compression formats can be used in Hadoop. Different compression codecs have different characteristics. For instance, some of them are fast when compressing the file while the others are slow. Thus, the suitable compression codec could be selected based on the different requirements when compressing the file.

Project Rhino offers the option to either compress, encrypt or both compress and encrypt the files stored in HDFS. The process of encrypting the file in HDFS is as follows: First the whole sensitive file which needs to be stored at HDFS is compressed and then it is encrypted. One encryption algorithm used is AES. AES cryptography is used under the splittable crypto codec. Then the whole file is divided into different splits. Each split is normally the same size as the HDFS block size and is measured by bytes. Each split is stored in a different datablock in a datanode. It is assumed that MapReduce will first decrypt and then decompress each of the blocks independently and starts processing them. This will provide the option that mapreduce can process different block independently and in parallel which results in higher speed, See Figure 4.1 [20].

Assumption: To improve the performance it is assumed that the whole sensitive file is encrypted at once rather than splitting the file and encrypt each split individually. This way encryption will be performed at once instead of doing it manytimes for many small splits. This will reduce the burden on the system by performing encryption one time.
Figure 4.1: Encryption enhancement in Hadoop
4.2.1 | Key storage and management

In this framework the real key which is used to encrypt or decrypt the data will not be stored in the plain text with the data. Instead the framework provides the ability to store a keyprofile with the data. This key profile could be an ID or alias which is stored with the encrypted data. Thus, whenever the key is needed to decrypt an encrypted file, keyprofile is used to get the keys which were used to encrypt the file. Whenever key expires or there is the need to use a new key, the new key will be used to encrypt the new data and there is no need to decrypt the data which was encrypted with old key and re-encrypt it with the new key again [21]. In large databases it can save a lot of time and reduce the burden of doing decryption and re-encryption of the previously encrypted file. This system provides the facility to implement different storage mechanisms such as the third party key management to keep the keys.

4.2.2 | MapReduce Key distribution

This framework provides different options to store keys and retrieve them. One way to store keys is that the key material can be stored on the protected keystore on each node. Second option is to store key material on a dedicated security server and then use the remote key management [21].

The second option has the risk of single point of failure for the dedicated security server. If the dedicated security server is down or out of order we lose access to all the keys stored in the keystore. But, in the first case which the key is stored securely on each node if one node is compromised, keys stored on other nodes will not be affected.

After keys are retrieved they will be stored as part of job configuration and will be encrypted by cluster specific key to prevent man-in-the-middle attack.

4.2.3 | Protecting Encryption Key

Keys which are used in cryptographic functions need to be secured. Keys need to be stored in a protected key management system. They will be retrieved from Keystore and will be submitted as a part of the job credentials. They will be transported from job submission process to MapReduce task. Keys need to be secured during this process [21]. This process is illustrated in Figure 4.2 and Figure 4.3.
Figure 4.2: Protecting Encryption key

Figure 4.3: Protecting Encryption key
1. Client calls key Provider.

2. Key Provider reads the encryption key from Keystore. Keystore is protected by a password. Also keys which are stored in keystore are protected by password. So, to read encryption keys from keystore one should know the keystore password and also password for encrypted keys to decrypt them.

3. Client will receive the key.

4. Client encrypts the key with cluster public key and stores it in the job credentials. Job credentials are any kind of information which is needed to run the job. Then, client submits the job.

5. TaskTracker will read the keys from Job Credentials and decrypt the keys with the cluster private key.

6. TaskTracker will put the decrypted keys in Env.

7. Map/reduce task will read the key from Env.
4.3 AUTHENTICATION ENHANCEMENT : CENTRALIZED HADOOP SINGLE SIGN ON

Centralized Hadoop Single Sign On (HSSO) provides pluggable authentication mechanism which provides flexibility for users to use different authentication mechanisms. HSSO also provides credential management framework which makes management of secrets, keys, and credentials easy. This framework will be discussed later [22].

HSSO will make authentication into a centralized service that is trusted in the Hadoop cluster. This centralized service makes authentication of the clients to multiple services in Hadoop more simplified. Using this system clients do not need to be authenticated separately by each service. Instead, the single server can do the authentication of clients for all the services. This centralized service will also use public-key cryptography to issue cryptographically verifiable tokens [22].

4.3.1 Client interaction with HDFS in HSSO

In this section client interaction with HSSO and HSSO architecture will be discussed in detail. One component used in HSSO is Identity Discovery Provider (IDP) discovery. Clients send IDP discovery request to HSSO to find out the accepted and trusted authentication mechanisms which the client can use. Authentication and authorization process in HSSO are described in detail below (as shown in Figure 4.4).

1. Client sends IDP request to IDP Discovery service in HSSO. This request contains Target Cluster ID, available ID types and the User Domain Identifier. IDP discovery service uses the best match algorithm to find the list of trusted IDP’s from the set of authentication providers. Then it sends back the IDP discovery response which contains IDP name, IDP description, and Authorization URL to get access to the cluster.

2. IDP discovery service will send the IDP response to client.

3. Client will be authenticated to IDP at the IDP authorization URL.

4. Based on different implementation Identity provider redirects user into the authorization URL and will send back the authentication response with the IDP specific token. IDP token consists of attributes such as 1-Username, 2-Groups, and 3-domain.
5. Client sends Cluster Access Token to the HSSO service provider and passes IDP token. HSSO service provider validates and normalizes the IDP token into a format which can be used to create the requested token. This process is as follows:

First, HSSO verifies the idp-token to see if it is valid according to the IDP specifications. Second, it normalizes the idp-token. After normalizing the token, HSSO will send the normalized token to the token builder and will request Cluster Access Token. Cluster Access Token is signed and encoded based on the JSON Web Token (JWT) practices [22].

**JWT** is a Javascript Object Notation based token encoding [23]. It provides a secure way to transmit security information and identity between the domains. Client can use JWT Bearer token to request for access token using an existing trust relationship. JSON Web Token is composed of three parts: 1-Header, 2-claim Set, and 3-Signature [23].

JSON header consists of the signing algorithm. In computing the Jason Web Signature (JWS) the two inputs used are the encoded JWT Header and encoded JWT claim set. JWT header and JWT claim are encoded based on Base64url encoding which provides resilience and prevents encoding changes. One signing algorithm to compute the JWS is Rivest Shamir Adelman (RSA)-Secure Hash Algorithm (SHA) 256 algorithm. JSON claim contains information about JWS which could be the target of the token, token issuer, token life time, and the time token was created [23].

6. HSSO service provider sends the response to the client. This response includes the expiration period of the token, name of the client who receives the response, issuer of the token, target service, token endpoint URL to get service access token, and the actual cluster access token. The actual cluster access token code is signed and encoded based on the JWT. Client will receive the response and will extract the cluster access token from the signed code.

7. Then the client will send the cluster access token to the token end point URL to request for a service access token. HSSO service token endpoint validates and accepts the cluster access token. Then it will create service access token and sends it to the client which will be used by client to be authenticated by a specific service.

8. After receiving the Service access token, client will send a request to the target Hadoop service at the provided URL and will also send the service access token within HTTP authentication in the format of JWT Bearer [24].
Figure 4.4: Client authentication and authorization process in HSSO [24]
Security Evaluation - Scenario 1

In HSSO client is not authenticated to the server again when presenting the cluster access token. So, anyone who has obtained the signed cluster access token can use it to request for service access token and access the resources. This can be considered as one of the design problems in HSSO, See Figure 4.5.

Attack Assumption:
These assumptions are related to the attacker and resources available to the attacker:

- Attacker has access to the network between the client and HSSO service provider, and client and NameNode/DataNode. Attacker might eavesdrops the communication between any of the mentioned parties. It is assumed that attacker does not access the communication between the HSSO Service Provider and NameNode/DataNode.
- Attacker has unlimited amount of resources to launch the attack.

To prevent the malicious user from accessing the cluster access token, one way is to secure the communication between the client and HSSO with TLS/SSL.
in two steps. First, when the client presents IDP token to HSSO to get the
cluster access token, the transmission should be encrypted by TLS/SSL. This
way we can protect the token being eavesdropped by any malicious user. Sec-
ond, when client presents the cluster access token to HSSO to be authenticated
and receives the service access token, the transmission should be protected by
the use of TLS/SSL so that the malicious user cannot steal the block access token.

TLS is chosen as a security protocol to solve the above problem since there is the
need to secure the transmission over untrusted system. TLS solves this problem
by encrypting the transmission. TLS also authenticates the server and client
which suits well in this scenario (Client who presents the cluster access token
needs to be authenticated to the HSSO server again). TLS provides option for
authentication mechanisms, encryption algorithms, and hashing algorithms in
a secure channel.

There are also some drawbacks using TLS. For instance, TLS causes increased
processor load. Public key operations are CPU-intensive. So, performance
varies while using TLS. Performance also depends on how often the connections
are established and how long they last. TLS uses great resources to establish
the connection. Also, TLS causes administrative overhead. TLS environment is
complex and needs maintenance. System administrator needs to configure the
system and manage certificates [25].

To protect the transmission using TLS, the below steps should be performed,
See Figure 4.6:

1. All the identity providers and Servers will receive the certificate from the
   Root Certificate Authority (RCA).
2. After that the client receives the certificate from the Identity Provider.
3. Then client and server start authenticating each other using certificates
   via TLS.
Detailed authentication of the clients and server using TLS is described in figure 4.7.
Figure 4.7: Client and server authentication using TLS

1. First client sends Client Hello message and specifies TLS version plus the list of ciphersuits it supports.

2. HSSO Server sends Server Hello message with the TLS version and chosen ciphersuits.

3. HSSO Service Provider sends its certificate plus its public key information ($k_s^+)$.

4. HSSO Server sends client certificate request to the client.

5. HSSO Server sends Server Hello Done message to the client and informs the client that the initial negotiation is done.

6. Client sends it certificate, public key information ($k_c^+$), generates a Pre-MasterSecret (Random number), encrypt it with the server public key and sends it to the server. (Pre MasterSecret is a random number generated...
by the client used to generate the MasterSecret to encrypt the whole message transfer).

7. After that client sends Certificate verify message used to verify the client’s Certificate. This message includes the signed hash messages by the client private key which could be decrypted by HSSO Server using clients public key. Then client and HSSO server use the previously exchanged random number and the PreMasterSecret to generate a common secret called "MasterSecret". From now on all other data transmitted between client and HSSO are encrypted using MasterSecret.

8. client sends change cipher spec message to HSSO Server to inform the server to change to encrypted mode.

9. Then client sends a Finished message to HSSO Server. Finish message contains the HMAC (MasterSecter,handshake messages). HSSO server decrypts this message using the previously exchanged secrets. If HSSO server fails to decrypts and verify this message, then connection is torn.

10. Then HSSO server sends change cipher spec message to the client.

11. HSSO server sends Finish message to the client (As described in step 10).

12. Now the secure connection is established. Authentication, encryption, and MAC algorithms are determined by HSSO server in the HSSO Server Hello message. Compression and random values are also negotiated in the hello messages. So, cluster access token is the shared MasterSecret between client and HSSO server.

13. After encrypted data is transferred, client and HSSO server send close-notify message to inform that the connection is closed.

*Security Evaluation- Scenario 2*

Based on what was discussed in the section 4.3.1, the cluster access token issued by HSSO Service Provider does not contain client ID. So, there is the risk that the misbehaving client shares the token among others to access the services.

Assumption:

1. Client in this scenario is considered as authenticated user.

2. It is assumed that the client does not share his ID with others.

3. It is assumed that the authenticated client misuse the cluster access token.
To prevent misusing of the cluster access token, solution is to bind client ID to cluster access token to be authenticated to the HSSO Service Endpoint. This scenario is discussed below (Figure 4.8):

1. First Client sends Identity Provider token to the HSSO Service provider.

2. After HSSO Service provider authenticates the user, it sends the cluster access token bounded to the Client ID to the client.

3. Client sends the cluster access token to the HSSO Service Endpoint to request for the service access token to access a specific service.

4. After HSSO service end point authenticates the client, saves the Client ID used for requesting a specific service access token in a database and sends the services access token to the client. This database is used to store and keep track of the ID of the client who accessed this service.

5. Client sends service access token to the Hadoop service and request for service.

6. After Hadoop Service authenticates the client, provides him with the services.
Figure 4.8: Cluster Access Token reuse

If another user except the trusted client sends the client’s cluster access token again to the HSSO Service endpoint to request for a specific service access token, HSSO Service endpoint checks in its database if this cluster access token bounded to the client ID is existing in its database or not. If it is existing this means that this cluster access token has been used by the trusted Client before. So, HSSO Service end point does not grant service access token to that user.

Below is the example of the cluster access token bounded to the Client ID which was discussed in Scenario 2.

```
"exp": "1366404909",
"sub": "joe",
"aud": "HSSO",
"iss": "https://www.example.com/gatekeeper/v1/authorize/BASIC",
"tke": "https://www.example.com/gatekeeper/v1/token",
"code": "eyJhbGciOiJtZWRpc2VjdCBdc3FjaGUgRGV2YXJpbiBJZCIKICAsImlkIjozODQ5MTM5NjQ1Niwic291dC5mb3VuZC1hdXRoX2RleF90aGuGc3VwYXVsdC0iX3N0dWRhY2siLCAiZW52IjogIjEzMDkxMDY2NjU2XCIKICB7InN1YiI6W1sibW9kZXMiOnsicRkFzIjp7InJ1dCI6XCJ2ZXJzaW9uIixcImlkZiI6XCJkYXRhZ2FkZSIsXCJ0aGUibm90IGRlaWJveC1uc2ZjLXN0cm9vbiIsXCJlcGlrZS10b2tlbnQiOiJyb21tZWRkZSIsXCJzaWduZXN0cml0eSIsXCJyb21lX3ZhbGlkX3Byb290IjowfX0sXCJ0b2tlbnpiX2lkIjoiMTExLjIiLCAibW9kZXQiOm51bGwsXCJlbmNvZGluZ3MiOlwiXCIKSW52ZW52TW9kZXg7fQ==
```

Another security concern is that when the client presents the service access token to the services, client and service are not mutually authenticated. So, there could be the risk that services which are not trusted might access the service access token which is presented by the client.

In order to prevent untrusted services have access to the service access token, TLS/SSL with server side authentication could be used while transferring the service access token. This way the service could be authenticated to the user which prevents untrusted services having access to the service access token.

There are various authorization extensions to TLS handshake. These extensions could be carried in the client and server hello messages to make sure that both parties support the authorization data types. Then if both the parties support the authorization data types, authorization information such as attribute certificates or Security Assertion Markup Language (SAML) are exchanged in the data handshake messages [24].
Client interaction with Mapreduce in HSSO

Client interaction with Mapreduce starts with step 8 as was illustrated in Figure 4.4. In this part, client interaction with Mapreduce is described. Also, the interaction of Map/Reduce task with the NameNode and DataNode is explained. Besides, Delegation token needs to be extended to contain service access token. More information about delegation token is provided in the section below [24].

When creating MapReduce job, client writes the job configuration, the job classes, input splits and metadata in the specific directory. Client submitting the job will get the delegation token from NameNode which will be given to the job to be authenticated to NameNode. Detailed interaction of the client with mapreduce is shown in Figure 4.9 and explained below [24]:

1. Client submits job to job tracker.
2. Task tracker sends heart beat to the job tracker and requests for task.
3. Job tracker sends back task to the task tracker.
4. Maptask will provide NameNode with the delegation token which it received from the client and will ask for the block access token.
5. NameNode will verify the delegation token and sends back the block access token to the Maptask.
6. Maptask will then send the block access token to the DataNode.
7. DataNode verifies block access token and sends back data blocks to Map Task.
Figure 4.9: Client interaction with MapReduce in HSSO
4.3.4 | Use of Delegation token in HSSO

In order for the job to access HDFS and other Hadoop services job should have access to the credentials. So, client needs to request delegation token from NameNode to provide it to the jobtracker to access HDFS. Name Node receives the service access token from the client and will use that to create delegation token. NameNode extracts the needed information from service access token and adds them to the Delegation token. This information includes attributes which are needed for fine grained authorization policy such as groups which the client belongs to or the roles of the client. This will require some modification in the delegation token.

Delegation Token Format

Delegation token consists of 1: Expiration Date, 2: KeyID, 3:Owner ID, 4:Block ID, 5:access modes, and 6:service access token. Delegation token includes the claims signed by HSSO service and it is trusted. Delegation token will be signed by the NameNode private key. This will make more simple key management of the signature keys. public-Key cryptography allows the signer of the keys to make public keys available by publishing them. Credential Management framework will provide facility to publish and get new public keys [24].

4.3.5 | Credential Management Framework:

Credential Management framework includes a library to secure storage of credentials to access Hadoop services. This storage is protected with password and the keys stored in it are also encrypted [24].
Project Rhino extends the access control for Hbase on a per cell basis. Providing Authorization on a cell rather than just table or column gives more fine grained authorization for users. One user might have the permission to access some cells in one column but the same user might not have access to the other cells in the same column [26].

The way project Rhino adds cell level security to Hbase

Project Rhino adds cell-level security. While inserting data to each cell, visibility tag expressions will be inserted to each cell. When user reads the data, user will provide his own authorization information including the group he belongs to, the role he has, and the permission which he has according to the groups and roles. Then the users authorization information will be checked against the visibility tag and based on that the user will be allowed to have access to that cell or not. So, to have cell based security authorization checks need to be made at each cell. This will incur some performance overhead. However, it provides more fine grained authorization [26].
4.5 | EVALUATION - SECURITY CONCERNS WITH HADOOP SINGLE SIGN ON

There are some security threats with the Hadoop Single Sign on in Project Rhino. Some of them will be described below:

4.5.1 | Client Authentication to services

Cluster access token is used by the authenticated client to request for service access token. Cluster Access token is used to represent the authenticated identity. It includes the authentication characteristics as well as strength and number of different factors which are involved in authentication. Its structure consists of expiration Date, Key ID, and Owner ID.

After client provides the HSSO service provider with the Identity token, Service provider will normalize and validate the token and send the response to the client. This response contains: Expiration period, Authenticated user, Target service which the code is provided, token issuer, token end point to get the service access token, and the cluster access token code. Token is encoded and signed based on JWT Bearer. The client will then present the cluster access token to the HSSO service provider to request for the service access token [27].
### 4.5.2 Summary of possible attacks in HSSO

<table>
<thead>
<tr>
<th>Security breach</th>
<th>Attack</th>
<th>Reason</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client and server are not mutually authenticated again when client presents the cluster access token to server</td>
<td>Eavesdrop</td>
<td>Design Problem</td>
<td>Encrypt the transmission using TLS/SSL</td>
</tr>
<tr>
<td>Block access token does not contain the ID of the authorized client</td>
<td>Reuse cluster access token</td>
<td>Design problem</td>
<td>Bind clients ID to the cluster access token</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of possible attacks in HSSO
Apache Sentry is one of the projects which provides fine grained authorization for the users of services such as Hive and Cloudera. Sentry is integrated with the SQL query framework Apache Hive and Cloudera Impala. This chapter discusses the security enhancements for existing security system in Hadoop made by Project Sentry.

5.1 SECURITY ENHANCEMENTS

5.1.1 Authentication

Sentry Service uses the existing Kerberos authentication in Hadoop. Other authentication mechanisms might be added if necessary [28].

5.1.2 Authorization

Without Sentry, Hadoop uses coarse-grained HDFS file authorization. In file level authorization in Hadoop, user can either have access to the file or not. So, in HDFS file authorization there is not the possibility to define multiple groups to have different access levels to data [28].

But, Sentry provides the ability to define multiple groups and roles for the authenticated users. With Sentry, authenticated users have the ability to control and have access to data based on the privileges [28].

Sentry provides fine-grained access control on data and metadata in Hadoop. Sentry can provide access control for table and view scopes at the different privilege levels which includes Select, Insert, and All. This will allow administrator to use view to control access to columns and rows [28].

Sentry provides role-based authorization. We can define multiple groups have access to the same set of data at different privilege models. For instance, for a specific set of data, we can permit Administrator the right to view all columns, Employee to have access to non-sensitive columns, and the Students...
not to read any columns [28].

5.2 | **Sentry Provides Authorization for Hive**

Sentry provides Authorization for Hive. Hive is a datawarehousing component on top of Hadoop. Hive structures the data into the database concepts such as table, rows, columns, and partitions. In this section Hive authorization will be checked first to see what Sentry adds to Hive Authorization. To better understand how hive works, the Hive architecture is presented in Figure 5.1 [29].

![Hive Architecture](image)

**Figure 5.1:** Hive architecture [29]

5.2.1 | **Hive Architecture**

Query language which is used in Hive is called HiveQl. HiveQl is used to query data.

Hive metastore:
It is used to store metadata about the tables and also the partitions in the Relational Database Management System (RDBMS).
Hiveserver2:
Hiveserver2 includes Hive execution engine. HiveServer2 allows multiple users to be connected to hive at the same time. Also, it performs authentication which prevents malicious users accessing the data. It uses Kerberos or Light Weight Directory Access Protocol (LDAP) to perform authentication for the clients who use Hive to access data [30]. Hive popularity is because it makes it able to access the data in the Hadoop cluster same as relational databases such as MYSQL. So, the authorization which is used in some traditional RDBMS could be expected from Hive too [30][31].

In RDBMS users cannot have access to data and metadata outside SQL engine. However, Hive allows users to have access to both data and metadata outside SQL engine. For instance, it allows users to have access to metadata outside HiveServer. This characteristic in Hive makes it flexible for some users but also makes security more complex in Hive. In RDBMS only SQL has access to data and metadata. So, the authorization model just needs to secure the SQL operation. But, in Hive both SQL operation and metadata operations which are exposed by metastore service need to be secured. See Figure 5.2 [32].

![Figure 5.2: Hive architecture](image-url)
5.3 EXISTING HIVE AUTHORIZATION

5.3.1 Traditional Hive Authorization

The default authorization model for Hive is the same as RDBMS style authorization which is based on users, groups, and roles and granting each of them the permission to operate on database or table. In this model permissions are stored directly in RDBMS metastore.

Evaluation

RDBMS Authorization model in Hadoop causes some problems due to the below reasons [33]:

Contrary to RDBMS, Hive cannot control all the data underneath it. The data is kept in a number of files and the file system has a separate permission authorization system based on HDFS file permission which could be different from Hive authorization system. For instance, Hive might grant permission to a user to have access to a file. But, HDFS file permission could prevent the same user to have access to the same file. So, here there is inconsistency between Hive authorization system and HDFS file permission which will make some problem accessing to files.

One solution for the above problem is that the user who wants to access files which are stored in HDFS and are accessed by Hive could be authenticated by Hive authentication system to prevent the malicious user accessing file in HDFS.
2. Storage Based Authorization

The default authorization model does not prevent malicious users accessing data which they should not. It just controls the good users not to do actions which they are not allowed to. Access to Meta Store Server needs to be secured. Storage based authorization provides security for metastore [34].

In this model, an authorization interface is implemented which makes the permission of the underlying file system as the basis for permission on each database, table, or partition. In fact, the security will be based on the access rights to the underlying storage system [34].

In Hive, when using a file system to store data, there will be a directory which corresponds to a database or table. With storage based authorization, the read or write permission which a user or a group will have for a directory, will determine on the permission the user could have on the database or table. For instance, for the user to alter table operation, he should have permission on the table directory before being able to do any operation. When the database uses a file system which has Unix-style permissions like HDFS, Read or Write permission could be set for the user, groups, or other.

Storage based authorization is used to add authorization to the metastore server API calls. It can only protect metastore contents against changes by malicious users. But, it cannot provide fine grained access control [34].
5.4  **META STORE SECURITY– ACCESSING THE USER PASSWORDS:**

It is possible that many clients need to have access to the same metastore. If many clients have access to the same metastore backed by the database, like MySQL, credentials of the users who connect to the same metastore will be visible in hive-site.XML. So, the malicious user can get access to user’s credentials and have access to the metadata even if the underneath data has been protected by HDFS access controls. Thus, it is necessary to protect the metastore [35].

Assumption: It is assumed that attacker has access to the file hive-site.XML. Attacker is not assumed to have access to the key used to encrypt the hash of the passwords.

5.4.1  **Proposed Solution:**

One way to secure Hive-site.XML is to use HMAC to hash the user’s password with salt. Salt could be generated using Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). They provide high level of randomness and are unpredictable. That is why CSPRNG could be used to create salt. Each password should have a unique salt.

These are different steps to store the password secure:

1. User creates account.
2. Generate long random salt with the use of CSPRNG.
3. Add Salt to the password and hash it with one standard cryptographic hash function such as SHA256.
4. Calculate the hash using the keyed hash algorithms such as HMAC.
5. The hashed password is stored in a directory which is readable only by the system administrator.

5.4.2  **Authorization process in Metastore**

After the user connects to the metastore server, the metastore will decide the username of the connecting user and the group which is associated with it.
This information is used to decide if the user should access the metadata which has been requested by comparing the privileges needed to do the task with the privileges which the user has. These tasks will be done by checking if the user is the member of the group which the privilege belongs to. It will also be checked that if the user or any of the groups he belongs to has a role which grants that privilege or not [36].

**Issue:**

In the existing authorization of Hive, authorization checks are done in the clients based on the HDFS file permission. Thus, there is the risk that users use other applications to read/write into files in HDFS.

**Solution**

As it was mentioned in the evaluation of Default authorization, to solve the problem of malicious users accessing into data that are stored in HDFS and are used by Hive, one solution is that Hive set permission for these files so that the file can be accessed by Hive.

5.5 **Sentry Architecture**

In order to have secure and fine-grained authorization for data and metadata in Hive, Sentry is developed. Sentry architecture is shown in Figure 5.3. Sentry is consisting of a core authorization provider and the binding layer. The authorization provider consists of Policy Engine and Policy Provider. Policy Engine is responsible for validating and evaluating the policy. Policy Provider is responsible for parsing the policy. The binding layer consists of a pluggable interface which can talk to the policy engine.

File-based provider is one of the Sentry components which is implemented to understand a specific policy file format. The policy file will be saved either in local file system or HDFS. Policy file stored in HDFS will be protected by HDFS file system permission.

Sentry architecture is extensible. Any developer can develop a binding for a different component such as Pig. Each component binding in Sentry implements a privilege model for that specific binding. For instance, the hive binding implements hive privilege model which performs fine-grained authorization
for row/column in a table $[37][3]$. 
Figure 5.3: Sentry Architecture

[3]
5.6 | DETAILED AUTHORIZATION PROCESS IN SENTRY

Following steps give detailed overview of how authorization is done in Sentry

5.6.1 | Authentication of the user who runs Sentry

1. First the user who runs Hive is authenticated based on the Kerberos Credentials.

2. Then user creates the policy file and stores it in HDFS or Local file system. This file defines groups, roles, privileges.

5.6.2 | How Sentry provides fine grained authorization for users

This section describes how the users of Hive will be authenticated and how Sentry provides authorization for the users of Hive.

1. The user who wants to access Hive will be authenticated based on Kerberos credentials.

2. Based on the userID, Policy Engine performs LDAP or ShellBased group mapping.

3. After that the list of groups are resolved for the user.

4. Then the policy engine will read the policies through policy provider from the policy file and then grant access based on the group membership, See Figure 5.4.
Figure 5.4: Authentication and Authorization of user who access Sentry
5.6.3 | Proposed improvement for authentication of the Hive users accessing Sentry

Sentry provides fine-grained authorization for the users of Hiveserver2. Here Hadoop Single Sign On which was proposed by Project Rhino (Refer to Section 4.3 for more information) could be used for the authentication of the users to Hiverserver 2 or any service which is bound to Sentry (as shown in Figure 5.5).

Assumption:

- Sentry system and bounded service (Hive) is considered one cluster.
- It is assumed that the service access token are securely cached.

Different steps to authenticate the user to Hive through Hadoop Single Sign is discussed below (Figure 5.6):
Figure 5.6: Improve Authentication in Sentry

1. First Hive User provides identity provider token to the HSSO Service Provider.

2. HSSO Service Provider authenticates the user and provides him with the cluster access token.

3. User sends the cluster access token to HSSO Service Endpoint.

4. HSSO Service Endpoint authenticates the client and sends service access token to him.

5. User sends the service access token to Hive and is authenticated to Hive.

6. After the user is authenticated, it can access Hive.

Service access token is cached for future authentication of the user to Hive. This way user does not need to be authenticated to Hive anytime he wants to access Hive. User is authenticated once through HSSO and receives the service access token from HSSO. This Service access token is used for future authentication of the users to any services such as Hive that are bound to Sentry.
This will reduce the burden on the system by reducing the times which the user should be authenticated to the service. However, service access tokens which are cached should be kept secure to prevent malicious user access them.
Most organizations nowadays are involved with big data. Large amount of data is stored and processed, then the results are used for data analysis.

One popular platform used to store large amount of data is Hadoop. There are some gaps with the existing security system in Hadoop. As security plays an important role in keeping the sensitive data protected, the goal of this thesis is to evaluate security for Hadoop.

Two of the projects which worked on improving the security for Hadoop are called Project Rhino and Sentry. The security enhancements made by these two projects from different security aspects such as encryption, authentication, and authorization have been studied.

Based on what was studied, Project Rhino offers good solution to encrypt the data at rest in HDFS if it is needed; thus, it provides confidentiality of data. Besides, it has implemented Hadoop Single Sign On which is used to reduce the burden on the framework by authenticating each of the client once and providing them the service access token which could be stored and used later for authentication to the service. Also, project Rhino provides cell-based security which provides more fine grained access control on cell level, in addition to table or column level access control.

Project Sentry provides fine grained authorization to data and metadata in Hadoop. It offers role-based authorization by defining different groups to have different access levels to different datasets. Project Sentry develops a framework which can be bound to services such as ImpalaD and HiveServer2 to provide fine-grained access for their users.

Table 6.1 summarizes the existing security system in Hadoop as well as the security enhancements done in project Rhino and Sentry.
<table>
<thead>
<tr>
<th><strong>Encryption</strong></th>
<th><strong>Existing Hadoop</strong></th>
<th><strong>Project Rhino</strong></th>
<th><strong>Sentry</strong></th>
</tr>
</thead>
</table>
|                | 1- Files stored in HDFS are not encrypted.  
2- SASL QOP for the RPC connections.  
3- SSL for web consoles and MapReduce shuffle operation.  
4- Encryption during HDFS file transfer. | 1- File stored in HDFS can be encrypted.  
2- Splittable Crypto Codec.  
3- Encryption keys are stored and encrypted in Keystore | Not any information found for the encryption of messages transmitted between Sentry service and the client who wants to access Sentry. |
| **Authentication** | 1- Kerberos mutual authentication between client and service  
2- Kerberos mutual authentication between Services (NameNode, DataNode, Jobtracker, and TaskTracker)  
3- Delegation token to authenticate MapReduce tasks to NameNode.  
4- Job Token to authenticate MapReduce task to TaskTracker | Hadoop Single Sign On in Project Rhino provides the below features:  
1- Supports different authentication mechanisms such as Public Key Cryptography.  
2- Single Server can do the authentication of clients for all services.  
3- Provides Credential Management Framework to store user credentials and encryption keys. | Uses the existing Kerberos system in Hadoop to authenticate users. |
| **Authorization** | 1- NameNode enforces HDFS file permission.  
2- DataNode authorizes access to blocks using block access token.  
3- Access control list for Job queues controls which user and groups can submit job to the job queue and change queue properties.  
4- Access control list for services controls which users and groups can use the service (Jobtracker, NameNode)  
5- Hbase provides table/column level access control | Extends Access control on Hbase on a cell basis | Fine Grained Authorization  
Role Based Authorization |
6.0.4 Future work

Some future work to improve security system of Hadoop is to evaluate security system in Hadoop Version2 in more detail. Some area to consider could be to work on the security tokens needed to do authentication or authorization between services and clients or between services when executing jobs.
BIBLIOGRAPHY


[31] shreepadma Venugopalan, “Hive authorization.”


