Hadoop security

https://library.oreilly.com/book/0636920033332/hadoop-security/toc.xhtml

Chapter 1. Introduction

Back in 2003, Google published a paper describing a scale-out architecture for storing massive amounts of data across clusters of servers, which it called the *Google File System (GFS)*. A year later, Google published another paper describing a programming model called *MapReduce*, which took advantage of GFS to process data in a parallel fashion, bringing the program to where the data resides. Around the same time, Doug Cutting and others were building an open source web crawler now called Apache Nutch. The Nutch developers realized that the MapReduce programming model and GFS were the perfect building blocks for a distributed web crawler, and they began implementing their own versions of both projects. These components would later split from Nutch and form the Apache Hadoop project. The ecosystem[1] of projects built around Hadoop's scale-out architecture brought about a different way of approaching problems by allowing the storage and processing of *all* data important to a business.

While all these new and exciting ways to process and store data in the Hadoop ecosystem have brought many use cases across different verticals to use this technology, it has become apparent that managing petabytes of data in a single centralized cluster can be dangerous. Hundreds if not thousands of servers linked together in a common application stack raises many questions about how to protect such a valuable asset. While other books focus on such things as writing MapReduce code, designing optimal ingest frameworks, or architecting complex low-latency processing systems on top of the Hadoop ecosystem, this one focuses on how to ensure that all of these things can be protected using the numerous security features available across the stack as part of a cohesive Hadoop security architecture.

Security Overview

Before this book can begin covering Hadoop-specific content, it is useful to understand some key theory and terminology related to information security. At the heart of information security theory is a model known as *CIA*, which stands for *confidentiality*, *integrity*, and *availability*. These three components of the model are high-level concepts that can be applied to a wide range of information systems, computing platforms, and—more specifically to this book—Hadoop. We also take a closer look at *authentication*, *authorization*, and *accounting*, which are critical components of secure computing that will be discussed in detail throughout the book.

Warning

While the CIA model helps to organize some information security principles, it is important to point out that this model is not a strict set of standards to follow. Security features in the Hadoop platform may span more than one of the CIA components, or possibly none at all.

Confidentiality

Confidentiality is a security principle focusing on the notion that information is only seen by the intended recipients. For example, if Alice sends a letter in the mail to Bob, it would only be deemed confidential if Bob were the only person able to read it. While this might seem straightforward enough, several important security concepts are necessary to ensure that confidentiality actually holds. For instance, how does Alice know that the letter she is sending is actually being read by the right Bob? If the correct Bob reads the letter, how does he know that the letter actually came from the right Alice? In order for both Alice and Bob to take part in this confidential information passing, they need to have an *identity* that uniquely distinguishes themselves from any other person. Additionally, both Alice and Bob need to prove their identities via a process known as *authentication*. Identity and authentication are key components of Hadoop security and are covered at length in Chapter 5.

Another important concept of confidentiality is *encryption*. Encryption is a mechanism to apply a mathematical algorithm to a piece of information where the output is something that unintended recipients are not able to read. Only the intended recipients are able to *decrypt* the encrypted message back to the original unencrypted message. Encryption of data can be applied both *at rest* and *in flight*. At-rest data encryption means that data resides in an encrypted format when not being accessed. A file that is encrypted and located on a hard drive is an example of at-rest encryption. In-flight encryption, also known as over-the-wire encryption, applies to data sent from one place to another over a network. Both modes of encryption can be used independently or together. At-rest encryption for Hadoop is covered in Chapter 9, and in-flight encryption is covered in Chapters 10 and 11.

Integrity

Integrity is an important part of information security. In the previous example where Alice sends a letter to Bob, what happens if Charles intercepts the letter in transit and makes changes to it unbeknownst to Alice and Bob? How can Bob ensure that the letter he receives is exactly the message that Alice sent? This concept is *data integrity*. The integrity of data is a critical component of information security, especially in industries with highly sensitive data. Imagine if a bank did not have a mechanism to prove the integrity of customer account balances? A hospital's data integrity of patient records? A government's data integrity of intelligence secrets? Even if confidentiality is guaranteed, data that doesn't have integrity guarantees is at risk of substantial damage. Integrity is covered in Chapters 9 and 10.

Availability

Availability is a different type of principle than the previous two. While confidentiality and integrity can closely be aligned to well-known security concepts, availability is largely covered by operational preparedness. For example, if Alice tries to send her letter to Bob, but the post office is closed, the letter cannot be sent to Bob, thus making it unavailable to him. The availability of data or services can be impacted by regular outages such as scheduled downtime for upgrades or

applying security patches, but it can also be impacted by security events such as distributed denial-of-service (DDoS) attacks. The handling of high-availability configurations is covered in *Hadoop Operations* and *Hadoop: The Definitive Guide*, but the concepts will be covered from a security perspective in Chapters 3 and 10.

Authentication, Authorization, and Accounting

Authentication, authorization, and accounting (often abbreviated, AAA) refer to an architectural pattern in computer security where users of a service prove their identity, are granted access based on rules, and where a recording of a user's actions is maintained for auditing purposes. Closely tied to AAA is the concept of identity. Identity refers to how a system distinguishes between different entities, users, and services, and is typically represented by an arbitrary string, such as a username or a unique number, such as a user ID (UID).

Before diving into how Hadoop supports identity, authentication, authorization, and accounting, consider how these concepts are used in the much simpler case of using the `sudo` command on a single Linux server. Let's take a look at the terminal session for two different users, Alice and Bob. On this server, Alice is given the username *alice* and Bob is given the username *bob*. Alice logs in first, as shown in Example 1-1.

Example 1-1. Authentication and authorization

```
$ ssh alice@hadoop01
alice@hadoop01's password:
Last login: Wed Feb 12 15:26:55 2014 from 172.18.12.166
[alice@hadoop01 ~]$ sudo service sshd status
openssh-daemon (pid  1260) is running...
[alice@hadoop01 ~]$
```

In Example 1-1, Alice logs in through SSH and she is immediately prompted for her password. Her username/password pair is used to verify her entry in the */etc/passwd* password file. When this step is completed, Alice has been *authenticated* with the identity *alice*. The next thing Alice does is use the `sudo` command to get the status of the `sshd` service, which requires superuser privileges. The command succeeds, indicating that Alice was *authorized* to perform that command. In the case of `sudo`, the rules that govern who is authorized to execute commands as the superuser are stored in the */etc/sudoers* file, shown in Example 1-2.

Example 1-2. /etc/sudoers

```
[root@hadoop01 ~]# cat /etc/sudoers
root ALL = (ALL) ALL
%wheel ALL = (ALL) NOPASSWD:ALL
[root@hadoop01 ~]#
```

In Example 1-2, we see that the `root` user is granted permission to execute any command with `sudo` and that members of the *wheel* group are granted permission to execute any command with `sudo` while not being prompted for a password. In this case, the system is relying on the authentication that was performed during login rather than issuing a new authentication challenge. The final question is, how does the system know that Alice is a member of the `wheel` group? In Unix and Linux systems, this is typically controlled by the */etc/group* file.

In this way, we can see that two files control Alice's identity: the */etc/passwd* file (see Example 1-4) assigns her username a unique UID as well as details such as her home directory, while the */etc/group* file (see Example 1-3) further provides information about the identity of groups on the system and which users belong to which groups. These sources of identity information are then used by the `sudo` command, along with authorization rules found in the */etc/sudoers* file, to verify that Alice is authorized to execute the requested command.

Example 1-3. /etc/group

```
[root@hadoop01 ~]# grep wheel /etc/group
wheel:x:10:alice
[root@hadoop01 ~]#
```

Example 1-4. /etc/passwd

```
[root@hadoop01 ~]# grep alice /etc/passwd
alice:x:1000:1000:Alice:/home/alice:/bin/bash
[root@hadoop01 ~]#
```

Now let's see how Bob's session turns out in Example 1-5.

Example 1-5. Authorization failure

```
$ ssh bob@hadoop01
bob@hadoop01's password:
Last login: Wed Feb 12 15:30:54 2014 from 172.18.12.166
[bob@hadoop01 ~]$ sudo service sshd status

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for bob:
bob is not in the sudoers file.  This incident will be reported.
[bob@hadoop01 ~]$
```

In this example, Bob is able to authenticate in much the same way that Alice does, but when he attempts to use `sudo` he sees very different behavior. First, he is again prompted for his password and after successfully supplying it, he is denied permission to run the `service` command with superuser privileges. This happens because, unlike Alice, Bob is not a member of the *wheel* group and is therefore *not authorized* to use the `sudo` command.

That covers identity, authentication, and authorization, but what about accounting? For actions that interact with secure services such as SSH and `sudo`, Linux generates a logfile called */var/log/secure*. This file records an account of certain actions including both successes and failures. If we take a look at this log after Alice and Bob have performed the preceding actions, we see the output in Example 1-6 (formatted for readability).

Example 1-6. /var/log/secure

```
[root@hadoop01 ~]# tail -n 6 /var/log/secure
Feb 12 20:32:04 ip-172-25-3-79 sshd[3774]: Accepted password for
  alice from 172.18.12.166 port 65012 ssh2
Feb 12 20:32:04 ip-172-25-3-79 sshd[3774]: pam_unix(sshd:session):
```

```
  session opened for user alice by (uid=0)
Feb 12 20:32:33 ip-172-25-3-79 sudo:    alice : TTY=pts/0 ;
  PWD=/home/alice ; USER=root ; COMMAND=/sbin/service sshd status
Feb 12 20:33:15 ip-172-25-3-79 sshd[3799]: Accepted password for
  bob from 172.18.12.166 port 65017 ssh2
Feb 12 20:33:15 ip-172-25-3-79 sshd[3799]: pam_unix(sshd:session):
  session opened for user bob by (uid=0)
Feb 12 20:33:39 ip-172-25-3-79 sudo:      bob : user NOT in sudoers;
  TTY=pts/2 ; PWD=/home/bob ; USER=root ; COMMAND=/sbin/service sshd status
[root@hadoop01 ~]#
```

For both users, the fact that they successfully logged in using SSH is recorded, as are their attempts to use `sudo`. In Alice's case, the system records that she successfully used `sudo` to execute the `/sbin/service sshd status` command as the user *root*. For Bob, on the other hand, the system records that he attempted to execute the `/sbin/service sshd status` command as the user *root* and was denied permission because he is not in */etc/sudoers*.

This example shows how the concepts of identity, authentication, authorization, and accounting are used to maintain a secure system in the relatively simple example of a single Linux server. These concepts are covered in detail in a Hadoop context in Part II.

Hadoop Security: A Brief History

Hadoop has its heart in storing and processing large amounts of data efficiently and as it turns out, cheaply (monetarily) when compared to other platforms. The focus early on in the project was around the actual technology to make this happen. Much of the code covered the logic on how to deal with the complexities inherent in distributed systems, such as handling of failures and coordination. Due to this focus, the early Hadoop project established a security stance that the entire cluster of machines and all of the users accessing it are part of a *trusted network*. What this effectively means is that Hadoop did not have strong security measures in place to enforce, well, much of anything.

As the project evolved, it became apparent that at a minimum there should be a mechanism for users to strongly authenticate to prove their identities. The mechanism chosen for the project was Kerberos, a well-established protocol that today is common in enterprise systems such as Microsoft Active Directory. After strong authentication came strong authorization. Strong authorization defined what an individual user could do after they had been authenticated. Initially, authorization was implemented on a per-component basis, meaning that administrators needed to define authorization controls in multiple places. Eventually this became easier with Apache Sentry (Incubating), but even today there is not a holistic view of authorization across the ecosystem, as we will see in Chapters 6 and 7.

Another aspect of Hadoop security that is still evolving is the protection of data through encryption and other confidentiality mechanisms. In the trusted network, it was assumed that data was inherently protected from unauthorized users because only authorized users were on the network. Since then, Hadoop has added encryption for data transmitted between nodes, as well as data stored on disk. We will see how this security evolution comes into play as we proceed, but first we will take a look at the Hadoop ecosystem to get our bearings.

Hadoop Components and Ecosystem

In this section, we will provide a 50,000-foot view of the Hadoop ecosystem components that are covered throughout the book. This will help to introduce components before talking about the security of them in later chapters. Readers that are well versed in the components listed can safely skip to the next section. Unless otherwise noted, security features described throughout this book apply to the versions of the associated project listed in Table 1-1.

Table 1-1. Project versions[a]

Project	Version
Apache HDFS	2.3.0
Apache MapReduce (for MR1)	1.2.1
Apache YARN (for MR2)	2.3.0
Apache Hive	0.12.0
Cloudera Impala	2.0.0
Apache HBase	0.98.0
Apache Accumulo	1.6.0
Apache Solr	4.4.0
Apache Oozie	4.0.0
Cloudera Hue	3.5.0
Apache ZooKeeper	3.4.5
Apache Flume	1.5.0
Apache Sqoop	1.4.4
Apache Sentry (Incubating)	1.4.0-incubating

[a] An astute reader will notice some omissions in the list of projects covered. In particular, there is no mention of Apache Spark, Apache Ranger, or Apache Knox. These projects were omitted due to time constraints and given their status as relatively new additions to the Hadoop ecosystem.

Apache HDFS

The *Hadoop Distributed File System*, or HDFS, is often considered the foundation component for the rest of the Hadoop ecosystem. HDFS is the storage layer for Hadoop and provides the ability to

store mass amounts of data while growing storage capacity and aggregate bandwidth in a linear fashion. HDFS is a *logical* filesystem that spans many servers, each with multiple hard drives. This is important to understand from a security perspective because a given file in HDFS can span many or all servers in the Hadoop cluster. This means that client interactions with a given file might require communication with every node in the cluster. This is made possible by a key implementation feature of HDFS that breaks up files into *blocks*. Each block of data for a given file can be stored on any physical drive on any node in the cluster. Because this is a complex topic that we cannot cover in depth here, we are omitting the details of how that works and recommend *Hadoop: The Definitive Guide, 3rd Edition* by Tom White (O'Reilly). The important security takeaway is that all files in HDFS are broken up into blocks, and clients using HDFS will communicate over the network to all of the servers in the Hadoop cluster when reading and writing files.

HDFS is built on a head/worker architecture and is comprised of two primary components: NameNode (head) and DataNode (worker). Additional components include JournalNode, HttpFS, and NFS Gateway:

NameNode

> The NameNode is responsible for keeping track of all the metadata related to the files in HDFS, such as filenames, block locations, file permissions, and replication. From a security perspective, it is important to know that clients of HDFS, such as those reading or writing files, *always* communicate with the NameNode. Additionally, the NameNode provides several important security functions for the entire Hadoop ecosystem, which are described later.

DataNode

> The DataNode is responsible for the actual storage and retrieval of data blocks in HDFS. Clients of HDFS reading a given file are told by the NameNode which DataNode in the cluster has the block of data requested. When writing data to HDFS, clients write a block of data to a DataNode determined by the NameNode. From there, that DataNode sets up a write pipeline to other DataNodes to complete the write based on the desired replication factor.

JournalNode

> The JournalNode is a special type of component for HDFS. When HDFS is configured for *high availability (HA)*, JournalNodes take over the NameNode responsibility for writing HDFS metadata information. Clusters typically have an odd number of JournalNodes (usually three or five) to ensure majority. For example, if a new file is written to HDFS, the metadata about the file is written to every JournalNode. When the majority of the JournalNodes successfully write this information, the change is considered durable. HDFS clients and DataNodes do not interact with JournalNodes directly.

HttpFS

> HttpFS is a component of HDFS that provides a proxy for clients to the NameNode and DataNodes. This proxy is a REST API and allows clients to communicate to the proxy to use HDFS without having direct connectivity to any of the other components in HDFS. HttpFS will be a key component in certain cluster architectures, as we will see later in the book.

NFS Gateway

: The NFS gateway, as the name implies, allows for clients to use HDFS like an NFS-mounted filesystem. The NFS gateway is an actual daemon process that facilitates the NFS protocol communication between clients and the underlying HDFS cluster. Much like HttpFS, the NFS gateway sits between HDFS and clients and therefore affords a security boundary that can be useful in certain cluster architectures.

KMS

: The *Hadoop Key Management Server*, or KMS, plays an important role in HDFS transparent encryption at rest. Its purpose is to act as the intermediary between HDFS clients, the NameNode, and a key server, handling encryption operations such as decrypting data encryption keys and managing encryption zone keys. This is covered in detail in Chapter 9.

Apache YARN

As Hadoop evolved, it became apparent that the MapReduce processing framework, while incredibly powerful, did not address the needs of additional use cases. Many problems are not easily solved, if at all, using the MapReduce programming paradigm. What was needed was a more generic framework that could better fit additional processing models. Apache YARN provides this capability. Other processing frameworks and applications, such as Impala and Spark, use YARN as the resource management framework. While YARN provides a more general resource management framework, MapReduce is still the canonical application that runs on it. MapReduce that runs on YARN is considered version 2, or MR2 for short. The YARN architecture consists of the following components:

ResourceManager

: The ResourceManager daemon is responsible for application submission requests, assigning ApplicationMaster tasks, and enforcing resource management policies.

JobHistory Server

: The JobHistory Server, as the name implies, keeps track of the history of all jobs that have run on the YARN framework. This includes job metrics like running time, number of tasks run, amount of data written to HDFS, and so on.

NodeManager

: The NodeManager daemon is responsible for launching individual tasks for jobs within YARN *containers*, which consist of virtual cores (CPU resources) and RAM resources. Individual tasks can request some number of virtual cores and memory depending on its needs. The minimum, maximum, and increment ranges are defined by the ResourceManager. Tasks execute as separate processes with their own JVM. One important role of the NodeManager is to launch a special task called the *ApplicationMaster*. This task is responsible for managing the status of all tasks for the given application. YARN separates resource management from task management to better scale YARN applications in large clusters as each job executes its own ApplicationMaster.

Apache MapReduce

MapReduce is the processing counterpart to HDFS and provides the most basic mechanism to batch process data. When MapReduce is executed on top of YARN, it is often called MapReduce2, or MR2. This distinguishes the YARN-based verison of MapReduce from the standalone MapReduce framework, which has been retroactively named MR1. MapReduce *jobs* are submitted by clients to the MapReduce framework and operate over a subset of data in HDFS, usually a specified directory. MapReduce itself is a programming paradigm that allows chunks of data, or blocks in the case of HDFS, to be processed by multiple servers in parallel, independent of one another. While a Hadoop developer needs to know the intricacies of how MapReduce works, a security architect largely does not. What a security architect needs to know is that clients submit their jobs to the MapReduce framework and from that point on, the MapReduce framework handles the distribution and execution of the client code across the cluster. Clients do not interact with any of the nodes in the cluster to make their job run. Jobs themselves require some number of *tasks* to be run to complete the work. Each task is started on a given node by the MapReduce framework's scheduling algorithm.

Note

Individual tasks started by the MapReduce framework on a given server are executed as different users depending on whether Kerberos is enabled. Without Kerberos enabled, individual tasks are run as the *mapred* system user. When Kerberos is enabled, the individual tasks are executed as the user that submitted the MapReduce job. However, even if Kerberos is enabled, it may not be immediately apparent which user is executing the underlying MapReduce tasks when another component or tool is submitting the MapReduce job. See "Impersonation" for a relevant detailed discussion regarding Hive impersonation.

Similar to HDFS, MapReduce is also a head/worker architecture and is comprised of two primary components:

JobTracker (head)

> When clients submit jobs to the MapReduce framework, they are communicating with the JobTracker. The JobTracker handles the submission of jobs by clients and determines how jobs are to be run by deciding things like how many tasks the job requires and which TaskTrackers will handle a given task. The JobTracker also handles security and operational features such as job queues, scheduling pools, and access control lists to determine authorization. Lastly, the JobTracker handles job metrics and other information about the job, which are communicated to it from the various TaskTrackers throughout the execution of a given job. The JobTracker includes both resource management and task management, which were split in MR2 between the ResourceManager and ApplicationMaster.

TaskTracker (worker)

> TaskTrackers are responsible for executing a given task that is part of a MapReduce job. TaskTrackers receive tasks to run from the JobTracker, and spawn off separate JVM processes for each task they run. TaskTrackers execute both map and reduce tasks, and the amount of each that can be run concurrently is part of the MapReduce configuration. The important takeaway from a security standpoint is that the JobTracker decides what tasks to be run and on which TaskTrackers. Clients do not have control over how tasks are assigned, nor do they

communicate with TaskTrackers as part of normal job execution.

A key point about MapReduce is that other Hadoop ecosystem components are frameworks and libraries on top of MapReduce, meaning that MapReduce handles the actual processing of data, but these frameworks and libraries abstract the MapReduce job execution from clients. Hive, Pig, and Sqoop are examples of components that use MapReduce in this fashion.

Tip

Understanding how MapReduce jobs are submitted is an important part of user auditing in Hadoop, and is discussed in detail in "Block access tokens". A user submitting her own Java MapReduce code is a much different activity from a security point of view than a user using Sqoop to import data from a RDBMS or executing a SQL query in Hive, even though all three of these activities use MapReduce.

Apache Hive

The Apache Hive project was started by Facebook. The company saw the utility of MapReduce to process data but found limitations in adoption of the framework due to the lack of Java programming skills in its analyst communities. Most of Facebook's analysts did have SQL skills, so the Hive project was started to serve as a SQL abstraction layer that uses MapReduce as the execution engine. The Hive architecture consists of the following components:

Metastore database

> The metastore database is a relational database that contains all the Hive metadata, such as information about databases, tables, columns, and data types. This information is used to apply structure to the underlying data in HDFS at the time of access, also known as *schema on read*.

Metastore server

> The Hive Metastore Server is a daemon that sits between Hive clients and the metastore database. This affords a layer of security by not allowing clients to have the database credentials to the Hive metastore.

HiveServer2

> HiveServer2 is the main access point for clients using Hive. HiveServer2 accepts JDBC and ODBC clients, and for this reason is leveraged by a variety of client tools and other third-party applications.

HCatalog

> HCatalog is a series of libraries that allow non-Hive frameworks to have access to Hive metadata. For example, users of Pig can use HCatalog to read schema information about a given directory of files in HDFS. The WebHCat server is a daemon process that exposes a REST interface to clients, which in turn access HCatalog APIs.

For more thorough coverage of Hive, have a look at *Programming Hive* by Edward Capriolo, Dean Wampler, and Jason Rutherglen (O'Reilly).

Cloudera Impala

Cloudera Impala is a massive parallel processing (MPP) framework that is purpose-built for analytic SQL. Impala reads data from HDFS and utilizes the Hive metastore for interpreting data structures and formats. The Impala architecture consists of the following components:

Impala daemon (impalad)

> The Impala daemon does all of the heavy lifting of data processing. These daemons are collocated with HDFS DataNodes to optimize for local reads.

StateStore

> The StateStore daemon process maintains state information about all of the Impala daemons running. It monitors whether Impala daemons are up or down, and broadcasts status to all of the daemons. The StateStore is not a required component in the Impala architecture, but it does provide for faster failure tolerance in the case where one or more daemons have gone down.

Catalog server

> The Catalog server is Impala's gateway into the Hive metastore. This process is responsible for pulling metadata from the Hive metastore and synchronizing metadata changes that have occurred by way of Impala clients. Having a separate Catalog server helps to reduce the load the Hive metastore server encounters, as well as to provide additional optimizations for Impala for speed.

Tip

New users to the Hadoop ecosystem often ask what the difference is between Hive and Impala because they both offer SQL access to data in HDFS. Hive was created to allow users that are familiar with SQL to process data in HDFS without needing to know anything about MapReduce. It was designed to abstract the innards of MapReduce to make the data in HDFS more accessible. Hive is largely used for batch access and ETL work. Impala, on the other hand, was designed from the ground up to be a fast analytic processing engine to support ad hoc queries and business intelligence (BI) tools. There is utility in both Hive and Impala, and they should be treated as complementary components.

For more thorough coverage of all things Impala, check out *Getting Started with Impala* (O'Reilly).

Apache Sentry (Incubating)

Sentry is the component that provides fine-grained role-based access controls (RBAC) to several of the other ecosystem components, such as Hive and Impala. While individual components may have their own authorization mechanism, Sentry provides a unified authorization that allows centralized policy enforcement across components. It is a critical component of Hadoop security, which is why we have dedicated an entire chapter to the topic (Chapter 7). Sentry consists of the following

components:

Sentry server

> The Sentry server is a daemon process that facilitates policy lookups made by other Hadoop ecosystem components. Client components of Sentry are configured to delegate authorization decisions based on the policies put in place by Sentry.

Policy database

> The Sentry policy database is the location where all authorization policies are stored. The Sentry server uses the policy database to determine if a user is allowed to perform a given action. Specifically, the Sentry server looks for a matching policy that grants access to a resource for the user. In earlier versions of Sentry, the policy database was a text file that contained all of the policies. The evolution of Sentry and the policy database is discussed in detail in Chapter 7.

Apache HBase

Apache HBase is a distributed key/value store inspired by Google's BigTable paper, "BigTable: A Distributed Storage System for Structured Data". HBase typically utilizes HDFS as the underlying storage layer for data, and for the purposes of this book we will assume that is the case. HBase *tables* are broken up into *regions*. These regions are partitioned by *row key*, which is the index portion of a given key. Row IDs are sorted, thus a given region has a range of sorted row keys. Regions are hosted by a *RegionServer*, where clients request data by a key. The key is comprised of several components: the row key, the *column family*, the *column qualifier*, and the *timestamp*. These components together uniquely identify a value stored in the table.

Clients accessing HBase first look up the RegionServers that are responsible for hosting a particular range of row keys. This lookup is done by scanning the `hbase:meta` table. When the right RegionServer is located, the client will make read/write requests directly to that RegionServer rather than through the master. The client caches the mapping of regions to RegionServers to avoid going through the lookup process. The location of the server hosting the `hbase:meta` table is looked up in ZooKeeper. HBase consists of the following components:

Master

> As stated, the HBase Master daemon is responsible for managing the regions that are hosted by which RegionServers. If a given RegionServer goes down, the HBase Master is responsible for reassigning the region to a different RegionServer. Multiple HBase Masters can be run simultaneously and the HBase Masters will use ZooKeeper to elect a single HBase Master to be active at any one time.

RegionServer

> RegionServers are responsible for serving regions of a given HBase table. Regions are sorted ranges of keys; they can either be defined manually using the HBase shell or automatically defined by HBase over time based upon the keys that are ingested into the table. One of HBase's goals is to evenly distribute the key-space, giving each RegionServer an equal responsibility in serving data. Each RegionServer typically hosts multiple regions.

REST server

> The HBase REST server provides a REST API to perform HBase operations. The default HBase API is provided by a Java API, just like many of the other Hadoop ecosystem projects. The REST API is commonly used as a language agnostic interface to allow clients to utilize any programming they wish.

Thrift server

> In addition to the REST server, HBase also has a Thrift server. This serves as yet another useful API interface for clients to leverage.

For more information on the architecture of HBase and the use cases it is best suited for, we recommend *HBase: The Definitive Guide* by Lars George (O'Reilly).

Apache Accumulo

Apache Accumulo is a sorted and distributed key/value store designed to be a robust, scalable, high-performance storage and retrieval system. Like HBase, Accumulo was originally based on the Google BigTable design, but was built on top of the Apache Hadoop ecosystem of projects (in particular, HDFS, ZooKeeper, and Apache Thrift). Accumulo uses roughly the same data model as HBase. Each Accumulo table is split into one or more tablets that contains a roughly equal number of records distributed by the record's row ID. Each record also has a multipart column key that includes a column family, column qualifier, and visibility label. The visibility label was one of Accumulo's first major departures from the original BigTable design. Visibility labels added the ability to implement cell-level security (we'll discuss them in more detail in Chapter 6). Finally, each record also contains a timestamp that allows users to store multiple versions of records that otherwise share the same record key. Collectively, the row ID, column, and timestamp make up a record's key, which is associated with a particular value.

The tablets are distributed by splitting up the set of row IDs. The split points are calculated automatically as data is inserted into a table. Each tablet is hosted by a single TabletServer that is responsible for serving reads and writes to data in the given tablet. Each TabletServer can host multiple tablets from the same tables and/or different tables. This makes the tablet the unit of distribution in the system.

When clients first access Accumulo, they look up the location of the TabletServer hosting the `accumulo.root` table. The `accumulo.root` table stores the information for how the `accumulo.meta` table is split into tablets. The client will directly communicate with the TabletServer hosting `accumulo.root` and then again for TabletServers that are hosting the tablets of the `accumulo.meta` table. Because the data in these tables—especially `accumulo.root`—changes relatively less frequently than other data, the client will maintain a cache of tablet locations read from these tables to avoid bottlenecks in the read/write pipeline. Once the client has the location of the tablets for the row IDs that it is reading/writing, it will communicate directly with the required TabletServers. At no point does the client have to interact with the Master, and this greatly aids scalability. Overall, Accumulo consists of the following components:

Master

: The Accumulo Master is responsible for coordinating the assignment of tablets to TabletServers. It ensures that each tablet is hosted by exactly one TabletServer and responds to events such as a TabletServer failing. It also handles administrative changes to a table and coordinates startup, shutdown, and write-ahead log recovery. Multiple Masters can be run simultaneously and they will elect a leader so that only one Master is active at a time.

TabletServer

: The TabletServer handles all read/write requests for a subset of the tablets in the Accumulo cluster. For writes, it handles writing the records to the write-ahead log and flushing the in-memory records to disk periodically. During recovery, the TabletServer replays the records from the write-ahead log into the tablet being recovered.

GarbageCollector

: The GarbageCollector periodically deletes files that are no longer needed by any Accumulo process. Multiple GarbageCollectors can be run simultaneously and they will elect a leader so that only one GarbageCollector is active at a time.

Tracer

: The Tracer monitors the rest of the cluster using Accumulo's distributed timing API and writes the data into an Accumulo table for future reference. Multiple Tracers can be run simultaneously and they will distribute the load evenly among them.

Monitor

: The Monitor is a web application for monitoring the state of the Accumulo cluster. It displays key metrics such as record count, cache hit/miss rates, and table information such as scan rate. The Monitor also acts as an endpoint for log forwarding so that errors and warnings can be diagnosed from a single interface.

Apache Solr

The Apache Solr project, and specifically *SolrCloud*, enables the search and retrieval of *documents* that are part of a larger *collection* that has been *sharded* across multiple physical servers. Search is one of the canonical use cases for big data and is one of the most common utilities used by anyone accessing the Internet. Solr is built on top of the Apache Lucene project, which actually handles the bulk of the indexing and search capabilities. Solr expands on these capabilities by providing enterprise search features such as faceted navigation, caching, hit highlighting, and an administration interface.

Solr has a single component, the server. There can be many Solr servers in a single deployment, which scale out linearly through the sharding provided by SolrCloud. SolrCloud also provides replication features to accommodate failures in a distributed environment.

Apache Oozie

Apache Oozie is a workflow management and orchestration system for Hadoop. It allows for setting up workflows that contain various *actions*, each of which can utilize a different component in the Hadoop ecosystem. For example, an Oozie workflow could start by executing a Sqoop import to move data into HDFS, then a Pig script to transform the data, followed by a Hive script to set up metadata structures. Oozie allows for more complex workflows, such as forks and joins that allow multiple steps to be executed in parallel, and other steps that rely on multiple steps to be completed before continuing. Oozie workflows can run on a repeatable schedule based on different types of input conditions such as running at a certain time or waiting until a certain path exists in HDFS.

Oozie consists of just a single server component, and this server is responsible for handling client workflow submissions, managing the execution of workflows, and reporting status.

Apache ZooKeeper

Apache ZooKeeper is a distributed coordination service that allows for distributed systems to store and read small amounts of data in a synchronized way. It is often used for storing common configuration information. Additionally, ZooKeeper is heavily used in the Hadoop ecosystem for synchronizing high availability (HA) services, such as NameNode HA and ResourceManager HA.

ZooKeeper itself is a distributed system that relies on an odd number of servers called a ZooKeeper *ensemble* to reach a *quorum*, or majority, to acknowledge a given transaction. ZooKeeper has only one component, the ZooKeeper server.

Apache Flume

Apache Flume is an event-based ingestion tool that is used primarily for ingestion into Hadoop, but can actually be used completely independent of it. Flume, as the name would imply, was initially created for the purpose of ingesting log events into HDFS. The Flume architecture consists of three main pieces: sources, sinks, and channels.

A Flume source defines how data is to be read from the upstream provider. This would include things like a syslog server, a JMS queue, or even polling a Linux directory. A Flume sink defines how data should be written downstream. Common Flume sinks include an HDFS sink and an HBase sink. Lastly, a Flume channel defines how data is stored between the source and sink. The two primary Flume channels are the memory channel and file channel. The memory channel affords speed at the cost of reliability, and the file channel provides reliability at the cost of speed.

Flume consists of a single component, a Flume *agent*. Agents contain the code for sources, sinks, and channels. An important part of the Flume architecture is that Flume agents can be connected to each other, where the sink of one agent connects to the source of another. A common interface in this case is using an Avro source and sink. Flume ingestion and security is covered in Chapter 10 and in *Using Flume*.

Apache Sqoop

Apache Sqoop provides the ability to do batch imports and exports of data to and from a traditional RDBMS, as well as other data sources such as FTP servers. Sqoop itself submits map-only MapReduce jobs that launch tasks to interact with the RDBMS in a parallel fashion. Sqoop is used both as an easy mechanism to initially seed a Hadoop cluster with data, as well as a tool used for regular ingestion and extraction routines. There are currently two different versions of Sqoop: Sqoop1 and Sqoop2. In this book, the focus is on Sqoop1. Sqoop2 is still not feature complete at the time of this writing, and is missing some fundamental security features, such as Kerberos authentication.

Sqoop1 is a set of client libraries that are invoked from the command line using the `sqoop` binary. These client libraries are responsible for the actual submission of the MapReduce job to the proper framework (e.g., traditional MapReduce or MapReduce2 on YARN). Sqoop is discussed in more detail in Chapter 10 and in *Apache Sqoop Cookbook*.

Cloudera Hue

Cloudera Hue is a web application that exposes many of the Hadoop ecosystem components in a user-friendly way. Hue allows for easy access into the Hadoop cluster without requiring users to be familiar with Linux or the various command-line interfaces the components have. Hue has several different security controls available, which we'll look at in Chapter 12. Hue is comprised of the following components:

Hue server

> This is the main component of Hue. It is effectively a web server that serves web content to users. Users are authenticated at first logon and from there, actions performed by the end user are actually done by Hue itself *on behalf of* the user. This concept is known as *impersonation* (covered in Chapter 5).

Kerberos Ticket Renewer

> As the name implies, this component is responsible for periodically renewing the *Kerberos ticket-granting ticket (TGT)*, which Hue uses to interact with the Hadoop cluster when the cluster has Kerberos enabled (Kerberos is discussed at length in Chapter 4).

Summary

This chapter introduced some common security terminology that builds the foundation of the topics covered throughout the rest of the book. A key takeaway from this chapter is to become comfortable with the fact that security for Hadoop is not a completely foreign discussion. Tried-and-true security principles such as CIA and AAA resonate in the Hadoop context and will be discussed at length in the chapters to come. Lastly, we took a look at many of the Hadoop ecosystem projects (and their individual components) to understand their purpose in the stack, and to get a sense at how security will apply.

In the next chapter, we will dive right into securing distributed systems. You will find that many of the security threats and mitigations that apply to Hadoop are generally applicable to distributed systems.

[1] Apache Hadoop itself consists of four subprojects: HDFS, YARN, MapReduce, and Hadoop Common. However, the Hadoop ecosystem, Hadoop, and the related projects that build on or integrate with Hadoop are often shortened to just Hadoop. We attempt to make it clear when we're referring to Hadoop the project versus Hadoop the ecosystem.

Part I. Security Architecture

Chapter 2. Securing Distributed Systems

In Chapter 1, we covered several key principles of secure computing. In this chapter, we will take a closer look at the interesting challenges that are present when considering the security of distributed systems. As we will see, being distributed considerably increases the potential threats to the system, thus also increasing the complexity of security measures needed to help mitigate those threats. A real-life example will help illustrate how security requirements increase when a system becomes more distributed.

Let's consider a bank as an example. Many years ago, everyday banking for the average person meant driving down to the local bank, visiting a bank teller, and conducting transactions in person. The bank's security measures would have included checking the person's identification, and account number, and verifying that the requested action could be performed, such as ensuring there was enough money in the account to cover a withdrawal.

Over the years, banks became larger. Your local hometown bank probably became a branch of a larger bank, thus giving you the ability to conduct banking not just at the bank's nearby location but also at any of its other locations. The security measures necessary to protect assets have grown because there is no longer just a single physical location to protect. Also, more bank tellers need to be properly trained.

Taking this a step further, banks eventually started making use of ATMs to allow customers to withdraw money without having to go to a branch location. As you might imagine, even more security controls are necessary to protect the bank beyond what was required when banking was a human interaction. Next, banks became interconnected with other banks, which allowed customers from one bank to use the ATMs of a different bank. Banks then needed to establish security controls between themselves to ensure that no security was lost as a result of this interconnectivity. Lastly, the Internet movement introduced the ability to do online banking through a website, or even from mobile devices. This dramatically increased potential threats and the security controls needed.

As you can see, what started as a straightforward security task to protect a small bank in your town has become orders of magnitude more difficult the more distributed and interconnected the bank became over decades of time. While this example might seem obvious, it starts to frame the problem of how to design a security architecture for a system that can be distributed across tens,

hundreds, or even thousands of machines. It is no small task but it can be made less intimidating by breaking it down into pieces, starting with understanding threats.

Threat Categories

A key component to arriving at a robust security architecture for a distributed system is to understand the threats that are likely to be present, and to be able to categorize them to better understand what security mechanisms need to be in place to help mitigate those threats. In this section, we will review a few common threat categories that are important to be aware of. The threat categories will help you identify where the threats are coming from, what security features are needed to protect against them, and how to respond to an incident if and when it happens.

Unauthorized Access/Masquerade

One of the most common threat categories comes in the form of unauthorized access. This happens when someone successfully accesses a system when he should have otherwise been denied access. One common way for this to happen is from a *masquerade* attack. Masquerade is the notion that an invalid user presents himself as a valid user in order to gain access. You might wonder how the invalid user presented himself as a valid user. The most likely answer that the attacker obtained a valid username and associated password.

Masquerade attacks are especially prominent since the age of the Internet, and specifically for distributed systems. Attackers have a variety of ways to obtain valid usernames and passwords, such as trying common words and phrases as passwords, or knowing words that are related to the valid user that might be used as a password. For example, attackers looking to obtain valid login credentials for a social media website, might collect keywords from a person's public posts to come up with a password list to try (e.g., if the attackers were focusing on New York–based users who list "baseball" as a hobby, they might try the password *yankees*).

In the case of an invalid user executing a successful masquerade attack, how would a security administrator know? After all, if an attacker logged in with a valid user's credentials, wouldn't this appear as normal from the distributed system's perspective? Not necessarily. Typically, masquerade attacks can be profiled by looking at audit logs for login attempts. If an attacker is using a list of possible passwords to try against a user account, the unsuccessful attempts should show up in audit logfiles. Seeing a high number of failed login attempts for a user can usually be attributed to an attack. A valid user might mistype or forget her password, leading to a small number of failed login attempts, but 20 successive failed login attempts, for example, would be unusual.

Another common footprint for masquerade attacks is to look at where, from a network perspective, the login attempts are coming from. Profiling login attempts by IP addresses can be a good way to discover if a masquerade attack is attempted. Are the IP addresses shown as the client attempting to log in consistent with what is expected, such as coming from a known subnet of company IP addresses, or are they sourced from another country on the other side of the world? Also, what time of day did the login attempts occur? Did Alice try to login to the system at 3:00 a.m., or did she log in during normal business hours?

Another form of unauthorized access comes from an attacker exploiting a vulnerability in the system, thus gaining entry without needing to present valid credentials. Vulnerabilities are discussed in “Vulnerabilities”.

Insider Threat

Arguably the single most damaging threat category is the *insider threat*. As the name implies, the attacker comes from inside the business and is a regular user. Insider threats can include employees, consultants, and contractors. What makes the insider threat so scary is that the attacker already has internal access to the system. The attacker can log in with valid credentials, get authorized by the system to perform a certain function, and pass any number of security checks along the way because she is *supposed* to be granted access. This can result in a blatant attack on a system, or something much more subtle like the attacker leaking sensitive data to unauthorized users by leveraging her own accesses.

Throughout this book, you will find security features that ensure that the right users are accessing only the data and services they should be. Combating insider threats requires effective auditing practices (described in Chapter 8). In addition to the technical tools available to help combat the insider threat, business policies need to be established to enforce proper auditing, and procedures that respond to incidents must be outlined. The need for these policies is true for all of the threat categories described in this chapter, though best practices for setting such policies are not covered.

Denial of Service

Denial of service (DoS), is a situation where a service is unavailable to one or more clients. The term *service* in this case is an umbrella that includes access to data, processing capabilities, and the general usability of the system in question. How the denial of service happens can come from a variety of different attack vectors. In the age of the Internet, a common attack vector is to simply overwhelm the system in question with excessive network traffic. This is done by using many computers in parallel, thus making the attack a *distributed denial of service (DDoS)*. When the system is bombarded with too many requests for it to handle, it starts failing in some way, from dropping other valid requests to outright failure of the system.

While distributed systems typically benefit from having fault tolerance of some kind, DoS attacks are still possible. For example, if a distributed system contains 50 servers, it might be difficult for attackers to disrupt service to all 50 machines. What if the distributed system is behind just a few network devices, such as a network firewall and an access switch? Attackers can use this to their advantage by targeting the gateway into the distributed system rather than the distributed system itself. This point is important and will be covered in Chapter 3 when discuss about architecting a network perimeter around the cluster.

Threats to Data

Data is the single most important component of a distributed system. Without data, a distributed system is nothing more than an idle hum of servers that rack up the electric and cooling bills in a data center. Because data is so important, it is also the focus of security attacks. Threats to data are

present in multiple places in a distributed system. First, data must be stored in a secure fashion to prevent unauthorized viewing, tampering, or deletion. Next, data must also be protected *in transit*, because distributed systems are, well, distributed. The passing of data across a network can be threatened by something disruptive like a DoS attack, or something more passive such as an attacker capturing the network traffic unbeknownst to the communicating parties. In Chapter 1, we discussed the CIA model and its components. Ultimately, the CIA model is all about mitigating threats to data.

Threat and Risk Assessment

The coverage of threat categories in the previous section probably was not the first time you have heard about these things. It's important that in addition to understanding these threat categories you also assess the *risk* to your particular distributed system. For example, while a denial-of-service attack may be highly likely to occur for systems that are directly connected to the Internet, systems that have no outside network access, such as those on a company intranet, have a much lower risk of this actually happening. Notice that the risk is *low* and not completely removed, an important distinction.

Assessing the threats to a distributed system involves taking a closer look at two key components: the users and the environment. Once you understand these components, assessing risk becomes more manageable.

User Assessment

It's important to understand what users your distributed system will be exposed to. This obviously includes users who will be accessing the system and directly interacting with the interfaces it provides. It also includes users who might be present elsewhere in the environment but won't directly access the system. Understanding users in this context leads to a better risk assessment. Users of a distributed system like Hadoop typically are first classified by their line of business. What do these users do? Are they business intelligence analysts? Developers? Risk analysts? Security auditors? Data quality analysts?

Once users are classified into groups by business function, you can start to identify access patterns and tools that these groups of users need in order to use the distributed system. For example, if the users of the distributed system are all developers, several assumptions can be made about the need for shell access to nodes in the system, logfiles to debug jobs, and developer tools. On the other hand, business intelligence analysts might not need any of those things and will instead require a suite of analytical tools that interact with the distributed system on the user's behalf.

There will also be users with indirect access to the system. These users won't need access to data or processing resources of the system. However, they'll still interact with it as a part of, for example, support functions such as system maintenance, health monitoring, and user auditing. These types of users need to be accounted for in the overall security model.

Environment Assessment

To assess the risk for our distributed system, we'll also need to understand the *environment* it resides in. Generally, this will mean assessing the operational environment both in relation to other logical systems and the physical world. We'll take a look at the specifics for Hadoop in Chapter 3.

One of the key criteria for assessing the environment, mentioned briefly, is to look at whether the distributed system is accessible to the Internet. If so, a whole host of threats are far more likely to be realized, such as DoS attacks, vulnerability exploits, and viruses. Distributed systems that are indeed connected to the Internet will require constant monitoring and alerting, as well as a regular cadence for applying software patches and updating various security software definitions.

Another criteria to evaluate the environment is to understand where the servers that comprise the distributed system are physically located. Are they located in your company data center? Are they in a third-party–managed data center? Are they in a public cloud infrastructure? Understanding the answer to these questions will start to frame the problem of providing a security assessment. For example, if the distributed system is hosted in a public cloud, a few threats are immediately apparent: the infrastructure is not owned by your company, so you do not definitively know who has direct access to the machines. This expands the scope of insider threat to include your hosting provider. Also, the usage of a public cloud begs the question of how your users are connecting to the distributed system and how data flows into and out of it. Again, threats to communications that occur across an open network to a shared public cloud have a much higher risk of happening than those that are within your own company data center.

The point is not to scare you into thinking that public clouds are bad and company data centers are good, but rather to impart that the existence of one versus another will vary the level of risk that a given threat may have against your distributed system. Regardless of the environment, the key to protecting your distributed system is to look at risk mitigation in a multitiered approach, as discussed in the next section.

Vulnerabilities

Vulnerabilities are a separate topic, but they are related to the discussion of threats and risk. Vulnerabilities exist in a variety of different forms in a distributed system. A common place for vulnerabilities is in the software itself. *All* software has vulnerabilities. This might seem like a harsh statement, but the truth of it is that no piece of software is 100% secure.

So what exactly is a software vulnerability? Put simply, it's a piece of code that is susceptible to some kind of error or failure condition that is not accounted for gracefully. For instance, consider the simple example of a piece of software with a password screen that allows users to change their password (we will assume that the intended logic for the software is to allow passwords up to 16 characters in length). What happens if the input field for a new password mistakenly has a maximum length of 8 characters, and thus truncates the chosen password? This could lead to users setting shorter passwords than they realized, and worse, less complex passwords that are easier for an attacker to guess.

Certainly, software vulnerabilities are not the only type of vulnerabilities that distributed systems

are susceptible to. Other vulnerabilities include those related to the network infrastructure that a distributed system relies on. For example, many years ago there was a vulnerability that allowed an attacker to send a ping to a network broadcast address, causing every host in the network range to reply with a ping response. The attacker crafted the ping request so that the source IP address was set to a computer that was the intended target of the attack. The result was that the target host of the attack was overwhelmed with network communication to the point of failure. This attack was known as the *ping of death*. It has been mitigated, but the point is that until this was fixed by network hardware vendors, this was a vulnerability that had nothing to do with the software stack of machines on the network, yet an attacker could use it to disrupt the service of a particular machine on the network.

Software patches are regularly released to fix vulnerabilities as they are discovered, thus regular schedules for applying patches to a distributed system's software stack should be an integral part of every administrator's standard operating procedures. As the ping-of-death example shows, the scope of patches should also include firmware for switches, routers, other networking equipment, disk controllers, and the server BIOS.

Defense in Depth

One of the challenges that security administrators face is how to mitigate all of the threat categories and vulnerabilities discussed in this chapter. When looking at the variety of threat categories, it becomes immediately apparent that there is no single silver bullet that can effectively stop these threats. In order to have a fighting chance, many security controls must be in place—and must work together—in order to provide a comfortable level of security. This idea of deploying multiple security controls and protection methods is called *defense in depth*.

Looking back in history, defense in depth was not regularly followed. Security typically meant *perimeter security*, in that security controls existed only on the outside, or perimeter, of whatever was to be protected. A canonical example of this is imagining a thick, tall wall surrounding a castle. The mindset was that as long as the wall stood, the castle was safe. If the wall was breached, that was bad news for the castle dwellers. Today, things have gotten better.

Defense-in-depth security now exists in our everyday lives. Take the example of going to a grocery store. The grocery store has a door with a lock on it, and is only unlocked during normal business hours. There is also an alarm system that is triggered if an intruder illegally enters the building after hours. During regular hours, shoppers are monitored with security cameras throughout the store. Finally, store employees are trained to watch for patrons behaving suspiciously.

All of these security measures are in place to protect the grocery store from a variety of different threats, such as break-ins, shoplifters, and robberies. Had the grocery store only relied on the "castle wall" approach by only relying on strong door locks, most threats would not be addressed. Defense in depth is important here because any single security measure is not likely to mitigate all threats to the store. The same is true for distributed systems. There are many places where individual security measures can be deployed, such as setting up a network firewall around the perimeter, restrictive permissions on data, or access controls to the servers. But implementing all of these measures

together helps to lower the chances that an attack will be successful.

Summary

In this chapter, we broke down distributed system security by analyzing threat categories and vulnerabilities, and demonstrating that applying a defense-in-depth security architecture will minimize security risks. We also discussed the insider threat and why it should not be overlooked when designing security architecture.

The next chapter focuses on protecting Hadoop in particular, and building a sound system architecture is the first step.

Chapter 3. System Architecture

In Chapter 2, we took a look at how the security landscape changes when going from individual isolated systems to a fully distributed network of systems. It becomes immediately apparent just how daunting a task it is to secure hundreds if not thousands of servers in a single Hadoop cluster. In this chapter, we dive into the details of taking on this challenge by breaking the cluster down into several components that can independently be secured as part of an overall security strategy. At a high level, the Hadoop cluster can be divided into two major areas: the network and the hosts. But before we do this, let's explore the operating environment in which the Hadoop cluster resides.

Operating Environment

In the early days of Hadoop, a *cluster* likely meant a hodgepodge of repurposed machines used to try out the new technology. You might even have used old desktop-class machines and a couple of extra access switches to wire them up. Things have changed dramatically over the years. The days of stacking a few machines in the corner of a room has been replaced by the notion that Hadoop clusters are first-class citizens in real enterprises. Where Hadoop clusters physically and logically fit into the enterprise is called the *operating environment*.

Numerous factors that contribute to the choice of operating environment for Hadoop are out of scope of this book. We will focus on the typical operating environments in use today. As a result of rapid advances in server and network hardware (thank Moore's law), Hadoop can live in a few different environments:

In-house

> This Hadoop environment consists of a collection of physical ("bare metal") machines that are owned and operated by the business, and live in data centers under the control of the business.

Managed

> This Hadoop environment is a variation of in-house in that it consists of physical machines, but the business does not own and operate them. They are rented from a separate business that handles the full provisioning and maintenance of the servers, and the servers live in their own

data centers.

Cloud

This Hadoop environment looks very different than the others. A cloud environment consists of virtual servers that may physically reside in many different locations. The most popular cloud provider for Hadoop environments is Amazon's Elastic Compute Cloud (EC2).

Network Security

Network security is a detailed topic and certainly cannot be covered exhaustively here. Instead, we will focus on a few important network security topics that are commonly used to secure a Hadoop cluster's network. The first of these is *network segmentation*.

Network Segmentation

Network segmentation is a common practice of isolating machines and services to a separate part of a larger network. This practice holds true no matter if we are talking about Hadoop clusters, web servers, department workstations, or some other system. Creating a network segment can be done in two different ways, often together.

The first option is *physical* network segmentation. This is achieved by sectioning off a portion of the network with devices such as routers, switches, and firewalls. While these devices operate at higher layers of the OSI model, from a physical-layer point of view the separation is just that all devices on one network segment are physically plugged into network devices that are separate from other devices on the larger network.

The second option is *logical* network segmentation. Logical segmentation operates at higher layers of the OSI model, most commonly at the network layer using *Internet Protocol* (IP) addressing. With logical separation, devices in the same network segment are grouped together in some way. The most common way this is achieved is through the use of network subnets. For example, if a Hadoop cluster has 150 nodes, it may be that these nodes are logically grouped on the same /24 subnet (e.g., an IP subnet mask of 255.255.255.0), which represents a maximum of 256 IP addresses (254 usable). Organizing hosts logically in this fashion makes it easy to administer and secure.

The most common method of network segmentation is a hybrid approach that uses aspects of both physical and logical network segmentation. The most common way of implementing the hybrid approach is through the use of *virtual local area networks* (VLANs). VLANs allow multiple network subnets to share physical switches. Each VLAN is a distinct broadcast domain even though all VLANs share a single layer-2 network. Depending on the capabilities of the network switches or routers, you might have to assign each physical port to a single VLAN or you may be able to take advantage of packet tagging to run multiple VLANs over the same port.

As briefly mentioned before, both physical and logical separation can be, and often are, used together. Physical and logical separation may be present in the in-house and managed environments where a Hadoop cluster has a logical subnet defined, and all machines are physically connected to the same group of dedicated network devices (e.g., top-of-rack switches and aggregation switches).

With the cloud operating environment, physical network segmentation is often more difficult. Cloud infrastructure design goals are such that the location of hardware is less important than the availability of services sized by operational need. Some cloud environments allow for users to choose machines to be in the same locality group. While this is certainly better from a performance point of view, such as in the case of network latencies, it does not usually help with security. Machines in the same locality group likely share the same physical network as other machines.

Now that we have a Hadoop cluster that resides on its own network segment, how is this segment protected? This is largely achieved with network firewalls, and intrusion detection and prevention systems.

Intrusion Detection and Prevention

In the previous section, network firewalls were introduced as a way to control flows into and out of the network that a Hadoop cluster lives in. While this works perfectly well when “normal” everyday traffic is flowing, what about when not-so-normal events are happening? What happens if a malicious attacker has bypassed the network firewall and is attempting exploits against machines in the cluster, such as buffer overflow attacks? How about distributed denial-of-service (DDoS) attacks? Intrusion detection and prevention systems can help stop these types of attacks. Before we dive into where these devices fit into the system architecture for the cluster, let’s cover a few basics about these systems.

Intrusion detection systems (IDS) and *intrusion prevention systems* (IPS) are often used interchangeably in the discussion of network security. However, these two systems are fundamentally different in the role they play in dealing with suspected intrusions. An IDS, as the name implies, *detects* an intrusion. It falls under the same category as monitoring and alerting systems. An IDS is typically connected to a switch listening in promiscuous mode, meaning that all traffic on the switch flows to the IDS in addition to the intended destination port(s). When an IDS finds a packet or stream of packets it suspects as an attack, it generates an alert. An alert might be an event that gets sent to a separate monitoring system, or even just an email alias that security administrators subscribe to. Figure 3-1 shows the network diagram when an IDS is in place; you will notice that the IDS is not in the network flow between the outside network and the cluster network.

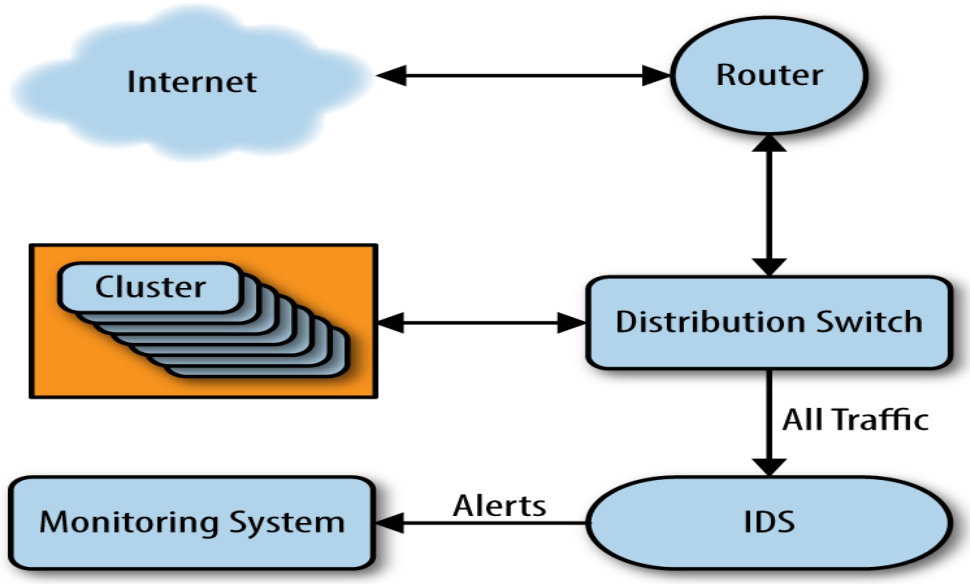

Figure 3-1. Network diagram with an IDS

An IPS, on the other hand, not only detects an intrusion, but actively tries to *prevent* or stop the intrusion as it is happening. This is made possible by the key difference between an IDS and IPS in that an IPS is not listening promiscuously on the network, but rather sitting *between* both sides of the network. Because of this fact, an IPS can actually stop the flow of intrusions to the other side of the network. A common feature of an IPS is to *fail close*. This means that upon failure of the IPS, such as being overwhelmed by an extensive DDoS attack to the point where it can no longer scan packets, it simply stops *all* packets from flowing through to the other side of the IPS. While this might seem like a successful DDoS attack, and in some ways it is, a fail close protects all the devices that are behind the IPS. Figure 3-2 shows the network diagram when an IPS is in place; you will notice that the IPS is actually in the network flow between the outside network and the cluster network.

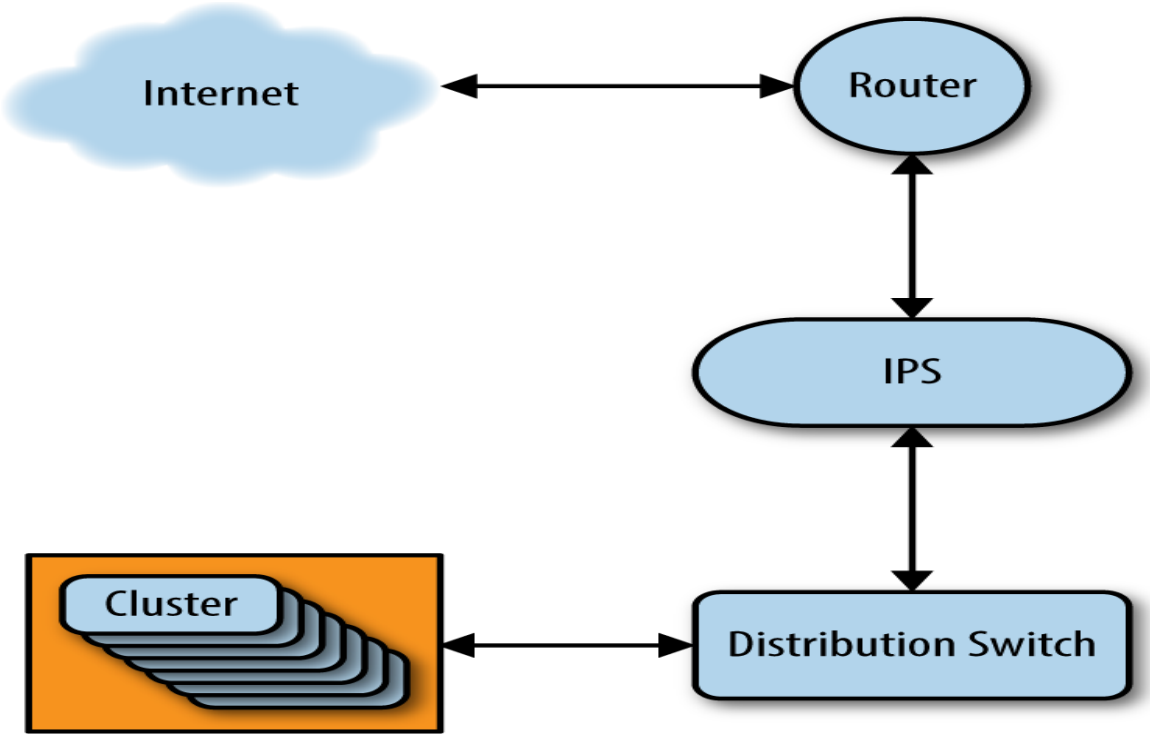

Figure 3-2. Network diagram with an IPS

Now that we have the 50,000-foot view of what these devices do, how does it help Hadoop? The answer is that it is another piece of the network security puzzle. Hadoop clusters inherently store massive amounts of data. Both detection and prevention of intrusion attempts to the cluster are critical to protecting the large swath of data. So where do these devices live relative to the rest of the network in which a Hadoop cluster lives? The answer: possibly several places.

In the discussion about network firewalls, it was mentioned that ingest pipelines that flow from the open Internet are likely to be treated differently from a security point of view. This is again the case with IDS and IPS devices. Intrusion attacks are largely sourced from malicious actors on the Internet. With that in mind, placing an IPS in the ingest path of an Internet data source is perfectly reasonable. The Internet might be a hotbed of malicious actors, but the insider threat to a business is also very real and should not be overlooked. Security architecture choices must always be made under the assumption that the malicious actor works inside the same building you do. Placing an IDS inside the trusted network can be a valuable tool to warn administrators against the insider threat.

An added bonus to the discussion of IDS and IPS is the fact that logging network traffic in high volumes is a fantastic Hadoop use case. Security companies often use Hadoop to collect IDS logs across many different customer networks in order to perform a variety of large-scale analytics and visualizations of the data, which can then feed into the advancement of rules engines used by firewalls and IDS/IPS devices.

Hadoop Roles and Separation Strategies

Earlier, we mentioned that nodes in the cluster can be classified into groups to aid in setting up an adequate security policy. In this section, we take a look at how to do that. Each node plays some kind of role in the cluster, and these roles will identify which security policies are necessary to protect it. First, let’s review the common Hadoop ecosystem components and the service roles that each have (we assume that you already knows what these service roles do, but if that’s not the case, refer back to Chapter 1 for a quick review):

HDFS

: NameNode (Active/Standby/Secondary), DataNode, JournalNode, FailoverController, HttpFS, NFSGateway

MapReduce

: JobTracker (Active/Standby), TaskTracker, FailoverController

YARN

: ResourceManager (Active/Standby), NodeManager, JobHistory Server

Hive

: Hive Metastore Server, HiveServer2, WebHCatServer

Impala

: Catalog Server, StateStore Server, Impalad

Hue

: HueServer, Beeswax, KerberosTicketRenewer

Oozie

: OozieServer

ZooKeeper

: ZooKeeper Server

HBase

: Master, RegionServer, ThriftServer, RESTServer

Accumulo

: Master, TabletServer, Tracer, GarbageCollector

Solr

: SolrServer

Management and monitoring services

: Cloudera Manager, Apache Ambari, Ganglia, Nagios, Puppet, Chef, etc.

Looking at this (nonexhaustive) list, you can see that many of the various ecosystem projects have a master/worker architecture. This lends itself well to organizing the service roles from a security architecture perspective. Additionally, some of the service roles are intended to be client-facing. Overall, the separation strategy is this: identify all of the master services to be run on *master nodes*, worker services on *worker nodes*, and management services on *management nodes*. Additionally, identify which components require client configuration files to be deployed such that users can access the services. These client configuration files, along with client-facing services, are placed on *edge nodes*. The classifications of nodes are explained in more detail in the following subsections.

Master Nodes

Master nodes are likely the most important of the node groups. They contain all of the primary services that are the backbone of Hadoop. Because of the importance of these roles to the components they represent, they carry an expectation of increased security policies to protect them. Following is a list of roles that should be run on dedicated master nodes:

- HDFS NameNode, Secondary NameNode (or Standby NameNode), FailoverController, JournalNode, and KMS

- MapReduce JobTracker and FailoverController
- YARN ResourceManager and JobHistory Server
- Hive Metastore Server
- Impala Catalog Server and StateStore Server
- Sentry Server
- ZooKeeper Server
- HBase Master
- Accumulo Master, Tracer, and GarbageCollector

Armed with this list of services, the first security question to ask is: Who needs access to a master node and for what purpose? The simple answer is administrators, to perform administrative functions (surprise, surprise). Clients to the cluster, be it actual end users or third-party tools, can access all of these services remotely using the standard interfaces that are exposed. For example, a user issuing the command `hdfs dfs -ls` can do so on any machine that has the proper client configuration for the HDFS service. The user does not need to execute this command on the master node that is running the HDFS NameNode for it to succeed. With that in mind, here are several important reasons for limiting access to master nodes to administrators:

Resource contention

> If regular end users are able to use master nodes to run arbitrary programs and thus use system resources, this takes away resources that may otherwise be needed by the master node roles. This can lead to a degradation of performance.

Security vulnerabilities

> Software has inherent vulnerabilities in it, and Hadoop is no different. Allowing users to have access to the same machines that have master node roles running can open the door for exploiting unpatched vulnerabilities in the Hadoop code (maliciously or accidentally). Restricting access to master nodes lowers the risk of exposing these security vulnerabilities.

Denial of service

> Users can do crazy things. There isn't really a nicer way to say it. If end users are sharing the same machines as master node roles, it inevitably sets the stage for a user to do something (for the sake of argument, accidentally) that will take down a master process. Going back to the resource contention argument, what happens if a user launches a runaway process that fills up the log directory? Will all of the master node roles handle it gracefully if they are unable to log anymore? Does an administrator want to find out? Another example would be a similar case where a runaway process maxed out CPU or RAM on the system, with the latter easily leading to out-of-memory errors.

Worker Nodes

Worker nodes handle the bulk of what a Hadoop cluster actually does, which is store and process

data. The typical roles found on worker nodes are the following:

- HDFS DataNode
- MapReduce TaskTracker
- YARN NodeManager
- Impala Daemon
- HBase RegionServer
- Accumulo TabletServer
- SolrServer

On the surface, it might seem like all cluster users need access to these nodes because these roles handle user requests for data and processing. However, this is most often not true. Typically, only administrators need remote access to worker nodes for maintenance tasks. End users can ingest data, submit jobs, and retrieve records by utilizing the corresponding interfaces and APIs available. Most of the time, as will be elaborated on a bit later, services provide a proxy mechanism that allows administrators to channel user activity to a certain set of nodes different from the actual worker nodes. These proxies communicate with worker nodes on behalf of the user, eliminating the need for direct access.

As with master nodes, there are reasons why limiting access to worker nodes to administrators makes sense:

Resource contention

> When regular end users are performing activities on a worker node outside the expected processes, it can create skew in resource management. For example, if YARN is configured to use a certain amount of system resources based on a calculation done by a Hadoop administrator taking into account the operating system needs and other software, what about end-user activity? It is often difficult to accurately profile user activity and account for it, so it is quite likely that heavily used worker nodes will not perform well or predictably compared to worker nodes that are not being used.

Worker role skew

> If end users are using worker nodes for daily activities, it can create undesirable skew in how the roles on the worker nodes behave. For example, if end users regularly log into a particular worker node that is running the DataNode role, data ingestion from this node will create skew in disk utilization because HDFS writes will try to write the first block locally before choosing locations elsewhere in the cluster. This means that if a user is trying to upload a 10 GB file into her home directory in HDFS, all 10 GB will be written to the local DataNode they are ingesting from.

Management Nodes

Management nodes are the lifeblood for administrators. These nodes provide the mechanism to install, configure, monitor, and otherwise maintain the Hadoop cluster. The typical roles found on

these nodes are:

- Configuration management
- Monitoring
- Alerting
- Software repositories
- Backend databases

These management nodes often contain the actual software repositories for the cluster. This is especially the case when the nodes in the Hadoop cluster do not have Internet access. The most critical role hosted on a management node is configuration management software. Whether it is Hadoop specific (e.g., Cloudera Manager, Apache Ambari) or not (e.g., Puppet, Chef), this is the place where administrators will set up and configure the cluster. The corollary to configuration management is monitoring and alerting. These roles are provided by software packages like Ganglia, Nagios, and the Hadoop-specific management consoles.

It goes without saying but will be said anyway: these nodes are not for regular users. Management and maintenance of a Hadoop cluster is an administrative function and thus should be protected as such. That being said, there are exceptions to the rule. A common exception is for developers to have access to cluster monitoring dashboards to observe metrics while jobs are running so they can ascertain performance characteristics of their code.

Edge Nodes

Edge nodes are the nodes that all of the users of the Hadoop cluster care about. These nodes host web interfaces, proxies, and client configurations that ultimately provide the mechanism for users to take advantage of the combined storage and computing system that is Hadoop. The following roles are typically found on edge nodes:

- HDFS HttpFS and NFS gateway
- Hive HiveServer2 and WebHCatServer
- Network proxy/load balancer for Impala
- Hue server and Kerberos ticket renewer
- Oozie server
- HBase Thrift server and REST server
- Flume agent
- Client configuration files

When looking at the list of common roles found on edge nodes, it becomes apparent that this node class is a bit different than the others. Edge nodes in general might not be treated as equivalent to one another, as is often the case with the other node classes. For example, ingest pipelines using

Flume agents will likely be on edge nodes not accessible by users, while edge nodes housing client configurations to facilitate command-line access would be accessible by users. How granular the classification of nodes within the edge node group will be dependent on a variety of factors, including cluster size and use cases. Here are some examples of further classifying edge nodes:

Data Gateway

: HDFS HttpFS and NFS gateway, HBase Thrift server and REST server, Flume agent

SQL Gateway

: Hive HiveServer2 and WebHCatServer, Impala load-balancing proxy (e.g., HAProxy)

User Portal

: Hue server and Kerberos ticket renewer, Oozie server, client configuration files

Tip

While the Impala daemon does not have to be collocated with an HDFS DataNode, it is not recommended to use a standalone Impala daemon as a proxy. A better option is to use a load-balancing proxy, such as HAProxy, to act as a load balancer. This is the recommended architecture in the case where clients cannot connect directly to an Impala daemon on a worker node because of a firewall or other restrictions.

Using the additional edge node classifications shown, it becomes easier to break down which nodes users are expected to have remote access to, and which nodes are only accessible remotely through the configured remote ports. While users need remote access to the user portal nodes to interact with the cluster from a shell, it is quite reasonable that both the data and SQL gateways are not accessible in this way. These nodes are accessible only via remote ports, which facilitates access to both command-line tools executed on the user portal, as well as additional business intelligence tools that might reside somewhere else in the network.

The groupings shown are just examples. It is important to understand not only the services installed in the cluster but also how the services are used and by whom. This circles back to earlier discussions about knowing the users and the operating environment.

Operating System Security

This section digs into how individual nodes should be protected at the operating-system level.

Remote Access Controls

In a typical server environment, remote access controls are pretty straightforward. For example, a server that hosts an RDBMS or web server is likely locked down to end users, allowing only privileged users and administrators to log into the machine. A Hadoop environment is not so simple. Because of the inherent complexity of the Hadoop ecosystem, a myriad of tools and access methods are available to interact with the cluster, in addition to the typical roles and responsibilities for basic administration.

While Hadoop clusters can span thousands of nodes, these nodes can be classified into groups, as we will see a bit later in this chapter. With that in mind, it is important to consider limiting remote access to machines by identifying which machines need to be accessed and why. Armed with this information, a remote access policy can be made to restrict remote access (typically SSH) to authorized users. On the surface, it might seem that authorized users are analogous to users of the Hadoop cluster, but this is typically not the case. For example, a developer writing Java MapReduce code or Pig scripts will likely require command-line access to one or more nodes in the cluster, whereas an analyst writing SQL queries for Hive and Impala might not need this access at all if they are using Hue or third-party business intelligence (BI) tools to interact with the cluster.

Host Firewalls

Remote access controls are a good way to limit which users are able to log into a given machine in the cluster. This is useful and necessary, but it is only a small component of protecting a given machine in the cluster. Host firewalls are an incredibly useful tool to limit the types of traffic going into and out of a node. In Linux systems, host firewalls are typically implemented using *iptables*. Certainly there are other third-party software packages that perform this function as well (e.g., commercial software), but we will focus on iptables, as it is largely available by default in most Linux distributions.

In order to leverage iptables, we must first understand and classify the network traffic in a Hadoop cluster. Table 3-1 shows common ports that are used by Hadoop ecosystem components. We will use this table to start building a host firewall policy for iptables.

Table 3-1. Common Hadoop service ports

Component	Service	Port(s)
Accumulo	Master	9999
	GarbageCollector	50091
	Tracer	12234
	ProxyServer	42424
	TabletServer	9997
	Monitor	4560, 50095
Cloudera Impala	Catalog Server	25020, 26000
	StateStore	24000, 25010
	Daemon	21000, 21050, 22000, 23000, 25000, 28000
	Llama ApplicationMaster	15000, 15001, 15002

Component	Service	Port(s)
Flume	Agent	41414
HBase	Master	60000, 60010
	REST Server	8085, 20550
	Thrift Server	9090, 9095
	RegionServer	60020, 60030
HDFS	NameNode	8020, 8022, 50070, 50470
	SecondaryNameNode	50090, 50495
	DataNode	1004, 1006, 50010, 50020, 50075, 50475
	JournalNode	8480, 8485
	HttpFS	14000, 14001
	NFS Gateway	111, 2049, 4242
	KMS	16000, 16001
Hive	Hive Metastore Server	9083
	HiveServer2	10000
	WebHCat Server	50111
Hue	Server	8888
MapReduce	JobTracker	8021, 8023, 9290, 50030
	FailoverController	8018
	TaskTracker	4867, 50060
Oozie	Server	11000, 11001, 11443
Sentry	Server	8038, 51000
Solr	Server	8983, 8984
YARN	ResourceManager	8030, 8031, 8032, 8033, 8088, 8090

Component	Service	Port(s)
	JobHistory Server	10020, 19888, 19890
	NodeManager	8040, 8041, 8042, 8044
ZooKeeper	Server	2181, 3181, 4181, 9010

Now that we have the common ports listed, we need to understand how strict of a policy needs to be enforced. Configuring iptables rules involves both ports and IP addresses, as well as the direction of communication. A typical basic firewall policy allows any host to reach the allowed ports, and all return (established) traffic is allowed. An example iptables policy for an HDFS NameNode might look like the one in Example 3-1.

Example 3-1. Basic NameNode iptables policy

```
iptables -N hdfs
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 8020 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 8022 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50070 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50470 -j ACCEPT
iptables -A INPUT -j hdfs
```

This policy is more relaxed in that it allows all hosts (0.0.0.0/0) to connect to the machine over the common HDFS NameNode service ports. However, this might be too open a policy. Let us say that the Hadoop cluster nodes are all part of the 10.1.1.0/24 subnet. Furthermore, a dedicated edge node is set up on the host 10.1.1.254 for all communication to the cluster. Finally, SSL is enabled for web consoles. The adjusted iptables policy for the NameNode machine might instead look like the one in Example 3-2.

Example 3-2. Secure NameNode iptables policy

```
iptables -N hdfs
iptables -A hdfs -p tcp -s 10.1.1.254/32 --dport 8020 -j ACCEPT
iptables -A hdfs -p tcp -s 10.1.1.254/32 --dport 8022 -j DROP
iptables -A hdfs -p tcp -s 10.1.1.0/24 --dport 8022 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50470 -j ACCEPT
iptables -A INPUT -j hdfs
```

The adjusted policy is now a lot more restrictive. It allows any user to get to the NameNode web console over SSL (port 50470), only cluster machines to connect to the NameNode over the dedicated DataNode RPC port (8022), and user traffic to the NameNode RPC port (8020) to occur only from the edge node.

Note

It might be necessary to insert the iptables jump target to a specific line number in the `INPUT` section of your policy for it to take effect. An append is shown for simplicity.

Chapter 4. Kerberos

Kerberos often intimidates even experienced system administrators and developers at the first mention of it. Applications and systems that rely on Kerberos often have many support calls and

trouble tickets filed to fix problems related to it. This chapter will introduce the basic Kerberos concepts that are necessary to understand how strong authentication works, and explain how it plays an important role with Hadoop authentication in Chapter 5.

So what exactly *is* Kerberos? From a mythological point of view, Kerberos is the Greek word for *Cerberus*, a multiheaded dog that guards the entrance to Hades to ensure that nobody who enters will ever leave. Kerberos from a technical (and more pleasant) point of view is the term given to an authentication mechanism developed at Massachusetts Institute of Technology (MIT). Kerberos evolved to become the de facto standard for strong authentication for computer systems large and small, with varying implementations ranging from MIT's Kerberos distribution to the authentication component of Microsoft's Active Directory.

Why Kerberos?

Playing devil's advocate here (pun intended), why does Hadoop need Kerberos at all? The reason becomes apparent when looking at the default model for Hadoop authentication. When presented with a username, Hadoop happily believes whatever you tell it, and ensures that every machine in the entire cluster believes it, too.

To use an analogy, if a person at a party approached you and introduced himself as "Bill," you naturally would believe that he is, in fact, Bill. How do you know that he really is Bill? Well, because he said so and you believed him without question. Hadoop without Kerberos behaves in much the same way, except that, to take the analogy a step further, Hadoop not only believes "Bill" is who he says he is but makes sure that everyone else believes it, too. This is a problem.

Hadoop by design is meant to store and process petabytes of data. As the old adage goes, with great power comes great responsibility. Hadoop in the enterprise can no longer get by with simplistic means for identifying (and trusting) users. Enter Kerberos. In the previous analogy, "Bill" introduces himself to you. Upon doing so, what if you responded by asking to see a valid passport and upon receiving it (naturally, because everyone brings a passport to a party…), checked the passport against a database to verify validity? This is the type of identify verification that Hadoop introduced by adding Kerberos authentication.

Kerberos Overview

The stage is now set and it is time to dig in and understand just how Kerberos works. Kerberos implementation is, as you might imagine, a client/server architecture. Before breaking down the components in detail, a bit of Kerberos terminology is needed.

First, identities in Kerberos are called *principals*. Every user and service that participates in the Kerberos authentication protocol requires a principal to uniquely identify itself. Principals are classified into two categories: *user* principals and *service* principals. *User principal names*, or UPNs, represent regular users. This closely resembles usernames or accounts in the operating system world. *Service principal names*, or SPNs, represent services that a user needs to access, such as a database on a specific server. The relationship between UPNs and SPNs will become more apparent when we work through an example later.

The next important Kerberos term is *realm*. A Kerberos realm is an authentication administrative domain. All principals are assigned to a specific Kerberos realm. A realm establishes a boundary, which makes administration easier.

Now that we have established what principals and realms are, the natural next step is to understand what stores and controls all of this information. The answer is a *key distribution center (KDC)*. The KDC is comprised of three components: the Kerberos database, the *authentication service (AS)*, and the *ticket-granting service (TGS)*. The Kerberos database stores all the information about the principals and the realm they belong to, among other things. Kerberos principals in the database are identified with a naming convention that looks like the following:

`alice@EXAMPLE.COM`

> A UPN that uniquely identifies the user (also called the *short name*): `alice` in the Kerberos realm `EXAMPLE.COM`. By convention, the realm name is always uppercase.

`bob/admin@EXAMPLE.COM`

> A variation of a regular UPN in that it identifies an administrator `bob` for the realm `EXAMPLE.COM`. The slash (/) in a UPN separates the short name and the admin distinction. The `admin` component convention is regularly used, but it is configurable as we will see later.

`hdfs/node1.example.com@EXAMPLE.COM`

> This principal represents an SPN for the `hdfs` service, on the host `node1.example.com`, in the Kerberos realm `EXAMPLE.COM`. The slash (/) in an SPN separates the short name `hdfs` and the hostname `node1.example.com`.

Note

The entire principal name is case sensitive! For instance, `hdfs/Node1.Hadoop.com@EXAMPLE.COM` is a different principal than the one in the third example. Typically, it is best practice to use all lowercase for the principal, except for the realm component, which is uppercase. The caveat here is, of course, that the underlying hostnames referred to in SPNs are also lowercase, which is also a best practice for host naming and DNS.

The second component of the KDC, the AS, is responsible for issuing a ticket-granting ticket (TGT) to a client when they initiate a request to the AS. The TGT is used to request access to other services.

The third component of the KDC, the TGS, is responsible for validating TGTs and granting *service tickets*. Service tickets allow an authenticated principal to use the service provided by the application server, identified by the SPN. The process flow of obtaining a TGT, presenting it to the TGS, and obtaining a service ticket is explained in the next section. For now, understand that the KDC has two components, the AS and TGS, which handle requests for authentication and access to services.

Note

There is a special principal of the form `krbtgt/<REALM>@<REALM>` within the Kerberos

database, such as `krbtgt/EXAMPLE.COM@EXAMPLE.COM`. This principal is used internally by both the AS and the TGS. The key for this principal is actually used to encrypt the content of the TGT that is issued to clients, thus ensuring that the TGT issued by the AS can only be validated by the TGS.

Table 4-1 provides a summary of the Kerberos terms and abbreviations introduced in this chapter.

Table 4-1. Kerberos term abbreviations

Term	Name	Description
UPN	User principal name	A principal that identifies a user in a given realm, with the format *<shortname><@REALM>* or *<shortname>/admin@<REALM>*
SPN	Service principal name	A principal that identifies a service on a specific host in a given realm, with the format *<shortname>/<hostname>@<REALM>*
TGT	Ticket-granting ticket	A special ticket type granted to a user after successfully authenticating to the AS
KDC	Key distribution center	A Kerberos server that contains three components: Kerberos database, AS, and TGS
AS	Authentication service	A KDC service that issues TGTs
TGS	Ticket-granting service	A KDC service that validates TGTs and grants service tickets

What has been presented thus far are a few of the basic Kerberos components needed to understand authentication at a high level. Kerberos in its own right is a very in-depth and complex topic that warrants an entire book on the subject. Thankfully, that has already been done. If you wish to dive far deeper than what is presented here, take a look at Jason Garman's excellent book, *Kerberos: The Definitive Guide* (O'Reilly).

Kerberos Workflow: A Simple Example

Now that the terminology and components have been introduced, we can now work through an example workflow showing how it all works at a high level. First, we will identify all of the components in play:

`EXAMPLE.COM`

: The Kerberos realm

Alice

: A user of the system, identified by the UPN `alice@EXAMPLE.COM`

`myservice`

: A service that will be hosted on `server1.example.com`, identified by the SPN `myservice/server1.example.com@EXAMPLE.COM`

`kdc.example.com`

: The KDC for the Kerberos realm `EXAMPLE.COM`

In order for Alice to use `myservice`, she needs to present a valid service ticket to `myservice`. The following list of steps shows how she does this (some details omitted for brevity):

1. Alice needs to obtain a TGT. To do this, she initiates a request to the AS at `kdc.example.com`, identifying herself as the principal `alice@EXAMPLE.COM`.
2. The AS responds by providing a TGT that is encrypted using the key (password) for the principal `alice@EXAMPLE.COM`.
3. Upon receipt of the encrypted message, Alice is prompted to enter the correct password for the principal `alice@EXAMPLE.COM` in order to decrypt the message.
4. After successfully decrypting the message containing the TGT, Alice now requests a service ticket from the TGS at `kdc.example.com` for the service identified by `myservice/server1.example.com@EXAMPLE.COM`, presenting the TGT along with the request.
5. The TGS validates the TGT and provides Alice a service ticket, encrypted with the `myservice/server1.example.com@EXAMPLE.COM` principal's key.
6. Alice now presents the service ticket to `myservice`, which can then decrypt it using the `myservice/server1.example.com@EXAMPLE.COM` key and validate the ticket.
7. The service `myservice` permits Alice to use the service because she has been properly authenticated.

This shows how Kerberos works at a high level. Obviously this is a greatly simplified example and many of the underlying details have not been presented. See Figure 4-1 for a sequence diagram of this example.

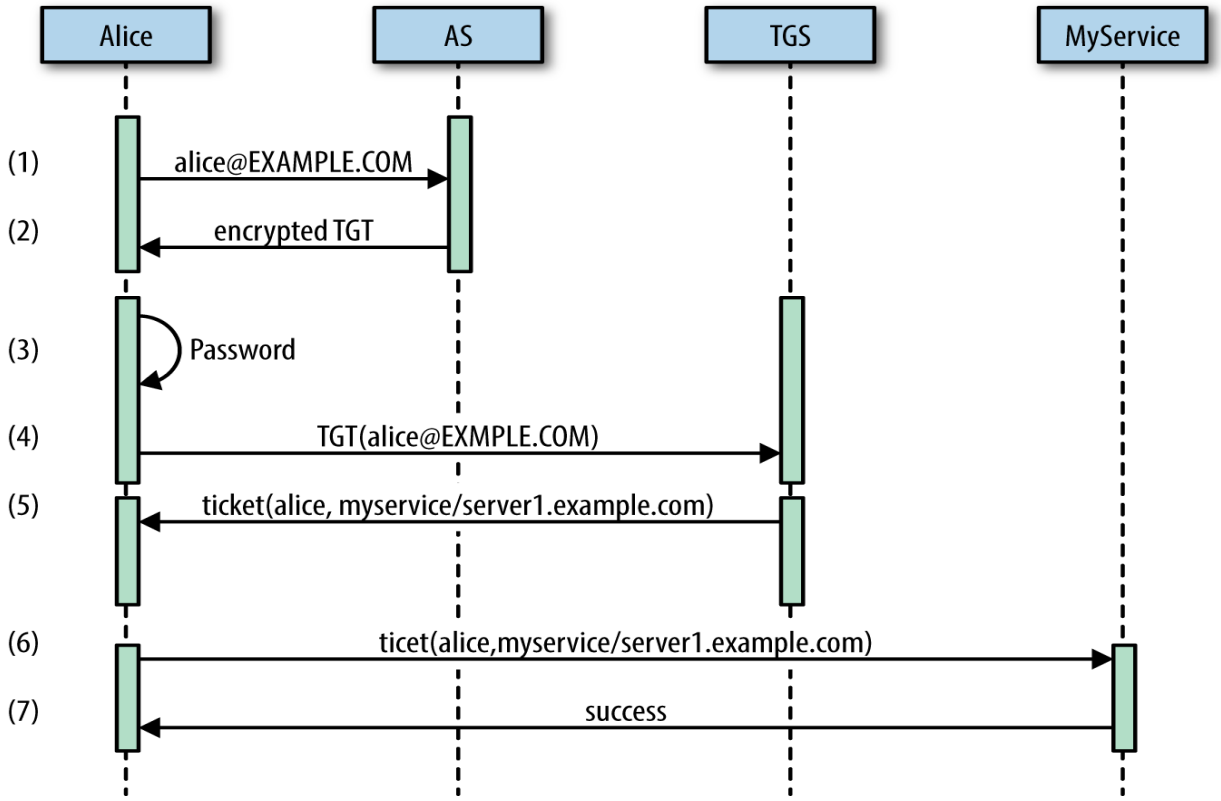

Figure 4-1. Kerberos workflow example

Kerberos Trusts

So far, Kerberos has been introduced under the implicit expectation that all users and services are contained within a single Kerberos realm. While this works well for introductory material, it is often not realistic given how large enterprises work. Over time, large enterprises end up with multiple Kerberos realms from things like mergers, acquisitions, or just simply wanting to segregate different parts of the enterprise. However, by default, a KDC only knows about its own realm and the principals in its own database. What if a user from one realm wants to use a service that is controlled by another realm? In order to make this happen, a Kerberos *trust* is needed between the two realms.

For example, suppose that Example is a very large corporation and has decided to create multiple realms to identify different lines of business, including `HR.EXAMPLE.COM` and `MARKETING.EXAMPLE.COM`. Because users in both realms might need to access services from both realms, the KDC for `HR.EXAMPLE.COM` needs to trust information from the `MARKETING.EXAMPLE.COM` realm and vice versa.

On the surface this seems pretty straightforward, except that there are actually two different types of trusts: *one-way trust* and *two-way trust* (sometimes called *bidirectional trust* or *full trust*). The example we just looked at represents a two-way trust.

What if there is also a `DEV.EXAMPLE.COM` realm where developers have principals that need to access the `DEV.EXAMPLE.COM` and `MARKETING.EXAMPLE.COM` realms, but marketing users should not be able to access the `DEV.EXAMPLE.COM` realm? This scenario requires a one-way

trust. A one-way trust is very common in Hadoop deployments when a KDC is installed and configured to contain all the information about the SPNs for the cluster nodes, but all UPNs for end users exist in a different realm, such as Active Directory. Oftentimes, Active Directory administrators or corporate policies prohibit full trusts for a variety of reasons.

So how does a Kerberos trust actually get established? Earlier in the chapter it was noted that a special principal is used internally by the AS and TGS, and it is of the form `krbtgt/<REALM>@<REALM>`. This principal becomes increasingly important for establishing trusts. With trusts, the principal instead takes the form of `krbtgt/<TRUSTING_REALM>@<TRUSTED_REALM>`. A key concept of this principal is that it exists in *both* realms. For example, if the `HR.EXAMPLE.COM` realm needs to trust the `MARKETING.EXAMPLE.COM` realm, the principal `krbtgt/HR.EXAMPLE.COM@MARKETING.EXAMPLE.COM` needs to exist in both realms.

Warning

The password for the `krbtgt/<TRUSTING_REALM>@<TRUSTED_REALM>` principal and the encryption types used *must* be the same in both realms in order for the trust to be established.

The previous example shows what is required for a one-way trust. In order to establish a full trust, the principal `krbtgt/MARKETING.EXAMPLE.COM@HR.EXAMPLE.COM` also needs to exist in both realms. To summarize, for the `HR.EXAMPLE.COM` realm to have a full trust with the `MARKETING.EXAMPLE.COM` realm, both realms need the principals `krbtgt/MARKETING.EXAMPLE.COM@HR.EXAMPLE.COM` and `krbtgt/HR.EXAMPLE.COM@MARKETING.EXAMPLE.COM`.

MIT Kerberos

As mentioned in the beginning of this chapter, Kerberos was first created at MIT. Over the years, it has undergone several revisions and the current version is *MIT Kerberos V5*, or *krb5* as it is often called. This section covers some of the components of the MIT Kerberos distribution to put some real examples into play with the conceptual examples introduced thus far.

Tip

For the most up-to-date definitive resource on the MIT Kerberos distribution, consult the excellent documentation at the official project website.

In the earlier example, we glossed over the fact that Alice initiated an authentication request. In practice, Alice does this by using the *kinit* tool (Example 4-1).

Example 4-1. kinit using the default user

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$
```

This example pairs the current Linux username *alice* with the *default realm* to come up with the suggested principal `alice@EXAMPLE.COM`. The default realm is explained later when we dive

into the configuration files. The `kinit` tool also allows the user to explicitly identify the principal to authenticate as (Example 4-2).

Example 4-2. kinit using a specified user

```
[alice@server1 ~]$ kinit alice/admin@EXAMPLE.COM
Enter password for alice/admin@EXAMPLE.COM:
[alice@server1 ~]$
```

Explicitly providing a principal name is often necessary to authenticate as an administrative user, as the preceding example depicts. Another option for authentication is by using a *keytab file*. A keytab file stores the actual encryption key that can be used in lieu of a password challenge for a given principal. Creating keytab files are useful for noninteractive principals, such as SPNs, which are often associated with long-running processes like Hadoop daemons. A keytab file does not have to be a 1:1 mapping to a single principal. Multiple different principal keys can be stored in a single keytab file. A user can use `kinit` with a keytab file by specifying the keytab file location, and the principal name to authenticate as (again, because multiple principal keys may exist in the keytab file), shown in Example 4-3.

Example 4-3. kinit using a keytab file

```
[alice@server1 ~]$ kinit -kt alice.keytab alice/admin@EXAMPLE.COM
[alice@server1 ~]$
```

Tip

The keytab file allows a user to authenticate without knowledge of the password. Because of this fact, keytabs should be protected with appropriate controls to prevent unauthorized users from authenticating with it. This is especially important when keytabs are created for administrative principals!

Another useful utility that is part of the MIT Kerberos distribution is called `klist`. This utility allows users to see what, if any, Kerberos credentials they have in their *credentials cache*. The credentials cache is the place on the local filesystem where, upon successful authentication to the AS, TGTs are stored. By default, this location is usually the file */tmp/krb5cc_<uid>* where *<uid>* is the numeric user ID on the local system. After a successful `kinit`, alice can view her credentials cache with `klist`, as shown in Example 4-4.

Example 4-4. Viewing the credentials cache with klist

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_5000
Default principal: alice@EXAMPLE.COM

Valid starting     Expires            Service principal
02/13/14 12:00:27  02/14/14 12:00:27  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/20/14 12:00:27
[alice@server1 ~]$
```

If a user tries to look at the credentials cache without having authenticated first, no credentials will be found (see Example 4-5).

Example 4-5. No credentials cache found

```
[alice@server1 ~]$ klist
No credentials cache found (ticket cache FILE:/tmp/krb5cc_5000
[alice@server1 ~]$
```

Another useful tool in the MIT Kerberos toolbox is `kdestroy`. As the name implies, this allows users to destroy credentials in their credentials cache. This is useful for switching users, or when trying out or debugging new configurations (see Example 4-6).

Example 4-6. Destroying the credentials cache with kdestroy

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_5000
Default principal: alice@EXAMPLE.COM

Valid starting     Expires            Service principal
02/13/14 12:00:27  02/14/14 12:00:27  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/20/14 12:00:27
[alice@server1 ~]$ kdestroy
[alice@server1 ~]$ klist
No credentials cache found (ticket cache FILE:/tmp/krb5cc_5000
[alice@server1 ~]$
```

So far, all of the MIT Kerberos examples shown "just work." Hidden away in these examples is the fact that there is a fair amount of configuration necessary to make it all work, both on the client and server side. The next two sections present basic configurations to tie together some of the concepts that have been presented thus far.

Server Configuration

Kerberos server configuration is primarily specified in the *kdc.conf* file, which is shown in Example 4-7. This file lives in */var/kerberos/krb5kdc/* on Red Hat/CentOS systems.

Example 4-7. kdc.conf

```
[kdcdefaults]
 kdc_ports = 88
 kdc_tcp_ports = 88

[realms]
 EXAMPLE.COM = {
  acl_file = /var/kerberos/krb5kdc/kadm5.acl
  dict_file = /usr/share/dict/words
  supported_enctypes = aes256-cts:normal aes128-cts:normal arcfour-hmac-
md5:normal
  max_renewable_life = 7d
 }
```

The first section, `kdcdefaults`, contains configurations that apply to all the realms listed, unless the specific realm configuration has values for the same configuration items. The configurations `kdc_ports` and `kdc_tcp_ports` specify the UDP and TCP ports the KDC should listen on, respectively. The next section, `realms`, contains all of the realms that the KDC is the server for. A single KDC can support multiple realms. The realm configuration items from this example are as follows:

`acl_file`

This specifies the file location to be used by the admin server for access controls (more on this later).

`dict_file`

This specifies the file that contains words that are not allowed to be used as passwords because they are easily cracked/guessed.

`supported_enctypes`

This specifies all of the encryption types supported by the KDC. When interacting with the KDC, clients must support at least one of the encryption types listed here. Be aware of using weak encryption types, such as DES, because they are easily exploitable.

`max_renewable_life`

This specifies the maximum amount of time that a ticket can be renewable. Clients can request a renewable lifetime up to this length. A typical value is seven days, denoted by `7d`.

Note

By default, encryption settings in MIT Kerberos are often set to a variety of encryption types, including weak choices such as DES. When possible, remove weak encryption types to ensure the best possible security. Weak encryption types are easily exploitable and well documented as such. When using AES-256, Java Cryptographic Extensions need to be installed on all nodes in the cluster to allow for unlimited strength encryption types. It is important to note that some countries prohibit the usage of these encryption types. Always follow the laws governing encryption strength for your country. A more detailed discussion of encryption is provided in Chapter 9.

The `acl_file` location (typically the file *kadm5.acl*) is used to control which users have privileged access to administer the Kerberos database. Administration of the Kerberos database is controlled by two different, but related, components: `kadmin.local` and `kadmin`. The first is a utility that allows the *root* user of the KDC server to modify the Kerberos database. As the name implies, it can *only* be run by the *root* user on the same machine where the Kerberos database resides. Administrators wishing to administer the Kerberos database remotely must use the `kadmin` server.

The `kadmin` server is a daemon process that allows remote connections to administer the Kerberos database. This is where the *kadm5.acl* file (shown in Example 4-8) comes into play. The `kadmin` utility uses Kerberos authentication, and the *kadm5.acl* file specifies which UPNs are allowed to perform privileged functions.

Example 4-8. kadm5.acl

```
*/admin@EXAMPLE.COM      *
cloudera-scm@EXAMPLE.COM *     hdfs/*@EXAMPLE.COM
cloudera-scm@EXAMPLE.COM *     mapred/*@EXAMPLE.COM
```

This allows any principal from the `EXAMPLE.COM` realm with the `/admin` distinction to perform any administrative action. While it is certainly acceptable to change the `admin` distinction to some other arbitrary name, it is recommended to follow the convention for simplicity and maintainability. Administrative users should only use their admin credentials for specific privileged actions, much in the same way administrators should not use the *root* user in Linux for everyday nonadministrative actions.

The example also shows how the ACL can be defined to restrict privileges to a *target* principal. It demonstrates that the user *cloudera-scm* can perform any action but only on SPNs that start with `hdfs` and `mapred`. This type of syntax is useful to grant access to a third-party tool to create and administer Hadoop principals, but not grant access to all of the admin functions.

As mentioned earlier, the `kadmin` tool allows for administration of the Kerberos database. This tool brings users to a shell-like interface where various commands can be entered to perform operations against the Kerberos database (see Examples 4-9 through 4-12.

Example 4-9. Adding a new principal to the Kerberos database

```
kadmin: addprinc alice@EXAMPLE.COM
WARNING: no policy specified for alice@EXAMPLE.COM; defaulting to no policy
Enter password for principal "alice@EXAMPLE.COM":
Re-enter password for principal "alice@EXAMPLE.COM":
Principal "alice@EXAMPLE.COM" created.
kadmin:
```

Example 4-10. Displaying the details of a principal in the Kerberos database

```
kadmin: getprinc alice@EXAMPLE.COM
Principal: alice@EXAMPLE.COM
Expiration date: [never]
Last password change: Tue Feb 18 20:48:15 EST 2014
Password expiration date: [none]
Maximum ticket life: 1 day 00:00:00
Maximum renewable life: 7 days 00:00:00
Last modified: Tue Feb 18 20:48:15 EST 2014 (root/admin@EXAMPLE.COM)
Last successful authentication: [never]
Last failed authentication: [never]
Failed password attempts: 0
Number of keys: 2
Key: vno 1, aes256-cts-hmac-sha1-96, no salt
Key: vno 1, aes128-cts-hmac-sha1-96, no salt
MKey: vno1
Attributes:
Policy: [none]
kadmin:
```

Example 4-11. Deleting a principal from the Kerberos database

```
kadmin: delprinc alice@EXAMPLE.COM
Are you sure you want to delete the principal "alice@EXAMPLE.COM"? (yes/no): yes
Principal "alice@EXAMPLE.COM" deleted.
Make sure that you have removed this principal from all ACLs before reusing.
kadmin:
```

Example 4-12. Listing all the principals in the Kerberos database

```
kadmin: listprincs
HTTP/server1.example.com@EXAMPLE.COM
K/M@EXAMPLE.COM
```

```
bob@EXAMPLE.COM
flume/server1.example.com@EXAMPLE.COM
hdfs/server1.example.com@EXAMPLE.COM
hdfs@EXAMPLE.COM
hive/server1.example.com@EXAMPLE.COM
hue/server1.example.com@EXAMPLE.COM
impala/server1.example.com@EXAMPLE.COM
kadmin/admin@EXAMPLE.COM
kadmin/server1.example.com@EXAMPLE.COM
kadmin/changepw@EXAMPLE.COM
krbtgt/EXAMPLE.COM@EXAMPLE.COM
mapred/server1.example.com@EXAMPLE.COM
oozie/server1.example.com@EXAMPLE.COM
yarn/server1.example.com@EXAMPLE.COM
zookeeper/server1.example.com@EXAMPLE.COM
kadmin:
```

Client Configuration

The default Kerberos client configuration file is typically named *krb5.conf*, and lives in the */etc/* directory on Unix/Linux systems. This configuration file is read whenever client applications need to use Kerberos, including the `kinit` utility. The *krb5.conf* shown in Example 4-13 configuration file is minimally configured from the default that comes with Red Hat/CentOS 6.4.

Example 4-13. krb5.conf

```
[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 default_realm = DEV.EXAMPLE.COM
 dns_lookup_realm = false
 dns_lookup_kdc = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 default_tkt_enctypes = aes256-cts aes128-cts
 default_tgs_enctypes = aes256-cts aes128-cts
 udp_preference_limit = 1

[realms]
 EXAMPLE.COM = {
  kdc = kdc.example.com
  admin_server = kdc.example.com
 }

 DEV.EXAMPLE.COM = {
   kdc = kdc.dev.example.com
   admin_server = kdc.dev.example.com
 }

[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
 .dev.example.com = DEV.EXAMPLE.COM
 dev.example.com = DEV.EXAMPLE.COM
```

In this example, there are several different sections. The first, `logging`, is self-explanatory. It

defines where logfiles are stored for the various Kerberos components that generate log events. The second section, `libdefaults`, contains general default configuration information. Let's take a closer look at the individual configurations in this section:

`default_realm`

> This defines what Kerberos realm should be assumed if no realm is provided. This is right in line with the earlier `kinit` example when a realm was not provided.

`dns_lookup_realm`

> DNS can be used to determine what Kerberos realm to use.

`dns_lookup_kdc`

> DNS can be used to find the location of the KDC.

`ticket_lifetime`

> This specifies how long a ticket lasts for. This can be any length of time up to the maximum specified by the KDC. A typical value is 24 hours, denoted by `24h`.

`renew_lifetime`

> This specifies how long a ticket can be *renewed* for. Tickets can be renewed by the KDC without having a client reauthenticate. This must be done prior to tickets expiring.

`forwardable`

> This specifies that tickets can be *forwardable*, which means that if a user has a TGT already but logs into a different remote system, the KDC can automatically reissue a new TGT without the client having to reauthenticate.

`default_tkt_enctypes`

> This specifies the encryption types to use for session keys when making requests to the AS. Preference from highest to lowest is left to right.

`default_tgs_enctypes`

> This specifies the encryption types to use for session keys when making requests to the TGS. Preference from highest to lowest is left to right.

`udp_preference_limit`

> This specifies the maximum packet size to use before switching to TCP instead of UDP. Setting this to 1 forces TCP to always be used.

The next section, `realms`, lists all the Kerberos realms that the client is aware of. The `kdc` and `admin_server` configurations tell the client which server is running the KDC and kadmin

processes, respectively. These configurations can specify the port along with the hostname. If no port is specified, it is assumed to use port 88 for the KDC and 749 for admin server. In this example, two realms are shown. This is a common configuration where a one-way trust exists between two realms, and clients need to know about both realms. In this example, perhaps the `EXAMPLE.COM` realm contains all of the end-user principals and `DEV.EXAMPLE.COM` contains all of the Hadoop service principals for a development cluster. Setting up Kerberos in this fashion allows users of this dev cluster to use their existing credentials in `EXAMPLE.COM` to access it.

The last section, `domain_realm`, maps DNS names to Kerberos realms. The first entry says all hosts under the `example.com` domain map to the `EXAMPLE.COM` realm, while the second entry says that `example.com` itself maps to the `EXAMPLE.COM` realm. This is similarly the case with `dev.example.com` and `DEV.EXAMPLE.COM`. If no matching entry is found in this section, the client will try to use the domain portion of the DNS name (converted to all uppercase) as the realm name.

Summary

The important takeaway from this chapter is that Kerberos authentication is a multistep client/server process to provide strong authentication of both users *and* services. We took a look at the MIT Kerberos distribution, which is a popular implementation choice. While this chapter covered some of the details of configuring the MIT Kerberos distribution, we strongly encourage you to refer to the official MIT Kerberos documentation, as it is the most up-to-date reference for the latest distribution; in addition, it serves as a more detailed guide about all of the configuration options available to a security administrator for setting up a Kerberos environment.

In the next chapter, the Kerberos concepts covered thus far will be taken a step further by putting them into the context of core Hadoop and the extended Hadoop ecosystem.

Part II. Authentication, Authorization, and Accounting

Chapter 5. Identity and Authentication

The first step necessary for any system securing data is to provide each user with a unique identity and to authenticate a user's claim of a particular identity. The reason authentication and identity are so essential is that no authorization scheme can control access to data if the scheme can't trust that users are who they claim to be.

In this chapter, we'll take a detailed look at how authentication and identity are managed for core Hadoop services. We start by looking at identity and how Hadoop integrates information from Kerberos KDCs and from LDAP and Active Directory domains to provide an integrated view of distributed identity. We'll also look at how Hadoop represents users internally and the options for

mapping external, global identities to those internal representations. Next, we revisit Kerberos and go into more details of how Hadoop uses Kerberos for strong authentication. From there, we'll take a look at how some core components use username/password–based authentication schemes and the role of distributed authentication tokens in the overall architecture. We finish the chapter with a discussion of user impersonation and a deep dive into the configuration of Hadoop authentication.

Identity

In the context of the Hadoop ecosystem, identity is a relatively complex topic. This is due to the fact that Hadoop goes to great lengths to be loosely coupled from authoritative identity sources. In Chapter 4, we introduced the Kerberos authentication protocol, a topic that will figure prominently in the following section, as it's the default secure authentication protocol used in Hadoop. While Kerberos provides support for robust authentication, it provides very little in the way of advanced identity features such as groups or roles. In particular, Kerberos exposes identity as a simple two-part string (or in the case of services, three-part string) consisting of a short name and a realm. While this is useful for giving every user a unique identifier, it is insufficient for the implementation of a robust authorization protocol.

In addition to users, most computing systems provide groups, which are typically defined as a collection of users. Because one of the goals of Hadoop is to integrate with existing enterprise systems, Hadoop took the pragmatic approach of using a pluggable system to provide the traditional group concept.

Mapping Kerberos Principals to Usernames

Before diving into more details on how Hadoop maps users to groups, we need to discuss how Hadoop translates Kerberos principal names to usernames. Recall from Chapter 4 that Kerberos uses a two-part string (e.g., `alice@EXAMPLE.COM`) or three-part string (e.g., `hdfs/namenode.example.com@EXAMPLE.COM`) that contains a short name, realm, and an optional instance name or hostname. To simplify working with usernames, Hadoop maps Kerberos principal names to local usernames. Hadoop can use the `auth_to_local` setting in the *krb5.conf* file, or Hadoop-specific rules can be configured in the `hadoop.security.auth_to_local` parameter in the *core-site.xml* file.

The value of `hadoop.security.auth_to_local` is set to one or more rules for mapping principal names to local usernames. A rule can either be the value `DEFAULT` or the string `RULE:` followed by three parts: the initial principal translation, the acceptance filter, and the substitution command. The special value `DEFAULT` maps names in Hadoop's local realm to just the first component (e.g., `alice/admin@EXAMPLE.COM` is mapped to `alice` by the `DEFAULT` rule).

The initial principal translation

The initial principal translation consists of a number followed by the substitution string. The number matches the number of components, not including the realm, of the principal. The

substitution string defines how the principal will be initially translated. The variable `$0` will be substituted with the realm, `$1` will be substituted with the first component, and `$2` will be substituted with the second component. See Table 5-1 for some example initial principal translations. The format of the initial principal translation is `[<number>:<string>]` and the output is called the *initial local name*.

Table 5-1. Example principal translations

Principal translation	Initial local name for `alice@EXAMPLE.com`	Initial local name for `hdfs/namenode.example.com@EXAMPLE.COM`
`[1:$1.$0]`	`alice.EXAMPLE.COM`	*No match*
`[1:$1]`	`alice`	*No match*
`[2:$1_$2@$0]`	No match	`hdfs_namenode.example.com@EXAMPLE.COM`
`[2:$1@$0]`	No match	`hdfs@EXAMPLE.COM`

The acceptance filter

The acceptance filter is a regular expression, and if the initial local name (i.e., the output from the first part of the rule) matches the regular expression, then the substitution command will be run over the string. The initial local name only matches if the entire string is matched by the regular expression. This is equivalent to having the regular expression start with a `^` and end with `$`. See Table 5-2 for some sample acceptance filters. The format of the acceptance filter is *`(<regular expression>)`*.

Table 5-2. Example acceptance filters

Acceptance filter	alice.EXAMPLE.COM	hdfs@EXAMPLE.COM
`(.*\.EXAMPLE\.COM)`	Match	No match
`(.*@EXAMPLE\.COM)`	No match	Match
`(.*EXAMPLE\.COM)`	Match	Match
`(EXAMPLE\.COM)`	No match	No match

The substitution command

The substitution command is a sed-style substitution with a regular expression pattern and a replacement string. Matching groups can be included by surrounding a portion of the regular expression in parentheses, and referenced in the replacement string by number (e.g., `\1`). The group number is determined by the order of the opening parentheses in the regular expression. See Table 5-3 for some sample substitution commands. The format of the substitution command is `s/<pattern>/<replacement>/g`. The `g` at the end is optional, and if it is present then the

substitution will be global over the entire string. If the `g` is omitted, then only the first substring that matches the pattern will be substituted.

Table 5-3. Example substitution commands

Substitution Command	alice.EXAMPLE.COM	hdfs@EXAMPLE.COM
`s/(.*)\.EXAMPLE.COM/\1/`	`alice`	Not applicable
`s/.EXAMPLE.COM//`	`alice`	`hdfs`
`s/E/Q/`	`alice.QXAMPLE.COM`	`hdfs@QXAMPLE.COM`
`s/E/Q/g`	`alice.QXAMPLQ.COM`	`hdfs@QXAMPLQ.COM`

The complete format for a rule is `RULE:[`*`<number>`*`:`*`<string>`*`](`*`<regular expression>`*`)s/`*`<pattern>`*`/`*`<replacement>`*`/`. Multiple rules are separated by new lines and rules are evaluated in order. Once a principal fully matches a rule (i.e., the principal matches the number in the initial principal translation and the initial local name matches the acceptance filter), the username becomes the output of that rule and no other rules are evaluated. Due to this order constraint, it's common to list the `DEFAULT` rule last.

The most common use of the `auth_to_local` setting is to configure how to handle principals from other Kerberos realms. A common scenario is to have one or more trusted realms. For example, if your Hadoop realm is `HADOOP.EXAMPLE.COM` but your corporate realm is `CORP.EXAMPLE.COM`, then you'd add rules to translate principals in the corporate realm into local users. See Example 5-1 for a sample configuration that only accepts users in the `HADOOP.EXAMPLE.COM` and `CORP.EXAMPLE.COM` realms, and maps users to the first component for both realms.

Example 5-1. Example auth_to_local configuration for a trusted realm

```
<property>
  <name>hadoop.security.auth_to_local</name>
  <value>
    RULE:[1:$1@$0](.*@CORP.EXAMPLE.COM)s/@CORP.EXAMPLE.COM//
    RULE:[2:$1@$0](.*@CORP.EXAMPLE.COM)s/@CORP.EXAMPLE.COM//
    DEFAULT
  </value>
</property>
```

Hadoop User to Group Mapping

Hadoop exposes a configuration parameter called `hadoop.security.group.mapping` to control how users are mapped to groups. The default implementation uses either native calls or local shell commands to look up user-to-group mappings using the standard UNIX interfaces. This means that only the groups that are configured on the server where the mapping is called are visible to Hadoop. In practice, this is not a major concern because it is important for all of the servers in your Hadoop cluster to have a consistent view of the users and groups that will be accessing the cluster.

Note

In addition to knowing how the user-to-group mapping system works, it is important to know where the mapping takes place. As described in Chapter 6, it is important for user-to-group mappings to get resolved consistently and at the point where authorization decisions are made. For Hadoop, that means that the mappings occur in the NameNode, JobTracker (for MR1), and ResourceManager (for YARN/MR2) processes. This is a very important detail, as the default user-to-group mapping implementation determines group membership by using standard UNIX interfaces; for a group to exist from Hadoop's perspective, it must exist from the perspective of the servers running the NameNode, JobTracker, and ResourceManager.

The `hadoop.security.group.mapping` configuration parameter can be set to any Java class that implements the `org.apache.hadoop.security.GroupMappingServiceProvider` interface. In addition to the default described earlier, Hadoop ships with a number of useful implementations of this interface which are summarized here:

`JniBasedUnixGroupsMapping`

: A JNI-based implementation that invokes the `getpwnam_r()` and `getgrouplist()` libc functions to determine group membership.

`JniBasedUnixGroupsNetgroupMapping`

: An extension of the `JniBasedUnixGroupsMapping` that invokes the `setnetgrent()`, `getnetgrent()`, and `endnetgrent()` libc functions to determine members of netgroups. Only netgroups that are used in service-level authorization access control lists are included in the mappings.

`ShellBasedUnixGroupsMapping`

: A shell-based implementation that uses the `id -Gn` command.

`ShellBasedUnixGroupsNetgroupMapping`

: An extension of the `ShellBasedUnixGroupsMapping` that uses the `getent netgroup` shell command to determine members of netgroups. Only netgroups that are used in service-level authorization access control lists are included in the mappings.

`JniBasedUnixGroupsMappingWithFallback`

: A wrapper around the `JniBasedUnixGroupsMapping` class that falls back to the `ShellBasedUnixGroupsMapping` class if the native libraries cannot be loaded (this is the default implementation).

`JniBasedUnixGroupsNetgroupMappingWithFallback`

: A wrapper around the `JniBasedUnixGroupsNetgroupMapping` class that falls back to the `ShellBasedUnixGroupsNetgroupMapping` class if the native libraries cannot be loaded.

`LdapGroupsMapping`

Connects directly to an LDAP or Active Directory server to determine group membership.

Warning

Regardless of the group mapping configured, Hadoop will cache group mappings and only call the group mapping implementation when entries in the cache expire. By default, the group cache is configured to expire every 300 seconds (5 minutes). If you want updates to your underlying groups to appear in Hadoop more frequently, then set the `hadoop.security.groups.cache.secs` property in *core-site.xml* to the number of seconds you want entries cached. This should be set small enough for updates to be reflected quickly, but not so small as to require unnecessary calls to your LDAP server or other group provider.

Mapping users to groups using LDAP

Most deployments can use the default group mapping provider. However, for environments where groups are only available directly from an LDAP or Active Directory server and not on the cluster nodes, Hadoop provides the `LdapGroupsMapping` implementation. This method can be configured by setting several required parameters in the *core-site.xml* file on the NameNode, JobTracker, and/or ResourceManager:

`hadoop.security.group.mapping.ldap.url`

The URL of the LDAP server to use for resolving groups. Must start with `ldap://` or `ldaps://` (if SSL is enabled).

`hadoop.security.group.mapping.ldap.bind.user`

The distinguished name of the user to bind as when connecting to the LDAP server. This user needs read access to the directory and need not be an administrator.

`hadoop.security.group.mapping.ldap.bind.password`

The password of the bind user. It is a best practice to not use this setting, but to put the password in a separate file and to configure the `hadoop.security.group.mapping.ldap.bind.password.file` property to point to that path.

Warning

If you're configuring Hadoop to directly use LDAP, you lose the local groups for Hadoop service accounts such as `hdfs`. This can lead to a large number of log messages similar to:

```
No groups available for user hdfs
```

For this reason, it's generally better to use the JNI or shell-based mappings and to integrate with LDAP/Active Directory at the operating system level. The System Security Services Daemon (SSSD) provides strong integration with a number of identity and authentication systems and handles common support for caching and offline access.

Using the parameters described earlier, Example 5-2 demonstrates how to implement

`LdapGroups Mapping` in *coresite.xml*.

Example 5-2. Example LDAP mapping in core-site.xml

```
...
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://ad.example.com</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>Hadoop@ad.example.com</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>password</value>
</property>
...
```

In addition to the required parameters, there are several optional parameters that can be set to control how users and groups are mapped.

`hadoop.security.group.mapping.ldap.bind.password.file`

> The path to a file that contains the password of the bind user. This file should only be readable by the Unix users that run the daemons (typically `hdfs`, `mapred`, and `yarn`).

`hadoop.security.group.mapping.ldap.ssl`

> Set to `true` to enable the use of SSL when conntecting to the LDAP server. If this setting is enabled, the `hadoop.security.group.mapping.ldap.url` must start with `ldaps://`.

`hadoop.security.group.mapping.ldap.ssl.keystore`

> The path to a Java keystore that contains the client certificate required by the LDAP server when connecting with SSL enabled. The keystore must be in the Java keystore (JKS) format.

`hadoop.security.group.mapping.ldap.ssl.keystore.password`

> The password to the `hadoop.security.group.mapping.ldap.ssl.keystore` file. It is a best practice to not use this setting, but to put the password in a separate file and configure the `hadoop.security.group.mapping.ldap.ssl.keystore.password.file` property to point to that path.

`hadoop.security.group.mapping.ldap.ssl.keystore.password.file`

> The path to a file that contains the password to the `hadoop.security.group.mapping.ldap.ssl.keystore` file. This file should only be readable by Unix users that run the daemons (typically `hdfs`, `mapred`, and `yarn`).

`hadoop.security.group.mapping.ldap.base`

The search base for searching the LDAP directory. This is a distinguished name and will typically be configured as specifically as possible while still covering all users who access the cluster.

`hadoop.security.group.mapping.ldap.search.filter.user`

A filter to use when searching the directory for LDAP users. The default setting, `(&(objectClass=user)(sAMAccountName={0}))`, is usually appropriate for Active Directory installations. For other LDAP servers, this setting must be changed. For OpenLDAP and compatible servers, the recommended setting is `(&(objectClass=inetOrgPerson)(uid={0}))`.

`hadoop.security.group.mapping.ldap.search.filter.group`

A filter to use when searching the directory for LDAP groups. The default setting, `(objectClass=group)`, is usually appropriate for Active Directory installations.

`hadoop.security.group.mapping.ldap.search.attr.member`

The attribute of the group object that identifies the users that are members of the group.

`hadoop.security.group.mapping.ldap.search.attr.group.name`

The attribute of the group object that identifies the group's name.

`hadoop.security.group.mapping.ldap.directory.search.timeout`

The maximum amount of time in milliseconds to wait for search results from the directory.

Provisioning of Hadoop Users

One of the most difficult requirements of Hadoop security to understand is that all users of a cluster must be provisioned on all servers in the cluster. This means they can either exist in the local */etc/passwd* password file or, more commonly, can be provisioned by having the servers access a network-based directory service, such as OpenLDAP or Active Directory. In order to understand this requirement, it's important to remember that Hadoop is effectively a service that lets you submit and execute arbitrary code across a cluster of machines. This means that if you don't trust your users, you need to restrict their access to any and all services running on those servers, including standard Linux services such as the local filesystem. Currently, the best way to enforce those restrictions is to execute individual tasks (the processes that make up a job) on the cluster using the username and UID of the user who submitted the job. In order to satisfy that requirement, it is necessary that every server in the cluster uses a consistent user database.

Note

While it is necessary for all users of the cluster to be provisioned on all of the servers in the cluster, it is not necessary to enable local or remote shell access to all of those users. A best practice is to

provision the users with a default shell of */sbin/nologin* and to disable SSH access using the `AllowUsers`, `DenyUsers`, `AllowGroups`, and `DenyGroups` settings in the */etc/ssh/sshd_config* file.

Authentication

Early versions of Hadoop and the related ecosystem projects did not support strong authentication. Hadoop is a complex distributed system, but fortunately most components in the ecosystem have standardized on a relatively small number of authentication options, depending on the service and protocol. In particular, Kerberos is used across most components of the ecosystem because Hadoop standardized on it early on in its development of security features. A summary of the authentication methods by service and protocol is shown in Table 5-4. In this section, we focus on authentication for HDFS, MapReduce, YARN, HBase, Accumulo, and ZooKeeper. Authentication for Hive, Impala, Hue, Oozie, and Solr are deferred to Chapters 11 and 12 because those are commonly accessed directly by clients.

Table 5-4. Hadoop ecosystem authentication methods

Service	Protocol	Methods
HDFS	RPC	Kerberos, delegation token
HDFS	Web UI	SPNEGO (Kerberos), pluggable
HDFS	REST (WebHDFS)	SPNEGO (Kerberos), delegation token
HDFS	REST (HttpFS)	SPNEGO (Kerberos), delegation token
MapReduce	RPC	Kerberos, delegation token
MapReduce	Web UI	SPNEGO (Kerberos), pluggable
YARN	RPC	Kerberos, delegation token
YARN	Web UI	SPNEGO (Kerberos), pluggable
Hive Server 2	Thrift	Kerberos, LDAP (username/password)
Hive Metastore	Thrift	Kerberos, LDAP (username/password)
Impala	Thrift	Kerberos, LDAP (username/password)
HBase	RPC	Kerberos, delegation token
HBase	Thrift Proxy	None

Service	Protocol	Methods
HBase	REST Proxy	SPNEGO (Kerberos)
Accumulo	RPC	Username/password, pluggable
Accumulo	Thrift Proxy	Username/password, pluggable
Solr	HTTP	Based on HTTP container
Oozie	REST	SPNEGO (Kerberos, delegation token)
Hue	Web UI	Username/password (database, PAM, LDAP), SAML, OAuth, SPNEGO (Kerberos), remote user (HTTP proxy)
ZooKeeper	RPC	Digest (username/password), IP, SASL (Kerberos), pluggable

Kerberos

Out of the box, Hadoop supports two authentication mechanisms: `simple` and `kerberos`. The `simple` mechanism, which is the default, uses the effective UID of the client process to determine the username, which it passes to Hadoop with no additional credentials. In this mode, Hadoop servers fully trust their clients. This default is sufficient for deployments where any user that can gain access to the cluster is fully trusted with access to all data and administrative functions on said cluster. For proof-of-concept systems or lab environments, it is often permissible to run in this mode and rely on firewalls and limiting the set of users that can log on to any system with client-access to the cluster. However, this is rarely acceptable for a production system or any system with multiple tenants. Simple authentication is similarly supported by HBase as its default mechanism.

HDFS, MapReduce, YARN, HBase, Oozie, and ZooKeeper all support Kerberos as an authentication mechanism for clients, though the implementations differ somewhat by service and interface. For RPC-based protocols, the *Simple Authentication and Security Layer* (SASL) framework is used to add authentication to the underlying protocol. In theory, any SASL mechanism could be supported, but in practice, the only mechanisms that are supported are GSSAPI (specifically Kerberos V5) and DIGEST-MD5 (see “Tokens” for details on DIGEST-MD5). Oozie does not have an RPC protocol and instead provides clients a REST interface. Oozie uses the *Simple and Protected GSSAPI Negotiation Mechanism* (SPNEGO), a protocol first implemented by Microsoft in Internet Explorer 5.0.1 and IIS 5.0 to do Kerberos authentication over HTTP. SPNEGO is also supported by the web interfaces for HDFS, MapReduce, YARN, Oozie, and Hue as well as the REST interfaces for HDFS (both WebHDFS and HttpFS) and HBase. For both SASL and SPNEGO, the authentication follows the standard Kerberos protocol and only the mechanism for presenting the service ticket changes.

Let’s see how Alice would authenticate against the HDFS NameNode using Kerberos:

1. Alice requests a service ticket from the TGS at `kdc.example.com` for the HDFS service identified by `hdfs/namenode.example.com@EXAMPLE.COM`, presenting her TGT

with the request.

2. The TGS validates the TGT and provides Alice a service ticket, encrypted with the `hdfs/namenode.example.com@EXAMPLE.COM` principal's key.
3. Alice presents the service ticket to the NameNode (over SASL), which can decrypt it using the `hdfs/namenode.example.com@EXAMPLE.COM` key and validate the ticket.

Username and Password Authentication

ZooKeeper supports authentication by username and password. Rather than using a database of usernames and passwords, ZooKeeper defers password checking to the authorization step (see "ZooKeeper ACLs"). When an ACL is attached to a ZNode, it includes the authentication scheme and a scheme-specific ID. The scheme-specific ID is verified using the authentication provider for the given scheme. Username and password authentication is implemented by the digest authentication provider, which generates a SHA-1 digest of the username and password. Because verification is deferred to the authorization check, the authentication step always succeeds. Users add their authentication details by calling the `addAuthInfo(String scheme, byte[] authData)` method with `"digest"` as the scheme and `"<username>:<password>.getBytes()"` as the authData where *<username>* and *<password>* are replaced with their appropriate values.

Accumulo also supports username and password–based authentication. Unlike ZooKeeper, Accumulo uses the more common approach of storing usernames and passwords and having an explicit login step that verifies if the password is valid. Accumulo's authentication system is pluggable through different implementations of the `AuthenticationToken` interface. The most common implementation is the `PasswordToken` class, which can be initialized from a `CharSequence` or a Java `Properties` file. Sample code for connecting to Accumulo using a username and password is shown in Example 5-3.

Example 5-3. Connecting to Accumulo with a username and password

```
// Create a handle to an Accumulo Instance
Instance instance = new ZooKeeperInstance("instance",
    "zk1.example.com,zk2.example.com,zk3.example.com");

// Create a token with the password
AuthenticationToken token = new PasswordToken("secret");

// Create the Connector; if the password is invalid, an
// AccumuloSecurityException will be thrown
Connector connector = instance.getConnector("alice", token);
```

Tokens

In any distributed system, it is necessary for all actions taken on behalf of a user to validate that user's identity. It is not sufficient to merely authenticate with the master of a service; authentication must happen at every interaction. Take the example of running a MapReduce job. Authentication

happens between the client and the NameNode in order to expand any wildcards in the command-line parameters, as well as between the client and the JobTracker, in order to submit the job.

The JobTracker then breaks the job into tasks that are subsequently launched by each TaskTracker in the cluster. Each task has to communicate with the NameNode in order to open the files that make up its input split. In order for the NameNode to enforce filesystem permissions, each task has to authenticate against the NameNode. If Kerberos was the only authentication mechanism, a user's TGT would have to be distributed to each task. The downside to that approach is it allows the tasks to authenticate against *any* Kerberos protected service, which is not desirable. Hadoop solves this problem by issuing authentication tokens that can be distributed to each task but are limited to a specific service.

Delegation tokens

Hadoop has multiple types of tokens that are used to allow subsequent authenticated access without a TGT or Kerberos service ticket. After authenticating against the NameNode using Kerberos, a client can obtain a *delegation token*. The delegation token is a shared secret between the client and the NameNode and can be used for RPC authentication using the DIGEST-MD5 mechanism.

Figure 5-1 shows two interactions between a client and the NameNode. First, the client requests a delegation token using the `getDelegationToken()` RPC call using a Kerberos service ticket for authentication (1). The NameNode replies with the delegation token (2). The client invokes the `getListing()` RPC call to request a directory listing, but this time it uses the delegation token for authentication. After validating the token, the NameNode responds with the requested `DirectoryListing` (4).

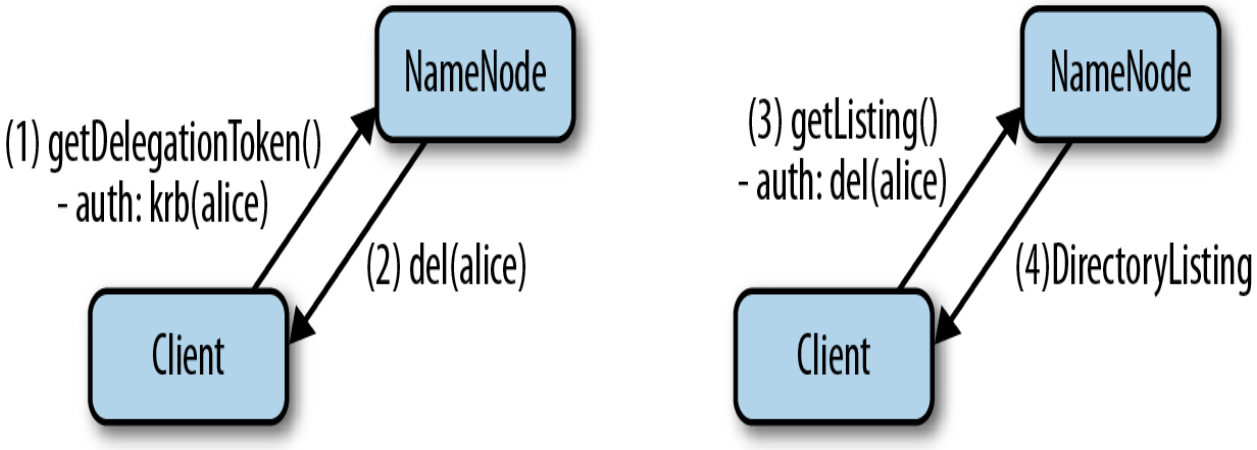

Figure 5-1. Retrieving and using a delegation token

The token has both an expiration date and a max issue date. The token will expire after the expiration date, but can be renewed even if expired up until the max issue date. A delegation token can be requested by the client after any initial Kerberos authentication to the NameNode. The token also has a designated *token renewer*. The token renewer authenticates using its Kerberos credentials when renewing a token on behalf of a user. The most common use of delegation tokens is for MapReduce jobs, in which case the client designates the JobTracker as the renewer. The delegation tokens are keyed by the NameNode's URL and stored in the JobTracker's system directory so they can be passed to the tasks. This allows the tasks to access HDFS without putting a user's TGT at

risk.

Block access tokens

File permission checks are performed by the NameNode, not the DataNode. By default, any client can access any block given only its block ID. To solve this, Hadoop introduced the notion of *block access tokens*. Block access tokens are generated by the NameNode and given to a client after the client is authenticated and the NameNode has performed the necessary authorization check for access to a file/block. The token includes the ID of the client, the block ID, and the permitted access mode (READ, WRITE, COPY, REPLACE) and is signed using a shared secret between the NameNode and DataNode. The shared secret is never shared with the client and when a block access token expires, the client has to request a new one from the NameNode.

Figure 5-2 shows how a client uses a block access token to read data. The client will first use Kerberos credentials to request the location of the block from the NameNode using the `getBlockLocations()` RPC call (1). The NameNode will respond with a `LocatedBlock` object which includes, among other details, a block access token for the requested block (2). The client will then request data from the DataNode using the `readBlock()` method in the data transfer protocol using the block access token for authentication (3). Finally, the DataNode will respond with the requested data (4).

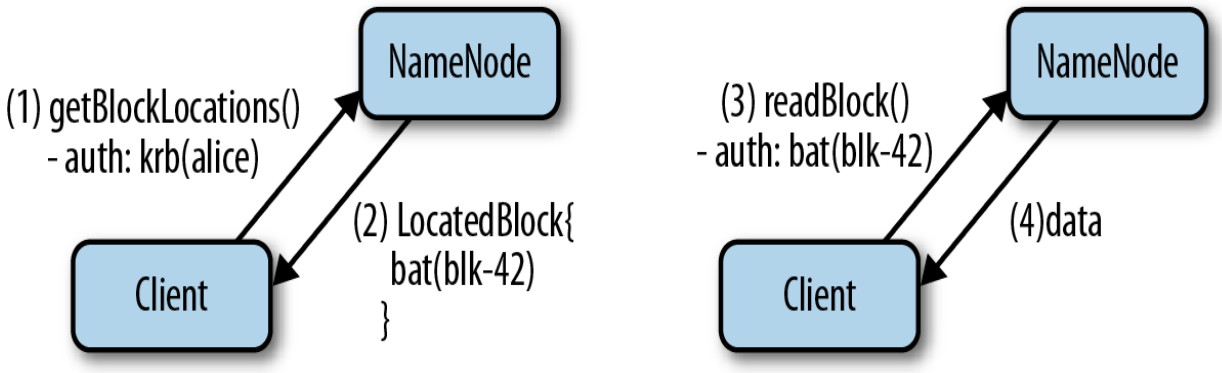

Figure 5-2. Accessing a block using a block access token

Job tokens

When submitting a MapReduce job, the JobTracker will create a secret key called a *job token* that is used by the tasks of the job to authenticate against the TaskTrackers. The JobTracker places the token in the JobTracker's system directory on HDFS and distributes it to the TaskTrackers over RPC. The TaskTrackers will place the token in the job directory on the local disk, which is only accessible to the job's user. The job token is used to authenticate RPC communication between the tasks and the TaskTrackers as well as to generate a hash, which ensures that intermediate outputs sent over HTTP in the shuffle phase are only accessible to the tasks of the job. Furthermore, the TaskTracker returning shuffle data calculates a hash that each task can use to verify that it is talking to a true TaskTracker and not an impostor.

Figure 5-3 is a time sequence diagram showing which authentication methods are used during job setup. First, the client requests the creation of a new job using Kerberos for authentication (1). The JobTracker responds with a job ID that's used to uniquely identify the job (2). The client then

requests a delegation token from the NameNode with the JobTracker as the renewer (3). The NameNode responds with the delegation token (4). Delegation tokens will only be issued if the client authenticates with Kerberos. Finally, the client uses Kerberos to authenticate with the JobTracker sending the delegation token and other required job details.

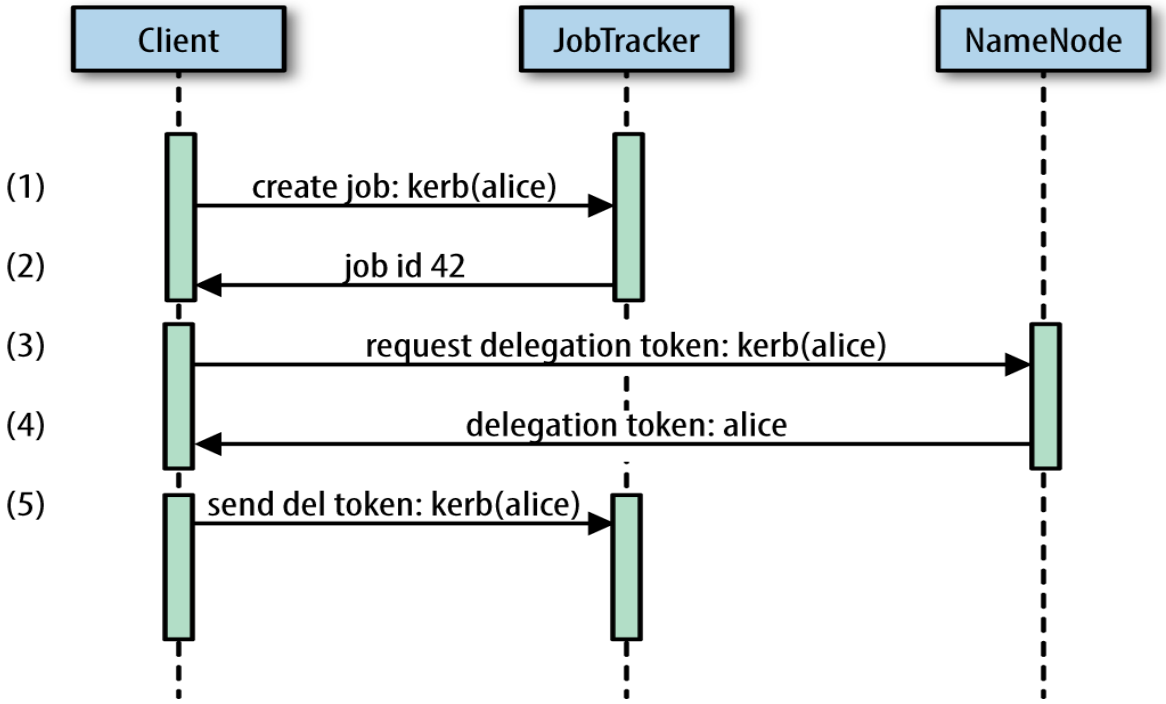

Figure 5-3. Job setup

Things get more interesting once the job starts executing, as Figure 5-4 shows.

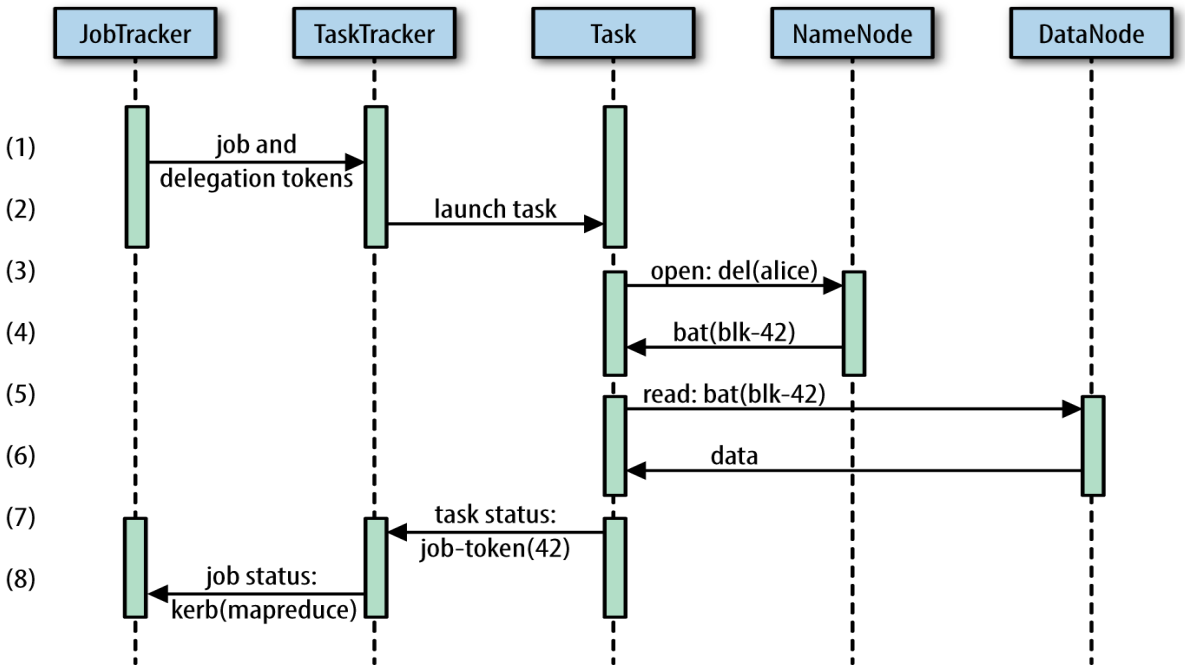

Figure 5-4. Job execution

The JobTracker will generate a job token for the job and then package up and send the job token, delegation token, and other required information to the TaskTracker (1). The JobTracker uses Kerberos authentication when talking to the TaskTracker. The TaskTracker will then place the tokens into a directory only accessible by the user who submitted the job, and will launch the tasks

(2). The Task uses the delegation token to open a file and request the block location for its input split (3). The NameNode will respond with the block location including a block access token for the given block (4). The Task then uses the block access token to read data from the DataNode (5) and the DataNode responds with the data (6). As the job progresses, the Task will report task status to the TaskTracker using the job token to authenticate (7). The TaskTracker will then report status back to the JobTracker using Kerberos authentication so that overall job status can be aggregated (8).

Impersonation

There are many services in the Hadoop ecosystem that perform actions on behalf of an end user. In order to maintain security, these services must authenticate their clients and be trusted to *impersonate* other users. Oozie, Hive (in HiveServer2), and Hue all support impersonating end users when accessing HDFS, MapReduce, YARN, or HBase. Secure impersonation works consistently across these services and is supported by designating which users are trusted to perform impersonation. When a trusted user needs to act on behalf of another user, she must authenticate as herself and supply the username of the user she is acting on behalf of. Trusted users can be limited to only impersonate specific groups of users, and only when accessing Hadoop from certain hosts to further constrain their privileges.

Impersonation is also sometimes called *proxying*. The user that can perform the impersonation (i.e., the user that can proxy other users) is called the *proxy*. The configuration parameters for enabling impersonation are `hadoop.proxyuser.<proxy>.hosts` and `hadoop.proxyuser.<proxy>.groups`, where `<proxy>` is the username of the user doing the impersonating. The values are comma-separated lists of hosts and groups, respectively, or `*` to mean all hosts/groups. If you want both Hue and Oozie to have proxy capabilities, but you want to limit the users that Oozie can proxy to members of the `oozie-users` group, then you'd use a configuration similar to that shown in Example 5-4.

Example 5-4. Example configuration for impersonation

```
<!-- Configure Hue impersonation from hue.example.com -->
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>hue.example.com</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
  <value>*</value>
</property>

<!--
     Configure Oozie impersonation from oozie01.example.com and
     oozie02.example.com for users in oozie-users
  -->
<property>
  <name>hadoop.proxyuser.oozie.hosts</name>
  <value>oozie01.example.com,oozie02.example.com</value>
</property>
<property>
  <name>hadoop.proxyuser.oozie.groups</name>
  <value>oozie-users</value>
</property>
```

Configuration

For production deployments, Hadoop supports the `kerberos` mechanism for authentication. When configured for Kerberos authentication, all users and daemons must provide valid credentials in order to access RPC interfaces. This means that you must create a Kerberos service principal for every server/daemon pair in the cluster. You'll recall that in Chapter 4 we described the concept of a service principal name (SPN), which consists of three parts: a service name, a hostname, and a realm. In Hadoop, each daemon that's part of a certain service uses that service's name (`hdfs` for HDFS, `mapred` for MapReduce, and `yarn` for YARN). Additionally, if you want to enable Kerberos authentication for the various web interfaces, then you also need to provision principals with the `HTTP` service name.

Let's see what needs to be done to configure a sample cluster with Kerberos authentication. For our example, we'll assume we have a cluster with hosts and services, as shown in Table 5-5.

Note

The service layout in Table 5-5 is meant to serve as an example, but it isn't the best way to provision a cluster. For starters, we're showing our example with both YARN and MR1 services configured. This is only meant to show the full range of configuration settings needed for both services. In a real deployment, you would only deploy one or the other. Similarly, you would not need to deploy a SecondaryNameNode if you're running two NameNodes with HA as we're doing here. Again, this is just to make our example configuration comprehensive.

Table 5-5. Service layout

Hostname	Daemon
nn1.example.com	NameNode
	JournalNode
nn2.example.com	NameNode
	JournalNode
snn.example.com	SecondaryNameNode
	JournalNode
rm.example.com	ResourceManager
jt.example.com	JobTracker
	JobHistoryServer
dn1.example.com	DataNode
	TaskTracker

Hostname	Daemon
	NodeManager
dn2.example.com	DataNode
	TaskTracker
	NodeManager
dn3.example.com	DataNode
	TaskTracker
	NodeManager

The first step is to create all of the required SPNs in your Kerberos KDC and to export a keytab file for each daemon on each server. The list of SPNs required for each host/role is shown in Table 5-6 along with a recommended name for their respective keytab files. You need to create different keytab files per server. We recommend using consistent names per daemon in order to use the same configuration files on all hosts even though keytab files with the same name on different hosts will contain different keys.

Table 5-6. Required Kerberos principals

Hostname	Daemon	Keytab file	SPN
nn1.example.com	NameNode/JournalNode	hdfs.keytab	hdfs/nn1.example.com@EXAMPLE.COM
			HTTP/nn1.example.com@EXAMPLE.COM
nn2.example.com	NameNode/JournalNode	hdfs.keytab	hdfs/nn2.example.com@EXAMPLE.COM
			HTTP/nn2.example.com@EXAMPLE.COM
snn.example.com	SecondaryNameNode/JournalNode	hdfs.keytab	hdfs/snn.example.com@EXAMPLE.COM
			HTTP/snn.example.com@EXAMPLE.COM
rm.example.com	ResourceManager	yarn.keytab	yarn/rm.example.com@EXAMPLE.COM
jt.example.com	JobTracker	mapred.keytab	mapred/jt.example.com@EXAMPLE.

Hostname	Daemon	Keytab file	SPN
			COM
			HTTP/jt.example.com@EXAMPLE.COM
	JobHistoryServer	mapred.keytab	mapred/jt.example.com@EXAMPLE.COM
dn1.example.com	DataNode	hdfs.keytab	hdfs/dn1.example.com@EXAMPLE.COM
			HTTP/dn1.example.com@EXAMPLE.COM
	TaskTracker	mapred.keytab	mapred/dn1.example.com@EXAMPLE.COM
			HTTP/dn1.example.com@EXAMPLE.COM
	NodeManager	yarn.keytab	yarn/dn1.example.com@EXAMPLE.COM
			HTTP/dn1.example.com@EXAMPLE.COM
dn2.example.com	DataNode	hdfs.keytab	hdfs/dn2.example.com@EXAMPLE.COM
			HTTP/dn2.example.com@EXAMPLE.COM
	TaskTracker	mapred.keytab	mapred/dn2.example.com@EXAMPLE.COM
			HTTP/dn2.example.com@EXAMPLE.COM
	NodeManager	yarn.keytab	yarn/dn2.example.com@EXAMPLE.COM
			HTTP/dn2.example.com@EXAMPLE.COM
dn3.example.com	DataNode	hdfs.keytab	hdfs/dn3.example.com@EXAMPLE.COM

Hostname	Daemon	Keytab file	SPN
			HTTP/dn3.example.com@EXAMPLE.COM
	TaskTracker	mapred.keytab	mapred/dn3.example.com@EXAMPLE.COM
			HTTP/dn3.example.com@EXAMPLE.COM
	NodeManager	yarn.keytab	yarn/dn3.example.com@EXAMPLE.COM
			HTTP/dn3.example.com@EXAMPLE.COM

Warning

Take care when exporting keytab files, as the default is to randomize the Kerberos key each time a principal is exported. You can export each principal once and then use the `ktutil` utility to combine the necessary keys into the keytab file for each daemon.

We recommend placing the appropriate keytab files into your *$HADOOP_CONF_DIR* directory (typically */etc/hadoop/conf*).

Full Example Configuration Files

A complete set of example configuration files are available in the example repository on GitHub that accompanies this book.

After you've created all of the required SPNs and distributed the keytab files, you need to configure Hadoop to use Kerberos for authentication. Start by setting `hadoop.security.authentication` to `kerberos` in the *core-site.xml* file, as shown in Example 5-5.

Example 5-5. Configuring the authentication type to Kerberos

```
<property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
</property>
```

HDFS

Next, we need to configure each daemon with its Kerberos principals and keytab files. For the NameNode, we also have to enable block access tokens by setting `dfs.block.access.token.enable` to `true`. The NameNode's configuration should be set in the *hdfs-site.xml* file, as shown in Example 5-6.

Example 5-6. Configuring the NameNode for Kerberos

```
<property>
  <name>dfs.block.access.token.enable</name>
  <value>true</value>
</property>
<property>
  <name>dfs.namenode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.namenode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

If you are not enabling high availability for HDFS, then you would next configure the SecondaryNameNode in the *hdfs-site.xml* file, as shown in Example 5-7.

Example 5-7. Configuring the SecondaryNameNode for Kerberos

```
<property>
  <name>dfs.secondary.namenode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

If you are enabling high availability for HDFS, then you need to configure the JournalNodes with the following settings in the *hdfs-site.xml* file, as shown in Example 5-8.

Example 5-8. Configuring the JournalNode for Kerberos

```
<property>
  <name>dfs.journalnode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.journalnode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.journalnode.kerberos.internal.spnego.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

Next, we'll configure the DataNode's with the following settings in the *hdfs-site.xml* file. In addition to configuring the keytab and principal name, you must configure the DataNode to use a privileged port for its RPC and HTTP servers. These ports need to be privileged because the DataNode does not use Hadoop's RPC framework for the data transfer protocol. By using privileged ports, the DataNode is authenticating that it was started by *root* using `jsvc`, as shown in

Example 5-9.

Example 5-9. Configuring the DataNode for Kerberos

```
<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:1004</value>
</property>
<property>
  <name>dfs.datanode.http.address</name>
  <value>0.0.0.0:1006</value>
</property>
<property>
  <name>dfs.datanode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.datanode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
```

WebHDFS is a REST-based protocol for accessing data in HDFS. WebHDFS scales by serving data over HTTP from the DataNode that stores the blocks being read. In order to secure access to WebHDFS, you need to set the following parameters in the *hdfs-site.xml* file of the NameNodes and DataNodes, as shown in Example 5-10.

Example 5-10. Configuring WebHDFS for Kerberos

```
<property>
  <name>dfs.web.authentication.kerberos.keytab</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.web.authentication.kerberos.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

The configuration of HDFS is now complete!

Chapter 6. Authorization

In "Authentication", we saw how the various Hadoop ecosystem projects support strong authentication to ensure that users are who they claim to be. However, authentication is only part of the overall security story—you also need a way to model which actions or data an authenticated user can access. The protection of resources in this manner is called *authorization* and is probably one of the most complex topics related to Hadoop security. Each service is relatively unique in the services it provides, and thus the authorization model it supports. The sections in this chapter are divided into subsections based on how each service implements authorization.

We start by looking at HDFS and its support for POSIX-style file permissions, as well as its support for service-level authorization to restrict user access to specific HDFS functions. Next, we turn our attention to MapReduce and YARN, which support a similar style of service-level authorization as well as a queue-based model controlling access to system resources. In the case of MapReduce and YARN, authorization is useful for both security and resource management/multitenancy (for more information on resource management, we recommend *Hadoop Operations* by Eric Sammer

[O'Reilly]). Finally, we cover the authorization features of the popular BigTable clones, Apache HBase and Apache Accumulo, including a discussion of the pros and cons of role-based and attribute-based security as well as a discussion of cell-level versus column-level security.

HDFS Authorization

Every attempt to access a file or directory in HDFS must first pass an authorization check. HDFS adopts the authorization scheme common to POSIX-compatible filesystems. Permissions are managed by three distinct classes of user: *owner*, *group*, and *others*. Each file or directory is owned by a specific user and that user makes up the object's *owner* class. Objects are also assigned a group and all of the members of that group make up the object's *group* class. All users that are not the owner and do not belong to the group assigned to the object make up the *others* class. Read, write, and execute permissions can be granted to each class independently.

These permissions are represented by a single octal integer that is calculated by summing the permission values (4 for read, 2 for write, and 1 for execute). For example, to represent that a class has read and execute permissions for a directory, an octal value of 5 (4+1) would be assigned. In HDFS, it is not meaningful, nor is it invalid, to assign the execute permission to a file. For directories, the execute bit gives permission to access a file's contents and metadata information if the name of the file is known. In order to list the names of files in a directory, you need read permission for the directory.

Regardless of the permissions on a file or a directory, the user that the NameNode runs as (typically *hdfs*) and any member of the group defined in `dfs.permissions.superusergroup` (defaults to *supergroup*), can read, write, or delete any file and directory. As far as HDFS is concerned, they are the equivalent of *root* on a Linux system.

The permissions assigned to the owner, group, and others can be represented by concatenating the three octal values in that order. For example, take a file for which the owner has read and write permissions and all other users have only read permission. This file's permissions would be represented as 644; 6 is assigned to the owner because she has both read and write (4+2), and 4 is assigned to the group and other classes because they only have read permissions. For a file for which all permissions have been granted to all users, the permissions would be 777.

In addition to the standard permissions, HDFS supports three additional special permissions: *setuid*, *setgid*, and *sticky*. These permissions are also represented as an octal value with 4 for setuid, 2 for setgid, and 1 for sticky. These permissions are optional and are included to the *left* of the regular permission bits if they are specified. Because files in HDFS can't be executed, setuid has no effect. Setgid similarly has no effect on files, but for directories it forces the group of newly created immediate child files and directories to that of the parent. This is the default behavior in HDFS, so it is not necessary to enable setgid on directories. The final permission is often called the sticky bit and it means that files in a directory can only be deleted by the owner of that file. Without the sticky bit set, a file can be deleted by anyone that has write access to the directory. In HDFS, the owner of a directory and the HDFS superuser can also delete files regardless of whether the sticky bit is set. The sticky bit is useful for directories, such as */tmp*, where you want all users to have write access to the directory but only the owner of the data should be able to delete data.

HDFS Extended ACLs

Using the basic POSIX permissions of owner, group, and world to allow access to a given file or directory is not always easy. What happens if two or more different groups of users need access to the same HDFS directory? With basic POSIX permissions, an administrator is left with two options: (1) make the directory world-accessible, or (2) create a group that encompasses all of the users that need access to the directory and assign group permissions to it. This is not ideal because option #1 risks making the data available to more than the intended users, and option #2 can quickly become a headache from a group management perspective. This problem becomes further compounded when one group of users requires read access and another group of users requires both read and write access.

With the release of Hadoop 2.4, HDFS is now equipped with extended ACLs. These ACLs work very much the same way as extended ACLs in a Unix environment. This allows files and directories in HDFS to have more permissions than the basic POSIX permissions.

To use HDFS extended ACLs, they must first be enabled on the NameNode. To do this, set the configuration property `dfs.namenode.acls.enabled` to `true` in *hdfs-site.xml*. Example 6-1 shows how HDFS extended ACLs are used.

Example 6-1. HDFS extended ACLs example

```
[alice@hadoop01 ~]$ hdfs dfs -ls /data
Found 1 items
drwxr-xr-x   - alice analysts          0 2014-10-25 19:03 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
group::r-x
other::r-x
[alice@hadoop01 ~]$ hdfs dfs -setfacl -m user:bob:r-x /data/alice
[alice@hadoop01 ~]$ hdfs dfs -setfacl -m group:developers:rwx /data/alice
[alice@hadoop01 ~]$ hdfs dfs -ls /data
Found 1 items
drwxr-xr-x+   - alice analysts          0 2014-10-25 19:03 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
user:bob:r-x
group::r-x
group:developers:rwx
mask::rwx
other::r-x
[alice@hadoop01 ~]$ hdfs dfs -chmod 750 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
group::r-x
group:developers:rwx    #effective:r-x
mask::r-x
other::---
[alice@hadoop01 ~]$ hdfs dfs -setfacl -b /data/alice
```

```
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
group::r-x
other::---
```

There are a few points worth highlighting. First, by default, files and directories do not have any ACLs. After adding an ACL entry to an object, the HDFS listing now appends a `+` to the permissions listing, such as in `drwxr-xr-x+`. Also, after adding an ACL entry, a new property is listed in the ACL called `mask`. The mask defines what the most restrictive permissions will be. For example, if user bob has `rwx` permissions, but the mask is `r-x`, bob's effective permissions are `r-x` and are noted as such in the output of `getfacl`, as shown in the example.

Another important part about the mask is that it gets adjusted to the least restrictive permissions that are set on an ACL. For example, if a mask is currently set to be `r-x` and a new ACL entry is added for a group to grant `rwx` permissions, the mask is adjusted to `rwx`.

Warning

Setting standard POSIX permissions on a file or directory that contains an extended ACL might immediately impact all entries because `hdfs dfs -chmod` will effectively set the mask, regardless of what ACL entries are present. For example, setting 700 permissions on a file or directory yields effective permissions of *no access* to all ACL entries defined, except the owner!

The last part of the example demonstrates how to completely remove all ACL entries for a directory, leaving just the basic POSIX permissions in place. One final point about extended ACLs is that they are limited to 32 entries per object (i.e., file or directory). That being said, four of the entries are taken up by `user`, `group`, `other`, and `mask`, so the net is 28 entries, which can be added before the NameNode throws an error: `setfacl: Invalid ACL: ACL has 33 entries, which exceeds maximum of 32`.

Another useful feature of extended ACLs is the usage of a *default ACL*. A default ACL applies only to a directory, and the effect is that all subdirectories and files created in that directory inherit the default ACL of the parent directory. For example, if a directory has a default ACL entry of `default:group:analysts:rwx`, then all files created in the directory will get a `group:analysts:rwx` entry, and subdirectories will get *both* the default ACL and the access ACL copied over. To set a default ACL, simply prepend `default:` to the user or group entry in the `setfacl` command. Remember that default ACLs *do not* themselves grant authorization. They simply define the inheritance behavior of newly created subdirectories and files.

MapReduce and YARN Authorization

Neither MapReduce nor YARN control access to data, but both provide access to cluster resources such as CPU, memory, disk I/O, and network I/O. Because these resources are finite, it is common for administrators to allocate resources to specific users or groups, especially in multitenant environments. The service-level authorizations described in the previous section control access to

specific protocols, such as who can and cannot submit a job to the cluster, but they are not granular enough to control access to cluster resources. Both MapReduce (MR1) and YARN support job queues as a way of putting limits on how jobs are allocated resources. In order to securely control those resources, Hadoop supports access control lists (ACLs) on the job queues. These ACLs control which users can submit to certain queues as well as which users can administer a queue. MapReduce defines different classes of users, which affect the way that ACLs are interpreted:

MapReduce/YARN cluster owner

> The user that starts the JobTracker process (MR1) or the ResourceManager process (YARN) is defined as the cluster owner. That user has permissions to submit jobs to any queue and can administer any queue or job. In most cases, the cluster owner is *mapred* for MapReduce (MR1) and *yarn* for YARN. Because it is dangerous to run jobs as the cluster owner, the LinuxTaskController defaults to blacklisting the *mapred* and *yarn* user accounts so they can't submit jobs.

MapReduce administrator

> There is a setting to create global MapReduce administrators that have the same privileges as the cluster owner. The advantage to defining specific users or groups as administrators is that you can still audit the individual actions of each administrator. This also lets you avoid having to distribute the password to a shared account, thus increasing the likelihood that the password could be compromised.

Job owner

> The owner of a job is the user that submitted it. Job owners can always administer their own jobs but can only submit jobs to queues for which they've been granted the submit permission.

Queue administrator

> Users or groups can be given administrative permissions over all of the jobs in a queue. Queue administrators can also submit jobs to the queues they administer.

MapReduce (MR1)

For MR1, ACLs are administered globally and apply to any job scheduler that supports ACLs. Both the CapacityScheduler and FairScheduler support ACLs; the FIFO (default) scheduler does not. Before configuring per-queue ACLs, you must enable MapReduce ACLs, configure the MapReduce administrators, and define the queue names in *mapred-site.xml*:

`mapred.acls.enabled`

> When set to `true`, ACLs will be checked when submitting or administering jobs. ACLs are also checked for authorizing the viewing and modification of jobs in the JobTracker interface.

`mapreduce.cluster.administrators`

> Configure administrators for the MapReduce cluster. Cluster administrators can always

administer any job or queue regardless of the configuration of job- or queue-specific ACLs. The format for this setting is a comma-delimited list of users and a comma-delimited list of groups that can access that protocol. The two lists are separated by a space. A leading space implies an empty list of users and a trailing space implies an empty list of groups. A special value of `*` can be used to signify that all users are granted access to that protocol (this is the default setting). See Table 6-1 for examples.

`mapred.queue.names`

A comma-delimited list of queue names. In order to configure ACLs for a queue, that queue must be listed in this property. MapReduce always supports at least one queue named *default*, so this parameter should always include *default* among the list of defined queues.

The configuration for per-queue ACLs is stored in *mapred-queue-acls.xml*. There are two types of ACLs that can be configured for each queue, a submit ACL and an administer ACL:

`mapred.queue.<queue_name>.acl-submit-job`

The access control list for users that can submit jobs to the queue named *queue_name*. The format for the submit job ACL is a comma-delimited list of users and a comma-delimited list of groups that are allowed to submit jobs to this queue, identical in format to hose described in "Service-Level Authorization" and depicted in Table 6-1. A special value of `*` can be used to signify that all users are granted access to that protocol (this is the default setting). Regardless of the value of this setting, the cluster owner and MapReduce administrators can submit jobs.

`mapred.queue.<queue_name>.acl-administer-jobs`

The access control list for users that are allowed to view job details, kill jobs, or modify a job's priority for all jobs in the queue named *queue_name*. The format for the administer-jobs ACL is a comma-delimited list of users and a comma delimited list of groups that are allowed to administer jobs in this queue, identical in format to those described in "Service-Level Authorization" and depicted in Table 6-1. A special value of `*` can be used to signify that all users are granted access to that protocol (this is the default setting). Regardless of the value of this setting, the cluster owner and MapReduce administrators can administer all the jobs in all the queues. The job owner can also administer jobs.

In addition to the per-queue ACLs, there are two types of ACLs that can be configured on a per-job basis. Defaults for these settings can be placed in the *mapred-site.xml* file used by clients and can be overridden by individual jobs:

`mapreduce.job.acl-view-job`

The access control list for users that are allowed to view job details. The format for the view-job ACL is a comma-delimited list of users and a comma-delimited list of groups that are allowed to view job details, identical in format to those described in "Service-Level Authorization" and depicted in Table 6-1. A special value of `*` can be used to signify that all users are granted access to that protocol (this is the default setting). Regardless of the value of this setting, the job owner, the cluster owner, MapReduce administrators, and administrators of the queue to which the job was submitted always have access to view a job. This ACL controls access to job-level counters, task-level counters, a task's diagnostic information, task

logs displayed on the TaskTracker web UI, and the *job.xml* shown by the JobTracker's web UI.

`mapreduce.job.acl-modify-job`

The access control list for users that are allowed to kill a job, kill a task, fail a task, and set the priority of a job. The format for the modify-job ACL is a comma-delimited list of users and a comma-delimited list of groups that are allowed to modify the job, identical in format to those described in "Service-Level Authorization" and depicted in Table 6-1. A special value of `*` can be used to signify that all users are granted access to that protocol (this is the default setting). Regardless of the value of this setting, the job owner, the cluster owner, MapReduce administrators, and administrators of the queue to which the job was submitted always have access to modify a job.

For deployments where you want a default deny policy for access to job details, a sensible default value for both settings is a single space, " " (without the quotes). This will deny access to job details to all users except the job owner, queue administrators, cluster administrators, and cluster owner.

Note

In order to control access to job details in the JobTracker web UI, you must configure MapReduce ACLs as described earlier, as well as enable web UI authentication as described in Chapter 11.

YARN (MR2)

With YARN/MR2, queue ACLs are no longer defined globally and each scheduler provides its own method of defining ACLs. ACLs are still enabled globally and there is a global ACL that defines YARN administrators. The settings to enable YARN ACLs and to define the admins are configured in the *yarn-site.xml*. Example values are provided in Example 6-8.

Example 6-8. YARN ACL configuration in yarn-site.xml

```
<property>
  <name>yarn.acl.enable</name>
  <value>true</value>
</property>
<property>
  <name>yarn.admin.acl</name>
  <value>yarn hadoop-admins</value>
</property>
```

Because each scheduler is configured differently, we will walk through setting up queue ACLs one scheduler at a time. For both examples, we will implement the same use case. Our cluster is primarily used for running production ETL pipelines, as well as production queries that generate regular reports. There is some ad hoc reporting as well, but production jobs should always take priority. In order to control access, we define two additional groups of users that contain only a subset of the *hadoop-users* we defined earlier. The *production-etl* group contains users that run production ETL jobs and the *production-queries* group contains users that run production queries. For this example, Alice is a member of the *production-etl* group while Bob is a member of the *production-queries* group. Let's start by configuring the FairScheduler.

FairScheduler

In order to guarantee the resources needed by the production jobs, we must first disable the default behavior of the FairScheduler, which is to place each user into their own queue that matches their username. This is done by setting two parameters, `yarn.scheduler.fair.user-as-default_queue` and `yarn.scheduler.fair.allow-undeclared-pools`, to `false`. The first parameter changes the default queue to `default` and the second ensures that users can't submit jobs to queues that have not be predefined. These settings, as well as the setting to enable the FairScheduler, are found in Example 6-9.

Example 6-9. FairScheduler configuration in yarn-site.xml

```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>
    org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler
  </value>
</property>
<property>
  <name>yarn.scheduler.fair.user-as-default-queue</name>
  <value>false</value>
</property>
<property>
  <name>yarn.scheduler.fair.allow-undeclared-pools</name>
  <value>false</value>
</property>
```

Next, we must define the queues and their ACLs within the *fair-scheduler.xml* file. The FairScheduler uses a hierarchical queue system and each queue is a descendant of the *root* queue. In our example, we want to provide 90% of the cluster resources to production jobs and 10% to ad hoc jobs. To achieve this, we define two direct children of the root queue: `prod` for production jobs and `default` for ad hoc jobs. We use the name "default" for the ad hoc queue because that is the queue jobs are submitted to if a queue is not specified. Resource management is a complex topic and we could tweak a lot of different settings to control the resources just so. Because our focus is on security, we'll use a simplified scheme and just control the resources with the weight of the queues. All that you need to understand is that for all queues that share a common parent, their resource allocation is defined as their weight divided by the weight of all of their siblings. In this case, we can assign `prod` a weight of 9.0 and `default` a weight of 1.0 to get the desired 90/10 split.

We also want to break up the production queue into two subqueues: one for ETL jobs and one for queries. For this example, we'll leave the two queues equally weighted by setting both queues to a weight of 1.0. It is important to note that the calculation of fair share happens in the context of your parent queue. In this example, that means that because we're giving both the `etl` and `queries` queues 50% of the resources of the `prod` queue, they'll end up with a global fair share of 45% each (50% × 90% = 45%).

Oozie Authorization

Apache Oozie has a very simple authorization model with two levels of accounts: *users* and *admin*

users. Users have the following permissions:

- Read access to all jobs
- Write access to their own jobs
- Write access to jobs based on a per-job access control list (list of users and groups)
- Read access to admin operations

Admin users have the following permissions:

- Write access to all jobs
- Writes access to admin operations

You can enable Oozie authorization by setting the following parameters in the *oozie-site.xml* file:

```
<property>
  <name>oozie.service.AuthorizationService.security.enabled</name>
  <value>true</value>
</property>
<property>
  <name>oozie.service.AuthorizationService.admin.groups</name>
  <value>oozie-admins</value>
</property>
```

If you don't set the `oozie.service.AuthorizationService.admin.groups` parameter, then you can specify a list of admin users, one per line, in the *adminusers.txt* file:

```
oozie
alice
```

In addition to owners and admin users having write access to a job, users can be granted write privileges through the use of a job-specific access control list. An Oozie ACL uses the same syntax as Hadoop ACLs (see Table 6-1) and is set in the `oozie.job.acl` property of a workflow, coordinator, or bundle *job.properties* file when submitting a job.

HBase and Accumulo Authorization

Apache HBase and Apache Accumulo are sorted, distributed key/value stores based on the design of Google's BigTable and built on top of HDFS and ZooKeeper. Both systems share a similar data model and are designed to enable random access and update workloads on top of HDFS which is a write-once filesystem. Data is stored in *rows* that contain one or more *columns*. Unlike a relational database, the columns in each row can differ. This makes it easier to implement complex data models where not every record shares the same schema. Each row is indexed with a primary key called a *row id* or *row key*; and within a row, each value is further indexed by a *column key* and *timestamp*. The intersection of a row key, column key and timestamp, along with the value they point to, is often called a *cell*. Internally, HBase and Accumulo store data as a sorted sequence of key/value pairs with the key consisting of the row ID, column key, and timestamp. Column keys are further split into two components; a *column family* and a *column qualifier*. In HBase, all of the columns in the same column family are stored in separate files on disk whereas in Accumulo

multiple column families can be grouped together into *locality groups*.

A collection of sorted rows is called a *table*. In HBase, the set of column families is predefined per table while Accumulo lets users create new column families on the fly. In both systems, column qualifiers do not need to be predefined and arbitrary qualifiers can be inserted into any row. A logical grouping of tables, similar to a database or schema in a relational database system, is called a *namespace*. Both HBase and Accumulo support permissions at the system, namespace, and table level. The available permissions and their semantics differ between Accumulo and HBase, so let's start by taking a look at Accumulo's permission model.

System, Namespace, and Table-Level Authorization

At the highest level, Accumulo supports system permissions. Generally, system permissions are reserved for the Accumulo root user or Accumulo administrators. Permissions set at a higher level are inherited by objects at a lower level. For example, if you have the system permission `CREATE_TABLE`, you can create a table in any namespace even if you don't have explicit permissions to create tables in that namespace. See Table 6-7 for a list of system-level permissions, their descriptions, and the equivalent namespace-level permission.

Warning

Throughout this section, you'll see many references to the Accumulo root user. This is not the same as the root system account. The Accumulo root user is automatically created when Accumulo is initialized, and that user is granted all of the system-level permissions. The root user can never have these permissions revoked, which prevents leaving Accumulo in a state where no one can administer it.

Table 6-7. System-level permissions in Accumulo

Permission	Description	Equivalent namespace permission
`System.GRANT`	Permission to grant permissions to other users; reserved for the Accumulo root user	`Namespace.ALTER_NAMESPACE`
`System.CREATE_TABLE`	Permission to create tables	`Namespace.CREATE_TABLE`
`System.DROP_TABLE`	Permission to delete tables	`Namespace.DROP_TABLE`
`System.ALTER_TABLE`	Permission to modify tables	`Namespace.ALTER_TABLE`
`System.DROP_NAMESPACE`	Permission to drop namespaces	`Namespace.DROP_NAMESPACE`
`System.ALTER_NAMESPACE`	Permission to modify namespaces	`Namespace.ALTER_NAMESPACE`
`System.CREATE_USE`	Permission to create new users	N/A

Permission	Description	Equivalent namespace permission
R		
System.DROP_USER	Permission to delete users	N/A
System.ALTER_USER	Permission to change user passwords, permissions, and authorizations	N/A
System.SYSTEM	Permission to perform administrative actions on tables or users	N/A
System.CREATE_NAMESPACE	Permission to create new namespaces	N/A

Namespaces are a logical collection of tables and are useful for organizing tables and delegating administrative functions to smaller groups. Suppose the marketing department needs to host a number of Accumulo tables to power some of its applications. In order to reduce the burden on the Accumulo administrator, we can create a marketing namespace and give `GRANT`, `CREATE_TABLE`, `DROP_TABLE`, and `ALTER_TABLE` permissions to an administrator in marketing. This will allow the department to create and manage its own tables without having to grant system-level permissions or wait for the Accumulo administrator. A number of namespace-level permissions are inherited by tables in the namespace. See Table 6-8 for the list of namespace-level permissions, their descriptions, and the equivalent table-level permission.

Table 6-8. Namespace-level permissions in Accumulo

Permission	Description	Equivalent table permission
Namespace.READ	Permission to read (scan) tables in the namespace	Table.READ
Namespace.WRITE	Permission to write (put/delete) to tables in the namespace	Table.WRITE
Namespace.GRANT	Permission to grant permissions to tables in the namespace	Table.GRANT
Namespace.BULK_IMPORT	Permission to bulk import data into tables in the namespace	Table.BULK_IMPORT
Namespace.ALTER_TABLE	Permission to set properties on tables in the namespace	Table.ALTER_TABLE
Namespace.DROP_TABLE	Permission to delete tables in the namespace	Table.DROP_TABLE
Namespace.CREATE_TABLE	Permission to create tables in the	N/A

Permission	Description	Equivalent table permission
	namespace	
`Namespace.ALTER_NAMESPACE`	Permission to set properties on the namespace	N/A
`Namespace.DROP_NAMESPACE`	Permission to delete the namespace	N/A

Chapter 7. Apache Sentry (Incubating)

Over the lifetime of the various Hadoop ecosystem projects, secure authorization has been added in a variety of different ways. It has become increasingly challenging for administrators to implement and maintain a common system of authorization across multiple components. To compound the problem, the various components have different levels of granularity and enforcement of authorization controls, which often leave an administrator confused as to what a given user can actually do (or not do) in the Hadoop environment. These issues, and many others, were the driving force behind the proposal for Apache Sentry (Incubating).

The Sentry proposal identified a need for fine-grained *role-based access controls (RBAC)* to give administrators more flexibility to control what users can access. Traditionally, and covered already, HDFS authorization controls are limited to simple POXIS-style permissions and extended ACLs. What about frameworks that work on top of HDFS, such as Hive, Cloudera Impala, Solr, HBase, and others? Sentry's goals are to implement authorization for Hadoop ecosystem components in a unified way so that security administrators can easily control what users and groups have access to without needing to know the ins and outs of every single component in the Hadoop stack.

Sentry Concepts

Each component that leverages Sentry for authorization must have a Sentry *binding*. The binding is a plug-in that the component uses to delegate authorization decisions to Sentry. This binding applies the relevant *model* to use for authorization decisions. For example, a SQL model would apply for the components Hive and Impala, a Search model would apply to Solr, and a BigTable model would apply to HBase and Accumulo. Sentry privilege models are discussed in detail a bit later.

With the appropriate model in place, Sentry uses a *policy engine* to determine if the requested action is authorized by checking the *policy provider*. The policy provider is the storage mechanism for the policies, such as a database or text file. Figure 7-1 shows how this looks conceptually.

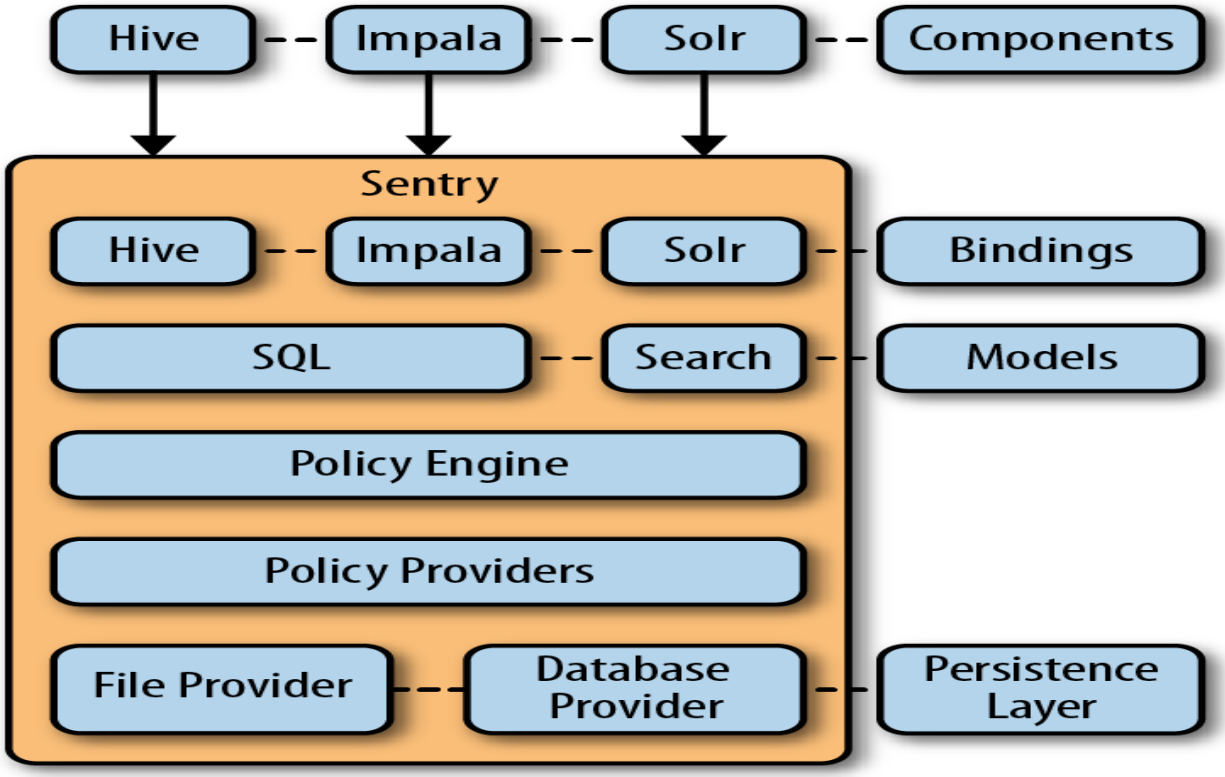

Figure 7-1. Sentry components

This flow makes sense for how components leverage Sentry at a high level, but what about the actual decision-making process for authorization by the policy engine? Regardless of the model in place for a given component, there are several key concepts that are common. *Users* are what you expect them to be. They are identities performing a specific action, such as executing a SQL query, searching a collection, reading a file, or retrieving a key/value pair. Users also belong to *groups*. In the Sentry context, groups are a collection of users that have the same needs and privileges. A *privilege* in Sentry is a unit of data access and is represented by a tuple of an object and an action to be performed on the object. For example, an object could be a `DATABASE`, `TABLE`, or `COLLECTION`, and the action could be `CREATE`, `READ`, `WRITE`.

Important

Sentry privileges are always defined in the positive case because, by default, Sentry denies access to every object. This is not to be confused with `REVOKE` syntax covered later, which simply removes the positive case privileges.

Lastly, a *role* is a collection of privileges and is the basic unit of grant within Sentry. A role typically aligns with a business function, such as a marketing analyst or database administrator. The relationship between users, groups, privileges, and roles is important in Sentry, and adheres to the following logic:

- A group contains multiple users
- A role is assigned a group
- A role is granted privileges

This is illustrated in Figure 7-2.

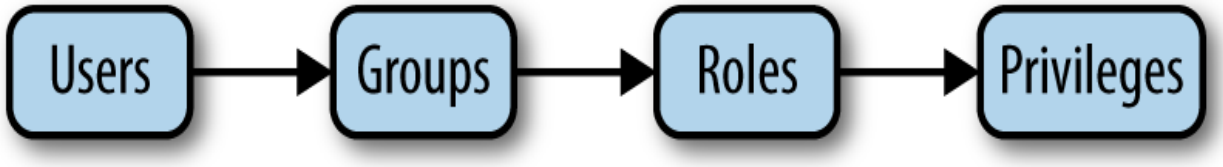

Figure 7-2. Sentry entity relationships

This relationship is strictly enforced in Sentry. It is not possible to assign a role to a user or grant privileges to a group, for example. While this relationship is strict, there are several many-to-many relationships in play here. A user can belong to many groups and a group can contain many users. For example, Alice could belong to both the Marketing and Developer groups, and the Developer group could contain both Alice and Bob.

Also, a role can be assigned to many groups and a group can have many roles. For example, the SQL Analyst role could be assigned to both the Marketing and Developer groups, and the Developer group could have both the SQL Analyst role and Database Administrator role.

Lastly, a role can be granted many privileges and a given privilege can be a part of many roles. For example, the SQL Analyst role could have `SELECT` privileges on the clickstream `TABLE` and `CREATE` privileges on the marketing `DATABASE`, and the same `CREATE` privilege on the marketing `DATABASE` could also be granted to the Database Administrator role.

Now that the high-level Sentry concepts have been covered, we can take a closer look at implementation, starting with the latest and greatest: the Sentry service.

The Sentry Service

When the Sentry project first made its way into the Apache incubator, the first release available to the public was one that utilized a plug-in–based approach. Services that leveraged Sentry were configured with a Sentry plug-in (the binding), and this plug-in ran inside the service in question and directly read the policy file. There was no daemon process for Sentry, like many of the other Hadoop ecosystem components. Furthermore, Sentry policies were configured in a plain text file that enumerated every policy. Whenever a policy was added, modified, or removed, it required a modification to the file. As you might imagine, this approach is rather simplistic, cumbersome to maintain, and prone to errors. To compound the problem, mistakes made in the policy file invalidated the *entire* file!

Thankfully, Sentry has largely moved beyond this early beginning and has grown into a first-class citizen in the Hadoop ecosystem. Starting with version 1.4, Sentry comes with a service that can be leveraged by Hive and Impala. This service utilizes a database backend instead of a text file for policy storage. Additionally, services that use Sentry are now configured with a binding that points to the Sentry service instead of a binding to handle all of the authorization decisions locally. Because of advancements in Sentry's architecture, it is not recommended to use the policy file–based configuration for Hive and Impala except on legacy systems. That being said, this chapter will include information about both configuration options. Figure 7-3 depicts how the Sentry service fits in with SQL access.

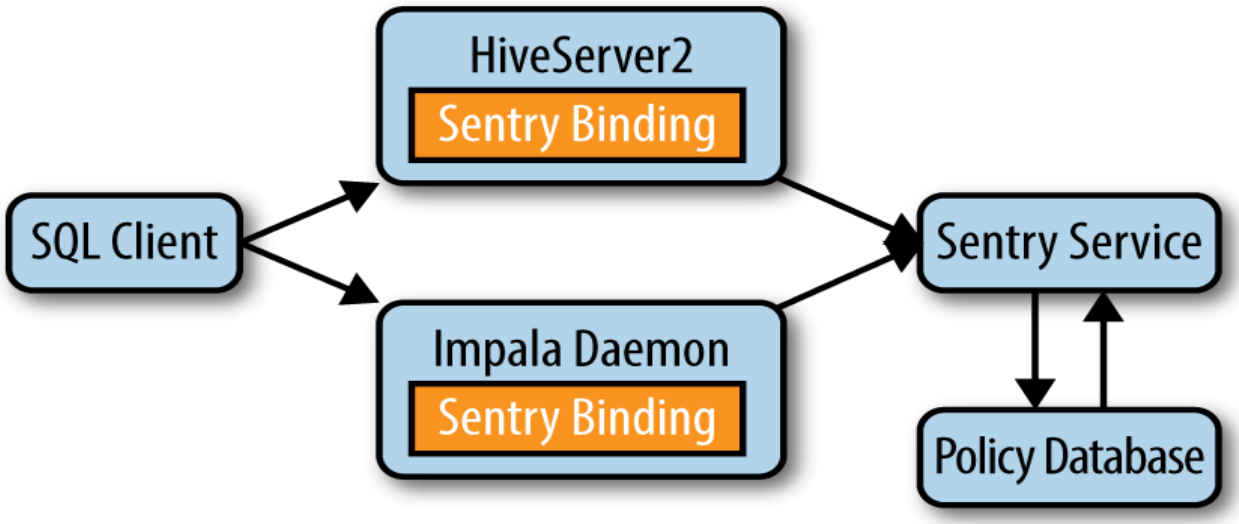

Figure 7-3. Sentry service architecture

At the time of this writing, Solr still utilizes policy files. It is expected that Solr as well as any other new Sentry-enabled services will move away from using policy file–based configurations.

Sentry Service Configuration

The first part of getting Sentry up and running in the cluster is to configure the Sentry service. The master configuration file for Sentry is called *sentry-site.xml*. Example 7-1 shows a typical configuration for the Sentry server in a Kerberos-enabled cluster, and Table 7-1 explains the configuration parameters. Later on in the chapter, we will take a look at how Hadoop ecosystem components utilize this Sentry service for authorization.

Example 7-1. Sentry service sentry-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <property>
    <name>sentry.service.server.rpc-address</name>
    <value>server1.example.com</value>
  </property>
  <property>
    <name>sentry.service.server.rpc-port</name>
    <value>8038</value>
  </property>
  <property>
    <name>sentry.service.admin.group</name>
    <value>hive,impala,hue</value>
  </property>
  <property>
    <name>sentry.service.allow.connect</name>
    <value>hive,impala,hue</value>
```

```
  </property>
  <property>
    <name>sentry.store.group.mapping</name>
    <value>org.apache.sentry.provider.common.HadoopGroupMappingService</value>
  </property>
  <property>
    <name>sentry.service.server.principal</name>
    <value>sentry/_HOST@EXAMPLE.COM</value>
  </property>
  <property>
    <name>sentry.service.security.mode</name>
    <value>kerberos</value>
  </property>
  <property>
    <name>sentry.service.server.keytab</name>
    <value>sentry.keytab</value>
  </property>
  <property>
    <name>sentry.store.jdbc.url</name>
    <value>jdbc:mysql://server2.example.com:3306/</value>
  </property>
  <property>
    <name>sentry.store.jdbc.driver</name>
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property>
    <name>sentry.store.jdbc.user</name>
    <value>sentry</value>
  </property>
  <property>
    <name>sentry.store.jdbc.password</name>
    <value>sentry_password</value>
  </property>
</configuration>
```

Table 7-1 shows all of the relevant configuration parameters for *sentry-site.xml*. This includes parameters that are used for configuring the Sentry service, as well as configurations for policy file–based implementations and component-specific configurations.

Table 7-1. sentry-site.xml configurations

Configuration	**Description**
`hive.sentry.provider`	Typically `org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProvider` for Hadoop groups; local groups can be defined only in a policy-file deployment and use `org.apache.sentry.provider.file.LocalGroupResourceAuthorizationProvider`
`hive.sentry.provider.resource`	The location of the policy file; can be both

	`file://` and `hdfs://` URIs
hive.sentry.server	The name of the Sentry server; can be anything
`sentry.hive.provider.backend`	Type of Sentry service: `org.apache.sentry.provider.file.SimpleFileProviderBackend` or `org.apache.sentry.provider.db.SimpleDBProviderBackend`
`sentry.metastore.service.users`	List of users allowed to bypass Sentry policies for the Hive metastore; only applies to Sentry service deployments
`sentry.provider`	Same options as `hive.sentry.provider`; used by Solr
`sentry.service.admin.group`	List of comma-separated groups that are administrators of the Sentry server
`sentry.service.allow.connect`	List of comma-separated users that are allowed to connect; typically only service users, not end users
`sentry.service.client.server.rpc-address`	Client configuration of the Sentry service endpoint
`sentry.service.client.server.rpc-port`	Client configuration of the Sentry service port
`sentry.service.security.mode`	The security mode the Sentry server is operating under; kerberos or none
`sentry.service.server.keytab`	Keytab filename that contains the credentials for `sentry.service.server.principal`
`sentry.service.server.principal`	Service principal name contained in `sentry.service.server.keytab` that the Sentry server identifies itself as
`sentry.service.server.rpc-address`	The hostname to start the Sentry server on
`sentry.service.server.rpc-port`	The port to listen on
`sentry.solr.provider.resource`	The location of the policy file for Solr; can be both `file://` and `hdfs://` URIs
`sentry.store.jdbc.driver`	The JDBC driver name to use to connect to the database
`sentry.store.jdbc.password`	The JDBC password to use
`sentry.store.jdbc.url`	The JDBC URL for the backend database the Sentry server should use
`sentry.store.jdbc.user`	The JDBC username to connect as
`sentry.store.group.mapping`	The class that provides the mapping of users to groups; typically `org.apache.sentry.provider.commo`

	`n.HadoopGroupMappingService`

Solr Authorization

Authorization for Solr starts with collections. Collections are the main entry point of access, much like how databases are for SQL. Sentry authorization initially started with defining privileges at the collection level. Sentry has since evolved to provide document-level authorization. Document-level authorization is done by tagging each document with a special field name containing the value that corresponds to an associated Sentry role name defined in the Sentry policy file (described later). The tagging of documents in this fashion would be done at ingest time, so it is important to have a good sense of role names to avoid needing to reprocess documents to change tag values.

Solr Sentry Configuration

This section explains how to set up Solr with Sentry authorization. Example 7-11 shows what is needed in the *sentry-site.xml* configuration file for the Solr servers.

Example 7-11. Solr sentry-site.xml policy file deployment

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <property>
    <name>sentry.provider</name>
    <value>
      org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProvider
    </value>
  </property>
  <property>
    <name>sentry.solr.provider.resource</name>
    <value>/user/solr/sentry/sentry-provider.ini</value>
  </property>
</configuration>
```

The two configuration properties shown in Example 7-11 should look very familiar at this point, but the configuration property names are slightly different with Solr. The `sentry.provider` configuration property works just like the `hive.sentry.provider` configuration for Hive and the `authorization_policy_provider_class` flag for Impala. The `sentry.solr.provider.resource` configuration property specifies the location of the Sentry policy file. Again, this policy file can be located either on the local filesystem or on HDFS. It needs to be readable by the user that the Solr servers are running as (typically the solr user).

To set up the Solr servers with Sentry authorization, some environment variables are needed. These can either be set as environment variables or as lines in the */etc/default/solr* configuration file. The first variable is `SOLR_SENTRY_ENABLED`. This obviously enables Sentry authorization when set to true. The next is `SOLR_AUTHORIZATION_SUPERUSER`. This variable defines the user that has superuser privileges, which typically should be the solr user. The last variable is `SOLR_AUTHORIZATION_SENTRY_SITE`, which specifies the location of the *sentry-site.xml*

configuration file described earlier. Example 7-12 shows how this looks.

Example 7-12. Solr environment variables in Sentry policy file deployment

```
...Other unrelated environment variables omitted for brevity
SOLR_SENTRY_ENABLED=true
SOLR_AUTHORIZATION_SUPERUSER=solr
SOLR_AUTHORIZATION_SENTRY_SITE=/etc/solr/conf/sentry-site.xml
```

It was mentioned earlier in this section that document-level authorization can be used. In order to make that happen, a few configurations are necessary for the collection. By default, collections are configured using the *solrconfig.xml* configuration file. This file needs to look like Example 7-13.

Example 7-13. Document-level security solrconfig.xml

```
<searchComponent name="queryDocAuthorization"
class="org.apache.solr.handler.component.QueryDocAuthorizationComponent">
    <bool name="enabled">true</bool>
    <str name="sentryAuthField">sentry_auth</str>
    <str name="allRolesToken">*</str>
</searchComponent>
```

Example 7-13 shows that the class `org.apache.solr.handler.component.QueryDocAuthorizationComponent` is used for document-level authorization decisions. It is turned on by setting the `enabled` property to `true`. The configuration property `sentryAuthField` defines the name of the field in a document that contains the authorization *token* to determine access. The default value of `sentry_auth` is shown, but this can be anything. Documents will use this tag to insert the role name that is required to access the document. The last configuration property `allRolesToken` defines the token that allows every role to access a given document. The default is *, and it makes sense to leave that as is to remain consistent with wildcard matches in other Sentry privileges.

Sentry Privilege Models

In this section, we take a look at the privilege models for the various services that Sentry provides authorization for. The privilege models identify privileges, object types that the privileges apply to, the scope of the privilege, and other useful information that will help a security administrator understand the authorization controls available, and to what granularity. Having a good grasp on the privilege models here will ensure that appropriate policies are selected to meet the desired level of authorization controls to protect data from unauthorized access.

SQL Privilege Model

Sentry provides three types of privileges for SQL access: SELECT, INSERT, and ALL. These privileges are not available for every object. Table 7-2 provides information on which privileges apply to which object in a SQL context. The SQL privilege model itself is a hierarchy, meaning privileges to container objects imply privileges to child objects. This is important to fully understand what users do or do not have access to.

Table 7-2. SQL privilege types[a]

Privilege	Object
INSERT	TABLE,URI
SELECT	TABLE,VIEW,URI
ALL	SERVER,DB,URI

[a] All privilege model tables are reproduced from cloudera.com with permission from Cloudera, Inc.

Table 7-3 lays out which container privilege yields the granular privilege on a given object. For example, the first line in the table should be interpreted as "ALL privileges on a SERVER object implies ALL privileges on a DATABASE object."

Table 7-3. SQL privilege hierarchy

Base Object	Granular Privilege	Container Object	Container Privilege That Implies Granular Privilege
DATABASE	ALL	SERVER	ALL
TABLE	INSERT	DATABASE	ALL
TABLE	SELECT	DATABASE	ALL
VIEW	SELECT	DATABASE	ALL

The final portion of the SQL privilege model is to understand how privileges map to SQL operations. Table 7-4 shows for a given SQL operation, what object scope does the operation apply to, and what privileges are required to perform the operation. For example, the first line in the table should be interpreted as "CREATE DATABASE applies to the SERVER object and requires ALL privileges on the SERVER object." Some of the SQL operations involve more than one privilege, such as creating views. Creating a new view requires ALL privileges on the DATABASE in which the view is to be created, as well as SELECT privileges on the TABLE/VIEW object(s) referenced by the view.

Table 7-4. SQL privileges

SQL Operation	Scope	Privileges
CREATE DATABASE	SERVER	ALL
DROP DATABASE	DATABASE	ALL
CREATE TABLE	DATABASE	ALL
DROP TABLE	TABLE	ALL
CREATE VIEW	DATABASE; SELECT on	ALL

SQL Operation	Scope	Privileges
	TABLE	
DROP VIEW	VIEW/TABLE	ALL
CREATE INDEX	TABLE	ALL
DROP INDEX	TABLE	ALL
ALTER TABLE ADD COLUMNS	TABLE	ALL
ALTER TABLE REPLACE COLUMNS	TABLE	ALL
ALTER TABLE CHANGE column	TABLE	ALL
ALTER TABLE RENAME	TABLE	ALL
ALTER TABLE SET TBLPROPERTIES	TABLE	ALL
ALTER TABLE SET FILEFORMAT	TABLE	ALL
ALTER TABLE SET LOCATION	TABLE	ALL
ALTER TABLE ADD PARTITION	TABLE	ALL
ALTER TABLE ADD PARTITION location	TABLE	ALL
ALTER TABLE DROP PARTITION	TABLE	ALL
ALTER TABLE PARTITION SET FILEFORMAT	TABLE	ALL
SHOW TBLPROPERTIES	TABLE	SELECT/INSERT
SHOW CREATE TABLE	TABLE	SELECT/INSERT
SHOW PARTITIONS	TABLE	SELECT/INSERT
DESCRIBE TABLE	TABLE	SELECT/INSERT
DESCRIBE TABLE PARTITION	TABLE	SELECT/INSERT
LOAD DATA	TABLE; URI	INSERT
SELECT	TABLE	SELECT
INSERT OVERWRITE TABLE	TABLE	INSERT

SQL Operation	Scope	Privileges
CREATE TABLE AS SELECT	DATABASE; SELECT on TABLE	ALL
USE database	ANY	ANY
ALTER TABLE SET SERDEPROPERTIES	TABLE	ALL
ALTER TABLE PARTITION SET SERDEPROPERTIES	TABLE	ALL
CREATE ROLE	SERVER	ALL
GRANT ROLE TO GROUP	SERVER	ALL
GRANT PRIVILEGE ON SERVER	SERVER	ALL
GRANT PRIVILEGE ON DATABASE	DATABASE	WITH GRANT OPTION
GRANT PRIVILEGE ON TABLE	TABLE	WITH GRANT OPTION

While most of the SQL operations are supported by both Hive and Impala, some operations are supported only by Hive or Impala, or have not been implemented yet. Table 7-5 lists the SQL privileges that only apply to Hive, and Table 7-6 lists the SQL privileges that only apply to Impala.

Table 7-5. Hive-only SQL privileges

SQL Operation	Scope	Privileges
INSERT OVERWRITE DIRECTORY	TABLE; URI	INSERT
ANALYZE TABLE	TABLE	SELECT + INSERT
IMPORT TABLE	DATABASE; URI	ALL
EXPORT TABLE	TABLE; URI	SELECT
ALTER TABLE TOUCH	TABLE	ALL
ALTER TABLE TOUCH PARTITION	TABLE	ALL
ALTER TABLE CLUSTERED BY SORTED BY	TABLE	ALL
ALTER TABLE ENABLE/DISABLE	TABLE	ALL

SQL Operation	Scope	Privileges
ALTER TABLE PARTITION ENABLE/DISABLE	TABLE	ALL
ALTER TABLE PARTITION RENAME TO PARTITION	TABLE	ALL
ALTER DATABASE	DATABASE	ALL
DESCRIBE DATABASE	DATABASE	SELECT/INSERT
SHOW COLUMNS	TABLE	SELECT/INSERT
SHOW INDEXES	TABLE	SELECT/INSERT

Table 7-6. Impala-only SQL privileges

SQL Operation	Scope	Privileges
EXPLAIN	TABLE	SELECT
INVALIDATE METADATA	SERVER	ALL
INVALIDATE METADATA table	TABLE	SELECT/INSERT
REFRESH table	TABLE	SELECT/INSERT
CREATE FUNCTION	SERVER	ALL
DROP FUNCTION	SERVER	ALL
COMPUTE STATS	TABLE	ALL

Solr Privilege Model

With Solr, Sentry provides three types of privileges: `QUERY`, `UPDATE`, and `*` (ALL). The privilege model for Solr is broken down between privileges that apply to request handlers and those that apply to collections. In Tables 7-8 through 7-10, the *admin* collection name is a special collection in Sentry that is used to represent administrative actions. In all of the Solr privilege model tables, *collection1* denotes an arbitrary collection name.

Table 7-7. Solr privilege table for nonadministrative request handlers

Request handler	Required privilege	Collections that require privilege
select	QUERY	collection1
query	QUERY	collection1
get	QUERY	collection1
browse	QUERY	collection1

Request handler	Required privilege	Collections that require privilege
tvrh	QUERY	collection1
clustering	QUERY	collection1
terms	QUERY	collection1
elevate	QUERY	collection1
analysis/field	QUERY	collection1
analysis/document	QUERY	collection1
update	UPDATE	collection1
update/json	UPDATE	collection1
update/csv	UPDATE	collection1

Table 7-8. Solr privilege table for collections admin actions

Collection action	Required privilege	Collections that require privilege
create	UPDATE	admin, collection1
delete	UPDATE	admin, collection1
reload	UPDATE	admin, collection1
createAlias	UPDATE	admin, collection1
deleteAlias	UPDATE	admin, collection1
syncShard	UPDATE	admin, collection1
splitShard	UPDATE	admin, collection1
deleteShard	UPDATE	admin, collection1

Table 7-9. Solr privilege table for core admin actions

Collection action	Required privilege	Collections that require privilege
create	UPDATE	admin, collection1
rename	UPDATE	admin, collection1
load	UPDATE	admin, collection1

Collection action	Required privilege	Collections that require privilege
unload	UPDATE	admin, collection1
status	UPDATE	admin, collection1
persist	UPDATE	admin
reload	UPDATE	admin, collection1
swap	UPDATE	admin, collection1
mergeIndexes	UPDATE	admin, collection1
split	UPDATE	admin, collection1
prepRecover	UPDATE	admin, collection1
requestRecover	UPDATE	admin, collection1
requestSyncShard	UPDATE	admin, collection1
requestApplyUpdates	UPDATE	admin, collection1

Table 7-10. Solr privilege table for info and AdminHandlers

Request handler	Required privilege	Collections that require privilege
LukeRequestHandler	QUERY	admin
SystemInfoHandler	QUERY	admin
SolrInfoMBeanHandler	QUERY	admin
PluginInfoHandler	QUERY	admin
ThreadDumpHandler	QUERY	admin
PropertiesRequestHandler	QUERY	admin
LogginHandler	QUERY, UPDATE (or *)	admin
ShowFileRequestHandler	QUERY	admin

Policy File Verification and Validation

When Sentry was first architected to use plain-text policy files, it was immediately apparent that administrators would need some kind of validation tool to perform basic sanity checks on the file prior to putting it in place. Sentry ships with a binary file, named `sentry` (surprise, surprise), which provides an important feature for policy file implementations: the `config-tool` command. This command allows an administrator to check the policy file for errors, but it also provides a mechanism to verify privileges for a given user. Example 7-17 demonstrates validating a policy file, where the first policy file has no errors and the second policy file has a typo (the word "sever" instead of "server").

Example 7-17. Sentry config-tool validation

```
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provider.ini
-v
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
No errors found in the policy file
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-
provider2.ini -v
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider2.ini
Sentry server: server1
 *** Found configuration problems ***
ERROR: Error processing file file:/etc/sentry/sentry-provider2.ini
 No authorizable found for sever=server1
ERROR: Failed to process global policy file
 file:/etc/sentry/sentry-provider2.ini
Sentry tool reported Errors:
org.apache.sentry.core.common.SentryConfigurationException:
        at org.apache.sentry.provider.file.SimpleFileProviderBackend.
         validatePolicy(SimpleFileProviderBackend.java:198)
        at org.apache.sentry.policy.db.SimpleDBPolicyEngine.
         validatePolicy(SimpleDBPolicyEngine.java:87)
        at org.apache.sentry.provider.common.ResourceAuthorizationProvider.
         validateResource(ResourceAuthorizationProvider.java:170)
        at org.apache.sentry.binding.hive.authz.SentryConfigTool.
         validatePolicy(SentryConfigTool.java:247)
        at org.apache.sentry.binding.hive.authz.
         SentryConfigTool$CommandImpl.run(SentryConfigTool.java:638)
        at org.apache.sentry.SentryMain.main(SentryMain.java:94)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.
         invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.
         invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.apache.hadoop.util.RunJar.main(RunJar.java:212
[root@server1 ~]#
```

Verifying a user's privileges is another powerful feature offered by the `config-tool`. This can be

done both by listing *all* privileges for a given user, or can be more specific by testing whether a given user would be authorized to execute a certain query. Example 7-18 demonstrates the usage of these features.

Example 7-18. Sentry config-tool Verification

```
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool \
 -s file:///etc/sentry/sentry-site.xml \
 -i file:///etc/sentry/sentry-provider.ini -l -u bob
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
Available privileges for user bob:
        server=server1
        server=server1->uri=hdfs://server1.example.com:8020/tmp
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool \
 -s file:///etc/sentry/sentry-site.xml \
 -i file:///etc/sentry/sentry-provider.ini -l -u alice
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
Available privileges for user alice:
        *** No permissions available ***
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provider.ini
 -u bob -e "select * from sample_08"
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
User bob has privileges to run the query
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-tool
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provider.ini
 -u alice  -e "select * from sample_08"
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
FAILED: SemanticException No valid privileges
*** Missing privileges for user alice:
         server=server1->db=default->table=sample_08->action=select
User alice does NOT have privileges to run the query
Sentry tool reported Errors: Compilation error: FAILED:
SemanticException No valid privileges
[root@server1 ~]#
```

Migrating From Policy Files

When the Sentry service was added to the project, a useful migration tool was also included. This tool allows an administrator to import the policies from the existing file into the Sentry service backend database. This alleviated the pains of needing to derive the SQL syntax for every policy and manually adding them to the database. The migration tool is a feature enhancement to the `config-tool` covered in the last section. Example 7-19 demonstrates the usage.

Example 7-19. Sentry Policy Import Tool

```
[root@server1 ~]# sentry --command config-tool --import \
-i file:///etc/sentry/sentry-provider.ini
Using hive-conf-dir /etc/hive/conf/
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:///etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
CREATE ROLE analyst_role;
GRANT ROLE analyst_role TO GROUP analysts;
# server=server1
GRANT SELECT ON DATABASE default TO ROLE analyst_role;
CREATE ROLE admin_role;
CREATE ROLE developer_role;
CREATE ROLE etl_role;
GRANT ROLE admin_role TO GROUP admins;
GRANT ALL ON SERVER server1 TO ROLE admin_role;
GRANT ROLE developer_role TO GROUP developers;
# server=server1
GRANT ALL ON DATABASE default TO ROLE developer_role;
GRANT ROLE etl_role TO GROUP etl;
# server=server1
GRANT INSERT ON DATABASE default TO ROLE etl_role;
# server=server1
GRANT SELECT ON DATABASE default TO ROLE etl_role;
[root@server1 ~]#
```

Summary

Conceptually, Sentry is a familiar and easy-to-understand concept, but as you have seen, the devil is in the details. Although Sentry is one of the newer components of the Hadoop ecosystem, it is quickly becoming an integral part of Hadoop security. Sentry has evolved rapidly in a short time and the expectation is that other ecosystem components will integrate with Sentry to provide strong authorization in a unified way.

Now that we have wrapped up the extensive topics of authentication and authorization, it is time to look at accounting to make sense of user activity in the cluster.

Chapter 8. Accounting

So far in this part of the book, we've described how to properly identify and authenticate users and services, as well as how authorization controls limit what users and services can do in the cluster. While all of these various controls do a good job defining and enforcing a security model for a Hadoop cluster, they do not complete a fundamental component of a security model: accounting.

Also referred to as auditing, accounting is the mechanism to keep track of what users and services are doing in the cluster. This is a critical piece of the security puzzle because without it, breaches in security can occur without anybody noticing. Accounting rounds out a security model by providing a record of what happened, which can be used for:

Active auditing

> This type of auditing is used in conjunction with some kind of alerting mechanism. For example, if a user tries to access a resource on the cluster and is denied, active auditing could generate an email to security administrators alerting them of this event.

Passive auditing

> This refers to auditing that does not generate some kind of alert. Passive auditing is often a bare-minimum requirement in a business so that designated auditors and security administrators can query audit events to look for certain events. For example, if there is a breach in security to the cluster, a security administrator can query the audit logs to find the data that was accessed during the breach.

Security compliance

> A business might be required to audit certain events to meet internal or legal compliance. This is most often the case where the data stored in HDFS contains sensitive information like personally identifiable information (PII), financial information such as credit card numbers and bank account numbers, and sensitive information about the business, like payroll records and business financials.

Hadoop components handle accounting differently depending on the purpose of the component. Components such as HDFS and HBase are data storage systems, so auditable events focus on reading, writing, and accessing data. Conversely, components such as MapReduce, Hive, and Impala are query engines and processing frameworks, so auditable events focus on end-user queries and jobs. The following subsections dig deeper into each component, and describe typical interactions with the component from an accounting point of view.

HDFS Audit Logs

HDFS provides two different audit logs that are used for two different purposes. The first, *hdfs-audit.log*, is used to audit general user activity such as when a user creates a new file, changes permissions of a file, requests a directory listing, and so on. The second, *SecurityAuth-hdfs.audit*, is used to audit service-level authorization activity. The setup for these logfiles involves hooking into `log4j.category.SecurityLogger` and `log4j.additivity.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit`. Example 8-1 shows how to do it.

Example 8-1. HDFS log4j.properties

```
# other logging settings omitted
hdfs.audit.logger=${log.threshold},RFAAUDIT
hdfs.audit.log.maxfilesize=256MB
hdfs.audit.log.maxbackupindex=20
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=
  ${hdfs.audit.logger}
```

```
log4j.additivity.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=false
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${log.dir}/hdfs-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p %c{2}: %m%n
log4j.appender.RFAAUDIT.MaxFileSize=${hdfs.audit.log.maxfilesize}
log4j.appender.RFAAUDIT.MaxBackupIndex=${hdfs.audit.log.maxbackupindex}
hadoop.security.logger=INFO,RFAS
hadoop.security.log.maxfilesize=256MB
hadoop.security.log.maxbackupindex=20
log4j.category.SecurityLogger=${hadoop.security.logger}
log4j.additivity.SecurityLogger=false
hadoop.security.log.file=SecurityAuth-${user.name}.audit
log4j.appender.RFAS=org.apache.log4j.RollingFileAppender
log4j.appender.RFAS.File=${log.dir}/${hadoop.security.log.file}
log4j.appender.RFAS.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAS.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
log4j.appender.RFAS.MaxFileSize=${hadoop.security.log.maxfilesize}
log4j.appender.RFAS.MaxBackupIndex=${hadoop.security.log.maxbackupindex}
```

So what actually shows up when an auditable event occurs? For this set of examples, let's assume the following:

- The user Alice is identified by the Kerberos principal `alice@EXAMPLE.COM`, and she has successfully used `kinit` to receive a valid TGT
- She does a directory listing on her HDFS home directory
- She creates an empty file named `test` in her HDFS home directory
- She changes the permissions of this file to be world-writable
- She attempts to move the file out of her home directory and into the */user* directory

In Example 8-2, Alice has done several actions with HDFS that are typical operations in HDFS. These are user activity events, so let's inspect *hdfs-audit.log* to see the trail that Alice left behind from her HDFS actions (the example logfile has been formatted for readability).

Example 8-2. hdfs-audit.log

```
...
2014-03-11 23:50:18,251 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice dst=null perm=null
2014-03-11 23:50:18,280 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=listStatus src=/user/alice dst=null perm=null
2014-03-11 23:50:32,058 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=null
perm=null
2014-03-11 23:50:32,073 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo   src=/user/alice dst=null
perm=null
2014-03-11 23:50:32,096 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=create src=/user/alice/test dst=null
 perm=alice:alice:rw-r-----
2014-03-11 23:50:39,558 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
```

```
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=null
perm=null
2014-03-11 23:50:39,587 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=setPermission src=/user/alice/test dst=null
 perm=alice:alice:rw-rw-rw-
2014-03-11 23:50:47,157 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user dst=null perm=null
2014-03-11 23:50:47,185 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=null
perm=null
2014-03-11 23:50:47,187 INFO FSNamesystem.audit: allowed=true
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/test dst=null perm=null
2014-03-11 23:50:47,190 INFO FSNamesystem.audit: allowed=false
ugi=alice@EXAMPLE.COM
 (auth:KERBEROS) ip=/10.1.1.1 cmd=rename src=/user/alice/test dst=/user/test
perm=nul
...
```

As you can see, the audit log shows pertinent information for each action Alice performed. Every action she performed required a `getfileinfo` command first, followed by the various actions she performed (listStatus, create, setPermission, and rename). In this log, it is clear who the user is that the event was for, what time it occurred, the IP address that action was performed from, and various other bits of information. The other important bit that was recorded was that Alice's last attempted action to move the file out of her home directory into a location she did not have permissions for was not allowed.

Sentry Audit Logs

In Chapter 7, we saw that the latest version of Sentry uses a service to facilitate authorization requests and manage interaction with the policy database. Auditing events that come as a result of modifying authorization policies is extremely critical in the accounting process. In order to do that, Sentry needs to be configured to capture audit events.
`sentry.hive.authorization.ddl.logger` logger class is the one that needs to be configured. Example 8-18 shows how this can be done.

Example 8-18. Sentry server log4j.properties

```
# other log settings omitted
log4j.logger.sentry.hive.authorization.ddl.logger=${sentry.audit.logger}
log4j.additivity.sentry.hive.authorization.ddl.logger=false
sentry.audit.logger=TRACE,RFAAUDIT
sentry.audit.log.maxfilesize=256MB
sentry.audit.log.maxbackupindex=20
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${log.dir}/audit/sentry-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p %c{2}: %m%n
log4j.appender.RFAAUDIT.MaxFileSize=${sentry.audit.log.maxfilesize}
log4j.appender.RFAAUDIT.MaxBackupIndex=${sentry.audit.log.maxbackupindex}
```

Now that Sentry is set up to log audit events, let's look at an example. For this example, Alice is a Sentry administrator and Bob is not. Alice uses the `beeline` shell to create a new role called

`analyst`, assign the role to the group `analystgrp`, and grant `SELECT` privileges on the `default` database to the role. Next, Bob tries to create a new role using the `impala-shell`, but is denied access. Example 8-19 shows the record of these actions.

Example 8-19. Sentry server audit log

```
2015-01-02 11:17:10,753 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"CREATE_ROLE","eventTime":"1420215430742","operationText":
  "CREATE ROLE analyst","allowed":"true","databaseName":null,
  "tableName":null,"resourcePath":null,"objectType":"ROLE"}
2015-01-02 11:17:37,537 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"ADD_ROLE_TO_GROUP","eventTime":"1420215457536",
  "operationText":"GRANT ROLE analyst TO GROUP analystgrp","allowed":"true",
  "databaseName":null,"tableName":null,"resourcePath":null,"objectType":"ROLE"}
2015-01-02 11:17:52,408 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"GRANT_PRIVILEGE","eventTime":"1420215472407","operationText":
  "GRANT SELECT ON DATABASE default TO ROLE analyst","allowed":"true",

"databaseName":"default","tableName":"","resourcePath":"","objectType":"PRINCIPA
L"}
2015-01-02 11:33:20,199 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"bob","impersonator":
  "impala/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.73",
  "operation":"CREATE_ROLE","eventTime":"1420216400199","operationText":
  "CREATE ROLE temp","allowed":"false","databaseName":null,"tableName":null,
  "resourcePath":null,"objectType":"ROLE"}
```

As you can see from the logs, the actual audit record is in JSON format. This makes for easy consumption by external log aggregation and management systems, which are important in larger enterprises.

Log Aggregation

Audit logs often span across many, if not all, nodes in the cluster. The sheer number of nodes multiplied by the individual audit log files generated can be a large undertaking to make sense of what is happening. It is typical, and highly recommended, to use some kind of log aggregation system to pull audit events from all of the nodes in the cluster into a central place for storage and analysis. There certainly are Hadoop-specific options out there that overlay additional intelligence as to what is going on the cluster. Even so, general-purpose log aggregation systems already in place in the enterprise can be a great way to manage Hadoop audit logs.

Another interesting option for log aggregation is to ingest them back into the Hadoop cluster for analysis. Security use cases for Hadoop are common and analyzing audit events from Hadoop fits the bill as well. As shown in this chapter, audit events are generally in a structured form and make for easy querying using SQL tools like Hive or Impala.

Summary

In this chapter, we took a look at several of the components in the Hadoop ecosystem and described the types of audit events that are recorded when users interact with the cluster. These log events are critical for accounting to ascertain what regular users are doing, but also to discover what unauthorized users are attempting to do. Although the Hadoop ecosystem does not have native alerting capabilities, the structure of the log events are conducive to allow additional tools to consume the events in a more general way. Active alerting is a newer capability that is still being worked on in the Hadoop ecosystem. Still, many general-purpose log aggregation tools possess the capabilities to alert when certain criteria are met, with many of these tools being common in the enterprise.

All of the auditing capabilities covered in this chapter wrap up the accounting portion of AAA. In the next part of the book, we will dive into how actual data, the lifeblood of Hadoop and big data, is secured.

Part III. Data Security
Chapter 9. Data Protection

By Eddie Garcia

So far, we have covered how Hadoop can be configured to enforce standard AAA controls. In this chapter, we will understand how these controls, along with the CIA principles discussed in Chapter 1, provide the foundation for protecting data. Data protection is a broad concept that involves topics ranging from data privacy to acceptable use. One of the topics we will specifically focus on is *encryption*.

Encryption is a common method to protect data. There are two primary flavors of data encryption: *data-at-rest encryption* and *data-in-transit encryption*, also referred to as *over-the-wire encryption*. Data at rest refers to data that is stored even after machines are powered off. This includes data on hard drives, flash drives, USB sticks, memory cards, CDs, DVDs, or even some old floppy drives or tapes in storage boxes. Data in transit, as its name implies, is data on the move, such as data traveling on the Internet, a USB cable, a coffee shop WiFi, cell phone towers, or from a remote space station to Earth.

Encryption Algorithms

Before diving into the two flavors of data encryption, we'll briefly discuss encryption algorithms. Encryption algorithms define the mathematical technique used to encrypt data. A common encryption algorithm is the *Advanced Encryption Standard*, or *AES*. It is a specification established by the U.S. National Institute of Standards and Technology (NIST) in FIPS-197.

Describing how AES encryption works is beyond the scope of this text, and we recommend Chapter 4 of *Understanding Cryptography* by Christof Paar and Jan Palzl (Springer, 2010). Other common

encryption algorithms include DES, RC4, Twofish, and Blowfish.

When encrypting data, key size is important. In general, the larger the key, the harder it is to crack. On the downside, encrypting data with larger keys is slower. When dealing with extremely small data, the encryption key should not be larger than the data itself.

When using AES, the commonly supported sizes are 128-bit, 192-bit, and 256-bit keys. The industry standard today is AES-256 (256-bit key) encryption, but history has shown that this can and will change. At one point, DES and triple DES (three rounds of DES) was the industry standard, but with today's computers both can be easily cracked with brute force.

Because of the performance overhead that encryption incurs, chip vendors created on-hardware functions to improve the performance of encryption. These enhancements can yield several orders of magnitude of improvement over software encryption. One popular hardware encryption technology is Intel's AES-NI.

With a basic understating of the encryption methods and algorithms, we can now dig a little deeper into two of the methods: full disk and filesystem encryption. These are much easier to implement because they do not require special hardware.

Encrypting Data at Rest

Let's say Alice places a message for Bob on a USB stick and hands it to him. But Bob somehow in his excitement misplaces the USB stick and Eve happens to find it. Eve, being curious, connects the USB and attempts to read the contents. Luckily, Alice encrypted the message; otherwise, it would have been an embarrassing situation for her. In this simple example, encryption has helped assure the confidentiality of the message. Nobody but Bob knows the password to decrypt the data.

At the core of the Hadoop ecosystem is HDFS, which is the filesystem for many other components. Until recently, encrypting data in HDFS was not natively supported, which means that other methods of encryption needed to be adopted.

Note

Over the years, there have been many cases of sensitive data breaches as a result of laptops and cell phones misplaced during transport, improper hard drive disposal, and physical hardware theft. Data-at-rest encryption helps mitigate these types of breaches because encryption makes it more difficult (but not impossible) to view the data.

In addition to native HDFS encryption, we will explore three other options, but we will not go into depth for every method because some are vendor specific. These methods work transparently below HDFS and thus don't require any Hadoop-specific configuration. All of these methods protect data in the case of a drive being physically removed from a drive array:

Encrypted drives

> This method is completely independent of the operating system. The physical drives on which HDFS stores its data support encryption natively. One limitation is that encrypted drives don't offer protection for data from rogue users and processes running on the system.

Full disk encryption

> This method typically works at system boot. This method does not require special drives or hardware like encrypted drives do. Several implementations of this technology exist, and typically vary by operating system. Some full disk encryption methods support operating system root partition encryption, while other methods only support encryption of the data partitions or volumes where the HDFS blocks are stored. Full disk encryption also has the limitation of encrypted drives in that it does not offer protection for data from rogue users and processes running on the system.

Filesystem encryption

> This method works at the operating-system level. This method also does not require special drives or hardware. Several implementations of this technology exist and vary by operating system. Filesystem encryption of the root partition is not supported because it becomes a chicken and the egg situation; the encrypted OS would need to boot to decrypt the OS. One of the benefits of filesystem encryption is that it offers protection for data against rogue users and processes running on the system. If an encrypted home directory is protected by a password known to a user and that user has not logged on to the system since boot, it would be impossible for a rogue user or process to gain access to the key to unlock the user's data, even as *root*.

Chapter 10. Securing Data Ingest

The preceding chapters have focused on securing Hadoop from a storage and data processing perspective. We've assumed that you have data in Hadoop and you want to secure access to it or to control how users share analytic resources, but we've neglected to explain how data gets into Hadoop in the first place.

There are many ways for data to be ingested into Hadoop. The simplest method is to copy files from a local filesystem (e.g., a local hard disk or an NFS mount) to HDFS using Hadoop's `put` command, as shown in Example 10-1.

Example 10-1. Ingesting files from the command line

```
[alice@hadoop01 ~]$ hdfs dfs -put /mnt/data/sea*.json /data/raw/sea_fire_911/
```

While this method might work for some datasets, it's much more common to ingest data from existing relational systems or set up flows of event- or log-oriented data. For these use cases, users use Sqoop and Flume, respectively.

Sqoop is designed to either pull data from a relational database into Hadoop or to push data from Hadoop into a remote database. In both cases, Sqoop launches a MapReduce job that does that actual data transfer. By default, Sqoop uses JDBC drivers to transport data between the map tasks and the database. This is called *generic mode* and it makes it easy to use Sqoop with new data stores, as the only requirement is the availability of JDBC drivers. For performance reasons, Sqoop also supports connectors that can use vendor-specific tools and interfaces to optimize the data transfer. To enable these optimizations, users specify the `--direct` option to enable *direct mode*.

For example, when enabling direct mode for MySQL, Sqoop will use the `mysqldump` and `mysqlimport` utilities to extract from or import to MySQL much more efficiently.

Flume is a distributed service for efficiently collecting, aggregating, and moving large volumes of event data. Users of Flume deploy agents, which are Java processes that transfer events. An event is the smallest unit of data that flows through Flume. Events have a binary (byte array) payload and an optional set of attributes defined by string key/value pairs. Each agent is configured with sources, sinks, and a channel. A source is a component that consumes events from an external data source; a source can either pull events from the external source or it can have events pushed to it by the external source. The Flume source connects to a channel, which makes the event available for consumption by a sink. The channel is completely passive in that it accepts events from sources and keeps them until they are consumed by a sink. A Flume sink transfers the event to an external data store or process.

Flume includes an AvroSource and an AvroSink that uses Avro RPC to transfer events. You can configure the AvroSink of one Flume agent to send events to the AvroSource of another Flume agent in order to build complex, distributed data flows. While Flume also supports a wide variety of sources and sinks, the primary ones used to implement inter-agent data flow are the AvroSource and AvroSink, so we'll restrict the rest of our discusion to this pair. The reliability of Flume is determined by the configuration of the channel. There are in-memory channels for data flows that prioritize speed over reliability, as well as disk-backed channels that support full recoverability. Figure 10-1 shows a two-tier Flume data flow showing the components internal to the agents as well as their interconnection.

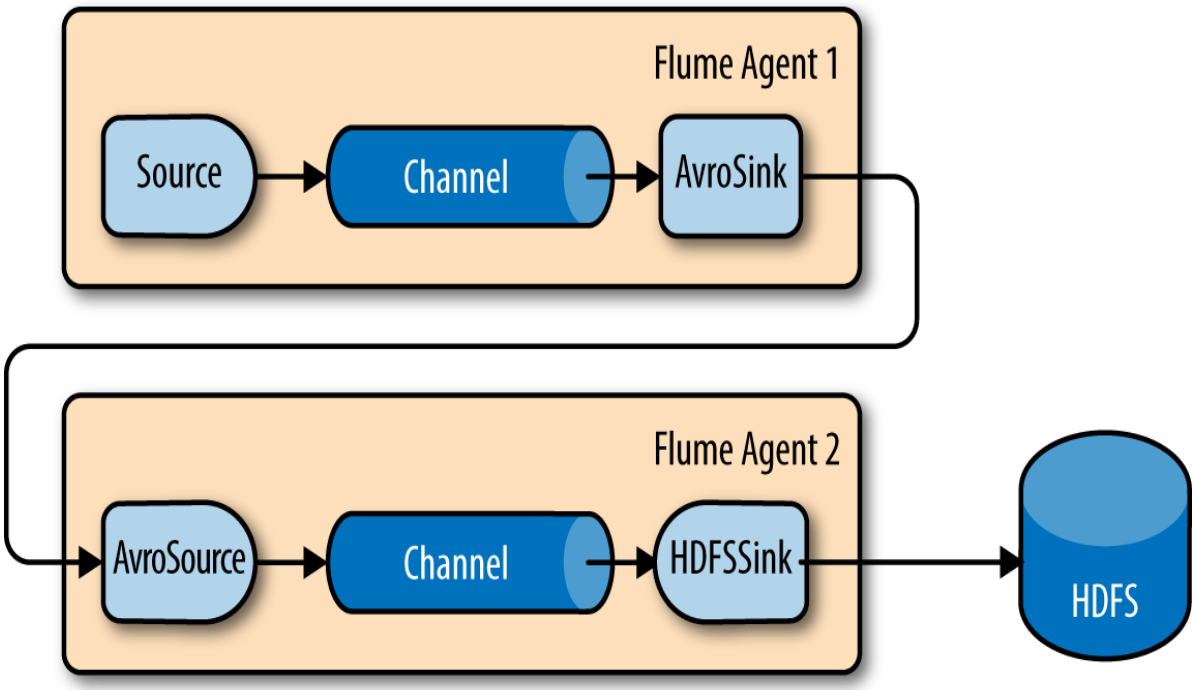

Figure 10-1. Flume architecture

Because Sqoop and Flume can be used to transfer sensitive data, it is important to consider the security implications of your ingest pipeline in the context of the overall deployment. In particular, you need to worry about the confidentiality, integrity, and availability (CIA) of your ingest pipeline.

Confidentiality refers to limiting access to the data to a set of authorized users. Systems typically guarantee confidentiality by a combination of authentication, authorization, and encryption. Integrity refers to how much you can trust that data hasn't been tampered with. Most systems employ checksums or signatures to verify the integrity of data. Availability refers to keeping information resources available. In the context of data ingest, it means that your ingestion system is robust against the loss of some capacity and that it has the ability to preserve in-transit data while it's dealing with the outage of some downstream system or service.

Ingest Workflows

So far, we've looked at how data is commonly ingested into a Hadoop environment and the different options for confidentiality, integrity, and availability for those ingest pipelines. However, ingesting data is typically part of an overall ETL process and additional considerations must be made in the context of the overall ETL flow.

One detail that we glossed over is *where* tools like Sqoop are launched from. Typically, you want to limit the interfaces that users have access to. As described in "Remote Access Controls", there are numerous ways that access to remote protocols can be secured, and the exact architecture will depend on your needs. The most common way of limiting access to edge services, including ingest, is to deploy services like Flume and Sqoop to *edge nodes*. Edge nodes are simply servers that have access to both the internal Hadoop cluster network and the outside world. Typical cluster deployments will lock down access to specific ports on specific hosts through the use of either host or network firewalls. In the context of data ingest, we can restrict the ability to *push* data to a Hadoop cluster through the edge nodes while still deploying *pull*-based mechanisms, such as Sqoop, to perform parallel ingest without opening up access to sensitive Hadoop services to the world.

Limiting remote login capabilities to only edge nodes goes a long way toward mitigating the risk of having users be physically logged into a Hadoop cluster. It allows you to concentrate your monitoring and security auditing while at the same time reducing the population of potential bad actors. When building production data flows, it's common to set up dedicated ETL accounts or groups that will execute the overall workflow. For organizations that require detailed auditing, it's recommended that actions be initiated by individual user accounts to better track activity back to a person.

In addition to Flume and Sqoop, edge nodes may run proxy services or other remote user protocols. For example, HDFS supports a proxy server called HttpFS, which exposes a read/write REST interface for HDFS. Just as with HDFS itself, HttpFS fully supports Kerberos-based authentication, and the same authorization controls built into HDFS apply when it's accessed through HttpFS. Running HttpFS on an edge node can be useful for allowing limited access to the data stored in HDFS and can even be used for certain data ingest use cases.

Another common edge node service is Oozie. Ooze is a workflow execution and scheduling tool. Complex workflows that combine Sqoop jobs, Hive queries, Pig scripts, and MapReduce jobs can be composed into single units and can be reliably executed and scheduled using Oozie. Oozie also provides a REST interface that supports Kerberos-based authentication and can be safely exposed to

an edge node.

For some use cases, it is necessary to stage files on an edge node before they are pushed into HDFS, HBase, or Accumulo. When creating these local disk (or sometimes NFS-mounted) staging directories, it is important to use your standard operating system controls to limit access to only those users authorized to access the data. Again, it's useful to define one or more ETL groups and limit the access to raw data to these relatively trusted groups.

Enterprise Architecture

This discussion of data ingest helps to illustrate a useful point: Hadoop is never deployed in a vacuum. Hadoop necessarily integrates with your existing and evolving enterprise architecture. This means that you can't consider Hadoop security on its own. When deciding how to secure your cluster, you must look at the requirements that already apply to your data and systems. These requirements will be driven by enterprise security standards, threat models, and specific dataset sensitivity. In particular, it doesn't make sense to lock down a Hadoop cluster or the data ingest pipeline that feeds the cluster if the source of the data has no security wrapped around it.

This is no different than when data warehousing systems were introduced to the enterprise. In a typical deployment, applications are tightly coupled with the transactional systems that back them. This makes security integration straightforward because access to the backend database is typically limited to the application that is generating and serving the data. Some attention to security detail gets introduced as soon as that transactional data is important enough to back up. However, this is still a relatively easy integration to make, because the backups can be restricted to trusted administrators that already have access to the source systems.

Where things get interesting is when you want to move data from these transactional systems into analytic data warehouses so that analysis can be performed independently of the application. Using a traditional data warehouse system, you would compare the security configuration of the transactional database with the features of the new data warehouse. This works fine for securing that data once it's in the warehouse and you can apply the same analysis to the database-based authorization features available in Sentry. However, care must be taken in how the data is handled between the transactional system and the analysis platform.

With these traditional systems, this comes down to securing the ETL grid that is used to load data into the data warehouse. It's clear that the same considerations that you make to your ETL grid would apply to the ingest pipeline of a Hadoop cluster. In particular, you have to consider when and where encryption was necessary to protect the confidentiality of data. You need to pay close attention to how to maintain the integrity of your data. This is especially true of traditional ETL grids that might not have enough storage capacity to maintain raw data after it has been transformed. And lastly, you care about the availability of the ETL grid to make sure that it does not impact the ability of the source systems or data warehouse to meet the requirements of their users. This is exactly the same process we went through in our discussion of data ingest into Hadoop in general, and with Flume and Sqoop in particular.

Again, this works in both directions. It doesn't make sense to apply security controls to Hadoop at

ingest or query time that are not maintained in source systems, just as it doesn't make sense to leave Hadoop wide open after careful work has gone into designing the security protections of your existing transactional or analytic tools. The perfect time to consider all of these factors is when you're designing your ingest pipeline. Because that is where Hadoop will integrate with the rest of your enterprise architecture, it's the perfect time to compare security and threat models and to carefully consider the security architecture of your overall Hadoop deployment.

Summary

In this chapter, we focused on the movement of data from external sources to Hadoop. After briefly talking about batch file ingest, we moved on to focus on the ingestion of event-based data with Flume, and the ingestion of data sourced from relational databases using Sqoop. What we found is that these common mechanisms for ingest have the ability to protect the integrity of data in transit. A key takeaway from this chapter is that the protection of data inside the cluster needs to be extended all the way to the source of ingest. This mode of protection should match the level in place at the source systems.

Now that we have covered protection of both data ingestion and data inside the cluster, we can move on to the final topic of data protection, which is to secure data extraction and client access.

[1] To make setup of dm-crypt/LUKS easier, you can use the cryptsetup tool. Instructions for setting up dm-crypt/LUKS using cryptsetup are available on the cryptsetup FAQ page.

[2] The Sqoop examples are based on the *Apache Sqoop Cookbook* by Kathleen Ting and Jarek Jarcec Cecho (O'Reilly). The example files and scripts used are available from the *Apache Sqoop Cookbook* project page.

[3] If SSL has not yet been configured for MySQL, you can follow the instructions in the MySQL manual.

Chapter 12. Cloudera Hue

Hue is a web application that provides an end-user focused interface for a large number of the projects in the Hadoop ecosystem. When Hadoop is configured with Kerberos authentication, then Hue must be configured with Kerberos credentials to properly access Hadoop. Kerberos is enabled by setting the following parameters in the *hue.ini* file:

`hue_principal`

: The Kerberos principal name for the Hue, including the fully qualified domain name of the Hue server

`hue_keytab`

: The path to the Kerberos keytab file containing Hue's service credentials

`kinit_path`

: The path to the Kerberos `kinit` command (not needed if `kinit` is on the path)

`reinit_frequency`

: The frequency in seconds for Hue to renew its Kerberos tickets

These settings should be placed under the `[[kerberos]]` subsection of the `[desktop]` top-level section in the *hue.ini* file. See Example 12-1 for a sample Hue kerberos configuration.

Example 12-1. Configuring Kerberos in Hue

```
[desktop]
[[kerberos]]
hue_principal=hue/hue.example.com@EXAMPLE.COM
hue_keytab=/etc/hue/conf/hue.keytab
reinit_frequency=3600
```

Hue has its own set of authentication backends and authenticates against Hadoop and other projects using Kerberos. In order to perform actions on behalf of other users, Hadoop must be configured to trust the Hue service. This is done by configuring Hadoop's proxy user/user impersonation capabilities. This is controlled by setting the hosts Hue can run on and the groups of users that Hue can impersonate. Either value can be set to * to indicate that impersonation is enabled from all hosts or from all groups, respectively. Example 12-2 shows how to enable Hue to impersonate users when accessing Hadoop from the host *hue.example.com* and for users in the `hadoop-users` group.

Example 12-2. Configuring Hue User Impersonation for Hadoop in core-site.xml

```
  <property>
    <name>hadoop.proxyuser.hue.hosts</name>
    <value>hue.example.com</value>
  </property>
  <property>
    <name>hadoop.proxyuser.hue.groups</name>
    <value>hadoop-users</value>
  </property>
```

HBase and Hive use the Hadoop impersonation configuration, but Oozie must be configured independently. If you want to use Oozie from Hue, you must set the `oozie.service.ProxyUserService.proxyuser.<user>.hosts` and `oozie.service.ProxyUserService.proxyuser.<user>.groups` properties in the *oozie-site.xml* file. Example 12-3 shows how to enable Hue to impersonate users when accessing Oozie from the host `hue.example.com` and for users in the *hadoop-users* group.

Example 12-3. Configuring Hue user impersonation for Oozie in oozie-site.xml

```
  <property>
    <name>oozie.service.ProxyUserService.proxyuser.hue.hosts</name>
    <value>hue.example.com</value>
  </property>
  <property>
    <name>oozie.service.ProxyUserService.proxyuser.hue.groups</name>
    <value>hadoop-users</value>
  </property>
```

If you're using the Hue search application, you also need to enable impersonation in Solr. This is done by setting the `SOLR_SECURITY_ALLOWED_PROXYUSERS`, `SOLR_SECURITY_PROXYUSER_<user>_HOSTS`, and `SOLR_SECURITY_PROXYUSER_<user>_GROUPS` environment variables in the */etc/default/solr* file. See Example 12-4 for a sample configuration to enable impersonation from the host `hue.example.com` and for users in the *hadoop-users* group.

Example 12-4. Configuring Hue user impersonation for Solr in /etc/default/solr

```
SOLR_SECURITY_ALLOWED_PROXYUSERS=hue
SOLR_SECURITY_PROXYUSER_hue_HOSTS=hue.example.com
SOLR_SECURITY_PROXYUSER_hue_GROUPS=hadoop-users
```

Hue HTTPS

By default, Hue runs over plain old HTTP. This is suitable for proofs of concept or for environments where the network between clients and Hue is fully trusted. However, for most environments it's strongly recommended that you configure Hue to use HTTPS. This is especially important if you don't fully trust the network between clients and Hue, as most of Hue's authentication backends support entering in a username and password through a browser form.

Fortunately, Hue makes configuring HTTPS easy. To do so, you simply configure the `ssl_certificate` and `ssl_private_key` settings, which are both under the `desktop` section of the *hue.ini* file. Both files should be in PEM format and the private key cannot be encrypted with a passphrase. See Example 12-5 for a sample configuration.

Example 12-5. Configuring Hue to use HTTPS

```
[desktop]
ssl_certificate=/etc/hue/conf/hue.crt
ssl_private_key=/etc/hue/conf/hue.pem
```

Warning

Hue does not currently support using a private key that is protected with a passphrase. This means it's very important that Hue's private key be protected to the greatest extent possible. Ensure that the key is owned by the `hue` user and is only readable by its owner (e.g., `chmod 400 /etc/hue/conf/hue.pem`). You might also configure filesystem-level encryption on the filesystem, storing the private key as described in "Filesystem Encryption". In cases where Hue is on a server that has other resources protected by TLS/SSL, it's strongly recommended that you issue a unique certificate just for Hue. This will lower the risk if Hue's private key is compromised by protecting other services running on the same machine.

Hue Authentication

Hue has a pluggable authentication framework and ships a number of useful authentication backends. The default authentication backend uses a private list of usernames and passwords stored in Hue's backing database. The backend is configured by setting the `backend` property to `desktop.auth.backend.AllowFirstUserDjangoBackend` under the `[[auth]]` subsection of the `[desktop]` section. See Example 12-6 for a sample *hue.ini* file where the

backend is explicitly set. Because this is the default, you can also leave this setting out entirely.

Example 12-6. Configuring the default Hue authentication backend

```
[desktop]
[[auth]]
backend=desktop.auth.backend.AllowFirstUserDjangoBackend
```

Hue also has support for using Kerberos/SPNEGO, LDAP, PAM, and SAML for authentication. We won't cover all of the options here, so refer to the `config_help` command for more information.[1]

SPNEGO Backend

Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO)[2] is a GSSAPI pseudo-mechanism for allowing clients and servers to negotiate the choice of authentication technology. SPNEGO is used any time a client wants to authenticate with a remote server but neither the client nor the server knows in advance the authentication protocols the other supports. The most common use of SPNEGO is with the HTTP negotiate protocol first proposed by Microsoft.[3]

Hue only supports SPNEGO with Kerberos V5 as the underlying mechanism. In particular this means you can't use Hue with the *Microsoft NT LAN Manager* (NTLM) protocol. Configuring SPNEGO with Hue requires setting the Hue authentication backend to SpnegoDjangoBackend (see Example 12-7), as well as setting the `KRB5_KTNAME` environment variable to the location of a keytab file that has the key for the `HTTP/<fully qualified domain name>@<REALM>` principal. If you're starting Hue by hand on the server `hue.example.com` and your keytab is located in */etc/hue/conf/hue.keytab*, then you'd start Hue as shown in Example 12-8.

Example 12-7. Configuring the SPNEGO Hue authentication backend

```
[desktop]
[[auth]]
backend=desktop.auth.backend.SpnegoDjangoBackend
```

Example 12-8. Setting KRB5_KTNAME and starting Hue manually

```
[hue@hue ~]$ export KRB5_KTNAME=/etc/hue/conf/hue.keytab
[hue@hue ~]$ ${HUE_HOME}/build/env/bin/supervisor
```

In order to use SPNEGO, you also need to have a TGT on your desktop (e.g., by running `kinit`) and you need to use a browser that supports SPNEGO. Internet Explorer and Safari both support SPNEGO without additional configuration. If you're using Firefox, you first must add the server or domain name you're authenticating against to the list of trusted URIs. This is done by typing `about:config` in the URL bar, then searching for `network.negotiate-auth.trusted-uris`, and then updating that preference to include the server name or domain name. For example, if you wanted to support SPNEGO with any server on the `example.com` domain, you would set `network.negotiate-auth.trusted-uris=example.com`. If you see the message `401 Unauthorized` while trying to connect to Hue, you likely don't have your trusted URIs configured correctly in Firefox.

Chapter 13. Case Studies

In this chapter, we present two case studies that cover many of the security topics in the book. First, we'll take a look at how Sentry can be used to control SQL access to data in a multitenancy environment. This will serve as a good warmup before we dive into a more detailed case study that shows a custom HBase application in action with various security features in place.

Case Study: Hadoop Data Warehouse

One of the key benefits of big data and Hadoop is the notion that many different and disparate datasets can be brought together to solve unique problems. What comes along with this are different types of users that span multiple lines of business. In this case study, we will take a look at how Sentry can be used to provide strong authorization of data in Hive and Impala in an environment consisting of multiple lines of business, multiple data owners, and different analysts.

First, let's list the assumptions we are making for this case study:

- The environment consists of three lines of business, which we will call `lob1`, `lob2`, and `lob3`
- Each line of business has analysts and administrators
    - The analysts are defined by the groups `lob1grp`, `lob2grp`, and `lob3grp`
    - The administrators are defined by the groups `lob1adm`, `lob2adm`, and `lob3adm`
    - Administrators are also in the analysts groups
- Each line of business needs to have its own sandbox area in HDFS to do ad hoc analysis, as well as to upload self-service data sources
- Each line of business has its own administrators that control access to their respective sandboxes
- Data inside the Hive warehouse is IT-managed, meaning only noninteractive ETL users add data
- Only Hive administrators create new objects in the Hive warehouse
- The Hive warehouse uses the default HDFS location */user/hive/warehouse*
- Kerberos has already been set up for the cluster
- Sentry has already been set up in the environment
- HDFS already has extended ACLs enabled
- The default umask for HDFS is set to 007

Environment Setup

Now that we have the basic assumptions, we need to set up the necessary directories in HDFS and prepare them for Sentry. The first thing we will do is lock down the Hive warehouse directory. HiveServer2 impersonation is disabled when enabling Sentry, so only the `hive` group should have access (which includes the `hive` and `impala` users). Here's what we need to do:

```
[root@server1 ~]# kinit hive
Password for hive@EXAMPLE.COM:
[root@server1 ~]# hdfs dfs -chmod -R 0771 /user/hive/warehouse
[root@server1 ~]# hdfs dfs -chown -R hive:hive /user/hive/warehouse
[root@server1 ~]#
```

As mentioned in the assumptions, each line of business needs a sandbox area. We will create the path */data/sandbox* as the root directory for all the sandboxes, and create the associated structures within it:

```
[root@server1 ~]# kinit hdfs
Password for hdfs@EC2.INTERNAL:
[root@server1 ~]# hdfs dfs -mkdir /data
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -chgrp lob1grp /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -chgrp lob2grp /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -chgrp lob3grp /data/sandbox/lob3
[root@server1 ~]#
```

Now that the basic directory structure is set up, we need to start thinking about what is needed to support Hive and Impala access to the sandbox. After all, these sandboxes are the place where all the users will be doing their ad hoc analytic work. Both the `hive` and `impala` users need access to these directories, so let's go ahead and set up HDFS-extended ACLs to allow the `hive` group full access:

```
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob3
[root@server1 ~]#
```

Warning

Remember, the default ACL is only applicable to directories, and it only dictates the ACLs that are copied to new subdirectories and files. Because of this fact, the parent directories still need a regular access ACL.

The next part we need to do is to make sure that regardless of who creates new files, all the intended accesses persist. If we left the permissions as they are right now, new directories and files created by the `hive` or `impala` users may actually be accessible by the analysts and administrators in the line of business. To fix that, let's go ahead and add those groups to the extended ACLs: