Table of Contents

Hadoop and Kerberos: The Madness beyond the Gate

> The most merciful thing in the world, I think, is the inability of the human mind to correlate all its contents. We live on a placid island of ignorance in the midst of black seas of infinity, and it was not meant that we should voyage far. The sciences, each straining in its own direction, have hitherto harmed us little; but some day the piecing together of dissociated knowledge will open up such terrifying vistas of reality, and of our frightful position therein, that we shall either go mad from the revelation or flee from the light into the peace and safety of a new dark age.
>
> *The Call of Cthulhu, HP Lovecraft, 1926.*

This manuscript discusses low-level issues related to Apache™ Hadoop® and Kerberos

Disclaimer

Just as the infamous Necronomicon is a collection of notes scrawled in blood as a warning to others, this book is

1. Incomplete.
2. Based on experience and superstition, rather than understanding and insight.
3. Contains information that will drive the reader insane.

Reading this book implies recognition of these facts and that the reader, their estate and their heirs accept all risk and liability. The author is not responsible if anything happens to their Apache Hadoop cluster, including all the data stored in HDFS disappearing into an unknown dimension, or the YARN scheduler starting to summon pre-human deities.

You have been warned

Implementation notes.

1. This is a work in progress book designed to built using the gitbook tool chain.
2. It is hosted on github. Pull requests are welcome.
3. All the content is Apache licensed.
4. This is not a formal support channel for Hadoop + Kerberos problems. If you have a support contract with Hortonworks then issues related to Kerberos may eventually reach the author. Otherwise: try
    - Hortonworks Answerhub
    - The users mailing list of Apache Hadoop, the application and you are using on top of it.

- Stack Overflow.

Hadoop and Kerberos: The Madness Beyond the Gate

Authors:

S.A. Loughran

Introduction

When HP Lovecraft wrote his books about forbidden knowledge which would reduce the reader to insanity, of "Elder Gods" to whom all of humanity were a passing inconvenience, most people assumed that he was making up a fantasy world. In fact he was documenting Kerberos.

What is remarkable is that he did this fifty years before kerberos was developed. This makes him less of an author, instead: a prophet.

What he wrote was true: there are some things humanity was not meant to know. Most people are better off living lives of naive innocence, never having to see an error message about SASL or GSS, never fear building up scripts of incantations to `kadmin.local`, incantations which you hope to keep evil and chaos away. To never stare in dismay at the code whose true name must never be spoken, but instead it's initials whispered, "UGI". For those of us who have done all this, our lives are forever ruined. From now on we will cherish any interaction with a secure Hadoop cluster —from a client application to HDFS, or application launch on a YARN cluster, and simply viewing a web page in a locked down web UI —all as a miracle against the odds, against the forces of chaos struggling to destroy order. And forever more, we shall fear those voices calling out to us in the night, the machines by our bed talking to us, saying things like "we have an urgent support call related to REST clients on a remote kerberos cluster —can you help?"

HP Lovecraft	Kerberos
Evil things lurking in New England towns and villages	MIT Project Athena
Ancient, evil deities oblivious to humanity	Kerberos Domain Controller
Books whose reading will drive the reader insane	IETF RFC 4120
Entities which are never spoken of aloud	UserGroupInformation
People driven insane by their knowledge	You

This documents contains the notes from previous people who have delved too deep into the mysteries of Apache™ Hadoop® and Kerberos, who have read the forbidden source code, maybe who have even contributed to it. If you wish to preserve your innocence, to view the world as a place of happiness: stop now.

Disclaimer

This document is a collection of notes based on the experience of the author. There are no guarantees that any of the information contained within was correct at the time of writing, let alone the time of reading. The author does not accept any responsibility for actions made on the basis of the information contained herein, be it correct or incorrect.

The reader of this document is likely to leave with some basic realisation that Kerberos, while important, is an uncontrolled force of suffering and devastation. The author does not accept any responsibility for the consequences of such knowledge.

What has been learned cannot be unlearned(*)

(*) Except for Kerberos workarounds you wrote 18 months ago and for which you now field support calls.

Foundational Concepts

What is the problem that Hadoop security is trying to address? Securing Hadoop.

Apache Hadoop is "an OS for data". A Hadoop cluster can rapidly become the largest stores of data in an organisation. That data can explicitly include sensitive information: financial, personal, business, and can often implicitly contain data which needs to be sensitive about the privacy of individuals (for example, log data of web accesses alone). Much of this data is protected by laws of different countries. This means that access to the data needs to be strictly controlled, and accesses made of that data potentially logged to provide an audit trail of use.

You have to also consider, "why do people have Hadoop clusters?".

It's not just because they have lots of data --its because they want to make use of it. A data-driven organisation needs to trust that data, or at least be confident of its origins. Allowing entities to tamper with that data is dangerous.

For the protection of data, then, read and write access to data stored directly in the HDFS filesystem needs to be protected. Applications which work with their data in HDFS also need to have their accesses restricted: Apache HBase and Apache Accumulo store their data in HDFS, Apache Hive submits SQL queries to HDFS-stored data, etc. All these accesses need to be secured; applications like HBase and Accumulo granted restricted access to their data, and themselves securing and authenticating communications with their clients.

YARN allows arbitrary applications to be deployed within a Hadoop cluster. This needs to be done without granting open access to the entire cluster from those user-launched applications, while isolating different users' work. A YARN application started by user Alice should not be able to directly manipulate an application launched by user "Bob", even if they are running on the same host. This means that not only do they need to run as different users on the same host (or in some

isolated virtual/container), the applications written by Alice and Bob themselves need to be secure. In particular, any web UI or IPC service they instantiate needs to have its access restricted to trusted users. here Alice and Bob

Authentication

The authentication problem: who is a caller identifying themselves as —and can you verify that they really are this person.

In an unsecure cluster, all callers to HDFS, YARN and other services are trusted to be who they say they are. In a secure cluster, services need to authenticate callers. That means some information must be passed with remote IPC/REST calls to declare a caller's identity and authenticate that identity

Authorization

Does an (authenticated) user have the permissions to perform the desired request?

This isn't handled by Keberos: this is Hadoop-side, and is generally done in various ways across systems. HDFS has file and directory permissions, with the user+group model now extended to ACLs. YARN allows job queues to be restricted to different users and groups, so restricting the memory & CPU limits of those users. When cluster node labels are used to differentiate parts of the cluster (e.g. servers with more RAM, GPUs or other features), then the queues can be used to restrict access to specific sets of nodes.

Similarly, HBase and Accumulo have their users and permissions, while Hive uses the permissions of the source files as its primary access control mechanism.

These various mechanisms are all a bit disjoint, hence the emergence of tools to work across the entire stack for a unified view, Apache Ranger being one example.

Encryption

Can data be intercepted on disk or over the wire?

Encrytion of Persistent Data.

HDFS now supports *at rest encryption*; the data is encrypted while stored on disk.

Before rushing to encrypt all the data, consider that it isn't a magic solution to security: the authentication and authorisation comes first. Encryption adds a new problem, secure key management, as well as the inevitable performance overhead. It also complicates some aspects of HDFS use.

Data stored in HDFS by applications is implicitly encrypted. However, applications like Hive have had to be reworked to ensure that when making queries across encrypted datasets, temporary data files are also stored in the same encryption zone, to stop the intermediate data being stored unencrypted. And of course, analytics code running in the servers may also intentionally or unintentionally persist the sensitive data in an unencrypted form: the local filesystem, OS swap space and even OS hibernate-time memory snapshots need to be managed.

Before rushing to enable persistent data encryption, then, you need to consider: what is the goal here?

What at-REST encryption does deliver is better guarantees that data stored in hard disks is not recoverable —at least on the HDFS side. However, as OS-level data can persist, (strongly) wiping HDDs prior to disposal is still going to be necessary to guarantee destruction of the data.

Auditing and Governance

Authenticated and Authorized users should not just be able to perform actions or read and write data —this should all be logged in *Audit Logs* so that if there is ever a need to see which files a user accessed, or what individual made specific requests of a service —that information is available. Audit logs should be

1. Separate log categories from normal processing logs, so log configurations can store them in separate locations, with different persistence policies.
2. Machine Parseable. This allows the audit logs themselves to be analyzed. This does not just have to be for security reasons; Spotify have disclosed that they run analysis over their HDFS audit logs to identify which files are most popular (and hence should have their replication factor increased), and which do not get used more then 7 days after their creation —and hence can be automatically deleted as part of a workflow.

What is Kerberos?

> Yog-Sothoth knows the gate. Yog-Sothoth is the gate. Yog-Sothoth is the key and guardian of the gate. Past, present, future, all are one in Yog-Sothoth. He knows where the Old Ones broke through of old, and where They shall break through again. He knows where They have trod earth's fields, and where They still tread them, and why no one can behold Them as They tread.
>
> *"The Dunwich Horror", HP Lovecraft, 1928*

Kerberos is a system for authenticating access to distributed services:

1. that callers to a service represent a `principal` in the system, or
2. That a caller to a service has been granted the right to act on behalf of a principal —a right which the principal can grant for a limited amount of time.

In Hadoop, feature #2 is key: a user or a process may *delegate* the authority to another process, which can then talk to the desired service with the delegated authority. These delegation rights are both limited in scope --- the principal delegates authority on a service-by-service basis --- and in time. The latter is for security reasons ---it guarantees that if the secret used to act as a delegate, the *token*, is stolen, there is only a finite time for which it can be used.

How does it work? That is beyond the scope of this book and its author.

It is covered in detail in [Coluris01], S7.6.2. However, anyone attempting to read this will generally come out with at least a light headache and no better informed.

For an approximate summary of the concepts

Kerberos Domain Controller, *the KDC*

The KDC is the gate, it is is the key and guardian of the gate, it is the gateway to the madness that is Kerberos.

Every Kerberos Realm needs at least one. There's one for Linux and Active Directory can act as a federated KDC infrastructure. Hadoop cluster management tools often aid in setting up a KDC for a Hadoop cluster. There's even a minature one, the `MiniKDC` in the Hadoop source for testing.

KDCs are managed by operations teams. If a developer finds themselves maintaining a KDC outside of a test environment, they are in trouble and probably out of their depth.

Kerberos Principal

A principal is an identity in the system; a person or a thing like the hadoop namenode which has been given an identity.

In Hadoop, a different principal is usually created for each service and machine in the cluster, such as `hdfs/node1`, `hdfs/node2`, ... etc. These principals would then be used for all HDFS daemons running on node1, node2, etc.

It's possible to shortcut this and skip the machine specific principal, downgrading to one per service, such as `hdfs`, `yarn`, `hbase` —or even one for all hadoop applications, such as `hadoop`. This can be done on a small cluster, but doesn't scale well, or makes working out WTF is going on difficult.

In particular, the bits of Kerberos which handle logins, the Kerberos Domain Controllers, treat repeated attempts to log in as the same principal within a short period of time as some attack on the system, such as a replay or key guessing attack. The requests are all automatically rejected (presumably without any validation, so as to reduce CPU load on the server). Even a small Hadoop cluster could generate enough authentication requests on a cluster restart for this to happen —hence a different principal for every service on every node.

How do the Hadoop services know which principal to identify themselves at? Generally, though Hadoop configuration files. They also determine the hostname, and use this to decide which of the possible principals in their keytab (see below) to identify themselves at. For this to work, machines have to know who they are.

Specifically

1. They have to have a name
2. That name has to be in their host table or DNS
3. It has to match the IP address of the host

what if there is more than one IP address on the host?

Generally Hadoop services are single-IP address, which is a limitation that is likely to be addressed at some time. So who knows. Actually, it comes from `org.apache.hadoop.net.NetUtils.getHostname()`, which invokes `InetAddress.getLocalHost()` and relies on this to return a hostname. It's because of this need to know hostnames for principals, that requests for Hadoop services to use IP Addresses over hostnames, such as MAPREDUCE-6463 are declined. It also means that if a machine does not know its own hostname, things do not work HADOOP-3426, HADOOP-10011.

Kerberos Realm

A Kerberos *Realm* is the security equivalent of a subnet: all principals live in a realm. It is conventional, though not mandatory, to use capital letters and a single name, rather than a dotted network address. Examples: `ENTERPRISE`, `HCLUSTER`

Kerberos allows different realms to have some form of trust of others. This would allow a Hadoop cluster with its own KDC and realm to trust the `ENTERPRISE` realm, but for the enterprise realm to not trust the HCLUSTER realm, and hence all its principals. This would prevent a principal

`hdfs/node1@HCLUSTER` from having access to the `ENTERPRISE` systems. While this is a bit tricky to set up, it means that keytabs created for the Hadoop cluster (see below) are only a security risk for the Hadoop cluster and all data kept in/processed by it, rather than the entire organisation.

Kerberos login, `kinit`

The command line program `kinit` is how a user authenticates with a KDC on a unix system; it uses the information stored in `/etc/krb`

Alongside `kinit`, comes `kdestroy`, to destroy credentials/log out, and `klist` to list the current status. The `kdestroy` command is invaluable if you want to verify that any program you start on the command line really is reading in and using keytab.

Here's what a full `klist -v` listing looks like

```
$ klist -v
Credentials cache: API:489E6666-45D0-4F04-9A1D-FCD5D48EEA07
        Principal: stevel@COTHAM
    Cache version: 0

Server: krbtgt/COTHAM@COTHAM
Client: stevel@COTHAM
Ticket etype: aes256-cts-hmac-sha1-96, kvno 1
Ticket length: 326
Auth time:  Sep  2 11:52:02 2015
End time:   Sep  3 11:52:01 2015
Renew till: Sep  2 11:52:02 2015
Ticket flags: enc-pa-rep, initial, renewable, forwardable
Addresses: addressless

Server: HTTP/devix.cotham.uk@COTHAM
Client: stevel@COTHAM
Ticket etype: aes256-cts-hmac-sha1-96, kvno 25
Ticket length: 333
Auth time:  Sep  2 11:52:02 2015
Start time: Sep  2 12:20:00 2015
End time:   Sep  3 11:52:01 2015
Ticket flags: enc-pa-rep, transited-policy-checked, forwardable
Addresses: addressless
```

A shorter summary comes from the basic `klist`

```
$ klist
Credentials cache: API:489E6666-45D0-4F04-9A1D-FCD5D48EEA07
        Principal: stevel@COTHAM

  Issued                Expires               Principal
Sep  2 11:52:02 2015  Sep  3 11:52:01 2015  krbtgt/COTHAM@COTHAM
Sep  2 12:20:00 2015  Sep  3 11:52:01 2015  HTTP/devix.cotham.uk@COTHAM
```

This shows that

1. The user is logged in as `stevel@COTHAM`

2. They have a ticket to work with the ticket granting service, `krbtgt/COTHAM@COTHAM`.
3. They have a ticket to authenticate with the principal `HTTP/devix.cotham.uk@COTHAM`. This is used by some HTTP services running on the host (`devix.cotham.uk`), specifically the Hadoop Namenode and Resource Manager web pages. These have both been configured to require Kerberos authentication via SPNEGO, and to use `HTTP` as the user. The full principal `HTTP/devix.cotham.uk` is determined from the host running the service.

Keytab

A (binary) file containing the secrets needed to log in as a principal

1. It contains all the information to log in as a principal, so is a sensitive file.
2. It can hold many principals, so one can be created for, say, hdfs, which contails all its principals, `hdfs/node1@HCLUSTER` , `hdfs/node2@HCLUSTER` , ...etc. Thus only one keytab per service is needed.
3. It is created by the KDC administrators, who must then securely propagate that file to where it can be used.

Keytabs are the only way in which programs can directly authenticate themselves with Kerberos, (though they can indirectly do this with credentials passed to them). This means that for any long-lived process, a keytab is needed.

Operations teams are generally very reluctant to provide keytabs. They will need to create them for all long-lived services which run in the cluster. For services such as HDFS and YARN this is generally done at cluster setup time. YARN services have to deal with this problem whenever a user wants to run a long lived *YARN service* within the cluster: the technical one of keytab management and the organisational one of getting the keytab in the first place.

To look at and work with keytabs, the `ktutil` command line program is the tool of choice.

Tickets

Kerberos is built around the notion of *tickets*.

A ticket is something which can be passed to a server to identify that the caller and to provide a secret key that can be used between the client an the server —for the duration of the ticket's lifetime. It is all that a server needs to authenticate a client: there's no need for the server to talk to the KDC.

What's important is that tickets can be passed on: an authenticated principal can obtain a ticket to a service, and pass that on to another process in the distributed system. The recipient can then issue requests on behalf of the original principal, using that ticket. That recipient only has the permissions granted to the ticket (it does not have any other permissions of the principal, unless those tickets are also provided), and those permissions are only valid for as long as the ticket is valid.

The limited lifetime of tickets ensures that even if a ticket is captured by a malicious attacker, they can only make use of the credential for the lifetime of the ticket. The ops team doesn't need to worry about lost/stolen tickets, to have a process for revoking them, as they expire within a short time period, usually a couple of days.

This notion of tickets starts right at the bottom of Kerberos. When a principal authenticates with the KDC, it doesn't get any special authentication secrets —it gets a ticket to the *Ticket Granting Service*. This ticket can then be used to get tickets to other services —and, like any other ticket, can be forwarded. Equally importantly, the ticket will expire —forcing the principal to re-authenticate via the command line or a keytab.

Kerberos Authentication Service

This is network-accessible service which runs in the KDC, and which is used to authenticate callers. The protocol to authenticate callers is one of those low level details found in text books. What is important to know is that

1. The KDC contains 'a secret' shared with the principal. There is no public/private key system here, just a shared secret.
2. When a client authenticates on the command line, the password is (somehow) used to generate some data which is passed to the authentication service to show that at least during the authentication process, the client had the password for the principal they were trying to authenticate as. (i.e., part of the process includes a challenge issued by the KDC, a challenged hashed by the password to show that's in the callers' possession).
3. When a client authenticates via the keytab, a similar challenge-reponse operation takes place to allow the client to show they have the principal's (secret) data in that keytab.
4. When the KDC's secret key for a principal is changed, all existing keytabs stop working.

Ticket Granting Service, *TGS*

1. A *Kerberos Domain Controller*, *KDC* exists on the network to manage Kerberos security
2. It contains an *Authentication Service*, which authenticates remote principals, and a *Ticket Granting Service*, *TGS*, which grants access to specific services.
3. The Authentication Service can authenticate via a password-based login, or though the principal having a stored copy of a shared secret, a *key*.
4. The TGS can issue *tickets*, secrets which declare that a caller has duration-limited access to a requested service, with the rights of the authenticated principal.
5. An authenticated principal can request tickets to services, which they can then use to authenticate directly with those services, and interact with them until the ticket expires.
6. A principal can also forward a ticket to any other process/service within the distributed system, to *delegate* rights.
7. This delegate can use the ticket to access the service, with the identity of the principal, for the lifespan of that ticket.

Note that Hadoop goes beyond this with the notion of *delegation tokens*, secrets which are similar to *tickets*, but which can be issued and renewed directly by Hadoop services. That will be covered in a later chapter.

Kerberos User Login

A user logs in with the Kerberos Authentication Service

Examples of Kerberos

To put things into the context of Hadoop, here are some examples of how it could be used.

Example: User listing an HDFS directory

A user wishes to submit some work to a Hadoop cluster, a new YARN application.

First, they must be logged in to the Kerberos infrastructure,

1. On unix, this is done by running `kinit`
2. The `kinit` program asks the user for their password.
3. This is used to authenticate the user with the *Authentication Service* of the KDC configured in `/etc/krb5.conf`.
4. The Kerberos *Authentication Service* authenticates the user and issues a TGT ticket, which is stored in the client's *Credentials Cache*. A call to `klist` can be used to verify this.

Then, they must run a hadoop command

```
hadoop fs -ls /
```

1. The HDFS client code attempts to talk to the HDFS Namenode via the `org.apache.hadoop.hdfs.protocol.ClientProtocol` IPC protocol
2. It checks to see if If security is enabled (via `UserGroupInformation.isSecurityEnabled()`)
3. If it is, it looks in metadata assocated with the protocol, metadata which is used to identify the Kerberos principal, the identity, of the namenode.

```
@InterfaceAudience.Private
@InterfaceStability.Evolving
@KerberosInfo(serverPrincipal ="dfs.namenode.kerberos.principal")
@TokenInfo(DelegationTokenSelector.class)
public interface ClientProtocol {
...
}
```

4. The Hadoop `Configuration` class instance used to initialise the client is used to retrieve the value of `"dfs.namenode.kerberos.principal"` —so identifying the service to which the client must have a valid ticket to talk to.

5. The Hadoop Kerberos code (this is in Java, not the OS), asks the Kerberos *Ticket Granting Service*, *the TGS*, for a ticket to talk to the Namenode's principal. It does this in a request authenticated with the *TGT* received during the `kinit` process.
6. This ticket is granted by the TGT, and cached in the memory of the JVM.
7. The Hadoop RPC layer then uses the ticket to authenticate the caller to the Namenode, and implicitly, authenticate the NameNode to the caller.
8. The Namenode can use the Kerberos information to determine the identity of the (authenticated) caller.
9. It can then look at the permissions of the user as recorded in the HDFS directory and file metadata and determine if they have the rights to perform the requested action.
10. If they do, the action is performed and the results returned to the caller.

(Note there's some glossing over of details here, specifically how the client to Namenode authentication takes place, how they stay authenticated, how a users principal gets mapped to user name and how its group membership is ascertained for authorization purposes.)

If a second request is made against the Namenode in the same Java process, there is no need to ask the TGT for a new ticket —not until the previous one expires. Instead cached authentication data is reused. This avoids involving the KDC in any further interactions with the Namenode.

In Hadoop —as we will see— things go one step further, with Delegation Tokens. For now: ignore them.

This example shows Kerberos at work, and the Hadoop IPC integration.

As described, this follows the original Kerberos architecture, one principal per user, tickets between users and services. Hadoop/Kerberos integration has to jump one step further to address the scale problem, to avoid overloading the KDC with requests, to avoid problems such as having to have the client ask the TGT for a ticket to talk to individual Datanodes when reading or writing a file across the HDFS filesystem, or even handle the problem with a tens of thousands of clients having to refresh their Namenode tickets every few hours.

This is done with a concept called *Hadoop Delegation Tokens*. These will be covered later.

For now, know that the core authentication between principals and services utterly depends upon the Hadoop infrastructure, with an initial process as describe above.

Kerberos and Windows Active Directory

A lot of people are blissfully unaware of Kerberos. Such a life is one to treasure. Many of these people, do, however, log in to an enterprise network by way of Microsoft Active Directory. "AD" is a Kerberos Controller. Kerberos Explained

If an organisation uses Active Directory to manage users, they are running Kerberos, so have the basic infrastructure needed to authenticate users and services within a Hadoop cluster. Users should be able to submit jobs as themselves, interacting with "Kerberized" Hadoop services.

Setting up Hadoop to work with Active Directory is beyond the scope of this book. Please consult the references in the bibliography, and/or any vendor-specific documentation.

For Developers, it is worth knowing that AD is subtly different from the MIT/Unix Kerberos controller, enough so that you should really test with a cluster using AD as the Kerberos infrastructure, alongside the MIT KDC.

Limitations of Kerberos

Kerberos is considered "the best there is" in terms of securing distributed systems. Its use of tickets is designed to limit the load on the KDC, as it is only interacted with when a principal requests a ticket, rather than having to validate every single request.

The ability to delegate tokens to other processes allows transitive authentication as the original principal. This can be used by core Hadoop services to act on a users behalf, and by processes launched by the user.

The fact that tickets/tokens are time limited means that if one is stolen, the time for which unauthorized access is possible is limited to the lifespan of the token.

Finally, the fact that kerberos clients are standard in Windows, Linux and OS/X, and built into the Java runtime, means that it is possible to use Kerberos widely.

This does not mean it is perfect. Known limitations are

1. The KDC is a Single Point of Failure, unless an HA system is set up (which Active Directory can do).
2. Excess load can overload the KDC.
3. The communications channels between services still need to be secure. Kerberos does not address data encryption. If those channels are not secure, then tickets can be intercepted or communications forged.
4. Time needs to be roughly consistent across machines, else the time-limited tokens won't work.
5. If time cannot be securely managed across machines (i.e. an insecure time synchronization, protocol is used), then it is theoretically possible to extend the lifetime of a stolen token.
6. Because a stolen ticket can be used directly against a service, there's no log of its use in the KDC. Every application needs to have its own audit log of actions performed by a user, so that the history of actions by a client authenticated with a stolen ticket can be traced.
7. It's an authentication service: it verifies callers and allows callers to pass that authentication information on. It doesn't deal with permissions *at all*.

There's some coverage of other issues in Kerberos in the Crosshairs: Golden Tickets, Silver Tickets, MITM, and More

Hadoop/Kerberos Integration Issues

Hadoop specific issues are:

1. While the ticketing process reduces KDC load, an entire Hadoop cluster starting up can generate the login requests of a few thousand principals over a short period of time. The Hadoop code contains some back-off logic to handle connection and authentication failures here.
2. Because granted tokens expire, long-lived YARN services need to have a mechanism for updating tokens.
3. It's both hard to code for kerberos, and test against it.

Finally, it is *necessary but not sufficient*.

Having a Kerberized application does not guarantee that it is secure: you need to think about possible weaknesses, ways in which untrusted callers can make use of the service, ways in which tokens and keytabs may be leaked (that includes log messages!) and defend against them.

Hadoop's support for Kerberos

Hadoop can use Kerberos to authenticate users, and processes running within a Hadoop cluster acting on behalf of the user. It is also used to authenticate services running within the Hadoop cluster itself -so that only authenticated HDFS Datanodes can join the HDFS filesystem, that only trusted Node Managers can heartbeat to the YARN Resource Manager and receive work.

- The exact means by which all this is done is one of the most complicated pieces of code to span the entire Hadoop codebase.*

Users of Hadoop do not need to worry about the implementation details, and, ideally, nor should the operations team.

Developers of core Hadoop code, anyone writing a YARN application, and anyone writing code to interact with a Hadoop cluster and applications running in it *do need to know those details*.

This is what this book attempts to cover.

Why do they inflict so much pain on us?

Before going in there, here's a recurring question: why? Why Kerberos and not, say some SSL-certificate like system? Or OAuth?

Kerberos was written to support centrally managed accounts in a local area network, one in which adminstrators manage individual accounts. This is actually much simpler to manage than PKI-certificate based systems: look at the effort it takes to revoke a certificate in a browser.

Introducing Hadoop Tokens

So far we've covered Kerberos and *Kerberos Tickets*. Hadoop complicates things by adding another form of delegated authentication, *Hadoop Tokens*.

Why does Hadoop have another layer on top of Kerberos?

That's a good question, one developers ask on a regular basis —at least once every hour based on our limited experiments.

Hadoop clusters are some of the largest "single" distributed systems on the planet in terms of numbers of services: a YARN cluster of 10,000 nodes would have 10,000 hdfs principals, 10,000 yarn principals and the principals of the users running the applications. That's a lot of principals, all talking to each other, all having to talk to the KDC, having to re-authenticate all the time, and making calls to the KDC whenever they wish to talk to another principal in the system.

Tokens are wire-serializable objects issued by Hadoop services, which grant access to services. Some services issue tokens to callers which are then used by those callers to directly interact with other services *without involving the KDC at all*.

As an example, The HDFS NameNode has to give callers access to the blocks comprising a file. This isn't done in the DataNodes: all filenames and the permissions are stored in the NN. All the DNs have is their set of blocks.

To get at these blocks, HDFS gives an authenticated caller a *Block Tokens* for every block they need to read in a file. The caller then requests the blocks of any of the datanodes hosting that block, including the block token in the request.

These HDFS Block Tokens do not contain any specific knowledge of the principal running the Datanodes, instead they declare that the caller has stated access rights to the specific block, up until the token expires.

```
public class BlockTokenIdentifier extends TokenIdentifier {
  static final Text KIND_NAME = new Text("HDFS_BLOCK_TOKEN");

  private long expiryDate;
  private int keyId;
  private String userId;
  private String blockPoolId;
  private long blockId;
  private final EnumSet<AccessMode> modes;
  private byte [] cache;

  ...
```

Alongside the fields covering the block and permissions, that `cache` data contains the token identifier

Kerberos Tickets vs Hadoop Tokens

Token	Function
Authentication Token	Directly authenticate a caller.
Delegation Token	A token which can be passed to another process.

Authentication Tokens

Authentication Tokens are explicitly issued by services to allow the caller to interact with the service without having to re-request tickets from the TGT.

When an Authentication Tokens expires, the caller must request a new one from the service. If the Kerberos ticket to interact with the service has expired, this may include re-requesting a ticket off the TGS, or even re-logging in to Kerberos to obtain a new TGT.

As such, they are almost equivalent to Kerberos Tickets -except that it is the distributed services themselves issuing the Authentication Token, not the TGS.

Delegation Tokens

A delegation token is requested by a client of a service; they can be passed to other processes.

When the token expires, the original client must request a new delegation token and pass it on to the other process, again.

What is more important is:

- delegation tokens can be renewed before they expire.*

This is a fundamental difference between Kerberos Tickets and Hadoop Delegation Tokens.

Holders of delegation tokens may renew them with a token-specific `TokenRenewer` service, so refresh them without needing the Kerberos credentials to log in to kerberos.

More subtly

1. The tokens must be renewed before they expire: once expired, a token is worthless.
2. Token renewers can be implemented as a Hadoop RPC service, or by other means, *including HTTP*.
3. Token renewal may simply be the updating of an expiry time in the server, without pushing out new tokens to the clients. This scales well when there are many processes across the cluster associated with a single application..

For the HDFS Client protocol, the client protocol itself is the token renewer. A client may talk to the Namenode using its current token, and request a new one, so refreshing it.

In contrast, the YARN timeline service is a pure REST API, which implements its token renewal over HTTP/HTTPS. To refresh the token, the client must issue an HTTP request (a PUT operation, interestingly enough), receiving a new token as a response.

Other delegation token renewal mechanisms alongside Hadoop RPC and HTTP could be implemented, that is a detail which client applications do not need to care about. All the matters is that they have the code to refresh tokens, usually code which lives alongside the RPC/REST client, *and keep renewing the tokens on a regularl basis*. Generally this is done by starting a thread in the background.

Delegation Token Revocation

Delegation tokens can be revoked —such as when the YARN which needed them completes.

In Kerberos, the client obtains a ticket off the KDC, then hands it to the service —a service which does not need to contain any information about the tickets issued to clients.

With delegation tokens, the specific services supporting them have to implement their own internal tracking of issued tokens. That comes with benefits as well as a cost.

The cost? The services now have to maintain some form of state, either locally or, in HA deployments, in some form of storage shared across the failover services.

The benefit: there's no need to involve the KDC in authenticating requests, yet short-lived access can be granted to applications running in the cluster. This explicitly avoid the problem of having 1000+ containers in a YARN application each trying to talk to the KDC. (Issue: surely tickets offer that feature?).

Example

Imagine a user deploying a YARN application in a cluster, one which needs access to the user's data stored in HDFS. The user would be required to be authenticated with the KDC, and have been granted a *Ticket Granting Ticket*; the ticket needed to work with the TGS.

The client-side launcher of the YARN application would be able to talk to HDFS and the YARN resource manager, because the user was logged in to Kerberos. This would be managed in the Hadoop RPC layer, requesting tickets to talk to the HDFS NameNode and YARN ResourceManager, if needed.

To give the YARN application the same rights to HDFS, the client-side application must request a Delegation Token to talk to HDFS, a key which is then passed to the YARN application in the `ContainerLaunchContext` within the `ApplicationSubmissionContext` used to define the application to launch: its required container resources, artifacts to download, "localize", environment to set up and command to run.

The YARN resource manager finds a location for the Application Master, and requests that hosts' Node Manager start the container/application.

The Node Manager uses the "delegated HDFS token" to download the launch-time resources into a local directory space, then executes the application.

Somehow, the HDFS token (and any other supplied tokens) are passed to the application that has been launched.

The launched application master can use this token to interact with HDFS *as the original user*.

The AM can also pass token(s) on to launched containers, so that they too have access to HDFS.

The Hadoop NameNode does not need to care whether the caller is the user themselves, the Node Manager localizing the container, the launched application or any launched containers. All it does is verify that when a caller requests access to the HDFS filesystem metadata or the contents of a file, it must have a ticket/token which declares that they are the specific user, and that the token is currently considered valid (based on the expiry time and the clock value of the Name Node)

What does this mean for my application?

If you are writing an application, what does this mean?

You need to worry about tokens in servers if:

1. You want an application deploying work into a YARN cluster to access your service *as the user submitting the job*.
2. You want to support secure connections without requiring Kerberos authentication at the rate of the maximum life of a kerberos ticket.
3. You want to allow applications to delegate authority, such as to YARN applications, or other services. (Example, filesystem delegation tokens provided to a Hive thrift server could be used to access the filesystem as that user).
4. You want a consistent client/server authentication and identification mechanism across secure and insecure clusters. This is exactly what YARN does: a token is issued by the YARN Resource Manager to an application instance's Application Manager at launch time; this is used in all communications from the AM to the RM. Using tokens *always* means there is no separate codepath between insecure and secure clusters.

You need to worry about tokens in client applications if you wish to interact with Hadoop services. If the client is required to run on a kerberos-authenticated account (e.g. kinit or keytab), then your main concern is simply making sure the principal is logged in.

If your application wishes to run code in the cluster using the YARN scheduler, you need to directly worry about Hadoop tokens. You will need to request delegation tokens from every service with which your application will interact, include them in the YARN launch information —and propagate them from your Application Master to all containers the application launches.

Design

(from Owen's design document)

Namenode Token

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

The token ID is used in messages from the client to identify the client; service can rebuild the `TokenAuthenticator` from it; this is the secret used for DIGEST-MD5 signing of requests.

Token renewal: caller asks service provider for a token to be renewed. The server updates the expiry date in its local table to `min(maxDate, now()+renew_period)`. A non-HA NN can use these renewal requests to actually rebuild its token table —provided the master key has been persisted.

Implementation Details

What is inside a Hadoop Token? Whateve marshallable data the service wishes to supply.

A token is treated as a byte array to be passed in communications, such as when setting up an IPC connection, or as a data to include on an HTTP header while negotiating with a remote REST endpoint.

The code on the server which issues tokens, the `SecretManager` is free to fill its byte arrays with structures of its choice. Sometimes serialized java objects are used; more recent code, such as that in YARN, serializes data as a protobuf structure and provides that in the byte array (example, `NMTokenIdentifier`).

`Token`

The abstract class `org.apache.hadoop.yarn.api.records.Token` is used to represent a token in Java code; it contains

field	type	role
identifier	`ByteBuffer`	the service-specific data within a token
password	`ByteBuffer`	a password
tokenKind	`String`	token kind for looking up tokens.

`TokenIdentifier implements Writable`

Implementations of the the abstract class `org.apache.hadoop.security.token.TokenIdentifier` must contain everything needed for a caller to be validated by a service which uses tokens for authentication.

The base class has:

field	type	role
`kind`	`Text`	Unique type of token identifier

The `kind` field must be unique across all tokens, as it is used in bonding to tokens.

`DelegationTokenIdentifier`

The abstract class `org.apache.hadoop.security.token.delegation.AbstractDelegationTokenIdentifier` is the base of all Delegation Tokens in HDFS and elsewhere.

It extends `TokenIdentifier` with:

field	type	role
`owner`	`Text`	owner of the token
`renewer`	`Text`	
`realUser`	`Text`	

It is straightforward to extend this with more data, which can be used to add information into a token which can then be read by the applications which have loaded the tokens. This has been used in HADOOP-14556 to pass Amazon AWS login secrets as if they were a filesystem token.

`SecretManager`

Every server which handles tokens through Hadoop RPC should implement an a `org.apache.hadoop.security.token.SecretManager` subclass.

In `org.apache.hadoop.ipc.Server` and `org.apache.hadoop.security.SaslRpcServer`, a specific implementation of the class `SecretManager<T extends TokenIdentifier>` creates instances of the TokenIdentifier

`DelegationKey`

This contains a "secret" (generated by the `javax.crypto` libraries), adding serialization and equality checks. Because of this the keys can be persisted (as HDFS does) or sent over a secure channel. Uses crop up in YARN's `ZKRMStateStore`, the MapReduce History server and the YARN Application Timeline Service.

Working with tokens

How tokens are issued

A first step is determining the Kerberos Principal for a service:

1. The Service name is derived from the URI (see `SecurityUtil.buildDTServiceName`)...different services on the same host have different service names
2. Every service has a protocol (usually defined by the RPC protocol API)
3. To find a token for a service, client enumerates all `SecurityInfo` instances; these return info about the provider. One class `AnnotatedSecurityInfo`, examines the annotations on the class to determine these values, including looking in the Hadoop configuration to determine the kerberos principal declared for that service (see IPC for specifics).

How tokens are renewed

Token renewal is the second part of token work. If you are implementing token support, it is easiest to postpone this until the core issuing is working.

Implement a subclass of `org.apache.hadoop.security.token.TokenRenewer`,

```
public abstract class TokenRenewer {

  /**
   * Does this renewer handle this kind of token?
   * @param kind the kind of the token
   * @return true if this renewer can renew it
   */
  public abstract boolean handleKind(Text kind);

  /**
   * Is the given token managed? Only managed tokens may be renewed or
   * cancelled.
   * @param token the token being checked
   * @return true if the token may be renewed or cancelled
   * @throws IOException
   */
  public abstract boolean isManaged(Token<?> token) throws IOException;

  /**
   * Renew the given token.
   * @return the new expiration time
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract long renew(Token<?> token,
                             Configuration conf
                             ) throws IOException, InterruptedException;

  /**
   * Cancel the given token
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract void cancel(Token<?> token,
                              Configuration conf
                              ) throws IOException, InterruptedException;
}
```

1. In `handleKind()`, verify the token kind is that which the renewer supports.
2. In `isManaged()`, return true, unless the ability to renew a token is made on a token-by-token basis.
3. In `reneww()`, renew the credential by notifying the bonded service that the token is to be renewed.
4. In `cancel()`, notify the bonded service that the token should be cancelled.

Finally, declare the class in the resource `META-INF/services/org.apache.hadoop.security.token.TokenRenewer`.

How delegation tokens are shared

DTs can be serialized; that is done when issued/renewed.

When making requests over Hadoop RPC, you don't need to include the DT, simply include the Hash to indicate that you have it (Revisit: what does this mean?)

Every token which is deserialized at the far end must have a service declaration in

`META-INF/services/org.apache.hadoop.security.token.TokenIdentifier`

Here is the one in `hadoop-common` as of Hadoop 3.2:

```
org.apache.hadoop.crypto.key.kms.KMSDelegationToken$KMSDelegationTokenIdentifier
```

When Hadoop is looking for an implementation of a token, it enumerates all available token identifiers registered this way through the java `ServiceLoader.load(class)` API.

`Token.decodeIdentifier()` is then invoked to extract the identifier information.

Important: All java classes directly invoked in a token implementation class must be on the classpath.

If this condition is not met, the identifier cannot be loaded.

Delegation Tokens

Token Propagation in YARN Applications

YARN applications depend on delegation tokens to gain access to cluster resources and data on behalf of the principal. It is the task of the client-side launcher code to collect the tokens needed, and pass them to the launch context used to launch the Application Master.

Proxy Users

Proxy users are a feature which was included in the Hadoop security model for services such as Oozie; a service which needs to be able to execute work on behalf of a user

Because the time at which Oozie would execute future work cannot be determined, delegation tokens cannot be used to authenticate requests issued by Oozie on behalf of a user. Kerberos keytabs are a possible solution here, but it would require every user submitting work to Oozie to have a keytab and to pass it to Oozie.

See Proxy user - Superusers Acting On Behalf Of Other Users.

Also a clarification in HADOOP-15758

> You cannot mimic a proxy user. A proxy user is specific construct. There is no substitute. A proxy user is a ugi that lacks its own authentication credentials, thus it explicitly encapsulates a "real" ugi that does contain kerberos credentials. The real ugi's user must be specifically configured on the target service to allow impersonation of the proxied user.
>
> There is no correlation between a proxy user and a ticket cache. The real ugi can supply ticket cache or keytab based credentials. All that matters is the real user has credentials

There's also a special bit of advice for service implementors

> An impersonating service should never ever be ticket cache based. Use a keytab. Otherwise you may be very surprised with proxy user service morphs into a different user if/when someone/something does a kinit as a different user.

Weaknesses

1. In HDFS Any compromised DN can create block tokens.
2. Possession of the tokens is sufficent to impersonate a user. This means it is critical to transport tokens over the network in an encrypted form. Typically, this is done by SASL-encrypting the Hadoop IPC channel.

How Delegation Tokens work in File System implementations

This is for relevance of people implementing/using FileSystem APIs.

1. Any FileSystem instance may issue zero or more delegation tokens when asked.
2. These can be used to grant time-limited access to the filesystem in different processes.
3. The reason more than one can be issued is to support aggregate filesystems where multiple filesystems may eventually be invoked (e.g ViewFS), or when tokens for multiple services need to be included (e.g KMS access tokens).
4. Clients must invoke `FileSystem.addDelegationTokens()` to get an array of tokens which may then be written to persistent storage, marshalled, and loaded in later.
5. Each filesystem must have a unique Canonical Name -a URI which will refer only to the FS instance to which the token matches.
6. When working with YARN, container and application launch contexts may include a list of tokens to include. the FS Tokens should be included here along with any needed to talk to the YARN RM, and, for Hadoop 3+, docker tokens. See `org.apache.hadoop.yarn.applications.distributedshell.Client` for an example of this.

In the remote application, these tokens can be unmarshalled from a the byte array, the tokens for specific services retrieved, and then used for authentication.

Technique: how to propagate secrets in TokenIdentifiers

A TokenIdentifier is a `Writable` object; it can contain arbitrary data. For this reason it can be used to propagate secrets from a client to deployed YARN applications and containers, even when there is no actual service which reads the token identifiers. All that is required is that the far end implements the token identifier and declares itself as such.

HDFS

> It seemed to be a sort of monster, or symbol representing a monster, of a form which only a diseased fancy could conceive. If I say that my somewhat extravagant imagination yielded simultaneous pictures of an octopus, a dragon, and a human caricature, I shall not be unfaithful to the spirit of the thing. A pulpy, tentacled head surmounted a grotesque and scaly body with rudimentary wings; but it was the general outline of the whole which made it most shockingly frightful. *The Call of Cthulhu, HP Lovecraft, 1926.*

HDFS uses Kerberos to

1. Authenticate caller's access to the Namenode and filesystem metadata (directory and file manipulation).
2. Authenticate Datanodes attempting to join the HDFS cluster. This prevents malicious code from claiming to be part of HDFS and having blocks passed to it.
3. Authenticate the Namenode with the Datanodes (prevents malicious code claiming to be the Namenode and granting access to data or just deleting it)
4. Grant callers read and write access to data within HDFS.

Kerberos is used to set up the initial trust between a client and the NN, by way of Hadoop tokens. A client with an Authentication Token can request a Delegation Token, which it can then pass to other services or YARN applications, so giving them time-bound access to HDFS with the rights of that user.

The namenode also issues "Block Tokens" which are needed to access HDFS data stored on the Datanodes: the DNs validate these tokens, rather than requiring clients to authenticate with the DNs in any way. This avoids any authentication overhead on block requests, and the need to somehow acquire/share delegation tokens to access specific DNs.

HDFS Block Tokens do not (as of August 2015) contain information about the identity of the caller or the process which is running. This is somewhat of an inconvenience, as it prevents the HDFS team from implementing user-specific priority/throttling of HDFS data access —something which would allow YARN containers to manage the IOPs and bandwith of containers, and allow multi-tenant Hadoop clusters to prioritise high-SLA applications over lower-priority code.

HDFS NameNode

1. NN reads in a keytab and initializes itself from there (i.e. no need to `kinit`; ticket renewal handed by `UGI`).
2. Generates a *Secret*

Delegation tokens in the NN are persisted to the edit log, the operations `OP_GET_DELEGATION_TOKEN` `OP_RENEW_DELEGATION_TOKEN` and `OP_CANCEL_DELEGATION_TOKEN` covering the actions. This ensures that on failover, the tokens are still valid

Block Keys

A `BlockKey` is the secret used to show that the caller has been granted access to a block in a DN.

The NN issues the block key to a client, which then asks a DN for that block, supplying the key as proof of authorization.

Block Keys are managed in the `BlockTokenSecretManager`, one in the NN and another in every DN to track the block keys to which it has access. It is the DNs which issue block keys as blocks are created; when they heartbeat to the NN they include the keys.

Block Tokens

A `BlockToken` is the token issued for access to a block; it includes

```
(userId, (BlockPoolId, BlockId), keyId, expiryDate, access-modes)
```

The block key itself isn't included, just the key to the referenced block. The access modes declare what access rights the caller has to the data

```
public enum AccessMode {
  READ, WRITE, COPY, REPLACE
}
```

It is the NN which has the permissions/ACLs on each file —DNs don't have access to that data. Thus it is the BlockToken which passes this information to the DN, by way of the client. Obviously, they need to be tamper-proof.

DataNodes

DataNodes do not use Hadoop RPC —they transfer data over HTTP. This delivers better performance, though the (historical) use of Jetty introduced other problems. At scale, obscure race conditions in Jetty surfaced. Hadoop now uses Netty for its DN block protocol.

DataNodes and SASL

Pre-2.6, all that could be done to secure the DN was to bring it up on a secure (<1024) port and so demonstrate that an OS superuser started the process. Hadoop 2.6 supports SASL authenticated HTTP connections, which works *provided all clients are all running Hadoop 2.6+*

See Secure DataNode

HDFS Bootstrap

1. NN reads in a keytab and initializes itself from there (i.e. no need to `kinit`; ticket renewal handed by `UGI`).

2. Generates a *Secret*

Delegation tokens in the NN are persisted to the edit log, the operations `OP_GET_DELEGATION_TOKEN` `OP_RENEW_DELEGATION_TOKEN` and `OP_CANCEL_DELEGATION_TOKEN` covering the actions. This ensures that on failover, the tokens are still valid

HDFS Client interaction

1. Client asks NN for access to a path, identifying via Kerberos or delegation token.
2. NN authenticates caller, if access to path is authorized, returns Block Token to the client.
3. Client talks to 1+ DNs with the block, using the Block Token.
4. DN authenticates Block Token using shared-secret with NameNode.
5. if authenticated, DN compares permissions in Block Token with operation requested, then grants or rejects the request.

The client does not have its identity checked by the DNs. That is done by the NN. This means that the client can in theory pass a Block Token on to another process for delegated access to a single block. It has another implication: DNs can't do IO throttling on a per-user basis, as they do not know the user requesting data.

WebHDFS

1. In a secure cluster, Web HDFS requires SPNEGO
2. After authenticating with a SPNEGO-negotiated mechanism, webhdfs sends an HTTP redirect, including the BlockTocken in the redirect

NN/Web UI

1. If web auth is enabled in a secure cluster, the DN web UI will requires SPNEGO
2. In a secure cluster, if webauth is disabled, kerberos/SPNEGO auth may still be needed to access the HDFS browser. This is a point of contention: its implicit from the delegation to WebHDFS --but a change across Hadoop versions, as before an unauthed user could still browse as "dr who".

UGI

> From the pictures I turned to the bulky, closely written letter itself; and for the next three hours was immersed in a gulf of unutterable horror. Where Akeley had given only outlines before, he now entered into minute details; presenting long transcripts of words overheard in the woods at night, long accounts of monstrous pinkish forms spied in thickets at twilight on the hills, and a terrible cosmic narrative derived from the application of profound and varied scholarship to the endless bygone discourses of the mad self-styled spy who had killed himself.
>
> HP Lovecraft The Whisperer in Darkness, 1931

If there is one class guaranteed to strike fear into anyone with experience in Hadoop+Kerberos code it is `UserGroupInformation`, abbreviated to "UGI"

Nobody says `UserGroupInformation` out loud; it is the *that which must not be named* of the stack

What does UGI do?

Here are some of the things it can do

1. Handles the initial login process, using any environmental `kinit`-ed tokens or a keytab.
2. Spawn off a thread to renew the TGT
3. Support an operation for-on demand verification/re-init of kerberos tickets details before issuing a request.
4. Appear in stack traces which warn the viewer of security related trouble.

UGI Strengths

- It's one place for almost all Kerberos/User authentication to live.
- Being fairly widely used, once you've learned it, your knowledge works through the entire Hadoop stack.

UGI Troublespots

- It's a singleton. Don't expect to have one "real user" per process. This does sort of makes sense. Even a single service has its "service" identity; as the
- Once initialized, it stays initialized *and cannot be reset*. This makes it critical to load in your configuration information including keytabs and principals, before that first initialization of the UGI. (There is actually a `UGI.reset()` call, but it is package scoped and purely to allow tests to reset the information).

- UGI initialization can take place in code which you don't expect. A specific example is in the Hadoop filesystem APIs. Create a Hadoop filesystem instance and UGI is likely to be inited immediately, even if it is a local file:// reference. As a result: init before you go near the filesystem, with the principal you want.
- It has to do some low-level reflection-based access to Java-version-specific Kerberos internal classes. This can break across Java versions, and JVM implementations. Specifically Java 8 has classes that Java 6 doesn't; the IBM JVM is very different.
- All its exceptions are basic `IOException` instances, so hard to match on without looking at the text, which is very brittle.
- Some invoked operations are relayed without the stack trace (this should now be fixed).
- Diagnostics could be improved. (this is one of those British understatements, it really means "it would be really nice if you could actually get any hint as to WTF is going inside the class as otherwise you are left with nothing to go on other than some message that a user at a random bit of code wasn't authorized)

The issues related to diagnostics, logging, exception types and inner causes could be addressed. It would be nice to also have an exception cached at init time, so that diagnostics code could even track down where the init took place. Volunteers welcome. That said, here are some bits of the code where patches would be vetoed

- Replacing the current text of exceptions. We don't know what is scanning for that text, or what documents go with them. Extending the text should be acceptable.
- All exceptions must remain subclasses of IOException.
- Logging must not leak secrets, such as tokens.

Core UGI Operations

`isSecurityEnabled()`

One of the foundational calls is the `UserGroupInformation.isSecurityEnabled()`

It crops up in code like this

```
if(!UserGroupInformation.isSecurityEnabled()) {
  stayInALifeOfNaiveInnocence();
} else {
  sufferTheEternalPainOfKerberos();
}
```

Having two branches of code, the "insecure" and "secure mode" is actually dangerous: the entire security-enabled branch only ever gets executed when run against a secure Hadoop cluster

Important

If you have separate secure and insecure codepaths, you must test on a secure cluster alongside an insecure one. Otherwise coverage of code and application state will be fundamentally lacking.

Unless you put in the effort, all your unit tests will be of the insecure codepath.

This means there's an entire codepath which won't get exercised until you run integration tests on a secure cluster, or worse: until you ship.

What to do? Alongside the testing, the other strategy is: keep the differences between the two branches down to a minimum. If you look at what YARN does, it always uses renewable tokens to authenticate communication between the YARN Resource Manager and a deployed application. As a result, one codepath for token creation, while token propagation and renewal is automatically tested on all applications.

Could your applications do the same? Certainly as far as token- and delegation-token based mechanisms for callers to show that they have been granted access rights to a service.

`getLoginUser()`

This returns the logged in user

```
UserGroupInformation user = UserGroupInformation.getLoginUser();
```

If there is no logged user --that is, the login process hasn't started yet, this triggers the login and the starting of the background refresh thread.

This makes it a point where the security kicks in: all configuration resources must be loaded in advance.

`checkTGTAndReloginFromKeytab()`

```
UserGroupInformation.checkTGTAndReloginFromKeytab();
```

If security is not enabled, this is a no-op.

If security is enabled, and the last login took place "long enough ago", this will trigger a re-login if needed (which may fail, of course).

If the last successful login was recent enough, this will be a no-op. This makes it a low cost operation to include in IPC/REST client operations so as to ensure that your tickets are up to date.

Important: If the login fails, UGI will remember this and not retry until a time limit has passed, even if other methods invoke the operation. The property `hadoop.kerberos.min.seconds.before.relogin` controls this delay; the default is 60s.

What does that mean? A failure lasts for a while, even if it is a transient one.

`getCurrentUser()`

This returns the *current* user.

Environment variable-managed UGI Initialization

There are some environment variables which configure UGI.

Environment Variable	Meaning
HADOOP_PROXY_USER	identity of a proxy user to authenticate as
HADOOP_TOKEN_FILE_LOCATION	local path to a token file

Why environment variables? They offer some features

1. Hadoop environment setup scripts can set them
2. When launching YARN containers, they may be set as environment variables.

As the UGI code is shared across all clients of HDFS and YARN; these environment variables can be used to configure *any* application which communicates with Hadoop services via the UGI-authenticated clients. Essentially: all Java IPC clients and those REST clients using (the Hadoop-implemented REST clients)[web_and_rest.html].

Debugging UGI

UGI supports low-level logging via the log4J log `org.apache.hadoop.security.UserGroupInformation`; set the system property `HADOOP_JAAS_DEBUG=true` to have the JAAS context logging at the debug level via some Java log API.

It's helpful to back this up with logging the `org.apache.hadoop.security.authentication` package in `hadoop-auth`

```
log4j.logger.org.apache.hadoop.security.authentication=DEBUG
log4j.logger.org.apache.hadoop.security=DEBUG
```

Proxy Users

Some applications need to act on behalf of other users. For example: Oozie wants to run scheduled jobs as people, YARN services

The current user is not always the same as the logged in user; it changes when a service performs an action on the user's behalf

`createProxyUser()`

Proxy users are a feature which was included in the Hadoop security model for services such as Oozie; a service which needs to be able to execute work on behalf of a user

`doAs()`

This method is at the core of UGI. A call to `doAs()` executes the inner code *as the user*. In secure, that means using the Kerberos tickets and Hadoop delegation tokens belonging to them.

Example: loading a filesystem as a user

```
UserGroupInformation proxy =
  UserGroupInformation.createProxyUser(user,
   UserGroupInformation.getLoginUser());

FileSystem userFS = proxy.doAs(
  new PrivilegedExceptionAction<FileSystem>() {
    public FileSystem run() throws Exception {
      return FileSystem.get(FileSystem.getDefaultUri(), conf);
    }
  });
```

Here the variable `userFS` contains a client of the Hadoop Filesystem with the home directory and access rights of the user `user`. If the user identity had come in via an RPC call, they'd

> Johansen, thank God, did not know quite all, even though he saw the city and the Thing, but I shall never sleep calmly again when I think of the horrors that lurk ceaselessly behind life in time and in space, and of those unhallowed blasphemies from elder stars which dream beneath the sea, known and favoured by a nightmare cult ready and eager to loose them upon the world whenever another earthquake shall heave their monstrous stone city again to the sun and air.
>
> HP Lovecraft The Call of Cthulhu, 1926

Java and JDK Versions

Kerberos support is built into the Java JRE. It comes in two parts

1. The public APIs with some guarantee of stability at the class/method level, along with assertions of functionality behind those classes and methods.
2. The internal implementation classes which are needed to do anything sophisticated.

The Hadoop UGI code uses part (2), the internal `com.sun` classes, as the public APIs are too simplistic for the authentication system.

This comes at a price

1. Different JRE vendors (e.g IBMs) use different implementation classes, so will not work.
2. Different Java versions can (and do) change those implementation classes.

Using internal classes is one of those "don't do this your code will be unreliable" rules; it wasn't something done lightly. In its defence, (a) it was needed and (b) it turns out that the public APIs are brittle across versions and JDKs too, so we haven't lost as much as you'd think.

Key things to know

- Hadoop is built and tested against the Oracle JDKs
- Open JDK has the same classes and methods, so will behave consistently; it's tested against too.
- It's left to the vendors of other JVMs to test their code; the patches are taken on trust.
- The Kerberos internal access usually needs fixing across Java versions. This means secure Hadoop clusters absolutely require the Java versions listed on the download requirements.
- Releases within a Java version may break the internals and/or the public API's behaviour.
- If you want to see the details of Hadoop's binding, look in `org.apache.hadoop.security.authentication.util.KerberosUtil` in the `hadoop-auth` module.

To put it differently:

The Hadoop security team is always scared of a new version of Java

JAAS

JAAS is a nightmare from the Enterprise Java Bean era, one which surfaces from the depths to pull the unwary under. You can see its heritage whenever you search for documentation; it's generally related to managing the context of callers to EJB operations.

JAAS provides for a standard configuration file format for specifying a *login context*; how code trying to run in a specific context/role should login and authenticate.

As a single `jaas.conf` file can have multiple contexts, the same file can be used to configure the server and clients of a service, each with different binding information. Different contexts can have different login/auth mechanisms, including Kerberos and LDAP, so that you can even specify different auth mechanisms for different roles.

In Hadoop, the JAAS context is invariably Kerberos when it comes to talking to HDFS, YARN, etc. However, if Zookeeper enters the mix, it may be interacted with differently —and so need a different JAAS context.

Fun facts about JAAS

1. Nobody ever mentions it, but the file takes backslashed-escapes like a Java string.
2. It needs escaped backlash directory separators on Windows, such as: `C:\\security\\krb5.conf`. Get that wrong and your code will fail with what will inevitably be an unintuitive message.
3. Each context must declare the authentication module to use. The kerberos authentication model on IBM JVMs is different from that on Oracle and OpenJDK JVMs. You need to know the target JVM for the context —or create separate contexts for the different JVMs.
4. The rules about when to use `=` within an entry, and when to complete an entry with a `;` appear to be: start with the login module, one key=value line per entry, quote strings, finish with a `;` within the same file.

Hadoop's UGI class will dynamically create a JAAS context for Hadoop logins, dynamically determining the name of the kerberos module to use. For interacting purely with HDFS and YARN, you may be able to avoid needing to know about or understand JAAS.

Example of a JAAS file valid for an Oracle JVM:

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=false
  useTicketCache=true
  doNotPrompt=true;
};
```

Setting a JAAS Config file for a Java process

```
-Djava.security.auth.login.config=/path/to/server/jaas/file.conf
```

In Hadoop applications, this has to be set in whichever environment variable is picked up by the command which your are invoking.

Keytabs

Keytabs are critical for secure Hadoop clusters, as they allow the services to be launched without prompts for passwords

Creating a Keytab

If your management tools sets up keytabs for you: use it.

```
kadmin.local

ktadd -k zk.service.keytab -norandkey zookeeper/devix@COTHAM
ktadd -k zk.service.keytab -norandkey zookeeper/devix.cotham.uk@COTHAM
exit
```

and of course, make it accessible

```
chgrp hadoop zk.service.keytab
chown zookeeper zk.service.keytab
```

check that the user can login

```
# sudo -u zookeeper klist -e -kt zk.service.keytab
# sudo -u zookeeper kinit -kt zk.service.keytab zookeeper/devix.cotham.uk
# sudo -u zookeeper klist
```

Keytab Expiry

Keytabs expire

That is: entries in them have a limited lifespan (default: 1 year)

This is actually a feature —it limits how long a lost/stolen keytab can have access to the system.

At the same time, it's a major inconvenience as (a) the keytabs expire and (b) it's never immediately obvious why your cluster has stopped working.

Keytab security

Keytabs are sensitive items. They need to be treated as having all the access to the data of that principal

Keytabs and YARN applications

SASL: Simple Authentication and Security Layer

This is an IETF-managed specification for securing network channels, with RFC4422 covering the core of it.

SASL is not an authentication mechanism. SASL is a mechanism for applications to set up an authenticated communications channel by way of a shared authentication mechanism. SASL covers the protocol for the applications to negotiate as to which authentication mechanism to use, then to perform whatever challenge/response exchanges are needed for that authentication to take place. Kerberos is one authentication mechanism, but SASL supports others, such as x.509 certificates.

Similarly, SASL does not address wire-encryption, or anything related to authorization. SASL is purely about a client authenticating its actual or delegated identity with a server, while the client verifies that the server also has the required identity (usually one declared in a configuration file).

As well as being independent of the authentication mechanism, SASL is independent of the underlying wire format/communications protocol. The SASL implementation libraries can be used by applications to secure whatever network protocol they've implemented.

In Hadoop, "SASL" can be taken to mean "authentication negotiated using SASL". It doesn't define which protocol itself is authenticated —and you don't really need to care. Furthermore, if you implement your own protocol, if you add SASL-based authentication to it, you get to use Kerberos, x509, Single-Sign-On, Open Auth (when completed), etc.

For Hadoop RPC, there are currently two protocols for authentication:

- KERBEROS: Kerberos ticket-based authentication
- DIGEST-MD5: MD5 checksum-based authentication; shows caller has a secret which the recipient also knows.

Note that there is also the protocol `PLAIN`; SASL-based negotiation to not have any authentication at all. That doesn't surface in Hadoop —yet— though it does crop up in JIRAs.

SASL-enabled services

Services which use SASL include

1. Hadoop RPC
2. Zookeeper
3. HDFS 2.6+ DataNode bulk IO protocol (HTTP based)

> Man's respect for the imponderables varies according to his mental constitution and environment. Through certain modes of thought and training it can be elevated tremendously, yet there is always a limit.
>
> *At the Root, HP Lovecraft, 1918.*

Hadoop IPC Security

The Hadoop IPC system handles Kerberos ticket and Hadoop token authentication automatically.

1. The identity of the principals of services are configured in the hadoop site configuration files.
2. Every IPC services uses java annotations, a metadata resource file and a custom `SecurityInfo` subclass to define the security information of the IPC, including the key in the configuration used to define the principal.
3. If a caller making a connection has a valid token (auth or delegate) it is used to authenticate with the remote principal.
4. If a caller lacks a token, the Hadoop ticket will be used to acquire an authentication token.
5. Applications may explicitly request delegation tokens to forward to other processes.
6. Delegation tokens are renewed in a background thread (which?).

IPC authentication options

Hadoop IPC uses SASL to authenticate, sign and potentially encrypt communications.

Use Kerberos to authenticate sender and recipient

```
<property>
<name>hadoop.rpc.protection</name>
<value>authentication</value>
</property>
```

Kerberos to authenticate sender and recipient, Checksums for tamper-protection

```
<property>
<name>hadoop.rpc.protection</name>
<value>integrity</value>
</property>
```

Kerberos to authenticate sender and recipient, Wire Encryption

```
<property>
<name>hadoop.rpc.protection</name>
<value>privacy</value>
</property>
```

Adding a new IPC interface to a Hadoop Service/Application

This is "fiddly". It's not impossible, it just involves effort.

In its favour: it's a lot easier than SPNEGO.

Annotating a service interface

```
@KerberosInfo(serverPrincipal = "my.kerberos.principal")
public interface MyRpc extends VersionedProtocol {
  long versionID = 0x01;
...
}
```

`SecurityInfo` subclass

Every exported RPC service will need its own extension of the `SecurityInfo` class, to provide two things:

1. The name of the principal to use in this communication
2. The token used to authenticate ongoing communications.

`PolicyProvider` subclass

```
public class MyRpcPolicyProvider extends PolicyProvider {

  public Service[] getServices() {
    return  new Service[] {
     new Service("my.protocol.acl", MyRpc.class)
    };
  }

}
```

This is used to inform the RPC infrastructure of the ACL policy: who may talk to the service. It must be explicitly passed to the RPC server

```
rpcService.getServer() .refreshServiceAcl(serviceConf, new MyRpcPolicyProvider());
```

In practise, the ACL list is usually configured with a list of groups, rather than a user.

`SecurityInfo` class

```
public class MyRpcSecurityInfo extends SecurityInfo { ... }
```

`SecurityInfo` resource file

The resource file `META-INF/services/org.apache.hadoop.security.SecurityInfo` lists all RPC APIs which have a matching SecurityInfo subclass in that JAR.

```
org.example.rpc.MyRpcSecurityInfo
```

The RPC framework will read this file and build up the security information for the APIs (server side? Client side? both?)

Authenticating a caller

How does an IPC endpoint validate the caller? If security is turned on, the client will have had to authenticate with Kerberos, ensuring that the server can determine the identity of the principal.

This is something it can ask for when handling the RPC Call:

```
UserGroupInformation callerUGI;

// #1: get the current user identity
try {
  callerUGI = UserGroupInformation.getCurrentUser();
} catch (IOException ie) {
  LOG.info("Error getting UGI ", ie);
  AuditLogger.logFailure("UNKNOWN", "Error getting UGI");
  throw RPCUtil.getRemoteException(ie);
}
```

The `callerUGI` variable is now set to the identity of the caller. If the caller has delegated authority (tickets, tokens) then they still authenticate as that principal they were acting as (possibly via a `doAs()` call).

```
// #2 verify their permissions
String user = callerUGI.getShortUserName();
if (!checkAccess(callerUGI, MODIFY)) {
  AuditLog.unauthorized(user,
    KILL_CONTAINER_REQUEST,
    "User doesn't have permissions to " + MODIFY);
  throw RPCUtil.getRemoteException(new AccessControlException(
    + user + " lacks access "
    + MODIFY_APP.name()));
}
AuditLog.authorized(user, KILL_CONTAINER_REQUEST)
```

In ths example, there's a check to see if the caller can make a request which modifies something in the service, if not the calls is rejected.

Note how failures are logged to an audit log; successful operations should be logged too. The purpose of the audit log is determine the actions of a principal —both successful and unsuccessful.

Downgrading to unauthed IPC

IPC can be set up on the client to fall back to unauthenticated IPC if it can't negotiate a kerberized connection. While convenient, this opens up some security vulnerabilitie -hence the feature is generally disabled on secure clusters. It can/should be enabled when needed

```
-D ipc.client.fallback-to-simple-auth-allowed=true
```

As an example, this is the option on the command line for DistCp to copy from a secure cluster to an insecure cluster, the destination only supporting simple authentication.

```
hadoop distcp -D ipc.client.fallback-to-simple-auth-allowed=true hdfs://secure:8020/lovecraft/
books hdfs://insecure:8020/lovecraft/books
```

Although you can set it in a core-site.xml, this is dangerous from a security perpective

```
<property>
  <name>ipc.client.fallback-to-simple-auth-allowed</name>
  <value>true</value>
</property>
```

warning it's tempting to turn this on during development, as it makes problems go away. As it is not recommended in production: avoid except on the CLI during attempts to debug problems.

Web, REST and SPNEGO

SPNEGO is the acronym of the protocol by which HTTP clients can authenticate with a web site using Kerberos. This allows the client to identify and authenticate itself to a web site or a web service. SPNEGO is supported by

- The standard browsers, to different levels of pain of use
- `curl` on the command line
- `java.net.URL`

The final point is key: it can be used programmatically in Java, so used by REST client applications to authenticate with a remote Web Service.

Exactly how the Java runtime implements its SPNEGO authentication is a mystery to all. Unlike, say Hadoop IPC, where the entire authentication code has been implemented by people whose email addresses you can identify from the change log and so ask hard questions, what the JDK does is a black hole.

The sole source of information is the JDK source, and anything which IDE decompilers can add if you end up stepping in to vendor-specific classes.

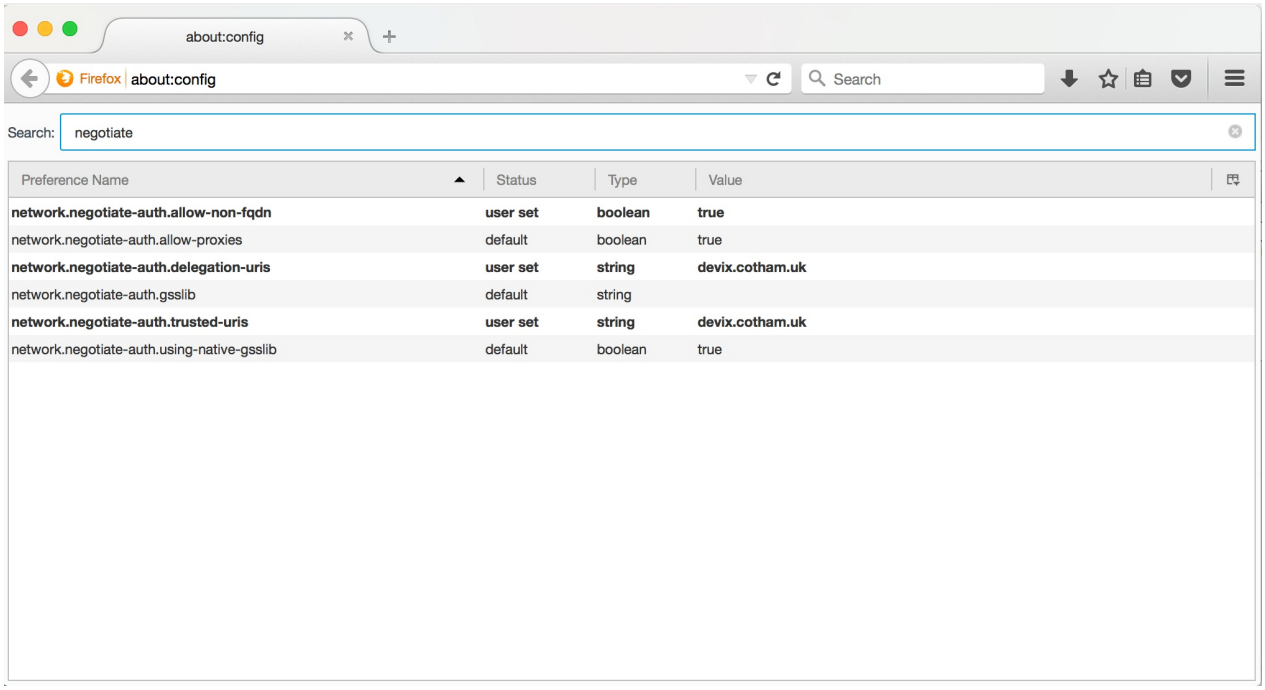
There is one readme file hidden in the test documentation.

Configuring Firefox to use SPNEGO

Firefox is the easiest browser to set up with SPNEGO support, as it is done in `about:config` and then persisted Here are the settings for a local VM, a VM which has an entry in the `/etc/hosts`:

```
192.168.1.134 devix.cotham.uk devix
```

This hostname is then listed in firefox's config as a URL to trust.



Chrome and SPNEGO

Historically, Chrome needed to be configured on the command line to use SPNEGO, which was complicated to the point of unusability.

Fortunately, there is a better way, Chromium Policy Templates.

See Google Chrome, SPNEGO, and WebHDFS on Hadoop

Why not use Apache HTTP Components?

The Apache HTTP Client/http components have a well-deserved reputation for being great libraries to work with remote HTTP servers.

Should you use them for Kerberos/SPNEGO authenticated applications?

No.

As the documentation says.

> There are a lot of issues that can happen but if lucky it'll work without too much of a problem. It should also provide some output to debug with.

That's not the kind of information you want to read when working out how to talk to a SPNEGO-authed server. In its favour: it's being honest, and "if you are lucky it will work" could probably be used to describe the entire JDK Kerberos libraries. However: they are being honest; it hasn't been a good experience trying to get Jersey to work with secure REST endpoints using the http components as the back end.

Don't waste time or make things worse: go with the JDK libraries from the outset

Jersey SPNEGO support

There is not enough space to describe how to do this; examine the code.

Apache CXF

I've been told that Apache CXF supports SPNEGO —but not yet experimented with it. Any insight here would be welcome.

SPNEGO REST clients in the Hadoop codebase

The first point to note is that there is more than one piece of code adding SPNEGO support to Jersey in the Hadoop libraries -there are at least three slightly different ones.

Code in:

WebHDFS

In the HDFS codebase.

KMS

This is probably the best starting point for any REST client which does not want to address the challenge of delegation token renewal.

YARN timeline server

This handles delegation token renewal by supporting an explicit renew-token REST operation. A scheduled operation in the client is used to issue this call regularly and so keep the token up to date.

Implementing a SPNEGO-authenticated endpoint

This isn't as hard as you think: you need to add an authentication filter

Fun facts

- HADOOP-10850 The Java SPNEGO code will blacklist any host where initializing the negotiation code fails. The blacklist lasts the duration of the JVM.

Adding Delegation token renewal

Simplest way to do this is to have something in the background which makes `OPTIONS` or `HEAD` calls of the endpoint (the former relies on `OPTIONS` not being disabled, the latter on `HEAD`) being inexpensive.

Supporting custom webauth initializers

Many large organizations implement their own authentication system. This can be a source of "entertainment", that is, if fielding support calls in stack traces which include private modules is considered entertaining.

TODO:

1. How to declare a custom webauth renderer in the RM proxy
2. How to handle it in a client

Identifying and Authenticating callers in Web/REST endpoints

```
private static UserGroupInformation getUser(HttpServletRequest req) {
  String remoteUser = req.getRemoteUser();
  UserGroupInformation callerUGI = null;
  if (remoteUser != null) {
    callerUGI = UserGroupInformation.createRemoteUser(remoteUser);
  }
  return callerUGI;
}
```

Note: the remote caller doesn't have any credentials. The service

This can then be used to process the events

```
@PUT
@Path("/jobs/{jobid}/tasks/{taskid}/attempts/{attemptid}/state")
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response updateJobTaskAttemptState(JobTaskAttemptState targetState,
    @Context HttpServletRequest request, @PathParam("jobid"))
        throws IOException, InterruptedException {
  init();
  UserGroupInformation callerUGI = getUser(request);
  // if the UGI is null, no remote user.
```

YARN and YARN Applications

YARN applications are somewhere where Hadoop authentication becomes some of its most complex.

Anyone writing a YARN application will encounter Hadoop security, and will end up spending time debugging the problems. This is "the price of security".

YARN Service security

YARN Resource Managers (RM) and Node Managers (NM) perform work on behalf of the user.

The NM's

1. `Localize` resources: Download from HDFS or other filesystem into a local directory. This is done using the delegation tokens attached to the container launch context. (For non-HDFS resources, using other credentials such as object store login details in cluster configuratio files)
2. Start the application as the user.

Securing YARN Application Web UIs and REST APIs

YARN provides a straightforward way of giving every YARN application SPNEGO authenticated web pages: it implements SPNEGO authentication in the Resource Manager Proxy. YARN web UI are expected to load the AM proxy filter when setting up its web UI; this filter will redirect all HTTP Requests coming from any host other than the RM Proxy hosts to an RM proxy, to which the client app/browser must re-issue the request. The client will authenticate against the principal of the RM PRoxy (usually `yarn`), and, once authenticated, have its request forwared.

As a result, all client interactions are SPNEGO-authenticated, without the YARN application itself needing any kerberos principal for the clients to authenticate against.

Known weaknesses in this approach are

1. As calls coming from the proxy hosts are not redirected, any application running on those hosts has unrestricted access to the YARN applications. This is why in a secure cluster the proxy hosts must run on cluster nodes which do not run end user code (i.e. not run YARN NodeManagers and hence schedule YARN containers).
2. The HTTP requests between proxy and YARN RM Server are not encrypted.

Securing YARN Application REST APIs

YARN REST APIs running on the same port as the registered web UI of a YARN application are automatically authenticated via SPNEGO authentication in the RM proxy.

Any REST endpoint (and equally, any web UI) brought up on a different port does not support SPNEGO authentication unless implemented in the YARN application itself.

Strategies for token renewal on YARN services

Keytabs for AM and containers

A keytab is provided for the application. This can be done by:

1. Installing it in every cluster node, then providing the path to this in a configuration directory. The keytab must be in a secure directory path, where only the service (and other trusted accounts) can read it.
2. Including the keytab as a resource for the container, relying on the Node Manager localizer to download it from HDFS and store it locally. This avoids the administration task of installing keytabs for specific services. It does require the client to have access to the keytab and as it is uploaded to the distributed filesystem, must be secured through the appropriate path permissions.

This is the strategy adopted by Apache Slider (incubating). Slider also pushes out specified keytabs for deployed applications such as HBase, with the Application Master declaring the HDFS paths to them in its Container Launch Requests.

AM keytab + renewal and forwarding of Delegation Tokens to containers

The Application Master is given the path to a keytab (usually a client-uploaded localized resource), so can stay authenticated with Kerberos. Launched containers are only given delegation tokens. Before a delegation token is due to expire, the processes running in the containers must request new tokens from the Application Master. Obviously, communications between containers and their Application Master must themselves be authenticated, so as to prevent malicious code from requesting the containers from an Application Master.

This is the strategy used by Spark 1.5+. Communications between container processes and the AM is over HTTPS-authenticated Akka channels.

Client-side push of renewed Delegation Tokens

This strategy may be the sole one acceptable to a strict operations team: a client process running on an account holding a kerberos TGT negotiates with all needed cluster services for delegation tokens, tokens which are then pushed out to the Application Master via some RPC interface.

This does require the client process to be re-executed on a regular basis; a cron or OOzie job can do this.

Zookeeper

Apache Zookeeper uses Kerberos + SASL to authenticate callers. The specifics are covered in Zookeeper and SASL

Other than SASL, its access control is all based around secrets "Digests" which are shared between client and server, and sent over the (unencrypted) channel. The Digest is stored in the ZK node; any client which provides the same Digest is considered to be that principal, and so gains those rights. They cannot be relied upon to securely identify callers.

What's generally more troublesome about ZK is that its ACL-based permission scheme is "confusing". Most developers are used to hierarchial permissions, in which the permissions of the parent paths propagate down. If a root directory is world writeable, then anyone with those permissions can implicity manipulate entries below it.

ZK nodes are not like this: permissions are there on a zknode-by-zknode basis.

For extra fun, ZK does not have any notion of groups; you can't have users in specific groups (i.e. 'administrators')

The Hadoop Security book covers the client-side ZK APIs briefly.

Enabling SASL in ZK

SASL is enabled in ZK by setting a system property. While adequate for a server, it's less than convenient when using ZK in an application as it means something very important: you cannot have a non-SASL and a SASL ZK connection at the same time. Although you could create theoretically create one connection, change the system properties and then create the next, Apache Curator, doesn't do this.

```
System.setProperty("zookeeper.sasl.client", "true");
```

As everyone sensible uses Curator to handle transient disconnections and ZK node failover, this isn't practicable. (Someone needs to fix this —volunteers welcome)

Working with ZK

If you want to use ZK in production you have to

1. Remember that even in a secure cluster, parts of the ZK path, including the root `/` znode, are usually world writeable. You may unintentionally be relying on this and be creating insecure paths. (Some of our production tests explicitly check this, BTW).

2. Remember that ZK permissions have to be explicitly asked for: there is no inheritance. Set them for every node you intend to work with.
3. Lock down node permissions in a secure cluster, so that only authenticated users can read secret data or manipulate data which must only be written by specific services. As an example, HBase and Accumulo both publish their binding information to ZK, The YARN Registry has per-user zknode paths set up so that all nodes under `/users/${username}` are implicitly written by the user `$username`, so have their authority.
4. On an insecure cluster, do not try to create an ACL with "the current user" until the user is actually authenticated.
5. If you want administrative accounts to have access to znodes, explicitly set it.

Basic code

```
List<ACL> perms = new ArrayList<>();
if (UserGroupInformation.isSecurityEnabled()) {
  perms(new ACL(ZooDefs.Perms.ALL, ZooDefs.Ids.AUTH_IDS));
  perms.add(new ACL(ZooDefs.Perms.READ,ZooDefs.Ids.ANYONE_ID_UNSAFE));
} else {
  perms.add(new ACL(ZooDefs.Perms.ALL, ZooDefs.Ids.ANYONE_ID_UNSAFE));
}
zk.createPath(path, null, perms, CreateMode.PERSISTENT);
```

Example YARN Registry

In the Hadoop yarn registry, in order to allow admin rights, we added a yarn property to list principals who would be given full access.

To avoiding requiring all configuration files to list the explicit realm to use, we added the concept that if the principal was listed purely as `user@`, rather than `user@REALM`, we'd append the value of `hadoop.registry.kerberos.realm` —and if that value was unset, the realm of the (logged in) caller.

This means that all users of the YARN registry in a secure cluster get znodes with admin access to `yarn`, `mapred` and `hdfs` users in the current Kerberos Realm, unless otherwise configured in the cluster's `core-site.xml`.

```
<property>
  <description>
    Key to set if the registry is secure. Turning it on
    changes the permissions policy from "open access"
    to restrictions on kerberos with the option of
    a user adding one or more auth key pairs down their
    own tree.
  </description>
  <name>hadoop.registry.secure</name>
  <value>false</value>
</property>

<property>
  <description>
    A comma separated list of Zookeeper ACL identifiers with
    system access to the registry in a secure cluster.

    These are given full access to all entries.

    If there is an "@" at the end of a SASL entry it
    instructs the registry client to append the default kerberos domain.
  </description>
  <name>hadoop.registry.system.acls</name>
  <value>sasl:yarn@, sasl:mapred@, sasl:hdfs@</value>
</property>

<property>
  <description>
    The kerberos realm: used to set the realm of
    system principals which do not declare their realm,
    and any other accounts that need the value.

    If empty, the default realm of the running process
    is used.

    If neither are known and the realm is needed, then the registry
    service/client will fail.
  </description>
  <name>hadoop.registry.kerberos.realm</name>
  <value></value>
</property>
```

ZK Client and JAAS

Zookeeper needs a jaas context in SASL mode.

It will actually attempt to fallback to unauthorized if it doesn't get one

ZK's example JAAS client config

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="/path/to/client/keytab"
  storeKey=true
  useTicketCache=false
  principal="yourzookeeperclient";
};
```

And here is one which should work for using your login credentials instead

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=false
  useTicketCache=true
  principal="user@REALM";
  doNotPrompt=true
};
```

How ZK reacts to authentication failures

The ZK server appears to react to a SASL authentication failure by closing the connection *without sending any error back to the client*

This means that for a client, authentication problems surface as connection failures

```
2015-12-15 13:56:30,066 [main] DEBUG zk.CuratorService (zkList(695)) - ls /registry
Exception: `/': Failure of ls() on /: org.apache.zookeeper.KeeperException$ConnectionLossExcep
tion: KeeperErrorCode = ConnectionLoss for /registry: KeeperErrorCode = ConnectionLoss for /re
gistry
2015-12-15 13:56:58,892 [main] ERROR main.ServiceLauncher (error(344)) - Exception: `/': Failu
re of ls() on /: org.apache.zookeeper.KeeperException$ConnectionLossException: KeeperErrorCode
 = ConnectionLoss for /registry: KeeperErrorCode = ConnectionLoss for /registry
org.apache.hadoop.registry.client.exceptions.RegistryIOException: `/': Failure of ls() on /: o
rg.apache.zookeeper.KeeperException$ConnectionLossException: KeeperErrorCode = ConnectionLoss
for /registry: KeeperErrorCode = ConnectionLoss for /registry
  at org.apache.hadoop.registry.client.impl.zk.CuratorService.operationFailure(CuratorService.
java:403)
  at org.apache.hadoop.registry.client.impl.zk.CuratorService.operationFailure(CuratorService.
java:360)
  at org.apache.hadoop.registry.client.impl.zk.CuratorService.zkList(CuratorService.java:701)
  at org.apache.hadoop.registry.client.impl.zk.RegistryOperationsService.list(RegistryOperatio
nsService.java:154)
  at org.apache.hadoop.registry.client.binding.RegistryUtils.statChildren(RegistryUtils.java:2
04)
  at org.apache.slider.client.SliderClient.actionResolve(SliderClient.java:3345)
  at org.apache.slider.client.SliderClient.exec(SliderClient.java:431)
  at org.apache.slider.client.SliderClient.runService(SliderClient.java:323)
  at org.apache.slider.core.main.ServiceLauncher.launchService(ServiceLauncher.java:188)
  at org.apache.slider.core.main.ServiceLauncher.launchServiceRobustly(ServiceLauncher.java:47
5)
  at org.apache.slider.core.main.ServiceLauncher.launchServiceAndExit(ServiceLauncher.java:403
)
  at org.apache.slider.core.main.ServiceLauncher.serviceMain(ServiceLauncher.java:630)
  at org.apache.slider.Slider.main(Slider.java:49)
Caused by: org.apache.zookeeper.KeeperException$ConnectionLossException: KeeperErrorCode = Con
nectionLoss for /registry
  at org.apache.zookeeper.KeeperException.create(KeeperException.java:99)
  at org.apache.zookeeper.KeeperException.create(KeeperException.java:51)
  at org.apache.zookeeper.ZooKeeper.getChildren(ZooKeeper.java:1590)
  at org.apache.curator.framework.imps.GetChildrenBuilderImpl$3.call(GetChildrenBuilderImpl.ja
va:214)
  at org.apache.curator.framework.imps.GetChildrenBuilderImpl$3.call(GetChildrenBuilderImpl.ja
va:203)
  at org.apache.curator.RetryLoop.callWithRetry(RetryLoop.java:107)
  at org.apache.curator.framework.imps.GetChildrenBuilderImpl.pathInForeground(GetChildrenBuil
derImpl.java:200)
  at org.apache.curator.framework.imps.GetChildrenBuilderImpl.forPath(GetChildrenBuilderImpl.j
ava:191)
  at org.apache.curator.framework.imps.GetChildrenBuilderImpl.forPath(GetChildrenBuilderImpl.j
ava:38)
  at org.apache.hadoop.registry.client.impl.zk.CuratorService.zkList(CuratorService.java:698)
  ... 10 more
```

If you can telnet into the ZK host & port then ZK is up, but rejecting authenticated calls.

You need to go to the server logs (e.g. `/var/log/zookeeper/zookeeper.out`) to see what actually went wrong:

```
2015-12-15 13:56:30,995 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactor
y@197] - Accepted socket connection from /192.168.56.1:55882
2015-12-15 13:56:31,004 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@868
] - Client attempting to establish new session at /192.168.56.1:55882
2015-12-15 13:56:31,031 - INFO  [SyncThread:0:ZooKeeperServer@617] - Established session 0x151
a5e1345d0003 with negotiated timeout 40000 for client /192.168.56.1:55882
2015-12-15 13:56:31,181 - WARN  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@969
] - Client failed to SASL authenticate: javax.security.sasl.SaslException:
 GSS initiate failed [Caused by GSSException: Failure unspecified at GSS-API level (Mechanism
level: Specified version of key is not available (44))]
2015-12-15 13:56:31,181 - WARN  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@975
] - Closing client connection due to SASL authentication failure.
2015-12-15 13:56:31,182 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@1007]
 - Closed socket connection for client /192.168.56.1:55882 which had
 sessionid 0x151a5e1345d0003
2015-12-15 13:56:31,182 - ERROR [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@178]
- Unexpected Exception:
java.nio.channels.CancelledKeyException
        at sun.nio.ch.SelectionKeyImpl.ensureValid(SelectionKeyImpl.java:73)
        at sun.nio.ch.SelectionKeyImpl.interestOps(SelectionKeyImpl.java:77)
        at org.apache.zookeeper.server.NIOServerCnxn.sendBuffer(NIOServerCnxn.java:151)
        at org.apache.zookeeper.server.NIOServerCnxn.sendResponse(NIOServerCnxn.java:1081)
        at org.apache.zookeeper.server.ZooKeeperServer.processPacket(ZooKeeperServer.java:936)
        at org.apache.zookeeper.server.NIOServerCnxn.readRequest(NIOServerCnxn.java:373)
        at org.apache.zookeeper.server.NIOServerCnxn.readPayload(NIOServerCnxn.java:200)
        at org.apache.zookeeper.server.NIOServerCnxn.doIO(NIOServerCnxn.java:244)
        at org.apache.zookeeper.server.NIOServerCnxnFactory.run(NIOServerCnxnFactory.java:208)
        at java.lang.Thread.run(Thread.java:745)
2015-12-15 13:56:31,186 - WARN  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@346]
- Exception causing close of session 0x151a5e1345d0003
 due to java.nio.channels.CancelledKeyException
2015-12-15 13:56:32,540 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactor
y@197] - Accepted socket connection from /192.168.56.1:55883
2015-12-15 13:56:32,542 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@861
] - Client attempting to renew session 0x151a5e1345d0003 at /192.168.56.1:55883
2015-12-15 13:56:32,543 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@617
] - Established session 0x151a5e1345d0003 with negotiated timeout 40000 for
 clie nt /192.168.56.1:55883
2015-12-15 13:56:32,547 - WARN  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:ZooKeeperServer@969
] - Client failed to SASL authenticate: javax.security.sasl.SaslException:
 GSS initiate failed [Caused by GSSException: Failure unspecified at GSS-API level (Mechanism
level: Specified version of key is not available (44))]
```

Troubleshooting ZK

There's a nice list from Jeremy Custenborder of what to do to troubleshoot ZK on ZOOKEEPER-2345

Testing

Writing Kerberos Tests with MiniKDC

The Hadoop project has an in-VM Kerberos Controller for tests, MiniKDC, which is packaged as its own JAR for downstream use. The core of this code actually comes from the Apache Directory Services project.

Testing against Kerberized Hadoop clusters

This is not actually the hardest form of testing; getting the MiniKDC working holds that honour. It does have some pre-requisites.

1. Everyone running the tests has set up a Hadoop cluster/single VM with Kerberos enabled.
2. The software project has a test runner capable of deploying applications into a remote Hadoop cluster/VM and assessing the outcome.

It's primarily the test runner which matters. Without that you cannot do functional tests against any Hadoop cluster. However, once you have such a test runner, you have a powerful tool: the ability to run tests against real Hadoop clusters, rather than simply minicluster and miniKDC tests which, while better than nothing, are unrealistic.

If this approach is so powerful, why not bypass the minicluster tests altogether?

1. Minicluster tests are easier to run. Build tools can run them; Jenkins can trivially run them as part of test runs.
2. The current state of the cluster affects the outcome of the tests. Its useful not only to have tests tear down properly, but for the setup phase of each test suite to verify that the cluster is in the valid initial state/get it into that state. For YARN applications, this generally means that there are no running applications in the cluster.
3. Setup often includes the overhead of copying files into HDFS. As the secure user.
4. The host launching the tests needs to be setup with kinit/keytabs.
5. Retrieving and interpreting the results is harder. Often it involved manually going to the YARN RM to get through to the logs (assuming that yarn-site is configured to preserve them), and/or collecting other service logs.
6. If you are working with nightly builds of Hadoop, VM setup needs to be automated.
7. Unless you can mandate and verify that all developers run the tests against secure clusters, they may not get run by everyone.
8. The tests can be slow.
9. Fault injection can be harder.

Overall, tests are less deterministic.

In the slider project, different team members have different test clusters, Linux and Windows, Kerberized and non-Kerberized, Java-7 and Java 8. This means that test runs do test a wide set of configurations without requiring every developer to have a VM of every form. The Hortonworks QE team also run these tests against the nightly HDP stack builds, catching regressions in both the HDP stack and in the Slider project.

For fault injection the Slider Application Master has an integral "chaos monkey" which can be configured to start after a defined period of time, then randomly kill worker containers and/or the application master. This is used in conjunction with the functional tests of the deployed applications to verify that they remain resilient to failure. When tests do fail, we are left with the problem of retrieving the logs and identifying problems from them. The QE test runs do collect all the logs from all the services across the test clusters —but this still leaves the problem of trying to correlate events from the logs across the machines.

Tuning a Hadoop cluster for aggressive token timeouts

Kinit

You can ask for a limited lifespan of a ticket when logging in on the console

```
kinit -l 15m
```

KDC

Here is an example `/etc/krb5.conf` which limits the lifespan of a ticket to 1h

```
[libdefaults]

  default_realm = DEVIX
  renew_lifetime = 2h
  forwardable = true

  ticket_lifetime = 1h
  dns_lookup_realm = false
  dns_lookup_kdc = false

[realms]

  DEVIX = {
    kdc = devix
    admin_server = devix
  }
```

The KDC host here, `devix` is a Linux VM. Turning off the DNS lookups avoids futile attempts to work with DNS/rDNS.

Hadoop tokens

TODO: Table of properties for hdfs, yarn, hive, ... listing token timeout properties

Enabling Kerberos for different Hadoop components

Core Hadoop

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

HBase

```
<property>
  <name>hbase.security.authentication</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.regionserver.kerberos.principal</name>
  <value>hbase/_HOST@YOUR-REALM.COM</value>
</property>
<property>
  <name>hbase.regionserver.keytab.file</name>
  <value>/etc/hbase/conf/keytab.krb5</value>
</property>
<property>
  <name>hbase.master.kerberos.principal</name>
  <value>hbase/_HOST@YOUR-REALM.COM</value>
</property>
<property>
  <name>hbase.master.keytab.file</name>
  <value>/etc/hbase/conf/keytab.krb5</value>
</property>
<property>
<name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.token.TokenProvider</value>
</property>
```

Tips

1. Use `kdestroy` to destroy your local ticket cache. Do this to ensure that code running locally is reading data in from a nominated keytab and not falling back to the user's ticket cache.
2. VMs: Make sure the clocks between VM and host are in sync; it's easy for a VM clock to drift when suspended and resumed.
3. VMs: Make sure that all hosts are listed in the host table, so that hostname lookup works.
4. Try to log in from a web browser without SPNEGO enabled; this will catch any WebUI that wasn't actually authenticating callers.
5. Try to issue RPC and REST calls from an unauthenticated client, and from a user that is not granted access rights.
6. YARN applications: verify that REST/Web requests against the real app URL (which can be determined from the YARN application record), are redirected to the RM proxy (i.e. that all GET calls result in a 30x redirect). If this does not take place, it means that the RM-hosted SPNEGO authentication layer can be bypassed.

Low-Level Secrets

> Among the agonies of these after days is that chief of torments — inarticulateness. What I learned and saw in those hours of impious exploration can never be told — for want of symbols or suggestions in any language.
>
> *The Shunned House, HP Lovecraft, 1924.*

`krb5.conf` and system property `java.security.krb5.conf`

You can do two things when setting up the JVM binding to the `krb5.conf` kerberos binding file.

1. Change the realm with System Property `java.security.krb5.realm`

This system property sets the realm for the kerberos binding. This allows you to use a different one from the default in the krb5.conf file.

Examples

```
-Djava.security.krb5.realm=PRODUCTION

System.setProperty("java.security.krb5.realm", "DEVELOPMENT");
```

The JVM property MUST be set before UGI is initialized.

2. Switch to an alternate `krb5.conf` *file.*

The JVM kerberos operations are configured via the `krb5.conf` file specified in the JVM option `java.security.krb5.conf` which can be done on the JVM command line, or inside the JVM

```
System.setProperty("java.security.krb5.conf", krbfilepath);
```

The JVM property MUST be set before UGI is initialized.

Notes

- use double backslash to escape paths on Windows platforms, e.g. `C:\\keys\\key1`, or `\\\\server4\\shared\\tokens`
- Different JVMs (e.g. IBM JVM) want different fields in their `krb5.conf` file. How can you tell? Kerberos will fail with a message

JVM Kerberos Library logging

You can turn Kerberos low-level logging on

```
-Dsun.security.krb5.debug=true
```

This doesn't come out via Log4J, or `java.util logging;` it just comes out on the console. Which is somewhat inconvenient —but bear in mind they are logging at a very low level part of the system. And it does at least log. If you find yourself down at this level you are in trouble. Bear that in mind.

JVM SPNEGO Logging

If you want to debug what is happening in SPNEGO, another system property lets you enable this:

```
-Dsun.security.spnego.debug=true
```

You can ask for both of these in the `HADOOP_OPTS` environment variable

```
export HADOOP_OPTS=-Dsun.security.krb5.debug=true -Dsun.security.spnego.debug=true
```

Hadoop-side JAAS debugging

Set the env variable `HADOOP_JAAS_DEBUG` to true and UGI will set the "debug" flag on any JAAS files it creates.

You can do this on the client, before issuing a `hadoop`, `hdfs` or `yarn` command, and set it in the environment script of a YARN service to turn it on there.

```
export HADOOP_JAAS_DEBUG=true
```

On the next Hadoop command, you'll see a trace like

```
    [UnixLoginModule]: succeeded importing info:
      uid = 503
      gid = 20
      supp gid = 20
      supp gid = 501
      supp gid = 12
      supp gid = 61
      supp gid = 79
      supp gid = 80
      supp gid = 81
      supp gid = 98
      supp gid = 399
      supp gid = 33
      supp gid = 100
      supp gid = 204
      supp gid = 395
      supp gid = 398
Debug is  true storeKey false useTicketCache true useKeyTab false doNotPrompt true ticketCache
 is null isInitiator true KeyTab is null refreshKrb5Config is false principal is null tryFirst
Pass is false useFirstPass is false storePass is false clearPass is false
Acquire TGT from Cache
Principal is stevel@COTHAM
    [UnixLoginModule]: added UnixPrincipal,
        UnixNumericUserPrincipal,
        UnixNumericGroupPrincipal(s),
       to Subject
Commit Succeeded

    [UnixLoginModule]: logged out Subject
    [Krb5LoginModule]: Entering logout
    [Krb5LoginModule]: logged out Subject
    [UnixLoginModule]: succeeded importing info:
      uid = 503
      gid = 20
      supp gid = 20
      supp gid = 501
      supp gid = 12
      supp gid = 61
      supp gid = 79
      supp gid = 80
      supp gid = 81
      supp gid = 98
      supp gid = 399
      supp gid = 33
      supp gid = 100
      supp gid = 204
      supp gid = 395
      supp gid = 398
Debug is  true storeKey false useTicketCache true useKeyTab false doNotPrompt true ticketCache
 is null isInitiator true KeyTab is null refreshKrb5Config is false principal is null tryFirst
Pass is false useFirstPass is false storePass is false clearPass is false
Acquire TGT from Cache
Principal is stevel@COTHAM
    [UnixLoginModule]: added UnixPrincipal,
        UnixNumericUserPrincipal,
        UnixNumericGroupPrincipal(s),
       to Subject
Commit Succeeded
```

OS-level Kerberos Debugging

Starting MIT Kerberos v1.9, Kerberos libraries introduced a debug option which is a boon to any person breaking his/her head over a nasty Kerberos issue. It is also a good way to understand how does Kerberos library work under the hood. User can set an environment variable called `KRB5_TRACE` to a filename or to `/dev/stdout` and Kerberos programs (like kinit, klist and kvno etc.) as well as Kerberos libraries (libkrb5*) will start printing more interesting details.

This is a very powerfull feature and can be used to debug any program which uses Kerberos libraries (e.g. CURL). It can also be used in conjunction with other debug options like `HADOOP_JAAS_DEBUG` and `sun.security.krb5.debug` .

```
export KRB5_TRACE=/tmp/kinit.log
```

After setting this up in the terminal, the kinit command will produce something similar to this:

```
# kinit admin/admin
Password for admin/admin@MYKDC.COM:

# cat /tmp/kinit.log
[5709] 1488484765.450285: Getting initial credentials for admin/admin@MYKDC.COM
[5709] 1488484765.450556: Sending request (200 bytes) to MYKDC.COM
[5709] 1488484765.450613: Resolving hostname sandbox.hortonworks.com
[5709] 1488484765.450954: Initiating TCP connection to stream 172.17.0.2:88
[5709] 1488484765.451060: Sending TCP request to stream 172.17.0.2:88
[5709] 1488484765.461681: Received answer from stream 172.17.0.2:88
[5709] 1488484765.461724: Response was not from master KDC
[5709] 1488484765.461752: Processing preauth types: 19
[5709] 1488484765.461764: Selected etype info: etype aes256-cts, salt "(null)", params ""
[5709] 1488484765.461767: Produced preauth for next request: (empty)
[5709] 1488484765.461771: Salt derived from principal: MYKDC.COMadminadmin
[5709] 1488484765.461773: Getting AS key, salt "MYKDC.COMadminadmin", params ""
[5709] 1488484770.985461: AS key obtained from gak_fct: aes256-cts/93FB
[5709] 1488484770.985518: Decrypted AS reply; session key is: aes256-cts/2C56
[5709] 1488484770.985531: FAST negotiation: available
[5709] 1488484770.985555: Initializing FILE:/tmp/krb5cc_0 with default princ admin/admin@MYKDC
.COM
[5709] 1488484770.985682: Removing admin/admin@MYKDC.COM -> krbtgt/MYKDC.COM@MYKDC.COM from FI
LE:/tmp/krb5cc_0
[5709] 1488484770.985688: Storing admin/admin@MYKDC.COM -> krbtgt/MYKDC.COM@MYKDC.COM in FILE:
/tmp/krb5cc_0
[5709] 1488484770.985742: Storing config in FILE:/tmp/krb5cc_0 for krbtgt/MYKDC.COM@MYKDC.COM:
 fast_avail: yes
[5709] 1488484770.985758: Removing admin/admin@MYKDC.COM -> krb5_ccache_conf_data/fast_avail/k
rbtgt\/MYKDC.COM\@MYKDC.COM@X-CACHECONF: from FILE:/tmp/krb5cc_0
[5709] 1488484770.985763: Storing admin/admin@MYKDC.COM -> krb5_ccache_conf_data/fast_avail/kr
btgt\/MYKDC.COM\@MYKDC.COM@X-CACHECONF: in FILE:/tmp/krb5cc_0
```

KRB5CCNAME

The environment variable `KRB5CCNAME` As the docs say:

If the KRB5CCNAME environment variable is set, its value is used to name the default ticket cache.

IP addresses vs. Hostnames

Kerberos principals are traditionally defined with hostnames of the form `hbase@worker3/EXAMPLE.COM`, not `hbase/10.10.15.1/EXAMPLE.COM`

The issue of whether Hadoop should support IP addresses has been raised HADOOP-9019 & HADOOP-7510 Current consensus is no: you need DNS set up, or at least a consistent and valid /etc/hosts file on every node in the cluster.

Windows

1. Windows does not reverse-DNS 127.0.0.1 to localhost or the local machine name; this can cause problems with MiniKDC tests in Windows, where adding a `user/127.0.0.1@REALM` principal will be needed example.
2. Windows hostnames are often upper case.

Kerberos's defences against replay attacks

From the javadocs of `org.apache.hadoop.ipc.Client.handleSaslConnectionFailure()`:

```
/**
 * If multiple clients with the same principal try to connect to the same
 * server at the same time, the server assumes a replay attack is in
 * progress. This is a feature of kerberos. In order to work around this,
 * what is done is that the client backs off randomly and tries to initiate
 * the connection again.
 */
```

That's a good defence on the surface, "multiple connections from same principal == attack", which doesn't scale to Hadoop clusters. Hence the sleep. It is also why large Hadoop clusters define a different principal for every service/host pair in the keytab, ensuring giving the principal for the HDFS blockserver on host1 an identity such as `hdfs/host1`, for host 2 `hdfs/host2`, etc. When a cluster is completely restarted, instead of the same principal trying to authenticate from 1000+ hosts, only the HDFS services on a single node try to authenticate as the same principal.

Asymmetric Kerberos Realm Trust

It is possible to configure Kerberos KDCs such that one realm, e.g `"hadoop-kdc"` can trust principals from a remote realm -but for that remote realm not to trust the principals from that `"hadoop-kdc"` realm. What does that permit? It means that a Hadoop-cluster-specific KDC can be

created and configured to trust principals from the enterprise-wide (Active-Directory Managed) KDC infrastructure. The hadoop cluster KDC will contain the principals for the various services, with these exported into keytabs.

As a result, even if the keytabs are compromised, *they do not grant any access to and enterprise-wide kerberos-authenticated services.

Error Messages to Fear

> The oldest and strongest emotion of mankind is fear, and the oldest and strongest kind of fear is fear of the unknown.
>
> *Supernatural Horror in Literature, HP Lovecraft, 1927.*

Security error messages appear to take pride in providing limited information. In particular, they are usually some generic `IOException` wrapping a generic security exception. There is some text in the message, but it is often `Failure unspecified at GSS-API level`, which means "something went wrong".

Generally a stack trace with UGI in it is a security problem, *though it can be a network problem surfacing in the security code*.

The underlying causes of problems are usually the standard ones of distributed systems: networking and configuration.

Some of the OS-level messages are covered in Oracle's Troubleshooting Kerberos docs.

- Common Kerberos Error Messages (A-M)
- Common Kerberos Error Messages (N-Z)

Here are some of the common ones seen in Hadoop stack traces *and what we think are possible causes*

That is: on one or more occasions, the listed cause was the one which, when corrected, made the stack trace go away.

`GSS initiate failed` —no further details provided

```
WARN  ipc.Client (Client.java:run(676)) - Couldn't setup connection for rm@EXAMPLE.COM to /172
.22.97.127:8020
org.apache.hadoop.ipc.RemoteException(javax.security.sasl.SaslException): GSS initiate failed
  at org.apache.hadoop.security.SaslRpcClient.saslConnect(SaslRpcClient.java:375)
  at org.apache.hadoop.ipc.Client$Connection.setupSaslConnection(Client.java:558)
```

This is widely agreed to be one of the most useless of error messages you can see. The only ones that are worse than this are those which disguise a Kerberos problem, such as when ZK closes the connection rather than saying "it couldn't authenticate".

If you see this connection, work out which service it was trying to talk to —and look in its logs instead. Maybe, just maybe, there will be more information there.

`Server not found in Kerberos database (7)` or `service ticket not found in the subject`

- DNS is a mess and your machine does not know its own name.
- Your machine has a hostname, but the service principal is a `/_HOST` wildcard and the hostname is not one there's an entry in the keytab for.

We've seen this in the stdout of a NN

```
TGS_REQ { ... }UNKNOWN_SERVER: authtime 0, hdfs@EXAMPLE.COM for krbtgt/NOVALOCAL@EXAMPLE.COM,
Server not found in Kerberos database
```

Variant as a client talking to a remote service over an HTTPS connection and with SPNEGO auth.

```
2019-01-28 11:44:13,345 [main] INFO  (DurationInfo.java:<init>(70)) - Starting: Fetching statu
s from https://server-1542663976389-48660-01-000003.example.org.site:8443/gateway/v1/
Debug is  true storeKey false useTicketCache true useKeyTab false doNotPrompt true ticketCache
 is null isInitiator true KeyTab is null refreshKrb5Config is false principal is null tryFirst
Pass is false useFirstPass is false storePass is false clearPass is false
Acquire TGT from Cache
Principal is qa@EXAMPLE.COM
Commit Succeeded

Search Subject for SPNEGO INIT cred (<<DEF>>, sun.security.jgss.spnego.SpNegoCredElement)
Search Subject for Kerberos V5 INIT cred (<<DEF>>, sun.security.jgss.krb5.Krb5InitCredential)
2019-01-28 11:44:14,642 [main] WARN  auth.HttpAuthenticator (HttpAuthenticator.java:generateAu
thResponse(207)) - NEGOTIATE authentication error: No valid credentials provided (Mechanism le
vel: No valid credentials provided (Mechanism level: Server not found in Kerberos database (7)
 - LOOKING_UP_SERVER))
```

After printing this warning, the application continued to make the HTTP Reqeust (using httpclient BTW) and then fail with 401 Unauth. That is: the failure to negotiate didn't tricker the immediate failure with a useful message, but simply downgraded to a 401 unauth, which is so broad it's not useful.

`No valid credentials provided (Mechanism level: Illegal key size)]`

Your JVM doesn't have the extended cryptography package and can't talk to the KDC. Switch to openjdk or go to your JVM supplier (Oracle, IBM) and download the JCE extension package, and install it in the hosts where you want Kerberos to work.

`Encryption type AES256 CTS mode with HMAC SHA1-96 is not supported/enabled`

```
[javax.security.sasl.SaslException: GSS initiate failed
[Caused by GSSException: Failure unspecified at GSS-API level
(Mechanism level: Encryption type AES256 CTS mode with HMAC SHA1-96 is not supported/enabled)]
]
```

This has surfaced in the distant past.

Assume it means the same as above: the JVM doesn't have the JCE JAR installed.

`No valid credentials provided (Mechanism level: Failed to find any Kerberos tgt`

This may appear in a stack trace starting with something like:

```
javax.security.sasl.SaslException:
GSS initiate failed [Caused by GSSException:
No valid credentials provided (Mechanism level: Failed to find any Kerberos tgt)]
```

It's very common, and essentially means "you weren't authenticated"

Possible causes:

1. You aren't logged in via `kinit`.
2. You have logged in with `kinit`, but the tickets you were issued with have expired.
3. Your process was issued with a ticket, which has now expired.
4. You did specify a keytab but it isn't there or is somehow otherwise invalid
5. You don't have the Java Cryptography Extensions installed.
6. The principal isn't in the same realm as the service, so a matching TGT cannot be found. That is: you have a TGT, it's just for the wrong realm.
7. Your Active Directory tree has the same principal in more than one place in the tree.
8. Your cached ticket list has been contaminated with a realmless-ticket, and the JVM is now unhappy. (See "The Principal With No Realm")
9. The program you are running may be trying to log in with a keytab, but it's got a Hadoop `FileSystem` instance *before that login took place*. Even if the service is not logged in. that FS instance is unauthenticated.

Apache HBase also has an unintentionally aggravating addition to the generic GSSException which is likely to further drive the user into madness.

```
ipc.AbstractRpcClient – SASL authentication failed.
The most likely cause is missing or invalid credentials. Consider ‘kinit’.
javax.security.sasl.SaslException: GSS initiate failed
[Caused by GSSException: No valid credentials provided (Mechanism level: Failed to find any Ke
rberos tgt)]
```

`Failure unspecified at GSS-API level (Mechanism level: Checksum failed)`

One of the classics

1. The password is wrong. A `kinit` command doesn't send the password to the KDC —it sends some hashed things to prove to the KDC that the caller has the password. If the password is wrong, so is the hash, hence an error about checksums.
2. There was a keytab, but it didn't work: the JVM has fallen back to trying to log in as the user.

3. Your keytab contains an old version of the keytab credentials, and cannot parse the information coming from the KDC, as it lacks the up to date credentials.
4. SPENGO/REST: Kerberos is very strict about hostnames and DNS; this can somehow trigger the problem. http://stackoverflow.com/questions/12229658/java-spnego-unwanted-spn-canonicalization;
5. SPENGO/REST: Java 8 behaves differently from Java 6 and 7 which can cause problems HADOOP-11628.

`javax.security.auth.login.LoginException: No password provided`

When this surfaces in a server log, it means the server couldn't log in as the user. That is, there isn't an entry in the supplied keytab for that user and the system (obviously) doesn't want to fall back to user-prompted password entry.

Some of the possible causes

- The wrong keytab was specified.
- The configuration key names used for specifying keytab or principal were wrong.
- There isn't an entry in the keytab for the user.
- The spelling of the principal is wrong.
- The hostname of the machine doesn't match that of a user in the keytab, so a match of `service/host` fails.

Ideally, services list the keytab and username at fault here. In a less than ideal world —that is the one we live in— things are sometimes less helpful

Here, for example, is a Zookeeper trace, saying it is the user `null` that is at fault.

```
2015-12-15 17:16:23,517 - WARN  [main:SaslServerCallbackHandler@105] - No password found for u
ser: null
2015-12-15 17:16:23,536 - ERROR [main:ZooKeeperServerMain@63] - Unexpected exception, exiting
abnormally
java.io.IOException: Could not configure server because SASL configuration did
not allow the ZooKeeper server to authenticate itself properly: javax.security.auth.login.Logi
nException: No password provided
  at org.apache.zookeeper.server.ServerCnxnFactory.configureSaslLogin(ServerCnxnFactory.java:2
07)
  at org.apache.zookeeper.server.NIOServerCnxnFactory.configure(NIOServerCnxnFactory.java:87)
  at org.apache.zookeeper.server.ZooKeeperServerMain.runFromConfig(ZooKeeperServerMain.java:11
1)
  at org.apache.zookeeper.server.ZooKeeperServerMain.initializeAndRun(ZooKeeperServerMain.java
:86)
  at org.apache.zookeeper.server.ZooKeeperServerMain.main(ZooKeeperServerMain.java:52)
  at org.apache.zookeeper.server.quorum.QuorumPeerMain.initializeAndRun(QuorumPeerMain.java:11
6)
  at org.apache.zookeeper.server.quorum.QuorumPeerMain.main(QuorumPeerMain.java:78)
```

`javax.security.auth.login.LoginException: Unable to obtain password from user`

Believed to be the same as the `No password provided`

```
Exception in thread "main" java.io.IOException:
   Login failure for alice@REALM from keytab /etc/security/keytabs/spark.headless.keytab:
  javax.security.auth.login.LoginException: Unable to obtain password from user
  at org.apache.hadoop.security.UserGroupInformation.loginUserFromKeytab(UserGroupInformation.
java:962)
  at org.apache.spark.deploy.SparkSubmit$.prepareSubmitEnvironment(SparkSubmit.scala:564)
  at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:154)
  at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:121)
  at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
Caused by: javax.security.auth.login.LoginException: Unable to obtain password from user
  at com.sun.security.auth.module.Krb5LoginModule.promptForPass(Krb5LoginModule.java:856)
  at com.sun.security.auth.module.Krb5LoginModule.attemptAuthentication(Krb5LoginModule.java:7
19)
```

The JVM Kerberos code needs to have the password for the user to login to kerberos with, but Hadoop has told it "don't ask for a password"', so the JVM raises an exception.

Root causes should be the same as for the other message.

`failure to login using ticket cache file`

You aren't logged via `kinit`, the application isn't configured to use a keytab. So: no ticket, no authentication, no access to cluster services.

you can use `klist -v` to show your current ticket cache

fix: log in with `kinit`

`Clock skew too great`

```
GSSException: No valid credentials provided
(Mechanism level: Attempt to obtain new INITIATE credentials failed! (null))
 . . . Caused by: javax.security.auth.login.LoginException: Clock skew too great

GSSException: No valid credentials provided (Mechanism level: Clock skew too great (37) - PROC
ESS_TGS

kinit: krb5_get_init_creds: time skew (343) larger than max (300)
```

This comes from the clocks on the machines being too far out of sync.

This can surface if you are doing Hadoop work on some VMs and have been suspending and resuming them; they've lost track of when they are. Reboot them.

If it's a physical cluster, make sure that your NTP daemons are pointing at the same NTP server, one that is actually reachable from the Hadoop cluster. And that the timezone settings of all the hosts are consistent.

`KDC has no support for encryption type`

This crops up on the MiniKDC if you are trying to be clever about encryption types. It doesn't support many.

`GSSException: No valid credentials provided (Mechanism level: Fail to create credential. (63) - No service creds)`

Rarely seen. Switching kerberos to use TCP rather than UDP makes it go away

In `/etc/krb5.conf`:

```
[libdefaults]
  udp_preference_limit = 1
```

Note also UDP is a lot slower to time out.

`Receive timed out`

Usually in a stack trace like

```
Caused by: java.net.SocketTimeoutException: Receive timed out
  at java.net.PlainDatagramSocketImpl.receive0(Native Method)
  at java.net.AbstractPlainDatagramSocketImpl.receive(AbstractPlainDatagramSocketImpl.java:146
)
  at java.net.DatagramSocket.receive(DatagramSocket.java:816)
  at sun.security.krb5.internal.UDPClient.receive(NetClient.java:207)
  at sun.security.krb5.KdcComm$KdcCommunication.run(KdcComm.java:390)
  at sun.security.krb5.KdcComm$KdcCommunication.run(KdcComm.java:343)
  at java.security.AccessController.doPrivileged(Native Method)
  at sun.security.krb5.KdcComm.send(KdcComm.java:327)
  at sun.security.krb5.KdcComm.send(KdcComm.java:219)
  at sun.security.krb5.KdcComm.send(KdcComm.java:191)
  at sun.security.krb5.KrbAsReqBuilder.send(KrbAsReqBuilder.java:319)
  at sun.security.krb5.KrbAsReqBuilder.action(KrbAsReqBuilder.java:364)
  at com.sun.security.auth.module.Krb5LoginModule.attemptAuthentication(Krb5LoginModule.java:7
35)
```

This means the UDP socket awaiting a response from KDC eventually gave up.

- The hostname of the KDC is wrong
- The IP address of the KDC is wrong
- There's nothing at the far end listening for requests.
- A firewall on either client or server is blocking UDP packets

Kerberos waits ~90 seconds before timing out, which is a long time to notice there's a problem.

Switch to TCP —at the very least, it will fail faster.

`javax.security.auth.login.LoginException: connect timed out`

Happens when the system is set up to use TCP as an authentication channel, and the far end KDC didn't respond in time.

- The hostname of the KDC is wrong
- The IP address of the KDC is wrong
- There's nothing at the far end listening for requests.
- A firewall somewhere is blocking TCP connections

`GSSException: No valid credentials provided (Mechanism level: Connection reset)`

We've seen this triggered in Hadoop tests after the MiniKDC through an exception; its thread exited and hence the Kerberos client got a connection error.

When you see this assume network connectivity problems, or something up at the KDC itself.

`Principal not found`

The hostname is wrong (or there is more than one hostname listed with different IP addresses) and so a principal of the form `user/_HOST@REALM` is coming back with the wrong host, and the KDC doesn't find it.

See the comments above about DNS for some more possibilities.

`Defective token detected (Mechanism level: GSSHeader did not find the right tag)`

Seen during SPNEGO Authentication: the token supplied by the client is not accepted by the server.

This apparently surfaces in Java 8 version 8u40; if Kerberos server doesn't support the first authentication mechanism which the client offers, then the client fails. Workaround: don't use those versions of Java.

This is now acknowledged by Oracle and has been fixed in 8u60.

`Specified version of key is not available (44)`

```
Client failed to SASL authenticate:
  javax.security.sasl.SaslException:
    GSS initiate failed [Caused by GSSException: Failure unspecified at GSS-API level
     (Mechanism level: Specified version of key is not available (44))]
```

The meaning of this message —or how to fix it— is a mystery to all.

There is some tentative coverage in Stack Overflow

One possibility is that the keys in your keytab have expired. Did you know that can happen? It does. One day your cluster works happily. The next your client requests are failing, with this message surfacing in the logs.

```
$ klist -kt zk.service.keytab
Keytab name: FILE:zk.service.keytab
KVNO Timestamp         Principal
---- ----------------- --------------------------------------------------------
   5 12/16/14 11:46:05 zookeeper/devix.cotham.uk@COTHAM
   5 12/16/14 11:46:05 zookeeper/devix.cotham.uk@COTHAM
   5 12/16/14 11:46:05 zookeeper/devix.cotham.uk@COTHAM
   5 12/16/14 11:46:05 zookeeper/devix.cotham.uk@COTHAM
```

One thing to see there is the version number in the KVNO table.

Oracle describe the JRE's handling of version numbers in their bug database.

From an account logged in to the system, you can look at the client's version number

```
$ kvno zookeeper/devix@COTHAM
zookeeper/devix@COTHAM: kvno = 1
```

Recommended strategy

Rebuild your keytabs.

1. Take a copy of your current keytab dir, for easy reverting.
2. Use `ktlist -kt` to list the entries in each keytab.
3. Use `ls -al` to record their user + group values + permissions.
4. In `kadmin.local`, re-export every key to the keytabs which needed it with `xst -norandkey`
5. Make sure the file owners and permissions are as before.
6. Restart everything.

`kinit: Client not found in Kerberos database while getting initial credentials`

This is fun: it means that the user is not known.

Possible causes

1. The user isn't in the database.
2. You are trying to connect to a different KDC than the one you thought you were using.
3. You aren't who you thought you were.

`SIMPLE authentication is not enabled. Available: [TOKEN]"`

This surfaces on RPC connections when the client is trying to use "SIMPLE" (i.e. unauthenticated) RPC, when the service is set to only support Kerberos ("TOKEN")

in the client configuration, set `hadoop.security.authentication` to `kerberos`.

(There is a configuration option to tell clients that they can support downgrading to simple, but as it shouldn't be used, this document doesn't list it.

`Request is a replay (34))`

The destination thinks the caller is attempting some kind of replay attack

1. The KDC is seeing too many attempts by the caller to authenticate as a specific principal, assumes some kind of attack and rejects the request. This can happen if you have too many processes/ nodes all sharing the same principal. Fix: make sure you have `service/_HOST@REALM` principals for all the services, rather than simple `service@REALM` principals.
2. The timestamps of the systems are out of sync, so it looks like an old token be re-issued. Check them all, including that of the KDC, make sure NTP is working, etc, etc.

`AuthenticationToken ignored`

This has been seen in the HTTP logs of Hadoop REST/Web UIs:

```
WARN org.apache.hadoop.security.authentication.server.AuthenticationFilter:
AuthenticationToken ignored:
org.apache.hadoop.security.authentication.util.SignerException: Invalid signature
```

This means that the caller did not have the credentials to talk to a Kerberos-secured channel.

1. The caller may not be logged in.
2. The caller may have been logged in, but its kerberos token has expired, so its authentication headers are not considered valid any more.
3. The time limit of a negotiated token for the HTTP connection has expired. Here the calling app is expected to recognise this, discard its old token and renegotiate a new one. If the calling app is a YARN hosted service, then something should have been refreshing the tokens for you.

`Found unsupported keytype (8)`

Happens when the keytype supported by the KDC isn't supported by the JVM

Generate a keytab with a supported key encryption type.

"User: MyUserName is not allowed to impersonate ProxyUser"

Your code has tried to create a proxy user and talk to a Hadoop service with it, but you do not have the right to impersonate that user. Only specific Hadoop services (Oozie, YARN RM) have the right to do what is the Hadoop equivalent of (`su $user`)

See Proxy user - Superusers Acting On Behalf Of Other Users

`Kerberos credential has expired`

Seen on an IBM JVM in HADOOP-9969

```
javax.security.sasl.SaslException:
  Failure to initialize security context [Caused by org.ietf.jgss.GSSException, major code: 8,
 minor code: 0
  major string: Credential expired
  minor string: Kerberos credential has expired]
```

The kerberos ticket has expired and not been renewed.

Possible causes

- The renewer thread somehow failed to start.
- The renewer did start, but didn't try to renew in time.
- A JVM/Hadoop code incompatibility stopped renewing from working.
- Renewal failed for some other reason.
- The client was kinited in and the token expired.
- Your VM clock has jumped forward and the ticket now out of date without any renewal taking place.

SASL `No common protection layer between client and server`

Not Kerberos, SASL itself:

```
16/01/22 09:44:17 WARN Client: Exception encountered while connecting to the server :
javax.security.sasl.SaslException: DIGEST-MD5: No common protection layer between client and s
erver
  at com.sun.security.sasl.digest.DigestMD5Client.checkQopSupport(DigestMD5Client.java:418)
  at com.sun.security.sasl.digest.DigestMD5Client.evaluateChallenge(DigestMD5Client.java:221)
  at org.apache.hadoop.security.SaslRpcClient.saslConnect(SaslRpcClient.java:413)
  at org.apache.hadoop.ipc.Client$Connection.setupSaslConnection(Client.java:558)
  at org.apache.hadoop.ipc.Client$Connection.access$1800(Client.java:373)
  at org.apache.hadoop.ipc.Client$Connection$2.run(Client.java:727)
  at org.apache.hadoop.ipc.Client$Connection$2.run(Client.java:723)
  at java.security.AccessController.doPrivileged(Native Method)
  at javax.security.auth.Subject.doAs(Subject.java:422)
  at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1657)
  at org.apache.hadoop.ipc.Client$Connection.setupIOstreams(Client.java:722)
  at org.apache.hadoop.ipc.Client$Connection.access$2800(Client.java:373)
  at org.apache.hadoop.ipc.Client.getConnection(Client.java:1493)
  at org.apache.hadoop.ipc.Client.call(Client.java:1397)
  at org.apache.hadoop.ipc.Client.call(Client.java:1358)
  at org.apache.hadoop.ipc.ProtobufRpcEngine$Invoker.invoke(ProtobufRpcEngine.java:229)
  at com.sun.proxy.$Proxy23.renewLease(Unknown Source)
  at org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolTranslatorPB.renewLease(ClientNam
enodeProtocolTranslatorPB.java:590)
  at sun.reflect.GeneratedMethodAccessor9.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:497)
```

On windows: `No authority could be contacted for authentication`

Reported on windows clients, especially related to the Hive ODBC client. This is kerberos, just someone else's library.

1. Make sure that your system is happy in the AD realm, etc. Do this first.
2. Make sure you've configured the ODBC driver according to the instructions.

During service startup `java.lang.RuntimeException: Could not resolve Kerberos principal name: + unknown error`

This something which can arise in the logs of a service. Here, for example, is a datanode failure.

```
Could not resolve Kerberos principal name: java.net.UnknownHostException: xubunty: xubunty: un
known error
```

This is *not* a kerberos problem. It is a network problem being misinterpreted as a Kerberos problem, purely because it surfaces in security code which assumes that all failures must be Kerberos related.

```
2016-04-06 11:00:35,796 ERROR org.apache.hadoop.hdfs.server.datanode.DataNode: Exception in se
cureMain java.io.IOException: java.lang.RuntimeException: Could not resolve Kerberos principal
 name: java.net.UnknownHostException: xubunty: xubunty: unknown error
  at org.apache.hadoop.http.HttpServer2.<init>(HttpServer2.java:347)
  at org.apache.hadoop.http.HttpServer2.<init>(HttpServer2.java:114)
  at org.apache.hadoop.http.HttpServer2$Builder.build(HttpServer2.java:290)
  at org.apache.hadoop.hdfs.server.datanode.web.DatanodeHttpServer.<init>(DatanodeHttpServer.j
ava:108)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.startInfoServer(DataNode.java:781)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.startDataNode(DataNode.java:1138)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.<init>(DataNode.java:432)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.makeInstance(DataNode.java:2423)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.instantiateDataNode(DataNode.java:2310)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.createDataNode(DataNode.java:2357)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.secureMain(DataNode.java:2538)
  at org.apache.hadoop.hdfs.server.datanode.DataNode.main(DataNode.java:2562)
Caused by: java.lang.RuntimeException: Could not resolve Kerberos principal name: java.net.Unk
nownHostException: xubunty: xubunty: unknown error
  at org.apache.hadoop.security.AuthenticationFilterInitializer.getFilterConfigMap(Authenticat
ionFilterInitializer.java:90)
  at org.apache.hadoop.http.HttpServer2.getFilterProperties(HttpServer2.java:455)
  at org.apache.hadoop.http.HttpServer2.constructSecretProvider(HttpServer2.java:445)
  at org.apache.hadoop.http.HttpServer2.<init>(HttpServer2.java:340)
  ... 11 more
Caused by: java.net.UnknownHostException: xubunty: xubunty: unknown error
  at java.net.InetAddress.getLocalHost(InetAddress.java:1505)
  at org.apache.hadoop.security.SecurityUtil.getLocalHostName(SecurityUtil.java:224)
  at org.apache.hadoop.security.SecurityUtil.replacePattern(SecurityUtil.java:192)
  at org.apache.hadoop.security.SecurityUtil.getServerPrincipal(SecurityUtil.java:147)
  at org.apache.hadoop.security.AuthenticationFilterInitializer.getFilterConfigMap(Authenticat
ionFilterInitializer.java:87)
  ... 14 more
Caused by: java.net.UnknownHostException: xubunty: unknown error
  at java.net.Inet4AddressImpl.lookupAllHostAddr(Native Method)
  at java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:928)
  at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1323)
  at java.net.InetAddress.getLocalHost(InetAddress.java:1500)
  ... 18 more2016-04-06 11:00:35,799 INFO org.apache.hadoop.util.ExitUtil: Exiting with status
 12016-04-06 11:00:35,806 INFO org.apache.hadoop.hdfs.server.datanode.DataNode: SHUTDOWN_MSG:
{code}
```

The root cause here was that the host `xubunty` which a service was configured to start on did not have an entry in `/etc/hosts`, nor DNS support. The attempt to look up the IP address of the local host failed.

The fix: add the short name of the host to `/etc/hosts`.

This example shows why errors reported as Kerberos problems, be they from the Hadoop stack or in the OS/Java code underneath, are not always Kerberos problems. Kerberos is fussy about networking; the Hadoop services have to initialize Kerberos before doing any other work. As a result, networking problems often surface first in stack traces belonging to the security classes, wrapped with exception messages implying a Kerberos problem. Always follow down to the innermost exception in a trace as the immediate symptom of a problem, the layers above attempts to interpret that, attempts which may or may not be correct.

Against Active Directory: `Realm not local to KDC while getting initial credentials`

Nobody quite knows.

It's believed to be related to Active Directory cross-realm/forest stuff, but there are hints that it can also be raised when the kerberos client is trying to auth with a KDC, but supplying a hostname rather than the realm.

This may be because you have intentionally or unintentionally created A Disjoint Namespace.aspx))

If you read that article, you will get the distinct impression that even the Microsoft Active Directory team are scared of Disjoint Namespaces, and so are going to a lot of effort to convince you not to go there. It may seem poignant that even the developers of AD are scared of this, but consider that these are probably inheritors of the codebase, not the original authors, and the final support line for when things don't work. Their very position in the company means that they get the worst-of-the-worst Kerberos-related problems. If they say "Don't go there", it'll be based on experience of fielding those support calls *and from having seen the Active Directory source code.*

- Kerberos and the Disjoint Namespace
- Kerberos Principal Name Canonicalization and Cross-Realm Referrals

`Can't get Master Kerberos principal`

The application wants to talk to a service (HDFS, YARN, HBase, ...), but it cannot determine the Kerberos principal which it should obtain a ticket for. This usually means that the client configuration doesn't declare the principal in the appropriate option —for example "dfs.namenode.kerberos.principal".

`SIMPLE authentication is not enabled. Available: [TOKEN, KERBEROS]`

This surfaces in the client:

```
org.apache.hadoop.security.AccessControlException: SIMPLE authentication is not enabled. Available:[TOKEN, KERBEROS]
```

The client doesn't think security is enabled; it's only trying to use "SIMPLE" authentication, —the caller is whoever they say they are. However, the server will only take Kerberos tickets or Hadoop delegation tokens which were previously acquired by by a caller with a valid Kerberos ticket. It is rejecting the authentication request.

`Null realm name (601)`

```
Null realm name (601) - default realm not specified
```

Comes from the JDK, Oracle docs say

> Cause: The default realm is not specified in the Kerberos configuration file krb5.conf (if used), provided as a part of the user name, or specified via the java.security.krb5.realm system property.
>
> Solution: Verify that your Kerberos configuration file (if used) contains an entry specifying the default realm, or directly specify it by setting the value of the java.security.krb5.realm system property and/or including it in your user name when authenticating using Kerberos."

This happens when the JVM option `java.security.krb5.realm` is set to "".

Who would do that? `hadoop-env.sh` does it by default on MacOS: in HADOOP-8719 someone deliberately patched `hadoop-env.sh` to clear the kerberos realm settings when running on macos

While this may have been valid in 2014, it is not valid any longer.

Fix: open the hadoop-env shell script, search for the string `java.security.krb5.realm`; delete the entire clause where it and its sibling options are cleared when OS == Darwin.

Fixed in HADOOP-15966 for Hadoop 3.0.4+, 3.1.3 and 3.2.1+

Tales of Terror

The following are all true stories. We welcome more submissions of these stories, especially covering the steps taken to determine what was wrong.

The Zookeeper's Birthday Present

A client program could not work with zookeeper: the connections were being broken. But it was working for everything else.

The cluster was one year old that day.

It turns out that ZK reacts to an auth failure by logging something in its logs, and breaking the client connection —without any notification to the client. Rather than a network problem (initial hypothesis), this a Kerberos problem. How was that worked out? By examining the Zookeeper logs —there was nothing client-side except the reports of connections being closed and the ZK client attempting to retry.

When a Kerberos keytab is created, the entries in it have a lifespan. The default value is one year. This was its first birthday, hence ZK wouldn't trust the client.

Fix: create new keytabs, valid for another year, and distribute them.

The Principal With No Realm

This one showed up during release testing —credit to Andras Bokor for tracking it all down.

A stack trace

```
16/01/16 01:42:39 WARN ipc.Client: Exception encountered while connecting to the server :
javax.security.sasl.SaslException: GSS initiate failed
[Caused by GSSException: No valid credentials provided (Mechanism level: Failed to find any Ke
rberos tgt)]
java.io.IOException: Failed on local exception: java.io.IOException:
javax.security.sasl.SaslException: GSS initiate failed
[Caused by GSSException: No valid credentials provided
(Mechanism level: Failed to find any Kerberos tgt)]; Host Details :
local host is: "os-u14-2-2.novalocal/172.22.73.243"; destination host is: "os-u14-2-3.novaloca
l":8020;
  at org.apache.hadoop.net.NetUtils.wrapException(NetUtils.java:773)
  at org.apache.hadoop.ipc.Client.call(Client.java:1431)
  at org.apache.hadoop.ipc.Client.call(Client.java:1358)
  at org.apache.hadoop.ipc.ProtobufRpcEngine$Invoker.invoke(ProtobufRpcEngine.java:229)
  at com.sun.proxy.$Proxy11.getFileInfo(Unknown Source)
  at org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolTranslatorPB.getFileInfo(ClientNa
menodeProtocolTranslatorPB.java:771)
  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.io.retry.RetryInvocationHandler.invokeMethod(RetryInvocationHandler.jav
a:252)
  at org.apache.hadoop.io.retry.RetryInvocationHandler.invoke(RetryInvocationHandler.java:104)
  at com.sun.proxy.$Proxy12.getFileInfo(Unknown Source)
  at org.apache.hadoop.hdfs.DFSClient.getFileInfo(DFSClient.java:2116)
  at org.apache.hadoop.hdfs.DistributedFileSystem$22.doCall(DistributedFileSystem.java:1315)
  at org.apache.hadoop.hdfs.DistributedFileSystem$22.doCall(DistributedFileSystem.java:1311)
  at org.apache.hadoop.fs.FileSystemLinkResolver.resolve(FileSystemLinkResolver.java:81)
  at org.apache.hadoop.hdfs.DistributedFileSystem.getFileStatus(DistributedFileSystem.java:131
1)
  at org.apache.hadoop.fs.FileSystem.exists(FileSystem.java:1424)
```

This looks like a normal "not logged in" problem, except for some little facts:

1. The user was logged in.
2. The failure was replicable.
3. It only surfaced on OpenJDK, not oracle JDK.
4. Everything worked on OpenJDK 7u51, but not on OpenJDK 7u91.

Something had changed in the JDK to reject the login on this system (ubuntu, virtual test cluster).

`Kdiag` didn't throw up anything obvious. What did show some warning was `klist`:

```
Ticket cache: FILE:/tmp/krb5cc_2529
Default principal: qe@REALM

Valid starting       Expires              Service principal
01/16/2016 11:07:23  01/16/2016 21:07:23  krbtgt/REALM@REALM
  renew until 01/23/2016 11:07:23
01/16/2016 13:13:11  01/16/2016 21:07:23  HTTP/hdfs-3-5@
  renew until 01/23/2016 11:07:23
01/16/2016 13:13:11  01/16/2016 21:07:23  HTTP/hdfs-3-5@REALM
  renew until 01/23/2016 11:07:23
```

See that? There's a principal which doesn't have a stated realm. Does that matter?

In OracleJDK, and OpenJDK 7u51, apparently not. In OpenJDK 7u91, yes

There's some new code in `sun.security.krb5.PrincipalName` (Oracle OpenJDK copyright)

```
// Validate a nameStrings argument
private static void validateNameStrings(String[] ns) {
    if (ns == null) {
        throw new IllegalArgumentException("Null nameStrings not allowed");
    }
    if (ns.length == 0) {
        throw new IllegalArgumentException("Empty nameStrings not allowed");
    }
    for (String s: ns) {
        if (s == null) {
            throw new IllegalArgumentException("Null nameString not allowed");
        }
        if (s.isEmpty()) {
            throw new IllegalArgumentException("Empty nameString not allowed");
        }
    }
}
```

This checks the code, and rejects if nothing is valid. Now, how does something invalid get in? Setting `HADOOP_JAAS_DEBUG=true` and logging at debug turned up output,

With 7u51:

```
16/01/20 15:13:20 DEBUG security.UserGroupInformation: using kerberos user:qe@REALM
```

With 7u91:

```
16/01/20 15:10:44 DEBUG security.UserGroupInformation: using kerberos user:null
```

Which means that the default principal wasn't being picked up, instead some JVM specific introspection had kicked in —and it was finding the principal without a realm, rather than the one that was.

Fix: add a `domain_realm` in `/etc/krb5.conf` mapping hostnames to realms

```
[domain_realm]
  hdfs-3-5.novalocal = REALM
```

A `klist` then returns a list of credentials without this realm-less one in.

```
Valid starting       Expires              Service principal
01/17/2016 14:49:08  01/18/2016 00:49:08  krbtgt/REALM@REALM
  renew until 01/24/2016 14:49:08
01/17/2016 14:49:16  01/18/2016 00:49:08  HTTP/hdfs-3-5@REALM
  renew until 01/24/2016 14:49:08
```

Because this was a virtual cluster, DNS/RDNS probably wasn't working, presumably kerberos didn't know what realm the host was in, and things went downhill. It just didn't show in any validation operations, merely in the classic "no TGT" error.

The AD realm redirection failure

Real-life example:

- Company ACME has one ActiveDirectory domain per continent.
- Domains have mutual trust enabled.
- AD is also used for Kerberos authentication.
- Kerberos trust is handled by a few AD servers in the "root" domain.
- Hadoop cluster is running in Europe.

When a South American user opens a SSH session on the edge node, authentication is done by LDAP, no issue the dirty work is done by the AD servers. But when a S.A. user tries to connect to HDFS or HiveServer2, with principal `Me@SAM.ACME.COM`, then the Kerberos client must make several hops...

1. AD server for @SAM.ACME.COM says "no, can't create ticket for svc/somehost@EUR.ACME.COM"
2. AD server for @SAM.ACME.COM says "OK, I can get you a credential to @ACME.COM, see what they can do there" alas,
3. There's no AD server defined in conf file for @ACME.COM
4. This leads to the all to familiar message, `Fail to create credential. (63) - No service creds`

Of course the only thing displayed in logs was the final error message. Even after enabling the "secret" debug flags, it was not clear what the client was trying to do with all these attempts. But the tell-tale was the "following capath" comment on each hop, because `CAPATH` is actually an optional section in `krb5.conf`. Fix: add the information about cross realm authentication to the `krb5.conf` file.

(CAPATH coverage: MIT, Redhat

The Limits of Hadoop Security

What are the limits of Hadoop security? Even with Kerberos enabled, what vulnerabilities exist?

Unpatched and 0-day holes in the layers underneath.

The underlying OS in a Hadoop cluster may have known or 0-day security holes, allowing a malicious (YARN?) application to gain root access to a host in the cluster. Once this is done it would have direct access to blocks stored by the datanode, and to secrets held in the various processes, including keytabs in the local filesystems.

Defences

1. Keep up to date with security issues. (SANS is worth tracking), and keep servers up to date.
2. Isolate the Hadoop cluster from the rest of your network infrastructure, apart from some "edge" nodes, so that only processes running in the cluster.
3. Developers: ensure that your code works with the more up to date versions of operating systems, JDKs and dependent libraries, so that you not holding back the upgrades. Do not increase the risk for the operations team.

Failure of users to keep their machines secure

The etnernal problem. Securing end-user machines is beyond the scope of the Hadoop project.

However, one area where Hadoop may impose risk on the end-user systems is the use of Java as the runtime for client-side code, so mandating an installation of the JVM on those users who need to directly talk to the Hadoop services. Ops teams should

- Make sure that an up to date JVM/JRE is installed, out of date ones are uninstalled, and that Java Applets in browsers are completely disabled.
- Control access to those Hadoop clusters and the services deployed on them.
- Use HDFS Quotas and YARN Queues to limit the resources malicious code can do.
- Collect the HDFS audit logs and learn how to use them to see if, after any possible security breach, you are in a position to even state what data was accessed by a specific user in a given time period.

We Hadoop developers need to

1. Make sure that our code works with current versions of Java, and test against forthcoming releases (a permanent trouble spot).
2. Make sure that our own systems are not vulnerable due to the tools installed locally.
3. Work to enable thin-client access to services, through REST APIs over Hadoop IPC and other

Java protocols, and by helping the native-client work.
4. Ensure our applications do not blindly trust users —and do as much as possible to prevent privilege escalation.
5. Log information for ops teams to use in security audits.

Denial of service attacks.

Hadoop is its own Distributed Denial of Service platform. A misconfiguration could easily trigger all datanodes to attempt to report in so frequently that the namenode gets overloaded, triggering apparent timeouts of some DN heartbeats, leading to the namenode assuming it has failed and starting block transfers of under-replicated blocks, so impacting network load and reporting even more. This is not a hypothetical example: Facebook had a cluster outage from precisely such an event, a failing switch partitioning the cluster and triggering a cascade failure. Nowadays IPC throttling (from Twitter) and the use of different ports on the namenode for heartbeating and filesystem operations (from Facebook) try to keep this under control.

We're not aware of any reported deliberate attempts to use a Hadoop cluster to overload local/remote services, though there are some anecdotes of the Yahoo! search engines having be written so as to deliberately stripe searches not just across hosts, but domains and countries, so as not to overload the DNS infrastructure of small countries. If you have some network service in your organisation which is considered critical (Examples: sharepoint, exchange), then configure the firewall rules to block access to those hosts and service ports from the Hadoop cluster.

Other examples of risk points and mitigation strategies

YARN resource overload

Too many applications asking for small numbers of containers, consuming resources in the Node Managers and RM. There are minimum size values for YARN container allocations for a reason: it's good to set them low on a single node development VM, but in production, they are needed

DNS overload.

This is easily done by accident. Many of the large clusters have local caching DNS servers for this reason, especially those doing any form of search.

CPU, network IO, disk IO, memory

YARN applications can consume so much local resources that they hurt the performance of other applications running on the same nodes.

In Linux and Windows, CPU can be throttled, the amount of physical and virtual memory limited. We could restrict disk and network IO (see relevant JIRAs), but that won't limit HDFS IO, which takes place in a different process. YARN labels do let you isolate parts of the cluster, so that low-latency YARN applications have access to machines across the racks which IO-heavy batch/background applications do not.

Deliberate insertion of malicious code into the Hadoop stack, dependent components or underlying OS.

We haven't encountered this yet. Is it conceivable? Yes: in the security interfaces and protocols themselves. Anything involving encryption protocols, random number generation and authentication checks would be the areas most appealing as targets: break the authentication or weaken the encryption and data in a Hadoop cluster becomes more accessible. As stated, we've not seen this. As Hadoop relies on external libraries for encryption, we have to trust them (and any hardware implementations), leaving random number generation and authentication code as targets. Given that few committers understand Hadoop Kerberos, especially at the REST/SPNEGO layer, it is hard for new code submissions in this area to be audited well.

One risk we have to consider is: if someone malicious had access to the committer credentials of a developer, could they insert malicious code? Everyone in the Hadoop team would notice changes in the code appearing without associated JIRA entries, though it's not clear how well reviewed the code is.

Mitigation strategies:

A key one has to be "identify those areas which would be vulnerable to deliberate weakening, and audit patch submissions extra rigorously there", "reject anything which appears to weaken security -even something as simple as allowing IP addresses instead of Hostnames in kerberos binding (cite: JIRA) could be dangerous. And while the submitters are probably well-meaning, we should assume maliciousness or incompetence in the high-risk areas. (* yes, this applies to my own patches too. The accusation of incompetence is defendable based on past submissions anyway).

Insecure applications

SQL injection attacks are the classic example here. It doesn't matter how secure the layers are underneath if the front end application isn't handling untrusted data. Then there are things like emergency patches to apple watches because of a binary parse error in fonts.

Mitigation strategies

1. Assume all incoming data is untrusted. In particular, all strings used in queries, while all documents (XML, HTML, binary) should be treated as potentially malformed, if not actually malicious.
2. Use source code auditing tools such as Coverity Scan to audit the code. Apache projects have free access to some of these tools.
3. Never have your programs ask for more rights than they need, to data, to database tables (and in HBase and Accumulo: columns)
4. Log data in a form which can be used for audit logs. (Issue: what is our story here? Logging to local/remote filesystems isn't it, not if malware could overwrite the logs)

Checklists

All programs

[ ] Sets up security before accessing any Hadoop services, including FileSystem APIs

[ ] Sets up security after loading local configuration files, such as `core-site.xml`. If you try to open an `hdfs://` filesystem, an `HdfsConfiguration` instance is created, which pulls in `hdfs-default.xml` and `hdfs-site.xml`. To load in the Yarn settings, create an instance of `YarnConfiguration`.

[ ] Are tested against a secure cluster from a logged in user.

[ ] Are tested against a secure cluster from a not logged in user, and the program set up to use a keytab (if supported).

[ ] Are tested from a logged out user with a token file containing the tokens and the environment variable `HADOOP_TOKEN_FILE_LOCATION` set to this file. This verifies Oozie can support it without needing a keytab.

[ ] Can get tokens for all services which they may optionally need.

[ ] Don't crash on a secure cluster if the cluster filesystem does not issue tokens. That is: Kerberized clusters where the FS is something other than HDFS.

Hadoop RPC Service

[ ] Principal for Service defined. This is generally a configuration property.

[ ] `SecurityInfo` subclass written.

[ ] `META-INF/services/org.apache.hadoop.security.SecurityInfo` resource lists.

[ ] the `SecurityInfo` subclass written

[ ] `PolicyProvider` subclass written.

[ ] RPC server handed `PolicyProvider` subclass during setup.

[ ] Service verifies that caller has authorization for the action before executing it.

[ ] Service records authorization failures to audit log.

[ ] Service records successful action to audit log.

[ ] Uses `doAs()` to perform operations as the user making the RPC call.

YARN Client/launcher

[ ] `HADOOP_USER` env variable set on AM launch context in insecure clusters, and in launched containers.

[ ] In secure cluster: all delegation tokens needed (HDFS, Hive, HBase, Zookeeper) created and added to launch context.

YARN Application

[ ] Delegation tokens extracted and saved.

[ ] When launching containers, the relevant subset of delegation tokens are passed to the containers. (This normally omits the RM/AM token).

[ ] Container Credentials are retrieved in AM and containers.

[ ] Delegation tokens revoked during (managed) teardown.

YARN Web UIs and REST endpoints

[ ] Primary Web server: `AmFilterInitializer` used to redirect requests to the RM Proxy.

[ ] Other web servers: a custom authentication strategy is chosen and implemented.

Yarn Service

[ ] A strategy for token renewal is chosen and implemented

Web Service

[ ] `AuthenticationFilter` added to web filter chain

[ ] Token renewal policy defined and implemented. (Look at `TimelineClientImpl` for an example of this)

Clients

All clients

[ ] Supports keytab login and calls `UserGroupInformation.loginUserFromKeytab(principalName, keytabFilename)` during initialization.

[ ] Issues `UserGroupInformation.getCurrentUser().checkTGTAndReloginFromKeytab()` call during connection setup/token reset. This is harmless on an insecure or non-keytab client.

[ ] Client supports Authentication Token option

[ ] Client supports Delegation Token option. (not so relevant for most YARN clients)

[ ] For Delegation-token authenticated connections, something runs in the background to regularly update delegation tokens.

[ ] Tested against secure clusters with user logged out (kdestroy).

[ ] Logs basic security operations at INFO, with detailed operations at DEBUG level.

RESTful client

[ ] Jersey: URL constructor handles SPNEGO Auth

[ ] Code invoking Jersey Client reacts to 401/403 exception responses when using Authentication Token by deleting creating a new Auth Token and re-issuing request. (this triggers re-authentication)

Debugging Workflow

[ ] host has an IP address (`ifconfig` / `ipconfig`)

[ ] host has an FQDN: `hostname -f`

[ ] FQDN resolves to hostname `nslookup $hostname`

[ ] hostname responds to pings `ping $hostname`

[ ] reverse DNS lookup of IPAddr returns hostname

[ ] clock is in sync with rest of cluster: `date`

[ ] JVM has Java Crypto Extensions

[ ] keytab exists

[ ] keytab is readable by account running service.

[ ] keytab contains principals in listing `ktlist -kt $keytab`

[ ] keytab FQDN is in entry of form `shortname/$FQDN`

Glossary

- KPW - Kerberos Password Used to encrypt and validate session keys
- TGT - Kerberos Ticket Granting Ticket A special KST granting user access to TGS Stored in user's keytab via kinit or windows login
- KST - Kerberos Service Ticket
- KST[P,S] - A KST granting principal P access to service S
- DT - Delegation Token
- DT[P,R] - Allows holder to impersonate principal P and renew DT with service R
- JT - Job Token
- A secure random value authenticating communication between JT and TT about a given task
- BT - Block Access Token
- BT[P,B] - A BT granting principal P access to block B

Bibliography

1. IETF RFC 4120
2. Java 7 Kerberos Requirements
3. Java 8 Kerberos Requirements
4. Troubleshooting Kerberos on Java 7
5. Troubleshooting Kerberos on Java 8
6. JAAS Configuration (Java 8)
7. For OS/X users, the GUI ticket viewer is `/System/Library/CoreServices/Ticket\ Viewer.app`
8. [Colouris01], Colouris, Dollimore & Kindberg, 2001, *Distributed System Concepts and Design*,
9. Java 8 GSS API
10. Ubuntu Kerberos Wiki
11. Kerberos FAQ. Dates from 2000; many of the links are worthless
12. Kerberos With Clocks Adrift: History, Protocols, and Implementation

Hadoop Security

1. Adding Security to Apache Hadoop
2. The Role of Delegation Tokens in Apache Hadoop Security
3. Chapter 8. Secure Apache HBase
4. Hadoop Operations p135+
5. Hadoop Security Architecture
6. [HADOOP-9621] Document/analyze current Hadoop security model, [HADOOP-9621) (https://issues.apache.org/jira/browse/HADOOP-9621)

Kerberos, Active Directory and Apache Hadoop

1. Microsoft Technet Introduction to Kerberos.aspx)
2. [Kabakov14], Kabakov, Securing Hadoop environments with Kerberos and active directory, IBM, 2014
3. [Cesir14], Cesir, Enabling Kerberos on HDP and Integrating with Active Directory, Hortonworks, 2014.
4. [Cloudera15] Cloudera, Integrating Hadoop Security with Active Directory, 2015
5. Troubleshooting Kerberos Encryption Types; Nathan Park, Ping Identity, 2013. Grat

Acknowledgements

- Everyone who has struggled to secure Hadoop deserves to be recognised, their sacrifice acknowledged.
- Everyone who has got their application to work within a secure Hadoop cluster will have suffered without any appreciation; without anyone appreciating their effort. Indeed, all that they are likely to have received is complaints about how their software is late.

However, our best praise, our greatest appreciation, has to go to everyone who added logging statements in the Hadoop codepath.

Some of the content in this document was copied from a 2013 presentation by Kevin Minder.

Contributors to this Document

It is through the work of these brave people that we shall prevail!

- Samson Scharfrichter
- Aloklal99
- chuckleberryfinn
- camypaj
- Sean Busbey
- Daniel Darabos
- Vipin Rathor
- Josh Elser
- Peter MacNaughton

We shall honour their memories, and mourn their messy and painful departures from the world of the sane, cherishing the github PRs they left as their glimpsed the underpinnings of the world, and so condemned themselvers forever for a world of Kerberos support calls.